THE PREMIER PRESS
GAME DEVELOPMENT SERIES

CD INCLUDED

GAME SCRIPTING MASTERY

Alex Varanese

Series Editor
André LaMothe
CEO Xtreme Games LLC

Premier Press

Game Scripting Mastery

Alex Varanese

PREMIER PRESS
GAME DEVELOPMENT

Premier
Press™

Publisher: Stacy L. Hiquet
Marketing Manager: Heather Hurley
Acquisitions Editor: Mitzi Koontz
Series Editor: André LaMothe
Project Editor: Estelle Manticas
Copy Editor: Kezia Endsley
Interior Layout: Bill Hartman
Cover Designer: Mike Tanamachi
Indexer: Kelly Talbot
Proofreader: Sara Gullion

ISBN: 1-931841-57-8

Library of Congress Catalog Card Number: 2001099849

Printed in the United States of America

03 04 05 06 07 BH 10 9 8 7 6 5 4 3 2 1

Premier Press, a division of Course Technology
2645 Erie Avenue, Suite 41
Cincinnati, Ohio 45208

This book is dedicated to my parents, Ray and Sue, and to my sister Katherine, if for no other reason than the simple fact that they'd put me in a body bag if I forgot to do so.

Foreword

Programming games is so fun! The simple reason is that you get to code so many different types of subsystems in a game, regardless of whether it's a simple *Pac Man* clone or a complex triple-A tactical shooter. Coding experience is very enriching, whether you're writing a renderer, sound system, AI system, or the game code itself; all of these types of programming contain challenges that you get to solve. The best way to code in any of these areas is with the most knowledge you can absorb beforehand. This is why you should have a ton of programming books close at hand.

One area of game coding that hasn't gotten much exposure is scripting. Some games don't need scripting—whether or not a game does is often dependant on your development environment and team—but in a lot of cases, using scripting is an ideal way of isolating game code from the main engine, or even handling in-game cinematics. Most programmers, when faced with solving a particular coding problem (let's say handling NPC interaction, for instance), will usually decide to write their own elaborate custom language that integrates with their game code. With the scripting tools available today this isn't strictly necessary, but boy is it fun!

Many coders aren't aware of the range of scripting solutions available today; that's where this fine book comes in. *Game Scripting Mastery* is the *best* way to dive into the mysterious world of game scripting languages. You'll learn what a scripting language is and how one is written; you'll get to learn about Lua, Python, and Tcl and how to make them work with your game (I'm a hardcore proponent for Lua, by the way); and, of course, you'll learn about compiler theory. You'll even get to examine how a full scripting language is developed! There's lots of knowledge contain herein, and if you love coding games, I'm confident that you'll enjoy finding out more about this aspect of game programming. Have "The Fun!"

John Romero

John Romero

Acknowledgments

It all started as I was standing around with some friends of mine on the second day of the 2001 Xtreme Game Developer's Conference in Santa Clara, California, discussing the Premier Press game development series. At the time, I'd been doing a lot of research on the subject of compiler theory—specifically, how it could be applied to game scripting—and at the exact moment I mentioned that a scripting book would be a good idea, André Lamothe just happened to walk by. "Let's see what he thinks," I said, and pulled him aside. "Hey André, have you ever thought about a book on game scripting for your series?" I expected something along the lines of "that's not a bad idea", or "sure– it's already in production." What I got was surprising, to say the least.

"Why don't you write it?"

That was literally what he said. Unless you're on some sort of weird version of *Jeopardy!* where the rules of the game require you to phrase your answer in the form of a book deal, this is a pretty startling response. I blinked, thought about it for about a nanosecond, and immediately said okay. This is how I handle most important decisions, but the sheer magnitude of the events that would be set into motion by this particular one could hardly have been predicted at the time. Never question the existence of fate.

With the obligatory anecdote out of the way, there are a number of very important people I'd like to thank for providing invaluable support during the production of this book. It'd be nothing short of criminal if this list didn't start with Mitzi Foster, my acquisitions editor who demonstrated what can only be described as superhuman patience during the turbulent submission and evolution of the book's manuscript. Having to handle the eleventh-hour rewrites of entire chapters (and large ones at that) after they've been submitted and processed is an editor's nightmare—and only one of the many she put up with—but she managed to handle it in stride, with a consistently friendly and supportive attitude.

Next up is my copy editor, Kezia Endsley; if you notice the thorough grammatical correctness of even the *comments* in this book's code listings, you'll have her to thank. Granted, it'll only be a matter of time before the latest version of Microsoft's compilers have a comment grammar checking paperclip, dancing monkey, robot dog, or ethnically ambiguous baby, but her eye for detail is safely appreciated for now.

Lastly, rounding out the Game Scripting Mastery pit crew is Estelle Manticas, my project editor who really stepped up to the plate during the later parts of the project, somehow maintaining a sense of humor while planet Earth crumbled around us. Few people have what it takes to manage the workload of an entire book when the pressure's on, and she managed to make it look easy.

Of course, due to my relatively young age and penchant for burning through cash like NASA, I've relied on others to provide a roof over my head. The honor here, not surprisingly, goes to my parents. I'd like to thank my mom for spreading news of my book deal to every friend, relative, teacher, and mailman our family has ever known, and my dad for deciding that the best time to work so loudly on rebuilding the deck directly outside my room is somewhere around *zero o'clock in the morning*. I also can't forget my sister, Katherine—her constant need for me to drive her to work is the only thing that keeps me waking up at a decent hour. Thanks a lot, guys!

And last, and most certainly least, I *suppose* I should thank that Lamothe guy. Seriously though—I may have toiled endlessly on the code and manuscript, but André is the real reason this book happened (and was also its technical editor). I've gotta say thanks for letting my raid your fridge on a regular basis, teaching me everything I know about electrical engineering, dumping so many free books on me, answering my incessant and apparently endless questions, restraining yourself from ending our more heated arguments with a golf club, and of course, extending such an obscenely generous offer to begin with. It should be known that there's literally no one else in the industry that goes out of their way to help people out this much, and I'm only one of many who've benefited from it.

I'd also like to give a big thanks to John Romero, who took time out of his understandably packed schedule to save the day and write the book's Foreword. If not for him, I probably would've had to get my mom to do it.

Oh and by the way, just because I think they'll get a kick out of it, I'd like to close with some horrendously geeky shout-outs: thanks to Ironblayde, xms and Protoman—three talented coders, and the few people I actually talk to regularly online—for listening to my constant ranting, and encouraging me to finish what I start (if for no other reason than the fact that I'll stop blabbering about it). You guys suck. Seriously.

Now if you'll excuse me, I'm gonna wrap this up. I feel like I'm signing a yearbook.

About the Author

Alex Varanese has been obsessed with game development since the mid-1980's when, at age five, he first laid eyes—with both fascination and a strange and unexplainable sense of familiarity—on the 8-bit Nintendo Entertainment System. He's been an avid artist since birth as well, but didn't really get going as a serious coder until later in life, at around age 15, with QBASIC. He got his start as a professional programmer at age 18 as a Java programmer in the Silicon Valley area, working on a number of upstart B2B projects on the J2EE platform before working for about a year as both a semi-freelance and in-house graphic designer.

Feeling that life in the office was too restrictive, however, he's since shifted his focus back to game development and the pursuit of future technology. He currently holds the position of head designer and systems architect for eGameZone (`http://www.egamezone.net`), the successor venture to André LaMothe's Xtreme Games LLC. He spends his free time programming, rendering, writing about himself in the third person, yelling at popup ads, starring in an off-Broadway production of *Dude, Where's My Car? The Musical*, and demonstrating a blatant disregard for the posted speed limit.

Alex Varanese can be reached at `alex@amvbooks.com`, and is always ready and willing to answer any questions you may have about the book. Please, don't hesitate to ask!

Letter from the Series Editor

A long, long, time ago on an 8-bit computer far, far, away, you could get away with hard coding all your game logic, artificial intelligence, and so forth. These days, as they say on the Sopranos "forget about it.…" Games are simply too complex to even think about coding anymore—in fact, 99 percent of all commercial games work like this: a 3D game engine is developed, then an interface to the engine is created via a scripting language system (usually a very high-level language) based on a virtual machine. The scripting language is used by the game programmers, and even more so the game designers, to create the actual game logic and behaviors for the entire game. Additionally, many of the rules of standard programming, such as strict typing and single threaded execution, are broken with scripting languages. In essence, the load of game development falls to the game designers for logic and game play, and to game programmers for the 3D engine, physics, and core technologies of the engine.

So where does one start when learning to use scripting in games? Well, there's a lot of stuff on the Internet of course, and you can try to interface languages like Python, Lau, and others to your game, but I say you should know how to do it yourself from the ground up. And that’s what *Game Scripting Mastery* is all about. This book is a monster—Alex covers every detail you can possibly imagine about game scripting.

This is hard stuff, relatively speaking—we are talking about compiler theory, virtual machines, and multithreading here. However, Alex starts off assuming you know nothing about scripting or compilers, so even if you’re a beginner you will be able to easily follow along, provided you take your time and work through the material. By the end of the book you’ll be able to write a compiler and a virtual machine, as well as interface your language to your existing C/C++ game engine—in essence, you will have mastered game scripting! Also, you will never want to write another parser as long as you live.

In conclusion, if game scripting is something you've been interested in, and you want to learn it in some serious detail, then this book is the book for you. Moreover, this is the *only* book on the market (as we go to publication) about this subject. As this is the flagship treatise on game scripting, we've tried to give you everything *we* needed when figuring it out on our own—and I think we have done much, much more. You be the judge!

Sincerely,

[illegible]

André LaMothe
Series Editor

CONTENTS AT A GLANCE

Introduction xliv

Part One
Scripting Fundamentals 1

Chapter 1
An Introduction to Scripting. 3
Chapter 2
Applications of Scripting Systems. 29

Part Two
Command-Based Scripting 61

Chapter 3
Introduction to Command-Based Scripting 63
Chapter 4
Advanced Command-Based Scripting 113

Part Three
Introduction to Procedural Scripting Languages 153

Chapter 5
Introduction to Procedural Scripting Systems 155

Chapter 6
Integration: Using Existing Scripting Systems 173
Chapter 7
Designing a Procedural Scripting Language 335

Part Four
Designing and Implementing a Low-Level Language 367
Chapter 8
Assembly Language Primer. 369
Chapter 9
Building the XASM Assembler 411

Part Five
Designing and Implementing a Virtual Machine 565
Chapter 10
Basic VM Design and Implementation 567
Chapter 11
Advanced VM Concepts and Issues 651

Part Six
Compiling High-Level Code................749
Chapter 12
Compiler Theory Overview. 751
Chapter 13
Lexical Analysis 783
Chapter 14
Building the XtremeScript Compiler Framework 857

Chapter 15
Parsing and Semantic Analysis 983

Part Seven
Completing Your Training 1137

Chapter 16
Applying the System to a Full Game 1139
Chapter 17
Where to Go From Here 1179

Appendix A
What's on the CD? 1203

INDEX..1207

Contents

INTRODUCTION XLIV

PART ONE
SCRIPTING FUNDAMENTALS 1

CHAPTER 1
AN INTRODUCTION TO SCRIPTING 3

What Is Scripting? 5
Structured Game Content—A Simple Approach 6
Improving the Method with Logical and Physical Separation .. 10
The Perils of Hardcoding 12
Storing Functionality in External Files 14
How Scripting Actually Works 15
An Overview of Computer Programming 16
An Overview of Scripting 18
The Fundamental Types of Scripting Systems 20
Procedural/Object-Oriented Language Systems 21
Command-Based Language Systems 22
Dynamically Linked Module Systems 23
Compiled versus Interpreted Code 24
Existing Scripting Solutions 26
Ruby 26
Lua 27
Java 27
Summary 27

Chapter 2
Applications of Scripting Systems........29

The General Purpose of Scripting 30

Role Playing Games (RPGs) 32

Complex, In-Depth Stories 32

The Solution 33

Non-Player Characters (NPCs) 34

The Solution 35

Items and Weapons 41

The Solution 43

Enemies 45

The Solution 46

First-Person Shooters (FPSs) 50

Objects, Puzzles, and Switches (Obligatory Oh My!) 51

The Solution 52

Enemy AI 57

The Solution 59

Summary 60

Part Two
Command-Based Scripting 61

Chapter 3
Introduction to Command-Based Scripting..63

The Basics of Command-Based Scripting. 64

High-Level Engine Control. 65

Commands 68

Master of Your Domain 68

Actually Getting Something Done 69

Command-Based Scripting Overview. 69

Engine Functionality Assessment 69

Loading and Executing Scripts 71

Looping Scripts 73

Implementing a Command-Based Language 74
Designing the Language 74
Writing the Script 75
Implementation 75
Basic Interface 75
Execution. 78
Command and Parameter Extraction. 81
The Command Handlers 87
Scripting a Game Intro Sequence. 90
The Language. 91
The Script 92
The Implementation 93
Scripting an RPG Character's Behavior 95
The Language. 95
Improving the Syntax. 96
Managing a Game Character 97
The Script 99
The Implementation 101
The Demo's Main Loop 105
Concurrent Script Execution 109
Summary 110
On the CD 111
Challenges 111

Chapter 4
Advanced Command-Based Scripting.....113

New Data Types 115
Boolean Constants 115
Floating-Point Support. 115
General-Purpose Symbolic Constants 116
An Internal Constant List. 117
A Two-Pass Approach. 120
Loading Before Executing. 124

Simple Iterative and Conditional Logic 125
Conditional Logic and Game Flags 125
Grouping Code with Blocks 128
The Block List 129
Iterative Logic 131
Nesting 133
Event-Based Scripting 135
Compiling Scripts to a Binary Format 137
Increased Execution Speed 137
Detecting Compile-Time Errors 139
Malicious Script Hacking 139
How a CBL Compiler Works 140
Executing Compiled Scripts 142
Compile-Time Preprocessing 143
Parameters 144
Basic Script Preprocessing 146
File-Inclusion Implementation 149
Summary 150

PART THREE
INTRODUCTION TO PROCEDURAL
SCRIPTING LANGUAGES 153

CHAPTER 5
INTRODUCTION TO PROCEDURAL
SCRIPTING SYSTEMS 155

Overall Scripting Architecture 156
High-Level Code 157
Low-Level Code 158
The Virtual Machine 159
A Deeper Look at XtremeScript 161
High-Level Code/Compilation 162
Lexical Analysis 164

Parsing/Syntactic Analysis 164
Semantic Analysis 165
Intermediate Code Generation 165
Optimization 165
Assembly Language Generation 166
The Symbol Table 166
The Front End versus the Back End 166
Low-Level Code/Assembly 167
The Assembler 167
The Disassembler 167
The Debugger 167
The Virtual Machine 168

The XtremeScript System 169
High-Level 170
Low-Level 170
Runtime 170

Summary 171

CHAPTER 6
INTEGRATION: USING EXISTING
SCRIPTING SYSTEMS 173

Integration 174
Implementation of Scripting Systems 179
The Bouncing Head Demo 181
Lua (and Basic Scripting Concepts) 185
The Lua System at a Glance 185
The Lua Library 185
The luac Compiler 185
The lua Interactive Interpreter 186
The Lua Language 187
Comments 188
Variables 188

Data Types 191
Tables. 193
Advanced String Features. 197
Expressions 198
Conditional Logic. 200
Iteration. 201
Functions 203
Integrating Lua with C 205
Compiling a Lua Project. 206
Initializing Lua 207
Loading Scripts. 208
The Lua Stack 209
Exporting C Functions to Lua 215
Executing Lua Scripts 219
Importing Lua Functions 221
Manipulating Global Lua Variables from C 226
Re-coding the Alien Demo in Lua 228
Advanced Lua Topics 241
Web Links 242

Python 242

The Python System at a Glance. 242
Directory Structure. 243
The Python Interactive Interpreter 243
The Python Language. 244
Comments. 244
Variables 244
Data Types 246
Basic Strings. 247
String Manipulation 248
Lists 251
Expressions 254
Conditional Logic. 256
Iteration. 258
Functions 261

Integrating Python with C 263
Compiling a Python Project 263
Initializing Python 265
Python Objects 265
Re-coding the Alien Head Demo 277
Advanced Topics 286
Web Links 286
Tcl 287
ActiveStateTcl 288
The Distribution at a Glance 288
The tclsh Interactive Interpreter 289
What, No Compiler? 290
Tcl Extensions 290
The Tcl Language 291
Commands—The Basis of Tcl 291
Substitution 292
Comments 297
Variables 298
Arrays 301
Expressions 303
Conditional Logic 306
Iteration 308
Functions (User-Defined Commands) 310
Integrating Tcl with C 312
Compiling a Tcl Project 312
Initializing Tcl 313
Loading and Running Scripts 314
Calling Tcl Commands from C 315
Exporting C Functions as Tcl Commands 316
Returning Values from Tcl Commands 319
Manipulating Global Tcl Variables from C 320
Recoding the Alien Head Demo 322
Advanced Topics 330
Web Links 330

Which Scripting System Should You Use? 331
Scripting an Actual Game 333
Summary 333
On the CD 334

CHAPTER 7
DESIGNING A PROCEDURAL SCRIPTING LANGUAGE 335

General Types of Languages 337
Assembly-Style Languages 337
Upping the Ante 340
Functions 344
Object-Oriented Programming 346
XtremeScript Language Overview 349
Design Goals 349
Syntax and Features 351
Data Structures 351
Operators and Expressions 354
Code Blocks 358
Control Structures 358
Functions 361
Escape Sequences 363
Comments 363
The Preprocessor 363
Reserved Word List 364
Summary 365

Part Four
Designing and Implementing
a Low-Level Language 367

Chapter 8
Assembly Language Primer 369

What Is Assembly Language? 370
Why Assembly Now? 371
How Assembly Works 372
Instructions 372
Operands. 372
Expressions 373
Jump Instructions. 375
Conditional Logic. 377
Iteration. 380
Mnemonics versus Opcodes 383
RISC versus CISC 386
Orthogonal Instruction Sets 388
Registers 389
The Stack 389
Stack Frames/Activation Records. 392
Local Variables and Scope. 395
Introducing XVM Assembly. 397
Initial Evaluations. 398
The XVM Instruction Set. 399
Memory. 399
Arithmetic 400
Bitwise. 401
String Processing 402
Conditional Branching 402
The Stack Interface 403
The Function Interface. 403
Miscellaneous. 404

XASM Directives 404
Stack and Data 405
Functions 406
Escape Sequences 407
Comments 407
Summary of XVM Assembly 408
Summary 409

Chapter 9
Building the XASM Assembler 411

How a Simple Assembler Works 413
Assembling Instructions 414
Assembling Variables 416
Assembling Operands 420
Assembling String Literals 422
Assembling Jumps and Function Calls 423
XASM Overview 428
Memory Management 429
Input: Structure of an XVM Assembly Script 430
Directives 431
Instructions 439
Line Labels 440
Host API Function Calls 440
The _Main () Function 441
The _RetVal Register 441
Comments 442
A Complete Example Script 442
Output: Structure of an XVM Executable 444
Overview 444
The Main Header 445
The Instruction Stream 447
The String Table 451
The Function Table 453
The Host API Call Table 454

Implementing the Assembler 455
Basic Lexing/Parsing Theory. 456
Lexing 457
Parsing 459
Basic String Processing. 462
Vocabulary 462
A String-Processing Library 464
The Assembler's Framework 469
The General Interface 470
A Structural Overview. 470
Lexical Analysis/Tokenization 495
The Lexer's Interface and Implementation 496
Error Handling. 525
Parsing. 527
Initializing the Parser 528
Directives 529
Line Labels. 542
Instructions 543
Building the .XSE Executable 552
The Header 552
The Instruction Stream 553
The String Table 555
The Function Table. 556
The Host API Call Table 557
The Assembly Process 558
Loading the Source File 558
The First Pass 559
The Second Pass 560
Producing the .XSE 562
Summary 563
On the CD 564
Challenges 564

Part Five
Designing and Implementing a Virtual Machine....................565

Chapter 10
Basic VM Design and Implementation ..567

Ghost in the Virtual Machine. 568
Mimicking Hardware 569
The VM's Major Components 570
The Instruction Stream 571
The Runtime Stack. 571
Global Data Tables 571
Multithreading 573
Integration with the Host Application 573
A Brief Overview of a VM's Lifecycle 574
Loading the Script 574
Beginning Execution at the Entry Point 576
The Execution Cycle 576
Function Calls 578
Calling a Function 578
Returning From a Function 580
Termination and Shut Down 581
Structural Overview of the XVM Prototype. 582
The Script Header. 583
Runtime Values. 583
The Instruction Stream 584
The Runtime Stack 585
The Frame Index 586
The Function Table. 587
The Host API Call Table. 587
The Final Script Structure 588
Building the XVM Prototype. 589
Loading an .XSE Executable 590

An .XSE Format Overview. 590
The Header 594
The Instruction Stream 595
The String Table 599
The Function Table. 601
The Host API Call Table 602
Structure Interfaces 603
The Instruction Stream 604
The Runtime Stack. 616
The Function Table. 621
The Host API Call Table 621
Summary 622
Initializing the VM. 624
The Execution Cycle 627
Instruction Set Implementation 628
Handling Script Pauses 633
Incrementing the Instruction Pointer 634
Operand Resolution. 636
Instruction Execution and Result Storage. 637
Termination and Shut Down 646
Summary 648
On the CD 649
Challenges 649

CHAPTER 11
ADVANCED VM CONCEPTS AND ISSUES. 651

A Next Generation Virtual Machine 652
Two Versions of the Machine 652
Multithreading 653
Multithreading Fundamentals 654
Cooperative vs. Preemptive Multitasking 654
From Tasks to Threads 658
Concurrent Execution Issues 659

Loading and Storing Multiple Scripts 667
The g_Script Structure. 667
Loading Scripts. 671
Initialization and Shutdown 674
Handling a Script Array 674
Executing Multiple Threads 677
Tracking Active Threads 678
The Scheduler 679
The First Completed XVM Demo 682
Host Application Integration. 682
Running Scripts in Parallel with the Host. 683
Manual Time Slicing vs. Native Threads. 684
Introducing the Integration Interface 686
Calling Host API Functions from a Script 686
Calling Script Functions from the Host 687
Tracking Global Variables 689
The XVM's Public Interface 694
Which Functions Should Be Public? 694
Name Clashes 695
Public Constants 696
Implementing the Integration Interface 696
Basic Script Control Functions. 697
Host API Calls 700
Script Function Calls 711
Invoking a Script Function: Synchronous Calls 713
Calling a Scripting Function: Asynchronous Calls 719
Adding Thread Priorities 728
Priority Ranks vs. Time Slice Durations 730
Updating the .XSE Format 731
Updating XASM. 733
Parsing the SetPriority Directive 734
Updating the XVM. 735
Demonstrating the Final XVM 739
The Host Application. 739
The Demo Script. 739

Embedding the XVM 741
Defining the Host API 742
The Main Program 742
The Output 745
Summary 746
On the CD 746
Challenges 747

PART SIX
COMPILING HIGH-LEVEL CODE 749

CHAPTER 12
COMPILER THEORY OVERVIEW 751

An Overview of Compiler Theory 752
Phases of Compilation 753
Lexical Analysis/Tokenization 755
Parsing 760
Semantic Analysis 764
I-Code 765
Single-Pass versus Multi-Pass Compilers 766
Target Code Emission 768
The Front and Back Ends 768
Compiler Compilers 769
How XtremeScript Works with XASM 769
Advanced Compiler Theory Topics 771
Optimization 771
Preprocessing 773
Retargeting 778
Linking, Loading, and Relocatable Code 779
Targeting Hardware Architectures 780
Summary 782

Chapter 13
Lexical Analysis 783

The Basics **785**
From Characters to Lexemes 785
Tokenization 787
Lexing Methods 787
Lexer Generation Utilities 788
Hand-Written Lexers.................................... 788
The Lexer's Framework **793**
Reading and Storing the Text File...................... 793
Displaying the Results 795
Error Handling... 797
A Numeric Lexer...................................... **797**
A Lexing Strategy 798
State Diagrams... 799
States and Token Types................................. 800
Initializing the Lexer.................................. 800
Beginning the Lexing Process........................... 801
The Lexing Loop 802
Completing the Demo.................................... 809
Lexing Identifiers and Reserved Words................ **811**
New States and Tokens 812
The Test File 813
Upgrading the Lexer 814
Completing the Demo.................................... 819
The Final Lexer: Delimiters, Operators, and Strings **822**
Lexing Delimiters 822
New States and Tokens 822
Upgrading the Lexer 823
Lexing Strings 827
New States and Tokens 827
Upgrading the Lexer 828

Operators 831
Breaking Operators Down. 832
Building Operator State Transition Tables 836
New States and Tokens 840
Upgrading the Lexer 841
Completing the Demo. 849
Summary 855
On the CD 855
Challenges 856

Chapter 14
Building the XtremeScript Compiler Framework. 857

A Strategic Overview 858
The Front End 859
The Loader Module 860
The Preprocessor Module 861
The Lexical Analyzer Module 861
The Parser Module 862
The I-Code Module. 862
The Back End 863
The Code Emitter Module. 863
The XASM Assembler 863
Major Structures 863
The Source Code. 863
The Script Header 864
The Symbol Table. 864
The Function Table. 865
The String Table 866
The I-Code Stream 866
Interfaces and Encapsulation 866
The Compiler's Lifespan 867
Reading the Command Line. 867
Loading the Source Code 867

Preprocessing 867
Parsing 867
Code Emission 868
Invoking XASM 868
The Compiler's main () Function 868

The Command-Line Interface 870

The Logo and Usage Info 870
Reading Filenames 871
Implementation 872
Reading Options 874
Implementation 875

Elementary Data Structures 880

Linked Lists 880
The Interface 881
Stacks 888
The Interface 888

Initialization and Shutdown 890

Global Variables and Structures 890
Initialization 891
Shutting Down 892

The Compiler's Modules 893

The Loader Module 895

The Preprocessor Module 897

Single-Line Comments 898
Block Comments 899
Preprocessor Directives 902
Implementing #include 902
Implementing #define 903

The Compiler's Tables 904

The Symbol Table 905
The SymbolNode Structure 905
The Interface 907

The Function Table. 910

The FuncNode Structure 911

The Interface 911

The String Table 915

Integrating the Lexical Analyzer Module 916

Rewinding the Token Stream 916

Lexer States. 917

A New Source Code Format. 919

New Miscellaneous Functions 922

Adding a Look-Ahead Character 922

Handling Invalid Tokens 923

Returning the Current Token 925

Copying the Current Lexeme 926

Error-Printing Helper Functions 927

Resetting the Lexer 928

The Parser Module 928

Error Handling. 928

General Errors 928

Code Errors 928

Cascading Errors 930

The I-Code Module 932

Approaches to I-Code. 932

A Simplified Instruction Set 933

The XtremeScript I-Code Instruction Set 935

The XtremeScript I-Code Implementation 935

Instructions 936

Jump Targets. 938

Source Code Annotation 940

The Interface. 942

Adding Instructions 943

Adding Operands. 944

Retrieving Operands 945

Adding Jump Targets. 946

Adding Source Code Annotation 947
Retrieving I-Code Nodes 948

The Code-Emitter Module 949

Code-Emission Basics 949
The General Format 950
Global Definitions 951
Emitting the Header. 952
Emitting Directives. 953
Emitting Symbol Declarations 955
Emitting Functions 958
Emitting a Complete XVM Assembly File 966

Generating the Final Executable. 969

Wrapping It All Up 972

Initiating the Compilation Process 972
Printing Compilation Statistics 972
Hard-coding a Test Script. 975
The Function 976
The Symbols 976
The Code 977
The Results 980

Summary 981

On the CD 981

Challenges 982

CHAPTER 15
PARSING AND SEMANTIC ANALYSIS 983

What Is Parsing? 985

Syntactic versus Semantic Analysis 985
Expressing Syntax 987
Syntax Diagrams 987
Backus-Naur Form. 988
Choosing a Method of Grammar Expression 989
Parse Trees 989

How Parsing Works 993
Recursive Descent Parsing 994
The XtremeScript Parser Module 996
The Basics 996
Tracking Scope 996
Reading Specific Tokens 997
The Parsing Strategy 1000
Parsing Statements and Code Blocks 1001
Syntax Diagrams 1002
The Implementation. 1004
ParseSourceCode () 1004
Statements 1005
Blocks 1007
Parsing Declarations 1008
Function Declarations 1008
Parsing and Verifying the Function Name 1010
Parsing the Parameter List 1011
Parsing the Function's Body 1015
Variable and Array Declarations. 1017
Host API Function Declarations. 1021
The host Keyword 1022
Upgrading the Lexer 1022
Parsing and Processing the host Keyword 1023
Testing Code Emitter Module 1026
Parsing Simple Expressions 1028
An Expression Parsing Strategy 1028
Parsing Addition and Subtraction 1028
Multiplication, Division, and Operator Precedence 1030
Stack-Based Expression Parsing 1031
Understanding the Expression Parser 1033
Coding the Expression Parser 1037
Parsing Full Expressions 1048
New Factor Types 1048
Parsing Function Calls 1051

New Unary Operators 1053
New Binary Operators 1054
Logical and Relational Operators 1054
The Logical And Operator 1055
Relational Greater Than or Equal 1056
The Rest 1058
L-Values and R-Values 1058
A Standalone Runtime Environment 1058
The Host Application 1059
Reading the Command Line 1060
Loading the Script 1061
Running the Script 1062
The Host API 1062
PrintString () 1063
PrintNewline () and PrintTab () 1063
Registering the API 1064
Parsing Advanced Statements and Constructs 1064
Assignment Statements 1065
Function Calls 1073
return 1075
while Loops 1079
while Loop Assembly Representation 1079
Parsing while Loops 1081
break 1086
Parsing break 1088
continue 1090
for Loops 1092
Branching with if 1092
if Block Assembly Representation 1092
Parsing if Blocks 1094
Syntax Diagram Summary 1099
The Test Drive 1099
Hello, World! 1099
Drawing Rectangles 1102

The Bouncing Head Demo 1106
Anatomy of the Program 1107
The Host Application 1109
The Low-Level XVM Assembly Script 1116
The High-Level XtremeScript Script 1127
The Results 1132
Summary 1134
On the CD 1134
Challenges 1135

Part Seven
Completing Your Training 1137

Chapter 16
Applying the System to a Full Game 1139

Introducing Lockdown 1140
The Premise 1140
Initial Planning and Setup 1142
Phase One—Game Logic and Storyboarding 1142
Phase Two—Asset Requirement Assessment 1150
Phase Three—Planning the Code 1155
Scripting Strategy 1157
Integrating XtremeScript 1158
The Host API 1158
Miscellaneous Functions 1159
Enemy Droid Functions 1159
Player Droid Functions 1159
Registering the Functions 1160
Writing the Scripts 1161
The Ambience Script 1161
The Blue Droid's Behavior Script 1162
The Grey Droid's Behavior Script 1163

The Red Droid's Behavior Script 1167
Compilation 1171
Loading and Running the Scripts 1171
Speed Issues 1173
Minimizing Expressions. 1174
The XVM's Internal Timer 1174
How to Play Lockdown 1175
Controls 1175
Interacting with Objects 1176
The Zone Map. 1176
Battle. 1176
Completing the Objective 1176
Summary 1177
On the CD 1177
Challenges 1178

Chapter 17
Where to Go From Here 1179

So What Now? 1180
Expanding Your Knowledge 1181
Compiler Theory 1181
More Advanced Parsing Methods 1182
Object-Orientation 1182
Optimization 1183
Runtime Environments. 1184
The Java Virtual Machine 1184
Alternative Operating Systems. 1185
Operating System Theory 1186
Advanced Topics and Ideas 1186
The Assembler and Runtime Environment. 1186
A Lower-Level Assembler. 1186
A Lower-Level Virtual Machine. 1187

Dynamic Memory Allocation 1189
The Compiler and High-Level Language 1190
Summary 1199

Appendix A
What's on the CD? 1201

The CD-ROM Interface 1202
Installation 1203
DirectX SDK 1203

INDEX 1205

Introduction

If you've been programming games for any reasonable amount of time, you've probably learned that at the end of the day, the *really* hard part of the job has nothing to do with illumination models, doppler shift, file formats, or frame rates, as the majority of game development books on the shelves would have you believe. These days, it's more or less evident that everyone knows everything. Gone are the days where game development gurus separated themselves from the common folk with their in-depth understanding of VGA registers or their ability to write an 8-bit mixer in 4K of code. Nowadays, impossibly fast hardware accelerators and monolithic APIs that do everything short of opening your mail pretty much have the technical details covered. No, what really make the creation of a phenomenal game difficult are the characters, the plot, and the suspension of disbelief.

Until Microsoft releases "DirectStoryline"—which probably won't be long, considering the amount of artificial intelligence driving DirectMusic—the true challenge will be immersing players in settings and worlds that exert a genuine sense of atmosphere and organic life. The floor should creak and groan when players walk across aging hardwood. The bowels of a ship should be alive with scurrying rats and the echoey drip-drop sounds of leaky pipes. Characters should converse and interact with both the player and one another in ways that suggest a substantial set of gears is turning inside their heads. In a nutshell, a world without compellingly animated detail and believable responsiveness won't be suitable for the games of today and tomorrow.

The problem, as the first chapter of this book will explain, is that the only solution to this problem directly offered by languages like C and C++ is to clump the code for implementing a peripheral character's quirky attitude together with code you use to multiply matrices and sort vertex lists. In other words, you're forced to write all of your game—from the low-level details to the high-level logic—in the same place. This is an illogical grouping and one that leads to all sorts of hazards and inconveniences.

And let's not forget the modding community. Every day it seems that players expect more flexibility and expansion capabilities from their games. Few PC titles last long on the shelves if a community of rabid, photosensitive code junkies can't tear it open and rewire its guts. The problem is, you can't just pop up an Open File dialog box and let the player chose a DLL or other dynamically linked solution, because doing so opens you up to all sorts of security holes. What if a malicious mod author decides that the penalty for taking a rocket blast to the gut is a freshly reformatted hard drive? Because of this, despite their power and speed, DLLs aren't necessarily the ideal solution.

This is where the book you're currently reading comes into play. As you'll soon find out, a solution that allows you to both easily script and control your in-game entities and environments, as well as give players the ability to write mods and extensions, can only really come in the form of a custom-designed language whose programs can run within an embeddable execution environment inside the game engine. This is scripting.

If that last paragraph seemed like a mouthful, don't worry. This book is like an elevator that truly starts from the bottom floor, containing everything you need to step out onto the roof and enjoy the view when you're finished. But as a mentally unstable associate of mine is often heard to say, "The devil is in the details." It's not enough to simply know what scripting is all about; in order to really make something happen, you need to know *everything*. From the upper echelons of the compiler, all the way down to the darkest corners of the virtual machine, you need to know what goes where, and most importantly, *why*. That's what this book aims to do. If you start at the beginning and follow along with me until the end, you should pick up everything you need to genuinely understand what's going on.

How This Book is Organized

With the dramatic proclamations out of the way, let's take a quick look at how this book is set up; then we'll be ready to get started.

This book is organized into a number of sections:

- **Part One: Scripting Fundamentals.** The majority of this material won't do you much good if you don't know what scripting is or why it's important. Like I said, you can follow this book whether or not you've even *heard* of scripting. The introduction provides enough background information to get you up to speed quick.
- **Part Two: Command-Based Scripting.** Developing a complete, high-level scripting system for a procedural language is a complex task. A *very* complex task. So, we start off by setting our sights a bit lower and implementing what I like to call a "command-based language." As you'll see, command-based languages are dead simple to implement and capable of performing rather interesting tasks.
- **Part Three: Introduction to Procedural Scripting Languages.** Part 3 is where things start to heat up, as we get our feet wet with real world, high-level scripting. Also covered in

this section are complete tutorials on using the Lua, Python and Tcl languages, as well as integrating their associated runtime environments with a host application.

- **Part Four: Designing and Implementing a Low-Level Langauge.** At the bottom of our scripting system will lie an assembly language and corresponding machine code (or *bytecode*). The design and implementation of this low-level environment will provide a vital foundation for the later chapters.
- **Part Five: Designing and Implementing a Virtual Machine.** Scripts—even compiled ones—don't matter much if you don't have a way to run them. This section of the book covers the design and implementation of a feature-packed virtual machine that's ready to be dropped into a game engine.
- **Part Six: Compiling High-Level Code.** The belly of the beast itself. Executing compiled bytecode is one thing, but being able to compile and ultimately run a high-level, procedural language of your own design is what real scripting is all about.
- **Part Seven: Completing Your Training.** Once you've earned your stripes, it's time to direct that knowledge somewhere. This final section aims to clear up any questions you may have in regards to furthering your study. You'll also see how the scripting system designed throughout the course of the book was applied to a complete game.

So that's it! You've got a roadmap firmly planted in your brain, and an interest in scripting that's hopefully piqued by now. It's time to roll our sleeves up and *turn this mutha out.*

Part One

Scripting Fundamentals

This page intentionally left blank

CHAPTER 1

An Introduction to Scripting

"We'll bring you the thrill of victory, the agony of defeat, and because we've got soccer highlights, the sheer pointlessness of a zero-zero tie."
—*Dan Rydel,* Sports Night

It goes without saying that modern game development is a multi-faceted task. As so many books on the subject love to ask, what other field involves such a perfect synthesis of art, music and sound, choreography and direction, and hardcore programming? Where else can you find each of these subjects sharing such equal levels of necessity, while at the same time working in complete unison to create a single, cohesive experience? For all intents and purposes, the answer is nowhere. A game development studio is just about the only place you're going to find so many different forms of talent working together in the pursuit of a common goal. It's the only place that requires as much art as it does science; that thrives on a truly equal blend of creativity and bleeding-edge technology. It's that technical side that we're going to be discussing for the next few hundred pages or so. Specifically, as the cover implies, you're going to learn about s*cripting*.

You might be wondering what scripting is. In fact, it's quite possible that you've never even heard the term before. And that's okay! It's not every day that you can pick up a book with absolutely no knowledge of the subject it teaches and expect to learn from it, but *Game Scripting Mastery* is most certainly an exception. Starting now, you're going to set out on a slow-paced and almost painfully in-depth stroll through the complex and esoteric world of feature-rich, professional grade game scripting. We're going to start from the very beginning, and we aren't even going to slow down until we've run circles around everything.

This book is going to explain everything you'll need to know, but don't relax too much. If you genuinely want to be the master that this book can turn you into, you're going to have to keep your eyes open and your mind sharp. I won't lie to you, reader. Every single man or woman who has stood their ground; *everyone* who has fought an agent has died. The other thing I'm not going to lie to you about is that the type of scripting we're going to learn—the seat-of-your-pants, pedal-to-the-asphalt techniques that pro development studios use for commercial products—is hard stuff.

So before going any further, take a nice deep breath and understand that, if anything, you're going to finish this book having learned *more* than you expected. Yes, this stuff can be difficult, but I'm going to explain it with that in mind. *Everything* becomes simple if it's taught properly, completely, and from the very beginning.

But enough with the drama! It's time to roll up your sleeves, take one last look at the real world, and dive headlong into the almost entirely uncharted territory that programmers call "game scripting." In this chapter you will find

- An overview of what scripting is and how it works.
- Discussion on the fundamental types of scripting systems.
- Brief coverage of existing scripting systems.

What *Is* Scripting?

Not surprisingly, your first step towards attaining scripting mastery is to understand precisely what it is. Actually, my usual first step is breaking open a crate of 20 oz. Coke bottles and binge-drinking myself into a caffeine-induced frenzy that blurs the line between a motivated work ethic and cardiac arrest…but maybe that's just me.

To be honest, this is the tricky part. I spent a lot of time going over the various ways I could explain this, and in the end, I felt that I'd explain scripting to you in the same order that I originally stumbled upon it. It worked for me, which means it'll probably work for you. So, put on your thinking cap, because it's time to use your imagination.

Here's a hypothetical situation. You and some friends have decided to create a role-playing game, or *RPG*. So, being the smart little programmers you are, you sit down and draft up a design document—a fully-detailed game plan that lets you get all of your ideas down on paper before attempting to code, draw, or compose anything. At this point I could go off on a three-hour lecture about the necessity of design documents, and why programs written without them are doomed to fail and how the programmers involved will all end up in horrible snowmobile accidents, but that's not why I'm here. Instead, I am going to quickly introduce this hypothetical RPG and cover the basic tasks involved in its production. Rather than explain what scripting is directly, I'll actually run into the problems that scripting solves so well, and thus learn the hard way. The hypothetical hard way, that is.

So anyway, let's say the design document is complete and you're ready to plow through this project from start to finish. The first thing you need is the game engine; something that allows players to walk around and explore the game world, interact with characters, and do battle with enemies. Sounds like a job for the programmer, right? Next up you're going to need graphics. Lots of 'em. So tell the artist to give the Playstation a rest and get to work. Now on to music and sound. Any good RPG needs to be dripping with atmosphere, and music and sound are a big part of that. Your musician should have this covered.

But something's missing. Sure, these three people can pump out a great demo of the engine, with all the graphics and sound you want, but what makes it a *game?* What makes it memorable and fun to play? The answer is the *content*—the quest and the storyline, the dialogue, the descriptions of each weapon, spell, enemy, and all those other details that separate a demo from the next platinum seller.

Structured Game Content—A Simple Approach

So how exactly do you create a complete game? The programmer uses a compiler to code the design document specifications into a functional program, the artist uses image processing and creation software like Photoshop and 3D Studio MAX to turn concept art and sketches into graphics, and musicians use a MIDI composer or other tracking software to transform the schizophrenic voices in their heads into game music. The problem is, there really isn't any tool or utility for "inputting" stories and character descriptions. You can't just open up Microsoft VisualStoryline, type in the plot to your game, press F5 and suddenly have a game full of characters and dialogue.

There doesn't seem to be a clear solution here, but the game *needs* these things—it really can't be a "game" without them. And somehow, every other RPG on the market has done it.

The first and perhaps most obvious approach is to have the programmer manually code all this data into the engine itself. Sounds like a reasonable way to handle the situation, doesn't it? Take the items, for instance. Each item in your game needs a unique description that tells the engine how it should look and function whenever the player uses it. In order to store this information, you might create a `struct` that will describe an item, and then create an array of these structures to hold all of them. Here's an idea of what that structure might look like:

```
typedef struct _Item
{
   char * pstrName;   // What is the item called?
   int iType;      // What general type of item is it?
   int iPrice;      // How much should it cost in shops?
   int iPower;      // How powerful is it?
} Item;
```

Let's go over this a bit. `pstrName` is of course what the item is called, which might be "Healing Potion" or "Armor Elixir." `iType` is the general *type* of the item, which the engine needs in order to know how it should function when used. It's an integer, so a list of constants that describe its functionality should be defined:

```
const HEAL          = 0;
const MAGIC_RESTORE    = 1;
const ARMOR_REPAIR      = 2;
const TELEPORT      = 3;
```

This provides a modest but useful selection of item types. If an item is of type `HEAL`, it restores the player's health points (or HP as they're often called). Items of type `MAGIC_RESTORE` are similar; they restore a player's magic points (MP). `ARMOR_REPAIR` repairs armor (not surprisingly), and `TELEPORT` lets the player immediately jump to another part of the game world under certain conditions (or something to that effect, I just threw that in there to mix things up a bit).

Up next is `iPrice`, which lets the merchants in your game's item shops know how much they should charge the player in order to buy it. Sounds simple enough, right? Last is `iPower`, which essentially means that whatever this item is trying to do, it should do it with this amount, or to this extent. In other words, if your item is meant to restore HP (meaning its of type `HEAL`), and `iPower` is 32, the player will get 32 HP back upon using the item. If the item is of type `MAGIC_RESTORE`, and `iPower` is 64, the player will get 64 MP back, and so on and so forth.

That pretty much wraps up the item description structure, but the real job still lies ahead. Now that the game's internal structure for representing items has been established, it needs to be filled. That's right, all those tens or even hundreds of items your game might need now must be written out, one by one:

```
const MAX_ITEM_COUNT = 128;      // 128 items should be enough

Item ItemArray [ MAX_ITEM_COUNT ];

// First, let's add something to heal injuries:
ItemArray [ 0 ].pstrName = "Health Potion Lv 1";
ItemArray [ 0 ].iType = HEAL;
ItemArray [ 0 ].iPrice = 20;
ItemArray [ 0 ].iPower = 10;

// Next, wizards and mages and all those guys are gonna need this:
ItemArray [ 1 ].pstrName = "Magic Potion Lv 6";
ItemArray [ 1 ].iType = MAGIC_RESTORE;
ItemArray [ 1 ].iPrice = 250;
ItemArray [ 1 ].iPower = 60;

// Big burly warriors may want some of this:
ItemArray [ 2 ].pstrName = "Armor Elixir Lv 2";
ItemArray [ 2 ].iType = ARMOR_REPAIR;
ItemArray [ 2 ].iPrice = 30;
ItemArray [ 2 ].iPower = 20;
```

```
// To be honest, I have no idea what on earth this thing is:
ItemArray [ 3 ].pstrName = "Orb of Sayjack";
ItemArray [ 3 ].iType = TELEPORT;
ItemArray [ 3 ].iPrice = 3000;
ItemArray [ 3 ].iPower = NULL;
```

Upon recompiling the game, four unique items will be available for use. With them in place, let's imagine you take them out for a field test, to make sure they're balanced and well suited for gameplay. To make this hypothetical situation a bit easier to follow, you can pretend that the rest of the engine and game content is finished; that you already have a working combat engine with a variety of enemies and weapons, you can navigate a 3D world, and so on. This way, you can focus solely on the items.

The first field test doesn't go so well. It's discovered in battle that "Health Potion Lv 1" isn't strong enough to provide a useful HP boost, and that it ultimately does little to help the player tip the scales back in their favor after taking significant damage. The obvious solution is to increase the power of the potion. So, you go back to the compiler and make your change:

```
ItemArray [ 0 ].iPower = 50;      // More healing power.
```

The engine will have to be recompiled in order for adjustment to take effect, of course. A second field test will follow.

The second test is equally disheartening; more items are clearly unbalanced. As it turns out, "Armor Elixir Lv 2" restores a lot less of the armor's vitality than is taken away during battle with various enemies, so it'll need to be turned up a notch. On the other hand, the modification to "Health Potion Lv 1" was too drastic; it now restores too *much* health and makes the game too easy. Once again, these items' properties must be tweaked.

```
// First let's fix the Health Potion issue
ItemArray [ 0 ].iPower = 40;      // Sounds more fair.

// Now the Armor Elixir
ItemArray [ 2 ].iPower = 50;      // Should be more helpful now.
```

…and once again, you sit on your hands while everything is recompiled. Due to the complexity of the game engine, the compilation of its source code takes a quite while. As a result, the constant retuning demanded by the game itself is putting a huge burden on the programmer and wasting a considerable amount of time. It's necessary, however, so you head out into your third field test, hoping that things work out better this time.

And they don't. The new problem? "Magic Potion Lv 6" is a bit too expensive. It's easy for the player to reach a point where he desperately needs to restore his magic points, but hasn't been given enough opportunities to collect gold, and thus gets stuck. This is very important and must be fixed immediately.

```
ItemArray [ 1 ].iPrice = 80;      // This tweaking is getting old.
```

Once again, (say it with me now) you recompile the engine to reflect the changes. The balancing of items in an RPG is not a trivial task, and requires a great deal of field testing and constant adjusting of properties. Unfortunately, the length of this process is extended considerably by the amount of time spent recompiling the engine. To make matters worse, 99.9% of the code being recompiled hasn't even changed—two out of three of these examples only changed a single line!

Can you imagine how many times you're going to have to recompile for a full set of 100+ items before they've all been perfected? And that's just one *aspect* of an RPG. You're still going to need a wide variety of weapons, armor, spells, characters, enemies, all of the dialogue, interactions, plot twists, and so on. That's a massive amount of information. For a full game's worth of content, you're going recompile everything thousands upon *thousands* of times. And that's an optimistic estimation. Hope you've got a fast machine.

Now let's really think about this. Every time you make even the *slightest* change to your items, you have to recompile the entire game along with it. That seems a bit wasteful, if flat out illogical, doesn't it? If all you want to do is make a healing potion more effective, why should you have to recompile the 3D engine and sound routines too? They're totally unrelated.

The answer is that you shouldn't. The content of your game is media, just like art, sound, and music. If an artist wants to modify some graphics, the programmer doesn't have to recompile, right? The artist just makes the changes and the next time you run the game these changes are reflected. Same goes for music and sound. The sound technician can rewrite "Battle Anthem in C Minor" as often as desired, and the programmer never has to know about it. Once again, you just restart the game and the new music plays fine.

So what gives? Why is the game content singled out like this? Why is it the only type of media that can't be easily changed? The first problem with this method is that when you write your item descriptions directly in your game code, you have to recompile everything with it. Which sucks. But that's by no means the only problem. Figure 1.1 demonstrates this.

The problem with all of this constant recompilation is mostly a physical issue; it wastes a lot of time, repeats a lot of processing unnecessarily, and so on. Another major problem with this method is one of organization. An RPG's engine is complicated enough as it is; managing graphics, sound, and player input is a huge task and requires a great deal of code. But consider how much *more* hectic and convoluted that code is going to become when another 5,000 lines or so of item descriptions, enemy profiles, and character dialogue are added. It's a terrible way to organize things. Imagine if your programmer (which will most likely be you) had to deal with all the *other* game media while coding at the same time—imagine if the IDE was further cluttered by endless piles of graphics, music, and sound. A nervous breakdown would be the only likely outcome.

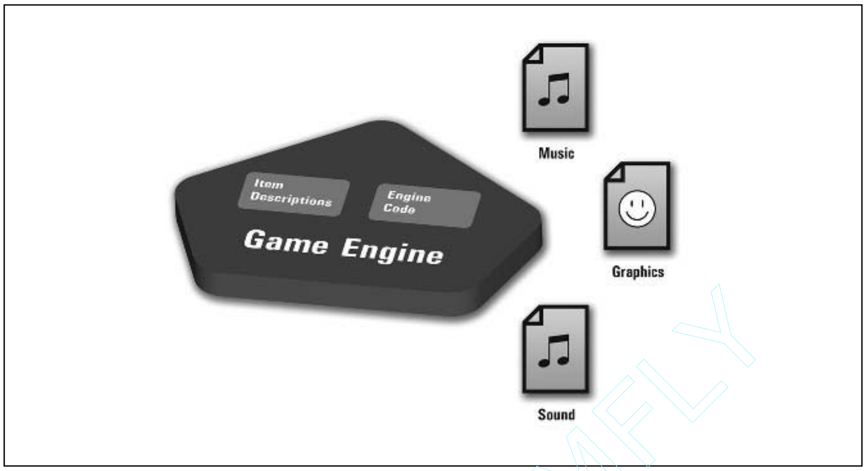


Figure 1.1

The engine code and item descriptions are part of the same source files, meaning you can't compile one without the other. Art, music, and sound, however, exist outside of the source code and are thus far more flexible.

Think about it this way—coding game content directly into your engine is a little like wearing a tuxedo every day of your life. Not only does it take a lot longer to put on a tux in the morning than it does to throw on a v-neck and some khakis, but it's inappropriate except for a few rare occasions. You're only going to go to a handful of weddings in your lifetime, so spending the time and effort involved in preparing for one on a daily basis will be a waste 98% of the time.

All bizarre analogies aside, however, it should now be clear why this is such a terrible way to organize things.

Improving the Method with Logical and Physical Separation

The situation in a nutshell is that you need an intelligent, highly structured way of *separating* your code from your game content. When you are working on the engine code, you shouldn't have to wade through endless item descriptions. Likewise, when you're working on item descriptions, the engine code should be miles away (metaphorically speaking, of course). You should also be able to change items drastically and as frequently as necessary, even after the game has been compiled, just like you can do with art, music, and sound. Imagine being able to get that slow, time-wasting compilation out of the way up front, mess with the items all you want, and have the changes show up immediately in the same executable! Sounds like quite an improvement, huh?

What's even better is how easy this is to accomplish. To determine how this is done, you need not look any further than that other game media—like the art and sound—that's been the subject of so much envy throughout this example. As you've learned rather painfully, they don't require a separate compile like the game content does; it's simply a matter of making changes and maybe restarting the game at worst. Why is this the case? Because they're stored in separate files. The game's only connection with this data is the code that reads it from the disk. They're loaded at *runtime*. At compile-time, they don't even have to be on the same hard drive, because they're unrelated to the source code. The game engine doesn't care what the data actually *is*, it just reads it and tosses it out there. So somehow, you need to offload your game content to external files as well. Then you can just write a single, compact block of code for loading in all of these items from the hard drive in one fell swoop. How slick is that? Check out Figure 1.2.

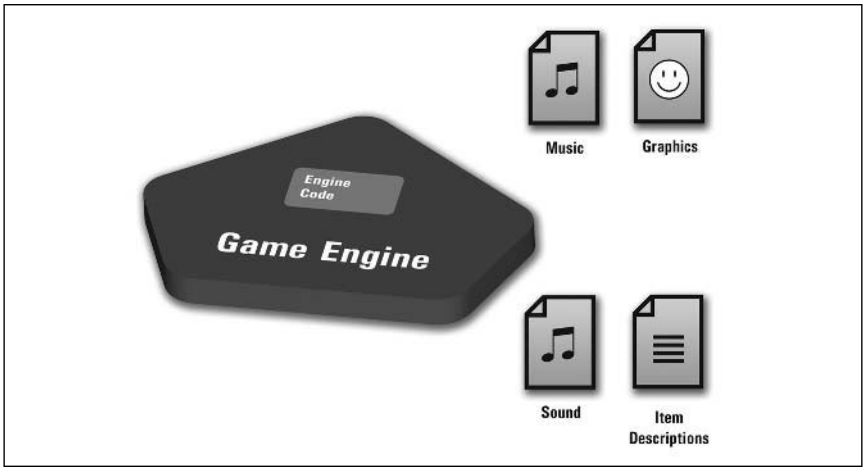


Figure 1.2

If you can get your item descriptions into external files, they'll be just as flexible as graphics and sound because they'll only be needed at runtime.

The first step in doing this is determining how you are going to store something like the following in a file:

```
ItemArray [ 1 ].pstrName = "Magic Potion Lv 6";
ItemArray [ 1 ].iType = MAGIC_RESTORE;
ItemArray [ 1 ].iPrice = 250;
ItemArray [ 1 ].iPower = 60;
```

In this example, the transition is going to be pretty simple. All you really need to do is take everything on the right side of the = sign and plop it into an ASCII file. After all, those are all of the actual values, whereas the assignment will be handled by the code responsible for loading it (called the *loader*). So here's what the Magic Potion looks like in its new, flexible, file-based form:

```
Magic Potion Lv 6
MAGIC_RESTORE
250
60
```

It's almost exactly the same! The only difference is that all the C/C++ code that it was wrapped up in has been separated and will be dealt with later. As you can see, the format of this item file is pretty simple; each attribute of the item gets its own line. Let's take a look at the steps you might take to load this into the game:

1. Open the file and determine which index of the item array to store its contents in. You'll probably be loading these in a loop, so it should just be a matter of referring to the loop counter.
2. Read the first string and store it in `pstrName`.
3. Read the next line. If the line is "`HEAL`", assign `HEAL` to `iType`. If it's "`MAGIC_RESTORE`" then assign `MAGIC_RESTORE`, and so on.
4. Read in the next line, convert it from a string to an integer, and store it in `iPrice`.
5. Read in the next line, convert it from a string to an integer, and store it in `iPower`.
6. Repeat steps 1-5 until all items have been loaded.

You'll notice that you can't just directly assign the item type to `iType` after reading it from the file. This is of course because the type is stored in the file as a string, but is represented in C/C++ as an integer constant. Also, note that steps 4 and 5 require you to convert the string to an integer before assigning it. This all stems from the fact that ASCII deals only with string data.

Well my friend, you've done it. You've saved yourself from the miserable fate that would've awaited you if you'd actually tried to code each item directly into the game. And as a result, you can now tweak and fine-tune your items without wasting any more time than you have to. You've also taken your first major step towards truly understanding the concepts of game scripting. Although this example was very specific and only a prelude to the real focus of the book (discussed shortly), it did teach the fundamental concept behind *all* forms of scripting: How to avoid *hardcoding*.

The Perils of Hardcoding

What is hardcoding? To put it simply, it's what you were doing when you tried coding your items directly into the engine. It's the practice of writing code or data in a rigid, fixed or hard-to-edit sort of way. Whether you decide to become a scripting guru or not, hardcoding is almost always something to avoid. It makes your code difficult to write, read, and edit. Take the following code block, for example:

```
const MAX_ARRAY_SIZE = 32;

int iArray [ MAX_ARRAY_SIZE ];
int iChecksum;

for ( int iIndex = 1; iIndex < MAX_ARRAY_SIZE; ++ iIndex )
{
   int iElement = iArray [ iIndex ];
   iArray [ iIndex - 1 ] = iElement;
   iChecksum += iElement;
}

iArray [ MAX_ARRAY_SIZE - 1 ] = iChecksum;
```

Regardless of what it's actually supposed to be doing the important thing to notice is that the size of the array, which is referred to a number of times, is stored in a handy constant beforehand. Why is this important? Well imagine if you suddenly wanted the array to contain 64 elements rather than 32. All you'd have to do is change the value of `MAX_ARRAY_SIZE`, and the rest of the program would immediately reflect the change. You wouldn't be so lucky if you happened to write the code like this:

```
int iArray [ 32 ];
int iChecksum;

for ( int iIndex = 1; iIndex < 32; ++ iIndex )
{
   int iElement = iArray [ iIndex ];
   iArray [ iIndex - 1 ] = iElement;
   iChecksum += iElement;
}
iArray [ 31 ] = iChecksum;
```

This is essentially the "hardcoded" version of the first code block, and it's obvious why it's so much less flexible. If you want to change the size of the array, you're going to have to do it in three separate places. Just like the items in the RPG, the `const` used in this small example is analogous to the external file—it allows you to make all of your changes in one, separate place, and watch the rest of the program automatically reflect them.

You aren't exactly scripting yet, but you're close! The item description files used in the RPG example are *almost* like very tiny scripts, so you're in good shape if you've understood everything so far. I just want to take you through one more chapter in the history of this hypothetical RPG project, which will bring you to the real heart of this introduction. After that, you should pretty much have the concept nailed.

So let's get back to these item description files. They're great; they take all the work of creating and fine-tuning game items off the programmer's shoulders while he or she is working on other things like the engine. But now it's time to consider some expansion issues. The item structure works pretty well for describing items, and it was certainly able to handle the basics like your typical health and magic potions, an armor elixir, and the mysterious Orb of Sayjack. But they're not going to cut it for long. Let's find out why.

Storing Functionality in External Files

Sooner or later, you're going to want more unique and complex items. The common thread between all of the items described so far is that they basically just increase or decrease various stats. It's something that's very easy to do, because each item only needs to tell the engine which stats it wants to change, and by how much. The problem is, it gets boring after a while because you can only do so much with a system like that.

So what happens when you want to create an item that does something very specific? Something that doesn't fit a mold as simple as "Tell me what stat to change and how much to change it by"? Something like an item that say, causes all ogres below a certain level to run away from battles? Or maybe an item that restores the MP of every wizard in the party that has a red cloak? What about one that gives the player the capability to see invisible treasure chests? These are all very specific tasks. So what can you do? Just add some item types to your list?

```
const HEAL          = 0;
const MAGIC_RESTORE    = 1;
const ARMOR_REPAIR      = 2;
const TELEPORT      = 3;
const MAKE_ALL_OGRES_BELOW_LEVEL_6_RUN_AWAY = 4;
const MAGIC_RESTORE_FOR_EVERY_WIZARD_WITH_RED_CLOAK = 5;
const MAKE_INVISIBLE_TREASURE_CHESTS_VISIBLE = 6;
```

No way *that's* gonna cut it. With a reasonably complex RPG, you might have as many item types as you do actual items! Observant readers might have also noticed that once again, this is dangerously close to a hardcoded solution. You are back in the game engine source code, adding code for specific items—additions that will once again require recompiles every time something needs to be changed. Isn't that the problem you were trying to solve in the first place?

The trouble though, is that the specific items like the ones mentioned previously simply *can't* be solved by any number of fields in an `Item` structure. They're too complex, too specific, and they even involve conditional logic (determining the level of the ogres, the color of the wizards' cloaks, and the visibility of the chests). The only way to actually implement these items is to *program* them—just like you'd program any other part of your game. I mean you pretty much have to; how are you going to test conditions without an `if` statement? But in order to write actual code, you have to go back to programming each item directly into the engine, right? Is there some magical way to actually store *code* in the item description files rather than just a list of values? And even if there is, how on earth would you execute it?

The answer is scripting. Scripting actually lets you write code *outside* of your engine, load that code *into* the engine, and execute it. Generally, scripts are written in their own language, which is often very similar to C/C++ (but usually simpler). These two types of code are separate—scripts use their own compiler and have no effect on your engine (unless you want them to). In essence, you can replace your item files, which currently just fill structure fields with values, with a block of code capable of doing anything your imagination can come up with. Want to create an item that only works if it's used at 8 PM on Thursdays if you're standing next to a certain castle holding a certain weapon? No problem!

Scripts are like little mini-programs that run inside your game. They work on all the same principals as a normal program; you write them in a text editor, pass them through a compiler, and are given a compiled file as a result. The difference, however, is that these executables don't run on your CPU like normal ones do. Because they run inside your game engine, they can do anything that normal game code can. But at the same time, they're separate. You load scripts just like you load images or sounds, or even like the item description files from earlier. But instead of displaying them on the screen or playing them through your speakers, you execute them. They can also talk to your game, and your game can talk back.

How cool is this? Can you feel yourself getting lost in the possibilities? You should be, because they're *endless*. Imagine the freedom and flexibility you'll suddenly be afforded with the ability to write separate mini-programs that all run inside your game! Suddenly your items can be written with as much control and detail as any other part of your game, but they still remain external and self-contained.

Anyway, this concludes the hypothetical RPG scenario. Now that you basically know what scripting is, you're ready to get a better feel for how it actually works. Sound good?

How Scripting Actually Works

If you're anything like I was back when I was first trying to piece together this whole scripting concept, you're probably wondering how you could possibly load code from a file and run it. I remember it sounding too complicated to be feasible for anyone other than Dennis Ritchie or Ken Thompson, (those are the guys who invented C, in case I lost you there) but trust me—although it is indeed a complex task, it's certainly not impossible. And with the proper reference material (which this book will graciously provide), it'll be fun, too! :)

Before going any further, however, let's refine the overall objective. What you basically want to be able do is write code in a high-level language similar to C/C++ that can be compiled independently of your game engine but loaded and executed by that engine whenever you want. The reason you want to do this is so you can separate *game content*, the artistic, creative, and design-oriented aspects of game development, from the *game engine*, the technological, generic side of things.

One of the most popular solutions to this problem literally involves designing and implementing a new language from the ground up. This language is called a *scripting language*, and as I've mentioned a number of times, is compiled with its own special compiler (so don't expect Microsoft VisualStudio to do this for you). Once this language is designed and implemented, you can write scripts and compile them to a special kind of executable that can be run inside your program. It's a lot more complicated than that, though, so you can start by getting acquainted with some of the details.

The first thing I want you to understand is that scripting is analogous to the traditional programming you're already familiar with. Actually, writing a script is pretty much identical to writing a program, the only real difference between the two is in how they're loaded and executed at runtime. Due to this fact, there exist a number of very strong parallels between scripting and programming. This means that the first step in explaining how *scripting* works is to make sure you understand how *programming* works, from start to finish.

An Overview of Computer Programming

Writing code that will execute on a computer is a complicated process, but it can be broken down into some rather simple steps. The overall goal behind computer programming is to be able to write code in a high-level, English-like language that humans can easily understand and follow, but ultimately translate that code into a low-level, machine-readable format. The reason for this is that code that looks like this:

```
int Y = 0;
int Z = 0;
for ( int X = 0; X < 32; ++ X )
{
   Y = X * 2;
   Z += Y;
}
```

which is quite simple and elementary to you and me, is pretty much impossible for your Intel or AMD processor to understand. Even if someone did build a processor capable of interpreting C/C++ like the previous code block, it'd be orders of magnitude slower than anything on the market now. Computers are designed to deal with things in their smallest, most fundamental form, and thus perform at optimal levels when the data in question is presented in such a fashion. As a result, you need a way to turn that fluffy, humanesque language you call C/C++ into a bare-bones, byte-for-byte stream of pure code.

That's where compilers come in. A compiler's job is to turn the C/C++, Java, or Pascal code that your brain can easily interpret and understand into *machine code*, a set of numeric codes (called *opcodes*, short for *op*eration *code*) that tell the processor to perform extremely fine-grained tasks like moving individual bytes of memory from one place to another or jumping to another instruction for iteration and branching. Designed to be blasted through your CPU at lightning speeds, machine code operates at the absolute lowest level of your computer. Because pure machine code is rather difficult to read by humans (because it's nothing more than a string of numbers), it is often written in a more understandable form called *assembly language*, which gives each numeric opcode a special tag called an *instruction mnemonic*. Here's the previous block of code from, after a compiler has translated it to assembly language:

```
mov   dword ptr [ebp-4],0
mov   dword ptr [ebp-8],0
mov   dword ptr [ebp-0Ch],0
jmp   00401048h
mov   eax,dword ptr [ebp-0Ch]
add   eax,1
mov   dword ptr [ebp-0Ch],eax
cmp   dword ptr [ebp-0Ch],20h
jge   00401061h
mov   ecx,dword ptr [ebp-0Ch]
shl   ecx,1
mov   dword ptr [ebp-4],ecx
mov    edx,dword ptr [ebp-8]
add   edx,dword ptr [ebp-4]
mov   dword ptr [ebp-8],edx
jmp   0040103fh
```

NOTE

For the remainder of this section, and in many places in this book, I'm going to use the terms *machine code* and *assembly language* interchangeably. Remember, the only difference between the two is what they look like. Although machine code is the numeric version and assembly is the human-readable form, they both represent the exact same data.

If you don't understand assembly language, that probably just looks like a big mess of ASCII characters. Either way, this is what the processor wants to see. All of those variable assignments, expressions, and even the `for` loop have been collapsed to just a handful of very quick instructions that the CPU can blast through without thinking twice. And the *really* useless stuff, like the actual names of those variables, is gone entirely. In addition to illustrating how simple and to-the-point machine code is, this example might also give you an idea of how complex a compiler's job is.

Anyway, once the code is compiled, it's ready to fly. The compiler hands all the compiled code to a program called a *linker*, which takes that massive volume of instructions, packages them all into a nice, tidy executable file along with a considerable amount of header information and slaps an .EXE on the end (or whatever extension your OS uses). When you run that executable, the operating system invokes the *program loader* (more commonly referred to simply as the *loader*), which is in charge of extracting the code from the .EXE file and loading it into memory. The loader then tells the CPU the address in memory of the first instruction to be processed, called the *program entry point*, (the `main ()` function in a typical C/C++ program), and the program begins executing. It might be displaying 3D graphics, playing a Chemical Brothers MP3, or accepting user input, but no matter what it's doing, the CPU is always processing instructions. This general process is illustrated in Figure 1.3.

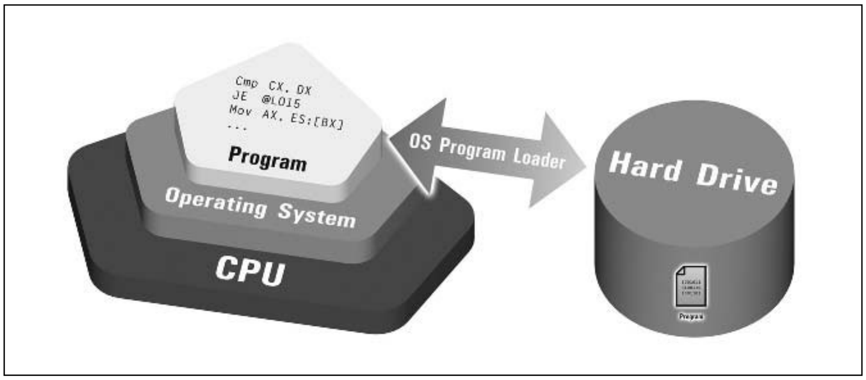


Figure 1.3

The OS program loader extracts machine code from the executable file and loads it into memory for execution.

This is basically the philosophy behind computer science in a nutshell: Turning problems and algorithms into high-level code, turning that high-level code into low-level code, executing that low-level code by feeding it through a processor, and (hopefully) solving the problem. Now that you've got that out of the way, you're ready to learn how this all applies to scripting.

An Overview of Scripting

You might be wondering why I spent the last section going over the processes behind general computer programming. For one thing, a lot of you probably already know this stuff like the back of your hand, and for another, this book is supposed to be about scripting, right? Well don't sweat it, because this is where you apply that knowledge. I just wanted to make sure that the programming process was fresh in your mind, because this next section will be quite similar and it's always good to make connections. As I mentioned earlier, there exist a great number of parallels between programming and scripting; the two subjects are based on almost identical concepts.

When you write a script, you write it just like you write a normal program. You open up a text editor of some sort (or maybe even an actual VisualStudio-style IDE if you go so far as to make one), and input your code in a high-level language, just like you do now with C/C++. When you're done, you hand that source file to a compiler, which reduces it to machine code. Until this point, nothing seems much different from the programming process discussed in the last section.

The changes, however, occur when the compiler is translating the high-level script code. Remember, the whole concept behind a script is that it's like a program that runs inside *another* program. As such, a script compiler can't translate it into 80X86 machine code like it would if it were compiling for an Intel CPU. In fact, it can't translate it to *any* CPU's machine code, because this code won't be running on a CPU.

So how's this code going to be executed, if not by a CPU? The answer is what's called a *virtual machine*, or *VM*. Aside from just being a cool-sounding term, a virtual machine is very similar to the CPU in your computer, except that it's implemented in software rather than silicon. A real CPU's job is basically to retrieve the next instruction to be executed, determine what that instruction is telling it to do, and do it. Seems pretty simple, huh? Well it's the same thing a virtual machine does. The only difference is that the VM understands its own special dialect of assembly language (often called *bytecode*, but you'll get to that later).

Another important attribute of a virtual machine is that, at least in the context of game scripting, it's not usually a standalone program. Rather, it's a special "module" that is built into (or "integrated with") other programs. This is also similar to your CPU, which is integrated with a motherboard, RAM, a hard drive, and a number of input and output devices. A CPU on its own is pretty much useless. Whatever program you integrate the VM with is called the *host application*, and it is this program that you are ultimately "scripting". So for example, if you integrated a VM into the hypothetical RPG discussed earlier, scripts would be *running inside* the VM, but they would be *scripting* the RPG. The VM is just a vehicle for getting the script's functionality to the host.

So a scripting system not only defines a high-level, C/C++-style language of its own, but also creates a new low-level assembly language, or *virtual machine code*. Script compilers translate scripts into this code, and the result is then run inside the host application's virtual machine. The virtual machine and the host application can talk to one another as well, and through this interface, the script can be given specific control the host. Figure 1.4 should help you visualize these interactions.

Notice that there are now two more layers above the program—the VM and the script(s) inside it.

So let's take a break from all this theory for a second and think about how this could be applied to your hypothetical RPG. Rather than define items by a simple set of values that the program blindly plugs into the item array, you could write a block of code that the program tells the VM to execute every time the item is used. Through the VM, this block of code could talk to the game, and the game could talk back. The script might ask the game how many hit points the player has, and what sort of armor is currently being worn. The game would pass this information to the

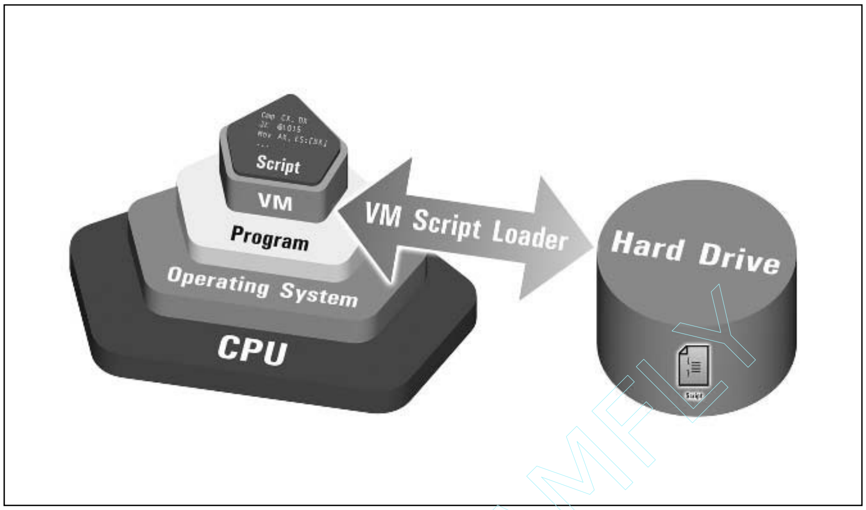


Figure 1.4

The VM's script loader loads virtual machine code from the script file, allowing the VM to execute it. In addition to a runtime environment, the VM also provides a communication layer, or interface, between the running script and the host program.

script and allow it process it, and ultimately the script would perform whatever functionality was associated with the item.

Host applications provide running scripts with a group of functions, called an *API* (which stands for Application Programming Interface), which they can call to affect the game. This API for an RPG might allow the script to move the player around in the game world, get items, change the background music, or whatever. With a system like this, *anything* is possible.

That was quite a bit of information to swallow, huh? Well, I've got some good and bad news. The bad news is that this *still* isn't everything; there are actually a number of ways to implement a game scripting system, and this was only one of them. The good news, though, is that this method is by far the most complex, and everything else will be a breeze if you've understood what's been covered so far.

So, without further ado…

The Fundamental Types of Scripting Systems

Like most complex subjects, scripting comes in a variety of forms. Some implementations involve highly structured, feature-rich compilers that understand full, procedural languages like C or even object oriented languages like C++, whereas others are based around simple command sets that look more like a LOGO program. The choices aren't always about design, however. There exists a huge selection of scripting systems these days, most of which have supportive and dedicated user communities, and almost all of which are free to download and use. Even after attaining scripting mastery, you still might feel that an existing package is right for you.

Regardless of the details, however, the motivation behind any choice in a scripting system should always be to match the project appropriately. With the huge number of features that can be either supported or left out, it's important to realize that the best script system is the one that offers just enough functionality to get the job done without overkill. Especially in the design phase, it can be easy to overdo it with the feature list. You don't need a Lamborghini to pick up milk from the grocery store, so this chapter will help you understand your options by discussing the fundamental types of scripting systems currently in use. Remember: Large, complicated feature lists do look cool, but they only serve to bulk up and slow down your programs when they aren't needed.

This section will cover:

- Procedural/object-oriented language systems
- Command-based language systems
- Dynamically linked module systems
- Compiled versus interpreted code
- Existing scripting solutions

Procedural/Object-Oriented Language Systems

Probably the most commonly used of the mainstream scripting systems are those built around procedural or object-oriented scripting languages, and employ the method of scripting discussed throughout this chapter.

In a nutshell, these systems work by writing scripts in a high-level, procedural or object oriented language which is then compiled to virtual machine code capable of running inside a virtual machine, or left uncompiled in order to be executed by an interpreter (more on the differences between compiled and interpreted code later). The VM or interpreter employed by these systems is integrated with a host application, giving that application the capability to invoke and communicate with scripts.

The languages designed for these systems are usually similar in syntax and design to C/C++, and thus are flexible, free-form languages suitable for virtually any major computing task. Although many scripting systems in this category are designed with a single type of program in mind, most can be (and are) effectively applied to any number of uses, ranging from games to Web servers to 3D modelers.

Unreal is a high-profile example of a game that's really put this method of scripting to good use. Its proprietary scripting language, *UnrealScript*, was designed specifically for use in *Unreal*, and provides a highly object oriented language similar to C/C++. Check out Figure 1.5.

Figure 1.5

Unreal, a first-person shooter based around a proprietary scripting system called UnrealScript.

Command-Based Language Systems

Command-based languages are generally built around extremely specialized LOGO-like languages that consist entirely of program-specific commands that accept zero or more parameters. For example, a command-based scripting system for the hypothetical RPG would allow scripts to call a number of game-specific functions for performing common tasks, such as moving the player around in the game world, getting items, talking to characters, and so on. For an example of what a script might look like, consider the following:

```
MovePlayer      10, 20
PlayerTalk      "Something is hidden in these bushes..."
PlayAnim      SEARCH_BUSHES
PlayerTalk      "It's the red sword!"
GetItem      RED_SWORD
```

As you can see, the commands that make up this hypothetical language are extremely specific to an RPG like the one in this chapter. As a result, it wouldn't be particularly practical to use this language to script another type of program, like a word processor. In that case, you'd want to revise the command set to be more appropriate. For example:

```
MoveCursor      2, 2
SetFont      "Times New Roman", 24, BLACK
PrintText      "Newsletter"
LineBreak
SetFontSize      12
PrintDate
LineBreak
```

Once again, the key characteristic behind these languages is how specialized they are. As you can see, both languages are written directly for their host application, with little to no flexibility. Although their lack of common language constructs such as variables and expressions, branching, iteration, and so on limit their use considerably, they're still handy for automating linear tasks into what are often called "macros". Programs like Photoshop and Microsoft Word allow the users to record their movements into macros, which can then be replayed later. Internally, these programs store macros in a similar fashion; recording each step of the actions in a program-specific, command-based language. In a lot of ways, you can think of HTML as command-based scripting, albeit in a more sophisticated fashion.

Dynamically Linked Module Systems

Something not yet discussed regarding the procedural scripting languages discussed so far are their inherent performance issues. You see, when a compiled script is run in a virtual machine, it executes at a significantly slower rate than native machine code running directly on your CPU. I'll discuss the specific reasons for this later, but for now, simply understand that they're definitely not to be used for speed-critical applications, because they're just too slow.

In order to avoid this, many games utilize *dynamically linked script modules*. In English, that basically means blocks of C/C++ code that are compiled to native machine code just like the game itself, and are linked and loaded at runtime. Because these are written in normal C/C++ and compiled by a native compiler like Microsoft Visual C++, they're extremely fast and very powerful. If you're a Windows user, you actually deal with these every day; but you probably know them by their more Windows-oriented name, *DLLs*. In fact, most (if not all) Windows games that implement this sort of scripting system actually use Win32 DLLs specifically. Examples of games that have used this method include id Software's *Quake II* and Valve's *Half-Life*.

Dynamically linked modules communicate with the game through an API that the game exposes to them. By using this API, the modules can retrieve and modify game state information, and thus control the game externally. Often times, this API is made public and distributed in what is called an *SDK* (Software Development Kit), so that other programmers can add to the game by writing their own modules. These add-ons are often called *mods* (an abbreviation for "modification") and are very popular with the previously mentioned games (Quake and Half-Life).

At first, dynamically linked modules seem like the ultimate scripting solution; they're separate and modularized from the host program they're associated with, but they've got all the speed and power of natively compiled C/C++. That unrestricted power, however, doubles as their most significant weakness. Because most commercial (and even many non-commercial) games are played by thousands and sometimes tens of thousands of gamers, often over the Internet, scripts and add-ons must be safe. Malicious and defective code is a serious issue in large-scale products—when that many people are playing your game, you'd better be sure that the external modules those games are running won't attempt to crash the server during multiplayer games, scan players' hard drives for personal information, or delete sensitive files. Furthermore, even non-malicious code can cause problems by freezing, causing memory leaks, or getting lost in endless loops.

If these modules are running inside a VM controlled directly by the host program, they can be dealt with safely and securely and the game can sometimes even continue uninterrupted simply by resetting an out-of-control script. Furthermore, VM security features can ensure that scripts won't have access to places they shouldn't be sticking their noses.

Dynamically linked script modules, however, don't run *inside* their host applications, but rather *along side* them. In these cases, hosts can assert very little control over these scripts' actions, often leaving both themselves and the system as a whole susceptible to whatever havoc they may intentionally or unintentionally wreak.

This pretty much wraps up the major types of scripting systems out there, so let's switch the focus a bit to a more subtle detail of this subject. A screenshot of Half-Life appears in Figure 1.6.

Compiled versus Interpreted Code

Earlier I mentioned compiled and interpreted code during the description of procedural language scripting systems. The difference between these two forms of code is simple: compiled code is reduced from its human-readable form to a series of machine-readable instructions called machine code, whereas interpreted code isn't.

So how does interpreted code run? It's a valid question, especially because I said earlier that no one's made a CPU capable of executing uncompiled C/C++ code. The answer is that the CPU doesn't run this code directly. Instead, it's run by a separate program, quite similar in nature to a virtual machine, called an *interpreter.* Interpreters are similar to VMs in the sense that they execute code in software and provide a suitable runtime environment. In many ways, however, interpreters are far more complex because they don't execute simplistic, fine-grained machine code.

Figure 1.6

Half-Life handles scripting and add-ons by allowing programmers to write game content in a typical C/C++ compiler using the proprietary Half-Life SDK.

Rather, they literally have to process and understand the exact same human-written, high-level C/C++ code you and I deal with every day.

If you think that sounds like a tough job, you're right. Interpreters are no picnic to implement. On the one hand, they're based on almost all of the complex, language parsing functionality of compilers, but on the other hand, they have to do it all fast enough to provide real-time performance.

However, contrary to what many believe, an interpreter isn't quite as black and white as it sounds. While it's true that an interpreter loads and executes raw source code directly without the aid of a separate compiler, virtually all modern interpreters actually perform an internal, *pre-compile* step, wherein the source code loaded from the disk is actually passed through a number of routines that encapsulate the functionality of a stand-alone compiler and produce a temporary, in-memory compiled version of the script or program that runs just as quickly as it would if it were an executable read from disk.

Most interpreters allow you the best of both worlds—fast execution time and the convenience of automatic, transparent compilation done entirely at runtime. There are still some trade-offs, however; for example, if you don't have the option to compile your scripts beforehand, you're forced to distribute human-readable script code with your game that leaves you wide open to modifications and hacks. Furthermore, the process of loading an ASCII-formatted script and compiling it at runtime means your scripts will take a longer time to load overall. Compiled scripts can be loaded faster and don't need any further processing once in memory.

As a result, this book will only casually mention interpreted code here and there, and instead focus entirely on compiled code. Again, while interpreters do function extremely well as debuggers and other development tools, the work involved in creating them outweighs their long-term usefulness (at least in the context of this book).

Existing Scripting Solutions

Creating your own scripting system might be the focus of this book, but an important step in designing anything is first learning all you can about the existing implementations. To this end, you can briefly check out some currently used scripting systems. All of the systems covered in this section are free to download and use, and are supported by loyal user communities. Even after attaining scripting mastery, using an existing scripting system is always a valid choice, and often a practical one. This section is merely an introduction, however; an in-depth description of both the design and use of existing scripting systems can be found in Chapter 6.

Ruby

`http://www.ruby-lang.org/en/index.html`

Ruby is a *strongly* object-oriented scripting language with an emphasis on system-management tasks. It boasts a number of advanced features, such as garbage collection, dynamic library loading, and multithreading (even on operating systems that don't support threads, such as DOS). If you download Ruby, however, you'll notice that it doesn't come with a compiler. This is because it is a fully interpreted language; you can immediately run scripts after writing them without compiling them to virtual machine code.

Taken directly from the official web site, here's a small sample of Ruby code (which defines a class called `Person`):

```
class Person
  attr_accessor :name, :age
  def initialize(name, age)
    @name = name
    @age  = age.to_i
  end
  def inspect
    "#@name (#@age)"
  end
end

p1 = Person.new('elmo', 4)
p2 = Person.new('zoe', 7)
```

Lua

`http://www.lua.org/`

As described by the official Lua web site, "Lua is a powerful, lightweight programming language designed for extending applications." Lua is a procedural scripting system that works well in any number of applications, including games. One of its most distinguishing features, however, lies in its ability to be expanded by programs written with it. As a result, the core language is rather small; it is often up to the user to implement additional features (such as classes). Lua is a compact, highly expandable and compiled language that interfaces well with C/C++, and is subsequently a common choice for game scripting.

Java

`http://java.sun.com/`

Strangely enough, Java has proven to be a viable and feature-rich scripting alternative. Although Java's true claim to fame is designing platform independent, standalone applications (often with a focus on the internet), Java's virtual machine, known as the *JVM*, can be easily integrated with C/C++ programs using the *Java Native Interface*, or *JNI*. Due to its common use in professional-grade e-commerce applications, the JVM is an optimized, multithreaded runtime environment for compiled scripts, and the language itself is flexible and highly object oriented.

SUMMARY

Phew! Not a bad way to start things off, eh? In only one chapter, you've taken a whirlwind tour of the world of game scripting, covering the basic concepts, a general overview of implementation, common variations on the traditional scripting method, and a whole lot of details. If you're new to this stuff, give yourself a big pat on the back for getting this far. If you aren't, then don't even think about patting your back yet. You aren't impressing anyone! (Just kidding)

In the coming chapters, you're going to do some really incredible things. So read on, because the only way you're going to understand the tough stuff is if you master the basics first! With that in mind, you might want to consider re-reading this chapter a few times. It covers a lot of ground in a very short time, and it's more than likely you missed a detail here or there, or still feel a bit fuzzy on a key concept or two. I personally find that even re-reading chapters I think I understood just fine turns out to be helpful in the end.

This page intentionally left blank

CHAPTER 2

Applications of Scripting Systems

"What's wrong with science being practical? Even profitable?"
——Dr. David Drumlin, Contact

As I mentioned in the last chapter, scripting systems should be designed to do as much as is necessary and no more. Because of this, understanding what the various forms of scripting systems can do, as well as their common applications, is essential in the process of attaining scripting mastery.

So that's what this chapter is all about: giving you some insight into how scripting is applied to real-world game projects. Seeing how something is actually used is often the best way to solidify something you've recently learned, so hopefully the material presented here will compliment that of the last chapter well. This has actually been covered to some extent already; the last chapter's hypothetical RPG project showed you by example how scripting can ease the production of games that require a lot of content. This chapter approaches the topic in a more detailed and directly informative way, and focuses on more than just role-playing games. In an effort to keep these examples of script applications as diverse as possible, the chapter also takes a look at a starkly contrasting game genre, but one that gets an equal amount of attention from the scripting community——the First-Person Shooter.

I should also briefly mention that if you're coming into the book with the sole purpose of applying what you learn to an existing project, you probably already know exactly why you need to build a scripting system and feel that you can sweat the background knowledge. Regardless of your skill level and intentions, however, I suggest you at least skim this stuff; not only is it a light and fairly non-technical read, but it sets the stage for the later chapters. The concepts introduced in this chapter will be carried on throughout the rest of the book and are definitely important to understand.

But enough with the setup, huh? Let's get going. This chapter will cover how scripting systems can be applied to the following problems:

- An RPG's story-related elements—non-player characters and plot details.
- RPG items, weapons and enemies.
- The objects, puzzles and switches of a first-person shooter.
- First-person shooter enemy behavior.

The General Purpose of Scripting

As was explained in the last chapter, the most basic reason to implement a scripting system is to avoid the perils of hardcoding. When the content of your game is separated from the engine, it allows the tweaking, testing, and general fine-tuning of a game's mechanics and features to be carried out without constant recompilation of the entire project. It also allows the game to be easily expanded even after it's been compiled, packaged, and shipped (see Figure 2.1). Modifications and extensions can be downloaded by players and immediately recognized by the game. With a system like this, gameplay can be extended indefinitely (so long as people produce new scripts and content, of course).

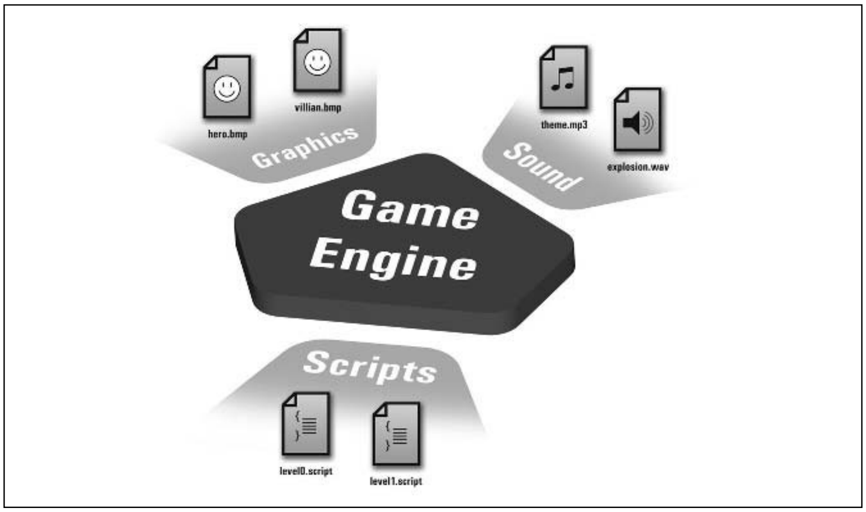


Figure 2.1

Game logic can be treated as modular content, allowing it to be just as flexible and interchangeable as graphics and sound.

Because the ideal separation of the game engine and its content allows the engine's executable to be compiled without a single line of game-specific code, the actual game the player experiences can be composed *entirely* of scripts and other media, like graphics and sound. What this means is that when players buy the game, they're actually getting two separate parts; a compiled game engine and a series of scripts that fleshes it out into the game itself. Because of this *modular architecture*, entirely new games such as sequels and spinoffs can be distributed in script-form only, running without modification on the engine that players already have.

One common application of this idea is distributing games in "episode" form; that means that stores only sell the first 25 percent or so of the game at the time of purchase, along with the executable engine capable of running it. After players finish the first episode, they're allowed to download or buy additional episodes as "patches" or "add-ons" for a smaller fee. This allows gamers to try games before committing to a full purchase, and it also lets the developers easily release new episodes as long as the game franchise is in demand. Rather than spend millions of dollars developing a full-blown sequel to the game, with a newly designed and coded engine, additional episodes can be produced for a fraction of the cost by basing them entirely on scripts and taking advantage of the existing engine, while still keeping players happy.

With this in mind, scripting seems applicable to all sorts of games; don't let the example from the first chapter imply that only RPGs need this sort of technology. Just about any type of game can benefit from scripting; even a *PacMan* clone could give the different colored ghosts their own unique AI by assigning them individual scripts to control their movement. So the first thing I want to impress upon you is how flexible and widely applicable these concepts are. All across the board, games of every genre and style can be reorganized and retooled for the better by introducing a scripting system in some capacity.

So to start things off on a solid footing, let's begin this tour of scripting applications with another look RPGs. This time I'll of course go into more detail, but at least this gets you going with some familiar terrain.

Role Playing Games (RPGs)

Although I've been going out of my way to assure you that RPGs are hardly the only types of games to which one can apply a scripting system, you do hear quite a bit of scripting-related conversation when hanging around RPG developers; often more so than other genres in fact. The reason for this is that RPGs lend themselves well to the concept of scripts because they require truly *massive* amounts of game content. Hundreds of maps, countless weapons, enemies and items, thousands of roaming characters, hundreds of megs worth of sound and music, and so on. So, naturally, RPG developers need a good way to develop this content in a structured and organized manner. Not surprisingly, scripting systems are the answer to this problem more often than not.

In order to understand why scripting can be so beneficial in the creation of RPGs, let's examine the typical content of these games. This section covers:

- Complex, in-depth stories
- Non-player characters (NPCs)
- Items and weapons
- Enemies

Complex, In-Depth Stories

Role playing games are in a class by themselves when it comes to their storylines. Although many games are satisfied with two paragraphs in the instruction manual that essentially boil down to "You've got 500 pounds of firepower strapped to your back. Blow up everything that moves and you'll save democracy!", RPGs play more like interactive novels. This means multi-dimensional characters with endless lines of dialogue and a heavily structured plot with numerous "plot points" that facilitate the progression of a player through the story.

At any given point in the player's adventure, the game is going to need to know every major thing the player has done up until that point in order to determine the current state of the game world, and thus, what will happen next. For example, if players can't stop the villain from burning the bridge to the hideout early in the game, they might be forced to find an alternate way in later.

The Solution

Many RPGs employ an array of "flags" that represent the current status of the plot or game world. Each flag represents an event in the game and can be either true or false (although similar systems allow flags to be more complex than simple Boolean values). At the beginning of the game, every flag will be `FALSE` because the player has yet to do anything. As players progress through the game, they're given the opportunity to either succeed or fail in various challenges, and the flags are updated accordingly. Therefore, at any given time, the flag array will provide a reasonably detailed history of the player's actions that the game can use to determine what to do next. For example, to find out if the villain's bridge has been burned down, it's necessary to check its corresponding flag. Check out figure 2.2.

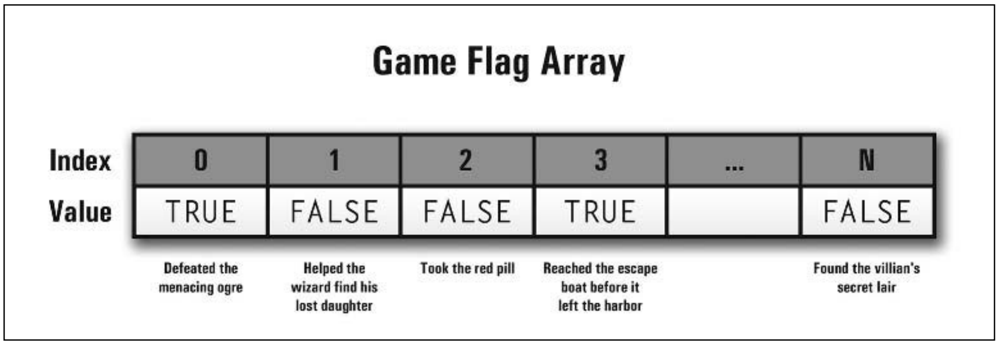

Game Flag Array
Index
Value
0
1
2
3
...
N
TRUE
FALSE
FALSE
TRUE
FALSE
Defeated the menacing ogre
Helped the wizard find his lost daughter
Took the red pill
Reached the escape boat before it left the harbor
Found the villian's secret lair

Figure 2.2

Every event in the game is represented by an element (commonly Boolean) in the game flag array. At any time, the array can be used to determine the general course the player has taken. This can be used to determine future events and conditions.

Implementation of this system can be approached in a number of ways. One method is to build the array of flags directly in the engine source code, and provide an interface to scripts that allows them to read and write to the array (basically just "get" and "set" functions). This way, most of the logic and functionality behind the flag system lies in external scripts; only the array itself needs to be built into the game engine. Depending on the capabilities of your scripting system, however, you might even be able to store the array itself in a script as well, and thus leave the engine entirely untouched. This is technically the ideal way to do it, because *all* game logic is offloaded from the main engine, but either way is certainly acceptable.

Non-Player Characters (NPCs)

One of the most commonly identifiable aspects of any RPG is the constant conversation with the characters that inhabit the game world. Whether it be the friendly population of the hero's home village or a surly guard keeping watch in front of a castle, virtually all RPGs require the player to talk to these non-player characters, or *NPCs*, in order to gather the information and clues necessary to solve puzzles and overcome challenges.

Generally speaking, the majority of the NPCs in an RPG will only spark trivial conversations, and their dialogue will consist of nothing more than a linear series of statements that never branch and always play out the same, no matter how many times you approach them. Kinda like that loopy uncle you see on holidays that no one likes to talk about.

Things aren't always so straightforward however. Some characters will do more than just ramble; they might ask a question that results in the player being prompted to choose from a list of responses, or ask the player to give them money in exchange for information or items, or any number of other things. In these cases, things like conditional logic, iteration, and the ability to read game flags become vital. An example of real character dialogue from Square's *Final Fantasy 9* can be found in Figure 2.3.

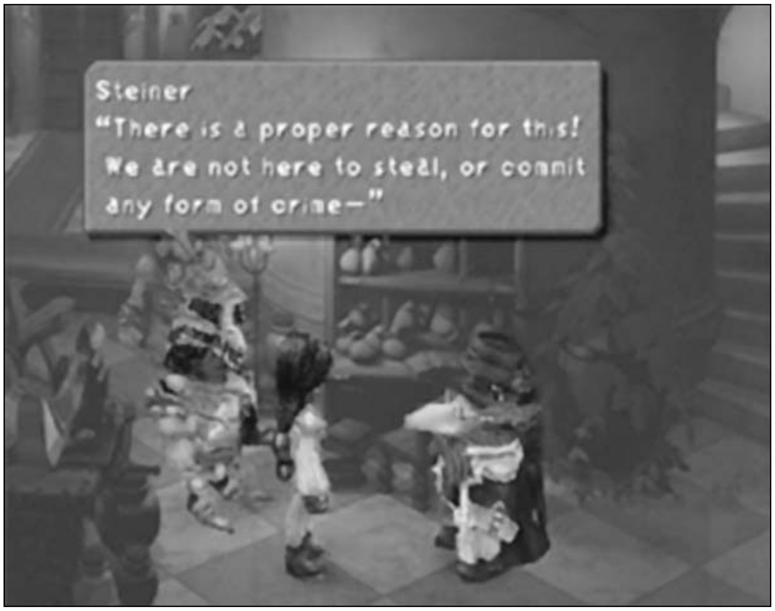

Steiner
"There is a proper reason for this! We are not here to steal, or commit any form of crime—"

Figure 2.3

Exchanging dialogue with an NPC in Squaresoft's Final Fantasy 9.

The Solution

First, let's discuss some of the simpler NPC conversations that you'll find in RPGs. In the case of conversations that don't require branching, a command-based language system is more than enough. For example, imagine you'd like the following exchange in your game:

NPC: "You look like you could use some garlic."

Player: "Excuse me?"

NPC: "You're the guy who's saving the world from the vampires, right?"

Player: "Yeah, that's me."

NPC: "So you're gonna need some garlic, won't you?"

Player: "I suppose I will, now that you mention it."

NPC: "Here ya go then!" *(Gives player garlic)*

Player: "Uh…thanks, I guess." *(Player scratches head)*

If you were paying attention, you might have noticed that only about four unique commands are necessary to implement this scene. And if you weren't paying attention, you probably still aren't, so I'll take advantage of this opportunity and plant some subliminal messages into your unknowing subconscious: *buy ten more copies of this book for no reason other than to inflate my royalty checks.* Anyway, here's a rundown of the functionality the scene requires:

- Both the player and the NPC need the ability to talk.
- The NPC needs to be able to give the player an item (vampire-thwarting garlic, in this case).
- There should also be a general animation-playing command to handle the head scratching.

Here's that same conversation, in command-based script form:

```
NPCTalk    "You look like you could use some garlic."
PlayerTalk    "Excuse me?
NPCTalk   "You're the guy who's saving the world from the vampires, right?"
PlayerTalk   "Yeah, that's me."
NPCTalk   "So you're gonna need some garlic, won't you?"
PlayerTalk   "I suppose I will, now that you mention it."
NPCTalk   "Here ya go then!"
GetItem   GARLIC
PlayerTalk   "Uh... thanks, I guess."
PlayAnim   PLAYER_SCRATCH_HEAD
```

Pretty straightforward, huh? Once written, this script would then be associated with the NPC, telling the game to run it whenever the player talks to him (or her, or it, or whatever your NPCs are classified as). It's a simple but elegant solution; all you need to establish is a one-to-one mapping of scripts to NPCs and you've got an easy and reasonably flexible way to externally control the inhabitants of your game world. To see this concept displayed in a more visual manner, check out Figure 2.4.

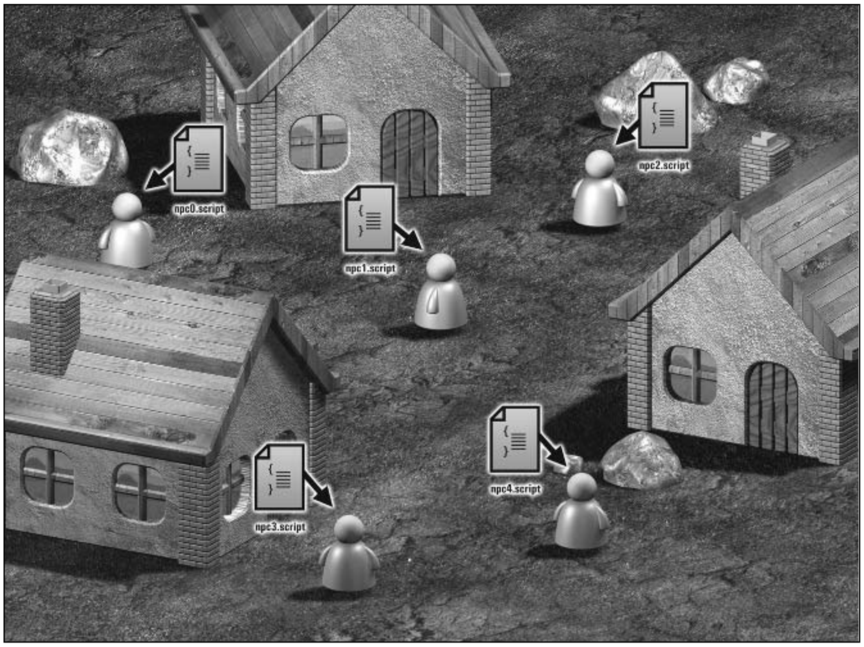


Figure 2.4

Every NPC in an RPG world is controlled and described by a unique script. The graphics simply personify them on-screen.

The honeymoon doesn't last forever, though, and sooner or later some of the more audacious characters roaming through your village will want to do more than just rattle off an unchanging batch of lines every time the player talks to them. They might want to ask the player a question that's accompanied by an on-screen list of answers to chose from, and have the conversation take different paths depending on the player's response. Maybe they'll need to be able to read the game flags and say different things depending on the player's history, or even write to the flags to change the course of future events. Or perhaps one of your characters is short-tempered and should become noticeably agitated if you attempt to talk to him repeatedly. The point is, a good RPG engine will allow its NPCs to be as flexible and lifelike as necessary, so you're going to need a far more descriptive and powerful language to program their behavior.

With this in mind, let's take a look at some of the more complex exchanges that can take place between the player and an NPC.

(Player talks to NPC for the first time)

NPC: "Hey, you look familiar." *(Squints at player's face)*

Player: "Do I? I don't believe we've met."

NPC: "Wait a sec— you're the guy who's gonna save the world from the vampires, right?"

NPC: (If player says Yes) "I knew it! Here, take this garlic!" *(Gives player garlic)*

Player: "Thanks!"

(Player talks to NPC again)

NPC: "Sorry, I don't have any more garlic. I gave you all I had last time we spoke."

Player: "Well *that* sucks. *(Stamps feet)* "

(Player talks to NPC a third time)

NPC: "Dude I told you, I gave you all my garlic. Leave me alone!"

Player: But I ran out, and there's still like 10 more vampires that need to be valiantly defeated!"

NPC: "Hmm…well, my brother lives in the next town over, and he owns a garlic processing plant. I'll tell him you're in the area, and to have a fresh batch ready for you. Next time you're there, just talk to him, and he'll give you all the garlic you need."

Player: "Thanks, mysterious garlic-dispensing stranger!"

NPC: "My name's Gary."

Player: "Whatever."

(Player talks to NPC more than three times)

NPC: "So, have you seen my brother yet?"

That's quite a step up from the previous style of conversation, isn't it? Don't bother trying to figure out how many commands you'd need to script it, because command-based languages just don't deliver in situations like this. So instead, let's look at the general features a language would need to describe this scene.

- Basic conversational capabilities are a given; both the NPC and the player need to be able to speak (which, more or less, just means printing their dialogue in a text box).
- There are a number of points during the conversation at which small animations would be nice, such as the NPC squinting his eyes and the player stamping his feet, so you'll need to be able to tell the engine which animations to play and when.
- Just like the previous example, the NPC gives the player garlic. Therefore, he'll need access to the player's inventory.

- As you can see in the first exchange, the NPC needs the ability to ask the player a question. At the very least, he needs to prompt the player for a yes or no response and branch out through the script's code depending on the result. It'd be nice to provide a custom list of possible answers as well, however, because not everything is going to be a yes or no question (unless the player is a walking magic 8 ball, but to be quite honest I can't see that game selling particularly well outside of Japan).
- Obviously, because the NPC clearly says different things depending on how many times the player has talked to him (up to four iterations, in this case), you need to keep track of the player's history with this character. Furthermore, because the player could theoretically quit and resume the game in between these separate conversations, you need not only the ability to preserve this information in memory during play, but also to save it to the disk in between game sessions. Generally speaking, you need the ability to store variable information associated with the NPC indefinitely.
- Lastly, you need to alter the game flags. How else would Gary's brother in the next town over be aware of the player's need for garlic cloves? To put it in more general terms, NPCs need to be able to tell the engine what they're up to so future events line up with the things they say. Likewise, because Gary's brother's script will need to read from the flags, this ability also lets NPCs base their dialogue on previous events. If you never talk to Gary a third time, his brother will have no idea who you are. Figure 2.5 illustrates the communication lines that exist between scripts, the game flags, and each other with this concept.

Judging by this list, the most prominent features you should notice are the ability to read and write variables and conditional logic that allows the script to behave differently depending on the situation. Now that you've really dissected it, I think this is starting to sound a lot less like a

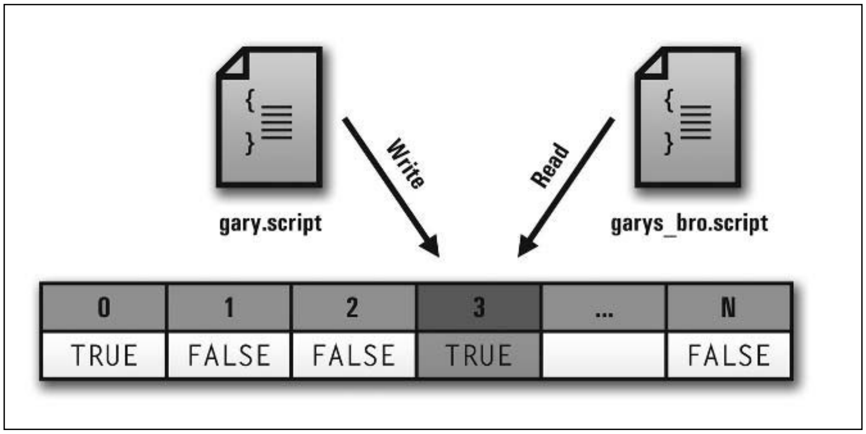


Figure 2.5

Scripts have the ability to both read and write to the game flag array. Reading allows the script to accurately respond to the player's previous actions, whereas writing allows them to affect the future.

macro-esque, command-based script and a lot more like the beginnings a C/C++ program! In essence, it will be. Let's take a look at some C/C++-like script code that you might write to implement this conversation.

```
static int iConverseCount = 0;
static bool bIsPlayerHero = FALSE;

main ()
{
  string strAnswer;

  if ( iConverseCount == 0 )
  {
    NPCTalk ( "Hey, you look familiar." );
    PlayAnim ( NPC, SQUINT );
    PlayerTalk ( "Do I? I don't believe we've met." );

    strAnswer = NPCAsk ( "Wait a sec-- you're the guy who's gonna save the world
from the vampires, right?", "Yes", "No" );
    if ( iAnswer == "Yes" )
    {
      NPCTalk ( "I knew it! Here, take this garlic!" );
      GiveItem ( GARLIC, 4 );
      PlayerTalk ( "Thanks!" );
      bIsPlayerHero = TRUE;
    }
    else
    {
      NPCTalk ( "Ah. My mistake." );
      bIsPlayerHero = FALSE;
    }
  }
  else
  {
    if ( bIsPlayerHero )
    {
      if ( iConverseCount == 1 )
      {
        NPCTalk ( "Sorry, I don't have any more garlic. I gave you all I had last
time we spoke." );
        PlayerTalk ( "Well that sucks." );
```

```
            PlayAnim ( PLAYER, STAMP_FEET );
        }
        elseif ( iConverseCount == 2 )
        {
            NPCTalk ( "Dude I told you, I gave you all my garlic. Leave me alone!" );
            PlayerTalk ( "But I ran out, and there's still like 10 more vampires that
need to be valiantly defeated!" );
            NPCTalk ( "Hmm... well, my brother lives in the next town over, and he owns
a garlic processing plant. I'll tell him you're in the area, and to have a fresh
batch ready for you. Next time you're there, just talk to him, and he'll give you
all the garlic you need." );
            PlayerTalk ( "Thanks, mysterious garlic-dispensing stranger!" );
            NPCTalk ( "My name's Gary." );
            PlayerTalk ( "Whatever." );

            SetGameFlag ( GET_GARLIC_FROM_GARYS_BROTHER );
        }
        else
        {
            NPCTalk ( "Seen my brother yet?" );
        }
    }
    else
    {
        NPCTalk ( "Hello again." );
    }
  }

  iConverseCount ++;
}
```

Pretty advanced for a script, huh? In just a short time, things have come quite a long way from simple command-based languages. As you can see, just adding a few new features can change the design and direction of your scripting system entirely.

You might also be wondering why, just because a few features were added, the language suddenly looks so much like C/C++. Although it would of course be possible to add variables, iteration constructs and conditional logic to the original language from the first example without going so far as to implement something as sophisticated as the C/C++-variant used in the previous example, the fact is that if you *already* need such advanced language features, you'll most likely need even more later. Throughout the course of an RPG project, you'll most likely find use for even more advanced features like arrays, pointers, dynamic resource allocation, and so on. It's a lot easier to decide to go with a C/C++-style syntax from the beginning and just add new things as you need them than it is to design both the syntax and overall structure of the language simultaneously. Using C/C++ syntax also keeps everything uniform and familiar; you don't have to "switch gears" every time to move from working on the engine to working on scripts.

Anyway, there's really no need to discuss the code; for one thing it's rather self explanatory to begin with, and for another, the point here isn't so much to teach you how to implement that specific conversation as it is to impress upon you the depth of real scripting languages. More or less, that *is* C/C++ code up there. There are certainly some small differences, but for the most part that's the same language you're coding the engine with. Obviously, if scripts need a language that's almost as sophisticated as the one used to write the game itself, it's a sign that this stuff can get very advanced, very quickly. NPCs probably seemed like a trivial issue 10 minutes ago, but after looking at how much is required just to ask a few questions and set a few flags, it's clear that even the simpler parts of an RPG benefit from, if not flat-out *require*, a fully procedural scripting language.

Items and Weapons

Items and weapons follow a similar pattern to most other game objects. Each weapon and item is associated with a script that's executed whenever it's used. Like NPCs, a number of items can be scripted using command-based languages because their behavior is very "macro-like". Others will require interaction with game flags and conditional logic. Iteration also becomes very important with items and weapons because they'll often require animated elements.

The last chapter took a look at the basic scripting of items. Actually, it really just looked at the offloading of simple item descriptions to external files, but also touched upon the theory of externally stored functionality. This chapter, however, goes into far more detail and looks at the creation of a complete, functional RPG weapon from start to finish.

Because RPGs are usually designed to present a convincingly detailed and realistic game world, there obviously has to be a large and diverse selection of items and weapons. It wouldn't make sense if, spread over the countless towns, cities, and even continents often found in role-playing games, there was only one type of sword or potion. Once again, this means you're looking for a structured and intelligent way to manage a huge amount of information. In a basic action game with only one or two types of weapons, hardcoding their functionality is no problem; in an RPG, however, anything less than a fully scripted solution is going to result in a tangled, unmanageable mess.

Furthermore, items and weapons in modern RPGs need to be attention-grabbers. Gone are the days of casting a spell or attacking with a sword that simply causes some lost hit points; today, gamers expect grandiose animations with detailed effects like glowing, morphing, and lens flares. Because graphics programming is a demanding and complicated field, a feature-rich scripting language is an absolute necessity.

Item and weapon scripts generally need to do a number of tasks. First to attend to is the actual behind-the-scenes functionality. What this is specifically of course depends on the item or weapon—it could be anything from damaging an enemy (decreasing its hit points) or healing a member of your party (increasing their hit points) to unlocking a door, clearing a passage, or whatever—the point though, is that it's always just a simple matter of updating game variables such as player/enemy statistics or game flags. It's a naturally basic task, and can usually be accomplished with only a few lines of code. In most cases, it can be handled with a command-based language just fine. Check out Figure 2.6.

The other side of things, however, is the version of the item or weapon's functionality that the player perceives. Granted, the player is well aware that the item is healing their party members, or that the weapon is damaging the ogre they're battling with simply because they're the ones who

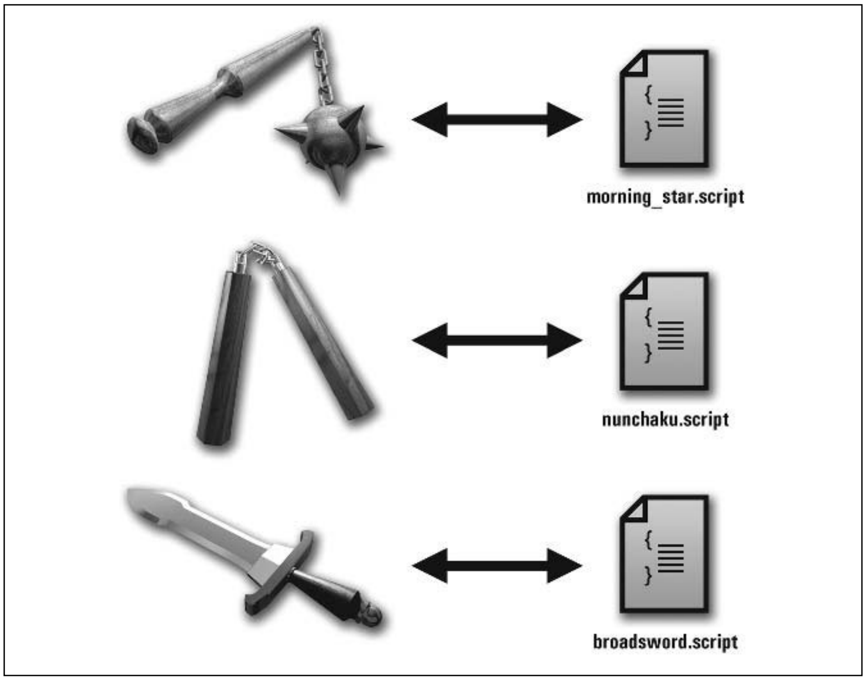


Figure 2.6

Like NPCs, weapons are mapped directly to corresponding script files. The script file defines their behavior by providing blocks of code for the game to run when the weapon is used.

selected and used it. But that's not enough; like I mentioned earlier, these things need to be *experienced*—they need to be seen and heard. What's the fun in using a weapon if you don't get to see some fireworks? So, the other thing you need to worry about when scripting items and weapons are the visuals. This is where command-based languages fall short. Granted, it'd be possible to code a bunch of effects directly in the engine and assign them commands that can be called from scripts, but that'll only result in your RPG having a processed, "cookie cutter" feel. You'll have a large number of items and weapons that all share a small collection of effects, resulting in a lot of redundancy. You'd also have a ton of game-specific effect code mixed up with your engine, which is rarely a good thing. As for coding the effects directly *with* the language, commands just aren't enough to describe unique and original visual effects

The Solution

Generally speaking, it's best to use a C/C++-style, procedural language that will allow items and weapons to define their own graphical effects, down to the tiniest details, from within the script itself. This way, the script not only updates statistics and alters game flags, it also provides its own eye candy. This whole process is actually pretty easy; it's just a matter of providing at least a basic set of graphical routines for scripts to call. All that's really necessary is the typical stuff—pixel plotting, drawing sprites, or maybe even playing movie files to allow for pre-rendered clips of animation—basically a refined subset of the stuff that your graphics API of choice like DirectX, OpenGL, or SDL provides. With these in place, you can code up graphical effects just as you would directly with C/C++.

Let's try creating an example weapon.

What we're going to design is a weapon called the Fire Sword (yeah I know, that sounds pretty generic, but it's just an example, so gimme a break, okay?). The Fire Sword is used to launch fireballs at enemies, and is especially powerful against aquatic or snow-based creatures such as hydras and ice monsters. Conversely, however, it's weaker against enemies that are used to hot, fiery environments, such as dragons, demons, and Mariah Carey. Also, just to make things interesting and force the player to think a bit more carefully about his strategy, the weapon, due to its heat, should cause a slight amount of damage to the player every time it's used. And, because it just wouldn't be fun without it, let's actually throw in a fireball animation to complete the illusion.

That's a pretty good description, but it's also important to consider the technical aspect of this weapon's functionality:

- You'll need the capability to alter the statistics of game characters; namely their hit points. You also need to factor in the fact that the sword causes serious damage to water- or snow-based enemies, but is less effective against fire-based creatures.

- The player needs to see an actual fireball being launched from the player's on-screen location to that of the enemy, as well as hear an explosion-like sound effect that's played upon impact. Because you're now dealing with animation and sound, you're definitely going to need conditional logic and iteration. Command-based languages are no longer an option. In addition, a basic multimedia API will have to be provided by the host application that allows scripts to, at the very least, draw sprites on the screen and play sound effects.
- Finally, the player must be dealt a small amount of damage due to the extreme heat levels expelled by the sword. Like the first task, this is just a matter of crunching some numbers and just means you need access to the player's stats.

And there you have it. Two of the three tasks up there are simple and easily handled by a command-based language. Unfortunately, the need for animation, as well as the need to deal different levels of damage based on the enemy's type, rules them out and pretty much forces you to adopt a language that gives you the capability to perform branches and loops. These concepts are the very basis of animation and pretty much all other graphical effects, so your hands are tied. So, let's see some C/C++-style code for this weapon:

```
Player.HP -= 4;

int Y = Player.OnScreenY;
for ( int X = Player.OnScreenY; X < Enemy.OnScreen.X; X ++ )
   BlitSprite ( FIREBALL, X, Y );
PlaySound ( KA_BOOM );

if ( Enemy.Type == ICE || Enemy.Type == WATER )
   Enemy.HP -= 16;
elseif ( Enemy.Type == FIRE )
   Enemy.HP -= 4;
else
   Enemy.HP -= 8;
```

Pretty straightforward, no? As you can see, once a reasonably powerful procedural language like the C/C++-variant is in place, actually coding the effects and functionality behind weapons like the Fire Sword becomes a relatively trivial task. In this case, it basically just boiled down to a `for` loop that moved a sprite across the screen and a call to a sound sample playing function. Obviously it's a simplistic example, but it should illustrate the fact that your imagination is the only real limitation with such a flexible scripting system, because it allows you to code pretty much anything you can imagine. This sort of power just isn't possible with command-based languages. Check out Figure 2.7 to see the fire sword in all its fiery glory.

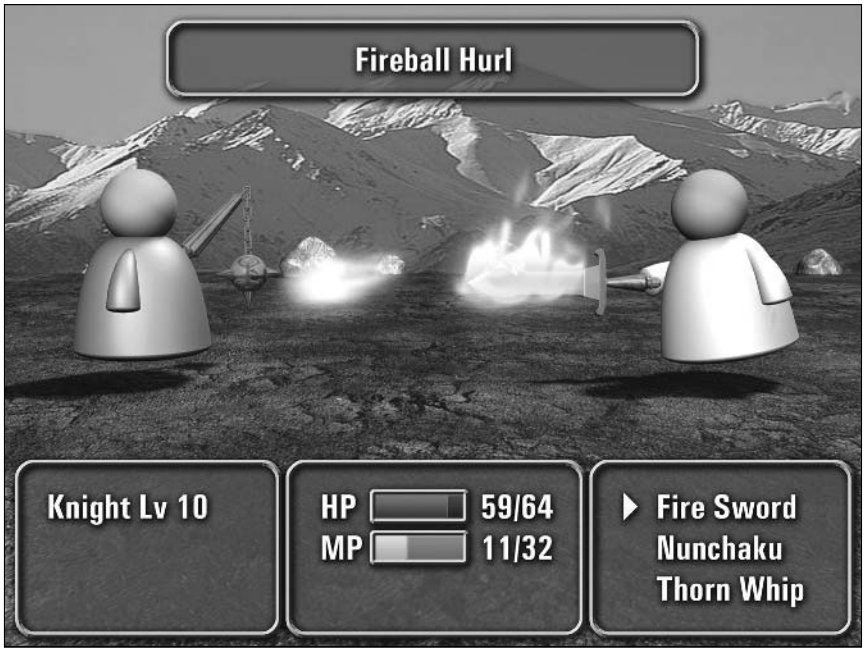


Figure 2.7

The fearsome Fire Sword being wielded in battle.

Enemies

I've covered the friendlier characters, like NPCs, and you understand the basis for the items and weapons you use to combat the forces of darkness, but what about the forces of darkness themselves?

Enemies are the evil, hostile characters in RPGs. They roam the game world and repeatedly attack the players in an attempt to stop them from fulfilling whatever it is their quest revolves around. During battle, a group of enemies is very similar to the players and their travel companions; both parties are fighting to defeat the other by attacking them with weapons and aiding themselves by using items such as healing elixirs and strength- or speed-enhancing potions.

In more general terms, they're the very reason you play RPGs in the first place; despite all of the conversing, exploring and puzzle solving, at least half of the gameplay time (and sometimes quite a bit more, depending on the game) is spent on the battlefield. Not surprisingly, the way enemies are implemented in an RPG project will have a huge effect on both the success of the project itself, as well as the quality of the final game. So don't screw it up! Figure 2.8 is a screenshot from *Breath of Fire*, a commercial RPG with battles in the style we're discussing.

The great thing about enemies though, is that they draw primarily on the two concepts you've already learned; they have the character- and personality-oriented aspects of NPCs, but they also

Figure 2.8

A battle sequence in Capcom's Breath of Fire *series.*

have the functional and destructive characteristics of items and weapons. As a result, determining how to define an enemy for use in your RPG engine is basically just a matter of combining the concepts behind these two other entities.

The Solution

You could approach this situation in any number of ways, but they all boil down to pretty familiar territory. As was the case with NPCs, the most important characteristic to establish when describing an enemy is its personality and behavior. Is it a strong, fast and powerful beast that attacks its opponents relentlessly and easily evades their counter-attacks? Or is it a meek, paranoid creature with a slow attack rate and relatively weak abilities? It could be either of these, but it'll most likely lie somewhere in between——a gray area that demands a sensitive and easily-tuned method of description.

You might be tempted to solve this problem by defining your enemies with a common set of parameters. For example, the behavior of enemies in your game might be described by:

- **Strength.** How powerful each attack is.
- **Speed.** How likely each attack is to connect with its target, as well as how likely the enemy is to successfully dodge a counter-attack.

- **Endurance.** How well the enemy will hold up after taking a significant amount of damage. Higher endurance allows enemies to maintain their intensity when the going gets rough.
- **Armor/Defense.** How much damage incoming attacks will cause. The lower the armor/defense level, the faster its hit points will decrease over the course of the battle due to its vulnerability.
- **Fear.** How likely the enemy is to run away from battles when approaching defeat.
- **Intelligence.** Determines the overall "strategy" of the enemy's moves during battle. Highly intelligent enemies might intentionally attack the weakest members of the player's party, or perhaps conserve their most powerful and physically draining attacks for the strongest. Less intelligent creatures are less likely to think this way and might waste their time attacking the wrong people with the wrong moves, plugging away with a brute force approach until the opponent is defeated.

You could keep adding parameters like these all day, but this seems like a pretty good list. It's clear that you can describe a wide variety of enemies this way; obviously a giant ogre-like beast would have super strength, endless endurance, rock-solid defense, and be nearly fearless. It wouldn't be particularly smart or fast, however. Likewise, a ninja or assassin would have speed and endurance to spare, as well as high intelligence and a reasonable level of strength. A lowly slime would probably have low levels of all of these things, whereas the final, ultimate villain might be maxed-out in every category. Overall, this is a simple system but it allows you to rapidly define large groups of diverse enemies with an adequate level of flexibility.

It should seem awfully suspicious, however, because as you learned in the last chapter with the item description files, defining such a broad group of entities in your game with nothing more than a set of common parameters can quickly paint you into a corner and deprive you of true creative control. As you've most certainly guessed by now, script code comes to the rescue once again.

But how do you actually organize the script's code? Despite the parallels I've drawn between enemy scripts and that of items and NPCs, astute readers might have noticed that there exists one major difference between them. Items, weapons, and NPCs are all invoked on a singular basis; they perform their functionality upon activation by some sort of trigger or event, and terminate upon completing their task. The Fire Sword is inactive until the moment you use it, at which point it hurls a fireball across the screen, decreases the enemy's hit points, and immediately returns control the game engine. Gary the NPC works the same way; the only real difference is that he talks about garlic rather than attacking anyone. In either case though, the idea is that NPCs and weapons work on a per-use basis.

Enemies, on the other hand, much like the player, are constantly affecting the game throughout the course of their battles. From the moment the battle starts to the point at which either the enemy or the player is defeated, the enemy must interpret to the player's input and make decisions based on it. It's in a constant state of activity, and as such, its script must be written in a different manner. Basically, the difference is that you need to think of the code as being part of a larger, constant loop rather than a single, self-contained event. Check out Figure 2.9 for a visual idea of this.

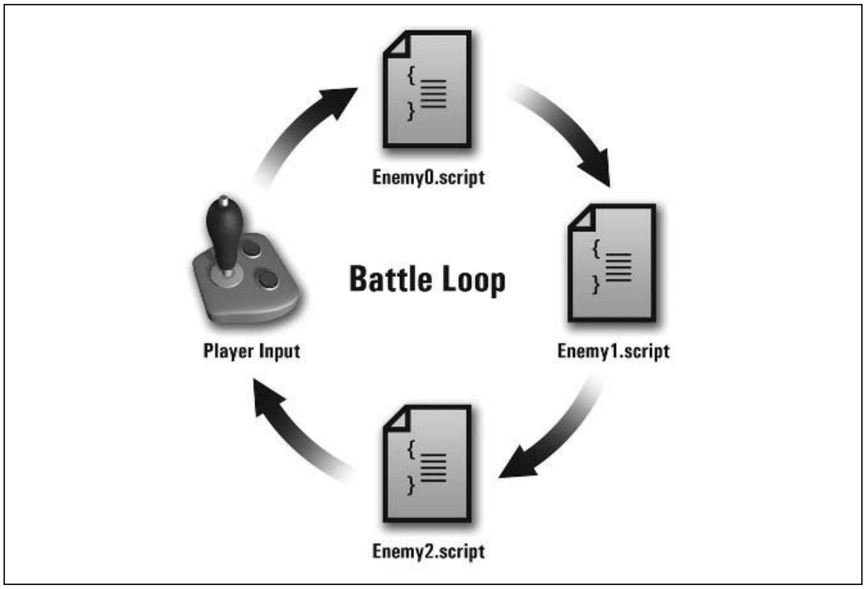


Figure 2.9

The basic outline of an RPG battle loop. At each iteration of the loop, the player and enemies are both polled for input. In the case of the player, this means handling incoming data from input devices; in the case of enemies, this means executing their battle scripts.

Like virtually all types of gameplay, an RPG battle is just a constantly repeating loop that, at each iteration, accepts input from the player and the enemy, manages their interactions, and calculates the overall results of their moves. It does this non-stop until either party is defeated, at which point it terminates and a victor is declared. So, rather than writing a chunk of code that's executed once and then forgotten, you need to write a much more specific and fine-grained routine that the game engine can automatically call every time the battle loop iterates. Instead of doing one thing and immediately being done with it, an enemy's AI script must repeatedly process whatever input was received since its last execution, and react to that input immediately. Here's a basic example:

```
void Act ()
{
   int iWeakestPlayer, iLastAttacker;

   if ( iHitPoints < 20 )
      if ( rand () % 10 == 1 )
         Flee ();
```

```
    else
    {
        iWeakestPlayer = GetWeakestPlayer ();

        if ( Player [ iWeakestPlayer ].iHitPoints < 20 )
            Attack ( iWeakestPlayer, METEOR_SHOWER );
        else
        {
            iLastAttacker = GetLastAttacker ();

            switch ( Player [ iLastAttacker ].iType )
            {
                case NINJA:
                {
                    Attack ( iLastAttacker, THROW_FIREBALL );
                    break;
                }

                case MAGE:
                {
                    Attack ( iLastAttacker, BROADSWORD );
                    break;
                }

                case WARRIOR:
                {
                    Attack ( iLastAttacker, SUMMON_DEMON );
                    break;
                }
            }
        }
    }
}
```

As you can see, it's a reasonably simple block of code. More importantly, note that it doesn't really have a beginning or an end; it's written to be "inserted" into an already running loop that will provide the initial input it uses to make its decisions.

In a nutshell, the AI works like this: First the enemy script determines how close to defeat it is. If it's lower than a certain threshold (fewer than 20 hit points in this case), it simulates an "attempt" to escape the battle by fleeing only if a random number generated between 1 and 10 is 1. If it feels strong enough to keep fighting, however, it calls a function provided by the battle engine to determine the identity of the weakest player. If the enemy deems the player suitably close to defeat (in this case, if his HP is less than 20), it wipes him out with the devastating "Meteor Shower" attack (whatever that is). If the weakest player isn't quite weak enough to finish off yet, the enemy instead goes after whoever attacked it last and chooses a specific counter-attack based on that player's type.

Not too shabby, huh? Parameter-based enemy descriptions hopefully aren't looking too appealing now, after seeing what's possible with procedural code.

Well that just about wraps up this discussion of RPG scripting, so you can now turn your attention to a more action-oriented game genre—first-person shooters.

First-Person Shooters (FPSs)

The first-person shooter is another hot spot for the research and development of scripting systems. Because such games are always on the cutting edge of realism in terms of both the game environment as well as the player's interaction with that environment's inhabitants, scripting plays an important role in breathing life into the creatures and objects of an FPS game world. Although the overall volume of media required for an FPS is usually less than that of an RPG, the flip side is that the expected detail and depth of both enemy AI as well as environmental interaction is much higher. While RPGs are usually more about the adventure and storyline as a whole, first-person shooters rely heavily on the immediate experience and reality of the game from one moment to the next. Figure 2.10 is a screenshot from *Halo*, a next-generation FPS.

As a result, players expect crates to explode into flying shards when they blow up; windows to shatter when they're shot; enemies to be intelligent and strategic, attacking in groups and coordinating their efforts to provide a realistic opposition; and powerful guns to fight their way from one side of the level to the other. There's no room in an FPS for cookie-cutter bad guys who all follow the same pattern, or weapons that are all the same basic projectile drawn with a different sprite. Even the levels themselves need a constantly changing atmosphere and sense of character. This all screams for a scripted solution that allows these elements to be externally coded and controlled with the same flexibility of the game's native language. Furthermore, communication between running scripts and the host application is emphasized to an almost unparalleled degree in an FPS in order to keep the illusion of a real, cohesive environment alive during the game.

Although a full-fledged FPS is of course based on a huge number of game elements, this section discusses the scripting possibilities behind two of the big ones: level objects, such as crates, retractable bridges and switches, as well as enemy AI.

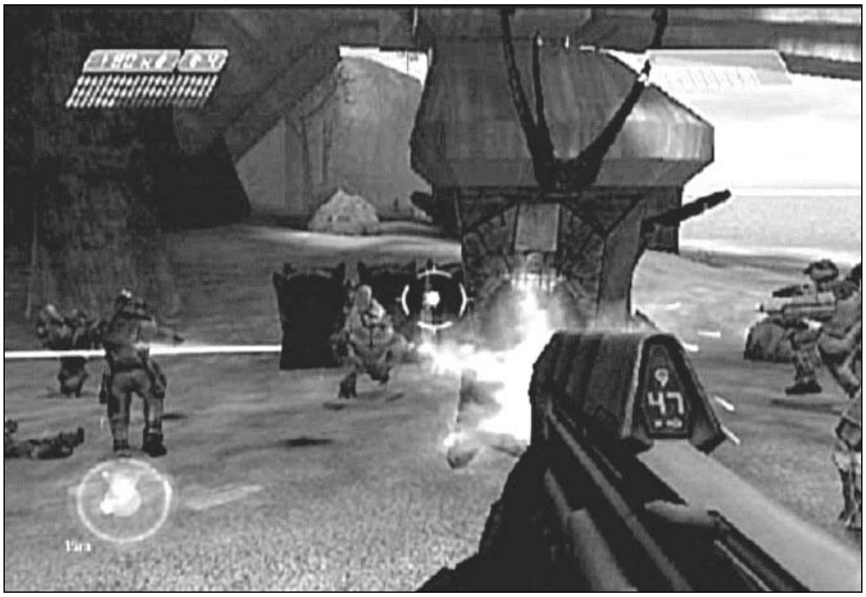


Figure 2.10

Halo, a popular first person shooter from Bungee. It might be harder to tell from a still, black-and-white image, but the game is rife with living, moving detail of all types. First person shooters thrive on this sort of relentless realism, and thus require sophisticated game engines, high-end hardware and intelligent use of scripting systems.

Objects, Puzzles, and Switches (Obligatory Oh My!)

The world of a highly developed FPS needs to feel "alive." Ideally, everything around you should properly react to your interaction with it, whether you're using it, activating it, shooting it, throwing grenades at it, or whatever else you like doing with crates and computer terminals.

If you see a light switch on the wall, you should be able to flip the lights on or off with it. If the door you want to open is locked and you see a computer terminal across the room, chances are that you can use the terminal to open the door. Crates, barrels, and pretty much any sort of generic storage container (the more toxic, the better) should explode or at least fall apart when a grenade goes off nearby. Bridges should retract and extend when their corresponding levers are thrown, windows should shatter when struck, lights should crack and dim when shot, and, well, you get the idea. The point is, objects in the game world need to react to you, and they should react differently depending on how you choose to interact with them.

But it's not entirely about property damage. As fun as it may be to blow up barrels, knock out windows and demolish light fixtures, interaction with game objects is also a common way for the player to advance through the level. Locating a hidden switch might be necessary in order to extend a bridge over a chasm, gaining access to a computer terminal might be the only way to lower the shields surrounding the reactor you want to destroy, or whatever. In these cases, objects are no longer self-contained, privately-operating entities. They now work together to create complex, interconnected systems, and can even be combined to form elaborate puzzles. Check out Figure 2.11.

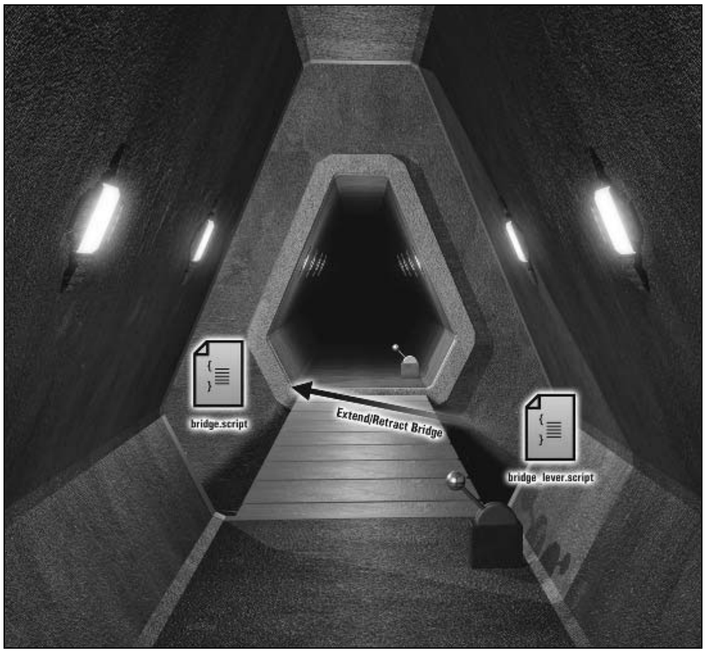


Figure 2.11

A mock-up hallway scene from an FPS. In scenes such as this, scripts are interconnected as functional objects that form a basic communication network. Pulling the lever will send a message to the bridge, telling it to either extend or retract. The bridge might then want to send a message to the lever on the other side, telling it to switch positions. This kind of object-to-object communication is common in such games.

First-person shooters often use switches and puzzles to increase the depth of gameplay; when pumping ammunition into aliens and zombies gets old, the player can focus instead on more intellectual challenges.

The Solution

Almost *everything* in an FPS environment has an associated script. These scripts give each object in the game world its own custom-tailored functionality, and are executed whenever said object comes into contact with some sort of outside force, such as the shockwave of an explosion, a few hundred rounds of bullets, or the player's prying hands.

Within the script, functionality is further refined and organized by associating blocks of code with *events*. Events tell the script who or what specifically invoked it, and allow the script to take appropriate action based on that information. Events are necessary because even the simplest objects need to behave differently depending on the circumstances; it wouldn't make much sense for a crate to violently explode when gently pushed, and it'd be equally confusing if the crate only slid over a few inches after being struck by a nuclear missile.

Events in a typical FPS relate to the abilities of the players and enemies who inhabit the game world. For example, players might be able to perform the following actions:

- **Fire.** Fires the weapon the player is currently armed with.
- **Use.** Attempts to use whatever is in front of the player. "Using" a crate would have little to no effect, but using a computer terminal could cause any number of things to happen. This action can also flip switches, throw levers, and open doors.
- **Push/Move.** Exerts a gentle force on whatever is in front of the player in an attempt to move it around. For example, if the player needs to reach the opening to an air vent that's a few feet too high, he or she might push a nearby crate under it to use as a intermediate step.
- **Collide.** Simply the result of walking into something. This is less of an "action" and more of a resulting event that might not have been intentional.

These form an almost one-to-one relationship with the events that ultimately affect the objects in question. For example, shooting a crate would cause the game engine to alert the crate's respective script that it's under fire by sending it a `SHOT` or `DESTROYED` event. It might even tell the crate what sort of weapon was used, and who was firing it. Using a computer terminal would send a `USE` event to the terminal's script, and so on. Once these events are received by scripts, they're routed to the proper block of code and the appropriate action is subsequently taken. Let's look at some example code. I'm going to show you three object scripts; one for a crate, one for a switch that opens a door, and one for an electric fence.

For the sake of the examples, let's pretend that `this` is a structure that contains the properties of each object, such as its visibility and location. Also, `Event` is a structure containing relevant event information, such as the type of event, the entity that caused it, and the direction and magnitude of force. Obviously, `InvokingEvent` is an instance of `Event` that is passed to each event script's `main ()` function automatically by the host application (the game engine).

Here's the crate:

```
/*
*   Crate
*
*   Can be shot and destroyed, as well as pushed around.
*/

main ( Event InvokingEvent )
{
   switch ( InvokingEvent.Type )
```

```
    {
        case SHOT:
        {
            /*
            The crate has been shot and thus destroyed, so
            first let's make it disappear.
            */

            this.bIsVisibile = FALSE;
                        /*
            Now let's tell the game engine to spawn an explosion
            in its place.
            */

            CreateExplosion ( this.iX, this.iY, this.iZ );

            /*
            To complete the effect, we'll tell the game engine to
            spawn a particle system of wooden shards, emanating from
            the explosion.
            */

            CreateParticleSystem ( this.iX, this.iY, this.iZ, WOOD );

            break;
        }

        case PUSH:
        {
            /*
            Something or someone is pushing the crate, so it's pretty much just a
simple matter of moving it in their direction. We'll assume that the game engine
will take care of collision detection. :) The force vector contains the force of the
event along each axis, so all we really need to do is add it to the location of the
crate.
            */

            this.iX += InvokingEvent.ForceVector.iX;
            this.iY += InvokingEvent.ForceVector.iY;
            this.iZ += InvokingEvent.ForceVector.iZ;
```

```
      }
   }
}
```

And the door switch:

```
/*
*   Door Switch
*
*   Can be shot and destroyed, and is also
*   used to open and close a door.
*/

main ( Event InvokingEvent )
{
   switch ( InvokingEvent.Type )
   {
      case SHOT:
      {
         /*
         Just to be evil, let's make the switch very fragile.
         Shooting it will destroy it and render it useless!
         Ha ha!
         */

         this.bIsBroken = TRUE;

         /*
         And just to make things a bit more realistic, let's
         emanate a small particle system of plastic shards.
         */

         CreateParticleSystem ( this.iX, this.iY, this.iZ, PLASTIC );

         break;
      }

      case USE:
      {
         /*
         This is the primary function of the switch. Let's
         assume that the level's doors exist in an array,
         and the one we want to open or close is at index
         zero.
         */

         if ( Door [ 0 ].IsOpen )
            CloseDoor ( 0 );
         else
            OpenDoor ( 0 );

         break;
      }
   }
}
```

And finally, the electric fence.

```
/*
*   Electric Fence
*
*   Simply exists to shock whoever or whatever comes in
*   contact with it.
*/

main ( Event InvokingEvent )
{
   switch ( InvokingEvent.Type )
   {
      case COLLIDE:
      {
         /*
         The fence only needs to react to COLLIDE events because
         its only purpose is to shock whatever touches it.
         Basically, this means decreasing the health of whatever
         it comes in contact with. The event structure will tell
         us which entity (which includes players and enemies)
         has come in contact with the fence.
         */
```

```
            Entity [ InvokingEvent.iEntityIndex ].Health -= 10;

            /*
            But what fun is electrocution without the visuals?
            */

            CreateParticleSystem ( this.iX, this.iY, this.iZ, SPARKS );

            /*
            And to really drive the point home...
            */

            PlaySound ( ZAP_AND_SIZZLE );
        }
    }
}
```

And there you go. Three fully-functional FPS game world objects, ready to be dropped into an alien corridor, a military compound, or a battle arena. As you can see, the real heart of this system is the ability of the game engine to pass event information to the script; once this is in place, objects can communicate with each other during gameplay via the game engine and form dynamic, lifelike systems. Switches can open doors; players and enemies can blow up kerosene barrels; or whatever else you can come up with.

Event-based script communication is an extremely important concept, and one that will be touched upon many times in the later chapters. In fact, let's discuss a topic that exploits it to an even greater extent right now.

Enemy AI

If nothing else, an FPS is all about mowing down bad guys. Whether they're lurking through corridors, hiding behind crates and under overhangs, or piling out of dropships, your job description is usually pretty straightforward—to reduce them to paint.

Of course, things aren't so simple. Enemies don't just stand there and accept your high-speed lead injections with open arms; they're designed to evade your attacks, return the favor with their own, and generally do anything they can to stop you in your tracks. Naturally, the actual strategies and techniques involved in combat such as this are complex, requiring constant awareness of the surrounding environment and a capable level of intelligence. This is all wrapped up into a nice tidy package called "enemy AI".

AI, or artificial intelligence, is what makes a good FPS such a convincing experience. Games just aren't fun if enemies don't seem lifelike and unique; if you're simply bombarded with lemming-like creatures that dive headlong into your gunfire, you're going to become very bored, very quickly. So, not surprisingly, the AI of FPS bad guys is a rapidly evolving field. With each new generation of shooter, players demand more and more intelligence and strategy on behalf of their computer-controlled opponents in hopes of a more realistic challenge.

As a result, the days of simply hardcoding a player-tracking algorithm and slapping it into the heads of every creature in your game are long gone. Different classes of enemies need to starkly contrast others, so as to provide an adequate level of variety and realism, and of course, to keep the player from getting bored. Furthermore, even enemies within the same class should ideally exhibit their own idiosyncrasies and nuances—anything to keep a particularly noticeable pattern from emerging. In addition to simply dodging attacks, however, enemies need to exhibit clearly realistic *strategies*; taking advantage of crates as hiding places, blowing up explosive objects near the player rather than directly shooting at him, and so on.

So far, so good; by now I think it's safe to say that you're sold on the flexibility of scripts; obviously, a C/C++-style scripting language with maybe a few built-in math routines for handling vectors and such should be more than enough to program lifelike AI and associate it with individual enemies. But smart enemies aren't enough if they simply operate alone. More and more, the concept of team play is taking over, and the real fun lies in taking on a hoard of enemies that have complete awareness of and communication with one another. Rather than simply acting as a chaotic mob that charges towards the player and relies solely on its size, enemies need to intelligently organize themselves to provide a unique and constantly evolving challenge. In games like Rainbow Six, when you're up against a team of terrorists, the illusion would be lost if they simply rushed you with guns blazing. Especially in the case of hostage situations, structured enemy communication and intelligence is an absolute must.

Returning to the general action genre of first person shooters, however, consider a number of group-based techniques enemies can employ when attacking the player:

- Breaking into simple groups for the purpose of attacking the player from a number of angles, depriving the player of a single target to focus on.
- Breaking into logical "task groups" that hinder the player in different ways; as one group directly attacks the player with a point-blank assault, other groups will set up more long-term defenses, such as blocking off power-ups or access to the rest of the level or arena.
- Literally surrounding the player on all sides (assuming the group is large enough), leaving no safe exit for the player.

As you can see, they're rather simple ideas, but they all share a common thread—the concept of enemy communication. In order to form any sort of group, pattern or formation, enemies need to be able to share ideas and information that help transition their current positions and objectives into the desired ones. So if one enemy, designated as the "leader" of sorts, decides that surrounding the player would be the most effective strategy, that leader needs the ability to spread that message around.

The Solution

If enemies need to communicate, and enemies are based on scripts, what I'm really talking about here is *inter-script communication*. So, for example, the script that controls the "leader" needs to be able to send messages directly to the scripts that control the other enemies. The enemy scripts are written specifically with this message system in mind, allowing them to interpret incoming messages and act appropriately.

I touched on this earlier in the section on FPS objects, where object scripts were passed event descriptions that allowed them to act differently depending on the entity's specific method of interaction with them. In that case, however, you relied on the game engine to send the messages; although players and enemies were of course responsible for invoking the events in the first place due to their actions, it was ultimately the game engine that noticed and identified the events and properly informed the object. Although engine-to-script communication is a useful and valuable capability in its own right, direct script-to-script communication is the basis for truly dynamic systems of game objects and entities that can, entirely on their own, work together to solve problems and achieve goals. Figure 2.12 depicts this process graphically.

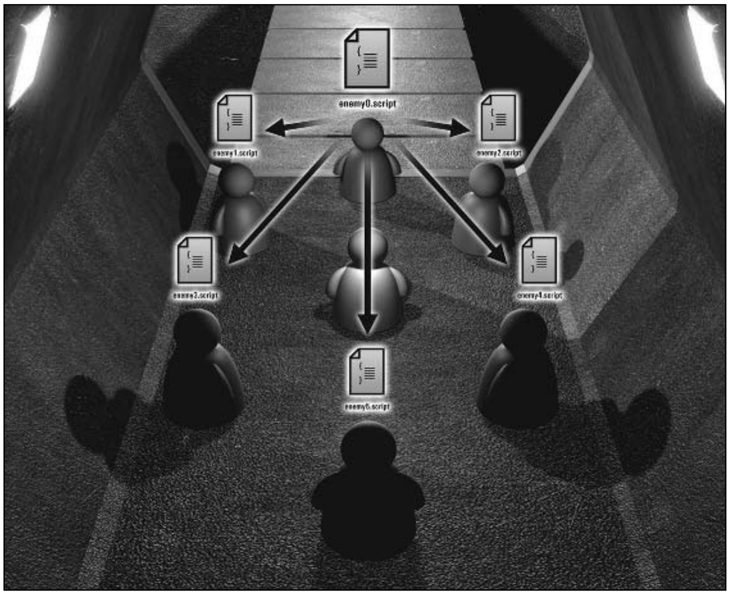


Figure 2.12

FPS enemies using scripting to communicate. In this case, they've used their communication abilities to form a surrounding formation around the player (the guy in the center).

An actual discussion of artificial intelligence, however, would be lengthy at best and is well beyond the scope of this book. The main lesson here is that script-to-script communication is a must for any FPS, because it's required for group-based enemy AI.

Summary

With any luck, your interest in scripting has taken on a more focused and educated form over the course of this chapter. This chapter took a brisk tour of a number of ways in which scripts can be applied to two vastly different styles of games, and certainly you've seen plenty of reasons why scripts are a godsend in more than a few situations. Fortunately, you're pretty much finished with the introductory and background-information chapters, which means actually getting your hands dirty with some real script system development is just around the corner.

Brace yourself, because the gloves are coming off and things are going to get messy!

Part Two

Command-Based Scripting

This page intentionally left blank

CHAPTER 3

Introduction to Command-Based Scripting

"It's not Irish, it's not English,
it's just... well... it's just Pikey."
——*Turkish,* Snatch

With the introductory stuff behind you, it's time to roll up your sleeves and take a stab at some basic scripting. To get started, you're going to explore a simple but useful method of scripting known as *command-based scripting*. Command-based scripts starkly contrast the types of scripts you'll ultimately write—they don't support common programming language features such as variables, loops, and conditional logic. Rather, as their name suggests, command-based languages are entirely based on specific commands that can be called with optional parameters. These commands directly cause the game engine to do something, such as move a player on the screen, change the background music, or display a bitmapped image. By calling a number of commands in a sequential fashion, you can externally control the engine's behavior (albeit in a rather simplistic way).

Command-based languages have a number of advantages and disadvantages, covered shortly. The most important lesson to learn about them, however, is that they're simple and relatively weak in terms of capabilities, but they're *very* easy to implement and can be used to achieve a lot of very cool results. In this chapter, you're going to

- Learn about the theory behind command-based languages, and how they're implemented.
- Implement a command-based language that manipulates the text console.
- Use a command-based language to script the intro sequence to a generic game.
- Apply command-based scripting to the behavior of the non-player characters in a basic RPG engine.

This chapter introduces a number of very important concepts that will ultimately prove vital later. Because of this, despite the relative simplicity of this chapter's contents, it's important that you make sure to read and understand all of it before moving on to the following chapters.

The Basics of Command-Based Scripting

Command-based languages are based on a very simple concept—high-level control of a game engine. I say high-level because command-based scripts are usually designed to do major things. Rather than rasterize individual polygons or rotate bitmaps, for example, they're more concerned with moving characters around in the game world, unlocking doors in fortresses, scripting the dialogue and events in cut scenes, and giving the player items and weapons. When you think in these terms, game engines really only perform a limited number of tasks. Even a game like Quake, for example, is based primarily on only a few major actions, such as:

- Player and robot movement within the game world.
- The firing of player and robot (bot) weapons.
- Managing the damage taken by collisions between players, bots, and projectiles.
- Assigning weapons and items to players and bots who find them, and decreasing ammo levels of those weapons as they're used.
- Loading new maps, changing background music, and other scene/background-oriented tasks.

Now don't get me wrong—Quake the engine is an extremely complex piece of software. Quake the game, however, despite being highly complex, can be easily boiled down to these far simpler concepts. This is true for virtually all games, and is the idea that command-based languages capitalize on, as shown in Figure 3.1.

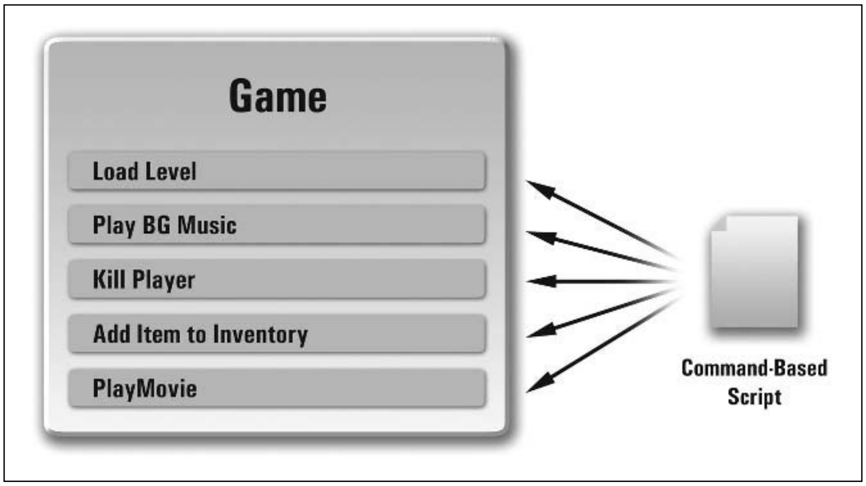


Figure 3.1

Command-based scripts control the game's basic functionality.

High-Level Engine Control

Because game engines are really only concerned with these high-level tasks, a lot can be accomplished by simply giving the engine a list of actions you want it to perform in a sequential order. As an example, think about how a Quake-like, first-person shooter game engine would switch arenas, on both a high- and low-level. Here's how it might work on a low-level:

- The screen freezes or is covered with a new bitmap to hide the inner workings of the process from the player.
- The memory allocated to hold the current level is freed.

- The file containing the new arena's geometry, textures, shadow maps, and other such resources is opened.
- The file format is parsed, headers are verified, and data is carefully extracted.
- New structures are allocated to store the arena, which are incrementally filled with the data from the file.
- The existing background music fades out.
- The existing background music is freed.
- Some sort of sound is made to give the player an auditory cue that the level change has taken place.
- The new background music is loaded.
- The new background music fades in.
- The screen freeze/bitmap is replaced by the next frame of the game engine running again, this time with the new level loaded.

As you can see, there are quite a lot of details to consider (and even now I'm skimming over countless intricacies). On a high-enough level, however, you can describe this sequence in much simpler terms:

- A background image is displayed (or the screen is frozen).
- A new level is loaded.
- The existing background music fades out.
- A level-change sound is played.
- A new background track is loaded.
- The new background music fades in.
- The game resumes execution.

Issues like the de-allocation of memory and the individual placement of blocks of data read from files can be glossed over entirely when explaining such a process in high-level terms, because all you care about is what's *conceptually* going on. In a lot of ways, it's almost like the difference between explaining this sequence to a technical person and a non-technical person. The techie will understand the importance of memory allocation and file handles, whereas such details will probably be lost on a less technical person, like your mail carrier. The mail carrier will, however, understand concepts like fading music in and out, switching levels, and so on (or just hand you some bills and catalogs and mysteriously stop delivering to your neighborhood the next day). Figure 3.2 illustrates how these high- and low-level entities interact.

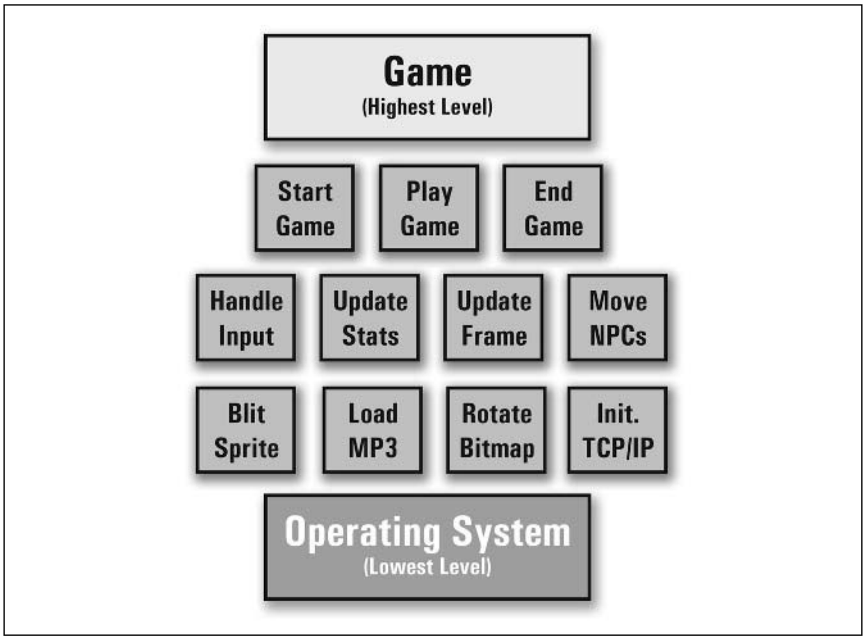


Figure 3.2

The functionality of a game and its engine is a multi-layered system of components.

The point to all this is that writing a command-based script is like articulating the high-level explanation of the process in a reasonably structured way. Let's just jump right in and see how the previous process would look as a command-based script:

```
ShowBitmap "Gfx/LevelLoading.bmp"
LoadLevel "Levels/Level4.lev"
FadeBGMusicOut
PlaySound "Sounds/LevelLoaded.wav"
LoadBGMusic "Music/Level4.mp3"
FadeBGMusicIn
```

As you can see, a command-based language is exactly that— a language based entirely on commands. Each command maps to a specific action the game engine can perform, like displaying bitmap images, loading MP3s, fading music in and out, and so on. As you can also see, these commands can accept (and indeed, often require) various parameters to help specify their tasks more precisely. In this regard, commands are highly analogous to functions, and can be thought of in more or less the same ways.

Commands

Specifically, a *command* is a symbolic name given to a specific game engine function or action. Commands can accept zero or more parameters, which can vary in data types but must always be literal values (command-based languages don't support variables or other methods of indirection). Here's the general syntax:

```
Command Param0 Param1 Param2
```

Imagine writing a C program that defines a `main ()` function and a number of other random functions, each of which accept zero to *N* parameters. Now imagine the `main ()` function cannot declare any local variables, or use any globals, and can only call the other functions with literal values. That's basically what it's like to code in a command-based language.

Of course, the syntax presented here is different. For simplicity's sake, extraneous whitespace is not allowed—the command and each of its parameters must be separated by a single space. There are no commas, tabs, or anything along those lines. Commands are always expressed on a single line and must begin at the line's first character.

Master of Your Domain

Another defining characteristic of command-based languages is that they're highly domain-specific. Because general-purpose structures like loops and branches don't exist, every line of code is just a call to a specific game engine feature. Because of this, each language is custom-designed around a single specific game, or type of game. This is known as the language's *domain.*

As you'll soon see, many of the underlying details of a command-based scripting system's implementation can be ported from one project, but the command list itself, and each command's implementation, is more or less hard-coded and generally only applicable to that specific project. For example, the following commands would suit an RPG or RPG-like game nicely:

```
MovePlayer
GetItem
CastSpell
PlayMovie
Teleport
InvokeBattle
```

These would hardly apply to a flight simulator or racing game, however.

Actually Getting Something Done

With all of these restrictions, you may be wondering if command-based languages (or CBLs, as the street kids are saying nowadays) are actually *useful* for anything. Admittedly, the inability to define or use variables, expressions, loops, branches, and other common features of programming languages is a serious setback. What this means, however, is not that command-based scripting is useless, but rather that it has different applications. For example, a 16 MHz CPU that can address 64KB of RAM might seem completely useless when compared to a 64-bit Pentium whose speeds are measured in GHz. However, such a chip might prove invaluable when developing a remote-controlled car or clock radio. Rather than thinking in terms of whether something is useful or useless, think in terms of its applications.

Remember, a command-based language is a quick and easy way to define a *sequential* and *static* series of events for the game engine to perform. Although this is obviously useless when attempting to script a particle system or complex AI logic for your game's final boss, it can be applied to simpler things like the details of your game's intro sequence, or the behavior of simple NPCs (non-player characters) in an RPG engine. In fact, you'll see examples of both of these applications in the following pages.

Command-Based Scripting Overview

Now that you understand the basics of command-based scripting, you're ready to take a brief look at how it's actually done.

Engine Functionality Assessment

Before doing anything else, the first step in designing and implementing a command-based language is determining two points:

- What the engine can do.
- What the engine's scripts will need to do.

It's important to differentiate between something the engine *can* do, and something scripts will actually *need* it to do. Also, just because an engine is capable of something doesn't mean a script can access or invoke it. All of the functionality you'd like to make available to scripts must first be wrapped in a command handler, which is a small piece of code that actually performs the action associated with each command.

For example, let's consider a simple, top-down, 2D RPG engine like the ones seen on the Nintendo, Super Nintendo, and Sega Saturn. These games were based around 2D maps composed of small, square graphics called *tiles*. These maps defined the background and general environment of each location in the game and could scroll in all four directions. On top of these maps, sprite-based characters would move around and interact with one another, as well the underlying background map. As you learned in the last chapter, one major issue of such games is the non-player characters (NPCs). NPCs need to appear lifelike, at least to some extent, and therefore can't simply stand still and wait for the player to approach them. They must move around on their own, which generally translates into code that must be written to define their actions.

In the case of this example, the commands listed in Table 3.1 might prove useful for scripts:

Table 3.1 RPG Engine Script Commands

Command	Description
`SetNPCDir`	Sets the direction in which the NPC is facing.
`MoveNPC`	Moves the NPC along the X and Y axes by the specified distances.
`Pause`	Causes the NPC to stand still for the specified duration.
`ShowTextBox`	Displays the specified string of text in a text box; used for dialogue.

Each of these commands requires some form of parameters to help direct its action. Such parameters can be expressed as one of two data types—integers and strings. Parameters are not separated by commas, but by a single space instead. The parameter list is also separated from the command itself by a single space, which means the overall syntax of a command in this language is as follows:

```
Command Param0 Param1 Param2
```

And *exactly* this. The language is in no way free-form, so arbitrary use of whitespace is not permitted.

With only four commands, this particular language is hardly feature-rich. You'd be surprised by how much these four simple commands can accomplish, however. Consider the following script.

```
SetNPCDir "Up"
MoveNPC 0 -20
Pause 200
SetNPCDir "Left"
MoveNPC -20 0
```

```
Pause 400
SetNPCDir "Down"
ShowTextBox "Hmmmmm... I know I left it here somewhere..."
Pause 400
```

Can you tell what this does just by looking at it? In only a few lines of simplistic script code, I've defined the behavior for an NPC who's clearly looking for something. He starts off in a given position, facing a given direction, and turns "up" (which actually just means north). He walks in that direction 20 pixels, pauses, and then turns left (west) and walks 20 more pixels. He pauses again, this time for a longer duration, and finally turns back towards the camera ("down", or south) and makes a comment about something he lost. The script then pauses briefly to allow the player a chance to read it, and, presumably, the script loops back to the beginning and starts over.

NOTE

You may be wondering why the cardinal directions in the NPC script like `"Up"` and `"Down"` are expressed as a string. This is because the language doesn't support symbolic constants like C's `#define` or C++'s `const`. It would be just as easy to create a `SetNPCDir` command that accepted integer codes that specified directions (0-3, for example), but it's a lot harder to remember an arbitrary number than it is to simply write the string. Regardless, this is still a messy solution, so keep reading—the next chapter will revisit this matter.

For such a simple scripting system, and even simpler script, this is quite a lively little character. Imagine how much personality you could squeeze out of your NPCs if you added just a few more commands! Hopefully, you're beginning to understand that you don't need too much complexity to get decent results when scripting.

Loading and Executing Scripts

The lifespan of a script spans multiple phases, each of which are illustrated in Figure 3.3. First, the script is loaded. In this simple language, where vertical whitespace and comments are not permitted, this simply means loading every line of the source file into a separate element of an array of strings. Once this process is complete, the array contains an in-memory copy of the script, ready to run. Check out Figure 3.4 for a visual idea of a script's in-memory form.

Once in memory, the script is executed by passing each line of code to a script handler (or executor, or whatever you want to call it) that processes each command, reads in parameters, and so forth. After a command and its parameters are processed and understood, the command handler performs whatever task the command is associated with. The command handler for `MoveNPC`, for example, uses the two integer parameters (the X and Y movement) to make direct changes to

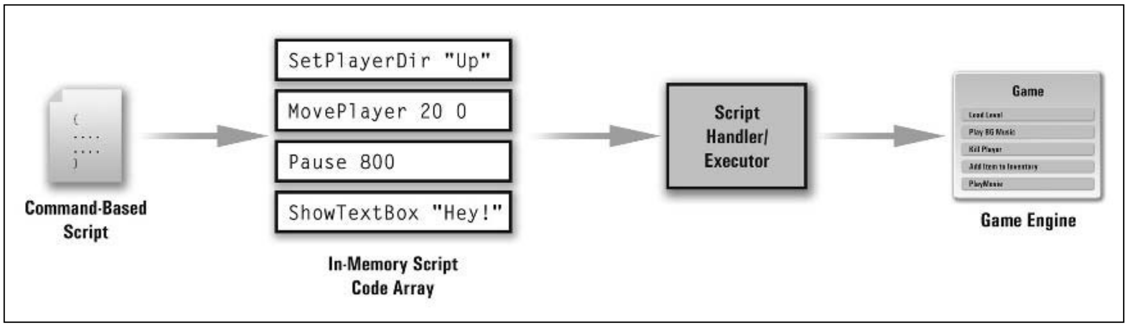


Figure 3.3

The lifespan of a script. The script is loaded into an array of strings, executed through the script handler, and finally exerts its control of the game engine.

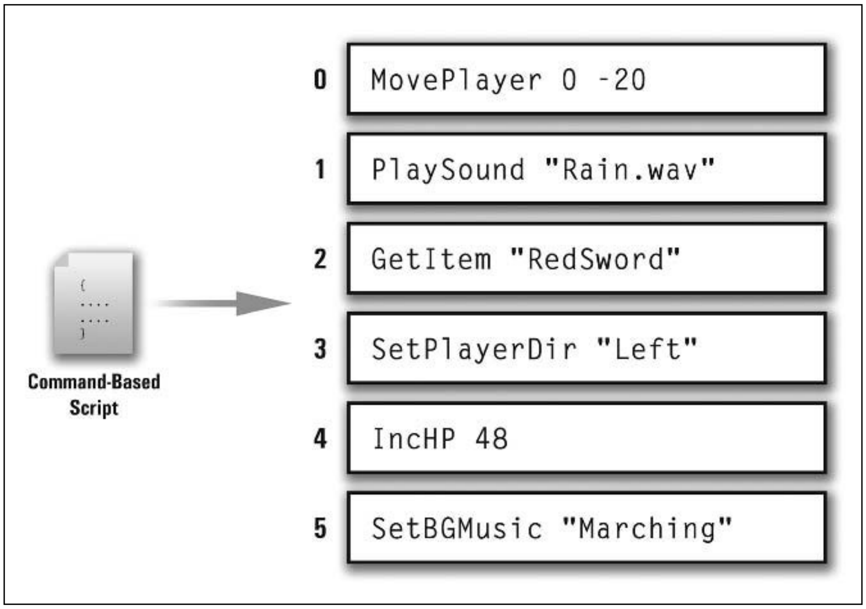


Figure 3.4

A script in memory.

the NPC data within the game engine. At this point, the script has succeeded in controlling the game engine.

The execution of command-based scripts is always purely sequential. This means that execution starts with the first command (line 0) and runs until the last command (line 5, in the case of Figure 3.4). At each step of the way, a global variable representing the current line of code within the script is updated to reflect the next command to process. This global might be called something like `g_iCurrLine`, for "current line". When this process is repeated in a loop, the script executes quickly and continually, simulating the execution of actual code. Once the last command in the script is reached, the script can either stop or loop back to the beginning and run again. Figure 3.5 illustrates the execution of a script.

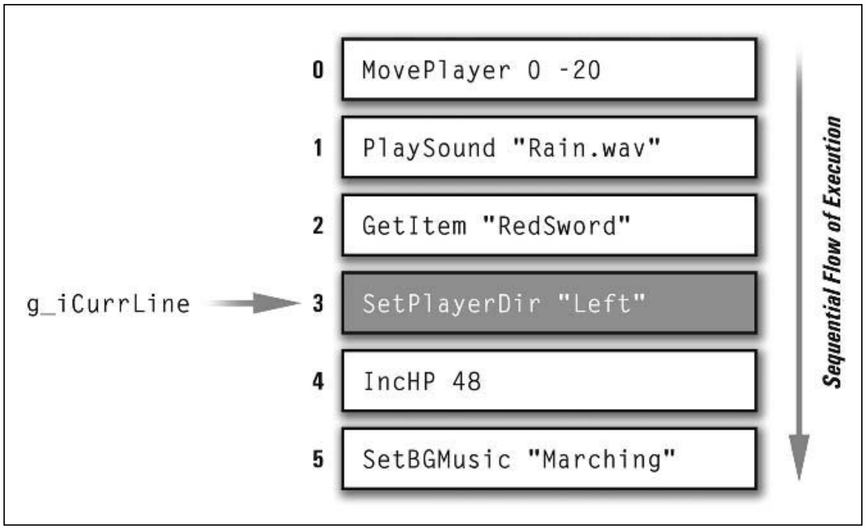


Figure 3.5

The execution of a script.

Looping Scripts

So should your scripts loop or stop when the last command ends? There's no straight answer to this question, because this is a decision that must be made on a per-script basis. For example, continuing on with the RPG engine theme, an example of a script that should execute once and immediately stop would be the script that defines the behavior of an item or weapon. When the player uses the item, the script needs to execute once, allowing the item to perform its task or action, and then immediately terminate. The item shouldn't operate more than once unless the player has specifically requested it to do so, or if the item has some sort of persistent nature to it (such as a torch that must remain lit).

> **TIP**
>
> **The issue of looping scripts and their tendency to appear contrived or predictable can be resolved in a number of ways. First of all, scripts that are sufficiently long can produce enough unique behavior before looping that players won't have the time (or interest) to notice a pattern develop. Also, it's possible to write a number of small scripts that all perform the same action in a slightly different way, which are then loaded at random by the game engine to produce behavior that is truly random (or nearly so).**

Scripts that should loop are those that primarily control background-related or otherwise ambient entities. For example, NPCs represent the living inhabitants of the game world, which means they should be constantly moving to keep the player's suspension of disbelieve intact. NPC scripts, therefore, should immediately revert to the first command after executing the last so that their actions never cease. Granted, this means that looped scripts will demonstrate a discernable pattern sooner or later, which might not be a good thing. I didn't say command-based scripts weren't without their disadvantages, though.

Implementing a Command-Based Language

With the theory out of the way, you can now actually implement a small, command-based language. To get things started, you're going to keep it simple and design a set of commands for scripting a scrolling text console like the ones seen in old text mode programs, or any Win32 console app.

Designing the Language

The first step is establishing a list of commands the language will need in order to effectively control the console. Table 3.2 lists them.

Again, just four commands. Because text consoles are pretty simple by nature, you don't need a lot of options and can get by with just a handful of commands. Remember, just because you *can* make something complex doesn't mean you should. Now that you have a language specification to work with, you're ready to write an initial script to test it.

Table 3.2 Text Console Commands

Command	Parameters	Description
`PrintString`	`String`	Prints the specified string.
`PrintStringLoop`	`String, Count`	Prints the specified string the specified number of times.
`Newline`	*None*	Prints an empty line.
`WaitForKeyPress`	*None*	Suspends execution until a key is pressed.

Writing the Script

It won't take much to test this language, because you can deem it functional after implementing just four commands. Here's a reasonable test script, though, that will help determine whether everything is working right in the following pages:

```
PrintString "This is a command-based language."
PrintString "Therefore, this is a command-based script."
Newline
PrintString "...and it's really quite boring."
Newline
PrintStringLoop "This string has been repeated four times." 4
Newline
PrintString "Okay, press a key already and put us both out of our misery."
PrintString "The next demo is cooler, I swear."
WaitForKeyPress
```

Yeah, this particular script is a bit of a downer, but it will get the job done. With your first script in hand, it's time to write a program that will execute it.

Implementation

Implementing a command-based language is a mostly straightforward task. Here's the general process:

- The script is loaded from the file into an in-memory string array.
- The line counter is reset to zero.
- The command is read from the first line of code. A line's command is considered to be everything from the first character of the string, all the way up to the first space.
- Based on the command, any of a number of command handlers is invoked to handle it. These command handlers need to access the command's parameters, so two functions are created for that (one for reading integer parameters, the other for reading strings). With the parameters processed, the command handler goes ahead and performs its task. At this point, the current line of the script is completely executed.
- The instruction counter is incremented and the process continues.
- After the script finishes executing, its array is freed.

Basic Interface

On a basic level, all the scripting system needs to do is load scripts, run them, and unload them. Let's look at the load and unload functions now.

`LoadScript ()` is used to load scripts into memory. It works like this:

- The file is opened in binary mode, and every instance of the `'\n'` (newline) character is counted to determine how many lines it contains.
- A string array is then allocated to hold the script based on this number.
- The script is then loaded, line-by-line, and the file is closed.

Here's the code behind `LoadScript ()`:

```
void LoadScript ( char * pstrFilename )
{
    // Create a file pointer for the script
    FILE * pScriptFile;

    // ---- Find out how many lines of code the script is

    // Open the source file in binary mode
    if ( ! ( pScriptFile = fopen ( pstrFilename, "rb" ) ) )
    {
        printf ( "File I/O error.\n" );
        exit ( 0 );
    }

    // Count the number of source lines
    while ( ! feof ( pScriptFile ) )
        if ( fgetc ( pScriptFile ) == '\n' )
            ++ g_iScriptSize;
    ++ g_iScriptSize;

    // Close the file
    fclose ( pScriptFile );

    // ---- Load the script

    // Open the script and print an error if it's not found
    if ( ! ( pScriptFile = fopen ( pstrFilename, "r" ) ) )
    {
        printf ( "File I/O error.\n" );
        exit ( 0 );
    }
    // Allocate a script of the proper size
    g_ppstrScript = ( char ** ) malloc ( g_iScriptSize * sizeof ( char * ) );

    // Load each line of code
    for ( int iCurrLineIndex = 0;
          iCurrLineIndex < g_iScriptSize;
          ++ iCurrLineIndex )
    {
        // Allocate space for the line and a null terminator
        g_ppstrScript [ iCurrLineIndex ] = ( char * )
            malloc ( MAX_SOURCE_LINE_SIZE + 1 );

        // Load the line
        fgets ( g_ppstrScript [ iCurrLineIndex ],
            MAX_SOURCE_LINE_SIZE, pScriptFile );
    }

    // Close the script
    fclose ( pScriptFile );
}
```

Notice that this function makes a reference to a constant called `MAX_SOURCE_LINE_SIZE`, which is used to read a specific amount of text from the script file. I usually set this value to `4096`, just to eliminate all possibilities of leaving something out, but this is overkill—especially in the case of a command-based language, I can virtually guarantee you'll never need more than 192 or so. The only possible exceptions will be huge string parameters, which may come up now and then when scripting complicated dialogue sequences. So no matter what, with a large enough value this constant will have you covered (besides, you're always free to change it).

Once the source is loaded into the array, it can be executed. Before getting to that, however, check out `UnloadScript ()`, which is called just before the program ends to free the script's resources:

```
void UnloadScript ()
{
    // Return immediately if the script is already free

    if ( ! g_ppstrScript )
        return;
    // Free each line of code individually

    for ( int iCurrLineIndex = 0;
              iCurrLineIndex < g_iScriptSize;
              ++ iCurrLineIndex )
        free ( g_ppstrScript [ iCurrLineIndex ] );

    // Free the script structure itself

    free ( g_ppstrScript );
}
```

The function first makes sure the `g_ppstrScript []` array is valid, and then manually frees each line of code. After this step, the string array pointer is freed, which completely unloads the script from memory.

Execution

With the script in memory, it's ready to run. This is accomplished with a call to `RunScript ()`, which will run until the entire script has been executed. The execution cycle for a command-based language is really quite simple. Here's the basic process:

- The command is read from the current line.
- The command is used to determine which command handler should be invoked, by comparing the command string found in the script to each command string the language supports. In this case, the strings are `PrintString`, `PrintStringLoop`, `Newline`, and `WaitForKeyPress`.
- Each of these commands is given a small block of code to handle its functionality. These blocks of code are wrapped in a chain of `if/else if` statements that are used to determine which command was specified.
- Once inside the command handler, an optional number of parameters are read from the current line and converted from strings to their actual values. These values are then used to help perform the commands action.
- The command block terminates, the line counter is incremented, and a check is made to determine whether the end of the script has been reached. If so, `RunScript ()` returns; otherwise the process repeats.

All in all, it's a pretty straightforward process. Just loop through each line of code and do what each command specifies. Now that you understand the basic logic behind `RunScript ()`, you can take a look at the code. By the way, there will be a number of functions referenced here that you haven't seen yet, but they should be pretty self-explanatory:

```
void RunScript ()
{
    // Allocate strings for holding source substrings
    char pstrCommand [ MAX_COMMAND_SIZE ];
    char pstrStringParam [ MAX_PARAM_SIZE ];

    // Loop through each line of code and execute it
    for ( g_iCurrScriptLine = 0;
          g_iCurrScriptLine < g_iScriptSize;
          ++ g_iCurrScriptLine )
    {
        // ---- Process the current line

        // Reset the current character
        g_iCurrScriptLineChar = 0;

        // Read the command
        GetCommand ( pstrCommand );

        // ---- Execute the command

        // PrintString
        if ( stricmp ( pstrCommand, COMMAND_PRINTSTRING ) == 0 )
        {
            // Get the string
            GetStringParam ( pstrStringParam );
            // Print the string
            printf ( "\t%s\n", pstrStringParam );
        }

        // PrintStringLoop
        else if ( stricmp ( pstrCommand, COMMAND_PRINTSTRINGLOOP ) == 0 )
        {
            // Get the string
            GetStringParam ( pstrStringParam );

            // Get the loop count
            int iLoopCount = GetIntParam ();

            // Print the string the specified number of times
            for ( int iCurrString = 0;
                    iCurrString < iLoopCount;
                    ++ iCurrString )
                printf ( "\t%d: %s\n", iCurrString, pstrStringParam );
        }

        // Newline
        else if ( stricmp ( pstrCommand, COMMAND_NEWLINE ) == 0 )
        {
            // Print a newline
            printf ( "\n" );
        }

        // WaitForKeyPress
        else if ( stricmp ( pstrCommand, COMMAND_WAITFORKEYPRESS ) == 0 )
        {
            // Suspend execution until a key is pressed
            while ( kbhit () )
                getch ();
            while ( ! kbhit () );
        }

        // Anything else is invalid
        else
        {
            printf ( "\tError: Invalid command.\n" );
            break;
        }
    }
}
```

The function begins by creating two strings—pstrCommand and pstrStringParam. As the script is executed, these two strings will be needed to hold both the current command and the current string parameter. Because it's possible that a command can have multiple string parameters, the command handler itself may have to declare more strings if they all need to be held at once, but because no command in this language does so, this will be fine. Note also that these two strings use constants as well to define their length. I have MAX_COMMAND_SIZE set to 64 and MAX_PARAM_SIZE set to 1024, just to make way for the potential huge dialogue strings mentioned earlier.

A for loop is then entered that takes you from the first command to the last. At each iteration, an index variable called g_iCurrScriptLineChar is set to zero, and a call is made to a function called

`GetCommand ()` that fills `pstrCommand` with a string containing the specified command (you'll learn more about `g_iCurrScriptLineChar` momentarily.) A series of `if/else if`'s is then entered to determine which command was found. `stricmp ()` is used to make the language case-insensitive, which I find convenient. As you can see, each comparison is made to a constant relating to the name of a specific command. The definitions for these constants are as follows:

```
#define COMMAND_PRINTSTRING         "PrintString"
#define COMMAND_PRINTSTRINGLOOP     "PrintStringLoop"
#define COMMAND_NEWLINE             "Newline"
#define COMMAND_WAITFORKEYPRESS     "WaitForKeyPress"
```

NOTE
Why are the command names case-insensitive? Don't C/C++ and indeed most other languages do just the opposite with their reserved words? Although it's true that most modern languages are largely case-sensitive, I personally find this approach arbitrary and annoying. All it seems case-sensitivity is good for is actually allowing you to create multiple identifiers with the same name, as long as their case differs, which is a practice I find messy and highly prone to logic errors. Unless you really want to differentiate between `MyCommand` and `myCommand` (which will only end in tears and turmoil), I suggest you stick with case-insensitivity.

The contents of each of these `if/else if` blocks are the command handlers themselves, which is where you'll find the command's implementation. You'll find calls to parameter-returning functions throughout these blocks of code—two of them, specifically—called `GetStringParam ()` and `GetIntParam ()`. Both of these functions scan through the current line of code and extract and convert the current parameter to its actual value for use within the command handler. I say "current" parameter, because repetitive calls to these functions will automatically return the command's next parameter, in sequence. You'll learn more about how parameters are dealt with in a second.

After the command handler ends, the `for` loop automatically handles the incrementing of the instruction counter (`g_iCurrScriptLine`) and makes sure the script hasn't ended. If it has, however, the `RunScript ()` simply returns and the job is done.

Command and Parameter Extraction

The last piece of the puzzle is determining how these parameters are read from the source file. To understand how this works, take a look first at how `GetCommand ()` works; the other functions do virtually the same thing it does.

GetCommand ()

The key to everything is `g_iCurrScriptLineChar`. Although `g_iCurrScriptLine` keeps track of the current *line* within the script, `g_iCurrScriptLineChar` keeps track of the current *character* within that line. Whenever a new line is executed by the execution loop, `g_iCurrScriptLineChar` is immediately set to zero. This puts the index within the source line string at the very beginning, which, coincidentally, is where the command begins. Remember, because of this language's strict whitespace policy, you know for sure that leading whitespace will never come before the command's first character. For example, in the following line of code:

```
PrintStringLoop "Loop" 4
```

The first character of the command, `P`, is found at character index zero. The name of the command extends all the way up to the first space, which, as you can see, comes just after `p`. Everything in between these two indexes, inclusive, composes a substring specifying the commands name. `GetCommand ()` does nothing more than scans through these characters and places them in the specified destination string. Check it out:

```
void GetCommand ( char * pstrDestString )
{
    // Keep track of the command's length
    int iCommandSize = 0;

    // Create a space for the current character
    char cCurrChar;

    // Read all characters until the first space to isolate the command
    while ( g_iCurrScriptLineChar <
        ( int ) strlen ( g_ppstrScript [ g_iCurrScriptLine ] ) )
    {
        // Read the next character from the line
        cCurrChar = g_ppstrScript
           [ g_iCurrScriptLine ][ g_iCurrScriptLineChar ];

        // If a space (or newline) has been read, the command is complete
        if ( cCurrChar == ' ' || cCurrChar == '\n' )
            break;

        // Otherwise, append it to the current command
        pstrDestString [ iCommandSize ] = cCurrChar;

        // Increment the length of the command
        ++ iCommandSize;
        // Move to the next character in the current line
        ++ g_iCurrScriptLineChar;
    }

    // Skip the trailing space
    ++ g_iCurrScriptLineChar;

    // Append a null terminator
    pstrDestString [ iCommandSize ] = '\0';

    // Convert it all to uppercase
    strupr ( pstrDestString );
}
```

Just as expected, this function is little more than a character-reading loop that incrementally builds a new string containing the name of the command. There are a few details to note, however. First of all, note that the loop checks for both single-space *and* newline characters to determine whether the command is complete. Remember, commands like `Newline` and `WaitForKeyPress` don't accept parameters, so in their cases, the end of the command is also the end of the line.

Also, after the loop finishes, you increment the `g_iCurrScriptLineChar` character index once more. This is because, as you know, a single space separates the command from the first parameter. It's much easier to simply get this space out of the way and save subsequent calls to the `Get*Param ()` functions from having to worry about it. A null terminator is then appended to the newly created string, and it's converted to uppercase.

By now, it should be clear why `g_iCurrScriptLineChar` is so important. Because this is a global value that persists between calls to `GetCommand ()` and `Get*Param ()`, each of these three functions can use it to determine where exactly in the current source line you are. This is why repeated calls to the parameter extraction functions always produce the *next* parameter, because they're all updating the same global character index.

NOTE

You may be wondering why I'm using both `strupr ()` to convert the command string to uppercase, and using `stricmp ()` when comparing it to each command name. `stricmp ()` is all I need to perform a case-insensitive comparison, but I'm a bit anal retentive when it comes to this sort of thing and like to simply convert all human-written input to uppercase for that added bit of cleanliness and order. Now if you'll excuse me, I'm going to adjust each of the objects on my desk until they're all at perfect 90-degree angles and make sure the oven is still off.

The process followed by GetCommand () is repeated for both GetIntParam () and GetStringParam (), so you should have no trouble following them. The only real difference is that unlike GetCommand (), both of these functions convert their substring in some form to create a "final value" that the command handler will use. For example, integer parameters found in the script will, by their very nature, not be integers. They'll be strings, and will have to be converted with a call to the atoi () function. This function will return an actual int value, which is the final value the command handler will want. Likewise, even though string parameters are already in string form, their surrounding double-quotes need to be dealt with, because the script writer obviously doesn't intend them to appear in the final output. In both cases, the substring extracted from the script code must first be converted before returning it to the caller.

GetIntParam ()

GetIntParam (), like GetCommand (), scans through the current line of code from the initial position of g_iCurrScriptLineChar, all the way until the first space character is encountered. Once this substring has been extracted, atoi () is used to convert it to a true integer value, which is returned to the caller. Have a look at the code:

```
int GetIntParam ()
{
    // Create some space for the integer's string representation
    char pstrString [ MAX_PARAM_SIZE ];

    // Keep track of the parameter's length
    int iParamSize = 0;

    // Create a space for the current character
    char cCurrChar;

    // Read all characters until the next space to isolate the integer
    while ( g_iCurrScriptLineChar <
            ( int ) strlen ( g_ppstrScript [ g_iCurrScriptLine ] ) )
    {
        // Read the next character from the line
        cCurrChar = g_ppstrScript
            [ g_iCurrScriptLine ][ g_iCurrScriptLineChar ];

        // If a space (or newline) has been read, the command is complete
        if ( cCurrChar == ' ' || cCurrChar == '\n' )
            break;
        // Otherwise, append it to the current command
        pstrString [ iParamSize ] = cCurrChar;

        // Increment the length of the command
        ++ iParamSize;

        // Move to the next character in the current line
        ++ g_iCurrScriptLineChar;
    }

    // Move past the trailing space
    ++ g_iCurrScriptLineChar;

    // Append a null terminator
    pstrString [ iParamSize ] = '\0';

    // Convert the string to an integer
    int iIntValue = atoi ( pstrString );

    // Return the integer value
    return iIntValue;
}
```

There shouldn't be any real surprises here, because it's virtually the same logic found in `GetCommand ()`. Remember that this function must also check for newlines before reading the next character, because the last parameter on the line will not be followed by a space.

GetStringParam ()

Lastly, there's `GetStringParam ()`. At this point, the function's code will almost seem redundant, because it shares so much logic with the last two functions you've looked at. You know the drill; dive right in:

```
void GetStringParam ( char * pstrDestString )
{
    // Keep track of the parameter's length
    int iParamSize = 0;

    // Create a space for the current character
    char cCurrChar;
    // Move past the opening double quote
    ++ g_iCurrScriptLineChar;

    // Read all characters until the closing double quote to isolate
    // the string
    while ( g_iCurrScriptLineChar <
            ( int ) strlen ( g_ppstrScript [ g_iCurrScriptLine ] ) )
    {
        // Read the next character from the line
        cCurrChar = g_ppstrScript
            [ g_iCurrScriptLine ][ g_iCurrScriptLineChar ];

        // If a double quote (or newline) has been read, the command
        // is complete
        if ( cCurrChar == '"' || cCurrChar == '\n' )
            break;

        // Otherwise, append it to the current command
        pstrDestString [ iParamSize ] = cCurrChar;

        // Increment the length of the command
        ++ iParamSize;

        // Move to the next character in the current line
        ++ g_iCurrScriptLineChar;
    }

    // Skip the trailing space and double quote
    g_iCurrScriptLineChar += 2;

    // Append a null terminator
    pstrDestString [ iParamSize ] = '\0';
}
```

As usual, it extracts the parameter's substring. However, there are a few subtle differences in the way this function works that are important to recognize. First of all, remember that a string parameter's final value is the version of the string without the double-quotes, as the parameter appears in the script. Rather than read the entire double-quote delimited string from the script and then attempt to perform some sort of physical processing to remove the quotes, the function just works around them entirely. Before entering the substring extraction loop, it increments g_iCurrScriptLineChar to avoid the first quote. It then runs until the next quote is found, without including it. This is why it's very important to note that GetStringParam () reads characters until a quote or newline character is encountered, rather than a space or newline, as the last two functions did.

Lastly, the function increments g_iCurrScriptLineChar by two. This is because, at the moment when the substring extraction loop has terminated, the character index will point directly to the string's closing double-quote character. This closing quote, as well as the space immediately following it, are both skipped by incrementing g_iCurrScriptLineChar by two, which once again sets things up nicely for the next call to a parameter-extracting function.

TIP

You may have noticed that each of these three functions share a main loop that is virtually identical. I did this purposely to help illustrate their individual functionality more clearly, but in practice, I suggest you base all three functions on a more basic function that simply extracts a substring starting from the current position of g_iCurrScriptLineChar until a space, double-quote, or newline is found. This function could then be used as a generic starting point for extracting commands and both types of parameters, saving you from the perils of such otherwise redundant code.

The Command Handlers

At this point, you've learned about every major aspect of the scripting system. You can load and unload scripts, run them, and manage the extraction and processing of each command and its parameters. At this point, you have everything you need to implement the commands themselves, and thus complete your first implementation of a command-based language.

With only four commands, and such simplistic ones at that, you'd be right in assuming that this is probably the easiest part of all. Let's take a look at the code first:

```
// PrintString
if ( stricmp ( pstrCommand, COMMAND_PRINTSTRING ) == 0 )
{
    // Get the string
    GetStringParam ( pstrStringParam );

    // Print the string
    printf ( "\t%s\n", pstrStringParam );
}
```

```
// PrintStringLoop

else if ( stricmp ( pstrCommand, COMMAND_PRINTSTRINGLOOP ) == 0 )
{
    // Get the string
    GetStringParam ( pstrStringParam );

    // Get the loop count
    int iLoopCount = GetIntParam ();

    // Print the string the specified number of times
    for ( int iCurrString = 0; iCurrString < iLoopCount; ++ iCurrString )
        printf ( "\t%d: %s\n", iCurrString, pstrStringParam );
}

// Newline
else if ( stricmp ( pstrCommand, COMMAND_NEWLINE ) == 0 )
{
    // Print a newline
    printf ( "\n" );
}

// WaitForKeyPress
else if ( stricmp ( pstrCommand, COMMAND_WAITFORKEYPRESS ) == 0 )
{
    // Suspend execution until a key is pressed
    while ( kbhit () )
        getch ();
    while ( ! kbhit () );
}
```

Just as you expected, right? `PrintString` is implemented by passing the specified string to `printf ()`. `PrintStringLoop` does the same thing, except it does so inside a `for` loop that runs until the specified integer parameter is reached. `Newline` is yet another example of a `printf ()`-based command, and `WaitForKeyPress` just enters an empty loop that checks the status of `kbhit ()` at each iteration. By the way, the two lines prior to this loop, as follows,

```
while ( kbhit () )
    getch ();
```

are just used to make sure the keyboard buffer is clear beforehand. Also, just to make things a bit more interesting, `PrintStringLoop` prints each string after a tab and a number that marks where it is in the loop.

Figure 3.6 illustrates this general process of the script controlling the text console.

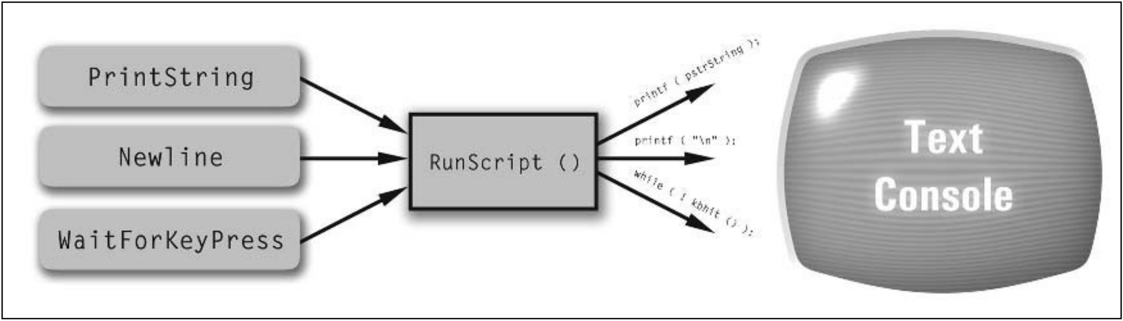


Figure 3.6

The process of commands in a script making their way to the text console.

Now, at long last, here's the mind-blowing output of the script. It's clearly the edge-of-your-seat thrill ride of the summer:

```
This is a command-based language.
Therefore, this is a command-based script.

...and it's really quite boring.

    0: This string has been repeated four times.
    1: This string has been repeated four times.
    2: This string has been repeated four times.
    3: This string has been repeated four times.

Okay, press a key already and put us both out of our misery.
The next demo is cooler, I swear.
```

Granted, slapping some strings of text onto the screen isn't exactly revolutionary, but it's a working basis for command-based scripts and can be almost *immediately* put to use in more exciting demos and applications. Hopefully, however, this section has taught you that even in the case of very simple scripting, there are a lot of details to consider.

Before moving on, there's an important lesson to be learned here about command-based languages. Because these languages consist entirely of domain-specific commands, the actual body of `RunScript ()` has to change almost entirely from project to project. Otherwise, the existing command handlers will almost invariably have to be removed entirely and replaced with new ones. This is one of the more severe downsides of command-based scripting. Although the script loading and unloading interface remains the same, as well as the helper functions like `GetCommand ()`, `GetStringParam ()`, and `GetIntParam ()`, the real guts of the system— the command handlers— are unfortunately rarely reusable.

Scripting a Game Intro Sequence

You'll now apply your newfound skills to something a bit flashier. One great application of command-based scripting is static game sequences, like cinematic cut scenes, or a game's intro. Game intros generally follow a basic pattern, wherein various copyright info and credits screens are displayed, followed by some sort of a title screen. These various screens are also generally linked together with transitions of some sort.

This will be the premise behind this next example of command-based scripting. I've prepared the graphics and some very basic transition code to be used in a simple game intro sequence you'll write a script to control. Figure 3.7 displays the general sequence of the intro as I've planned it:

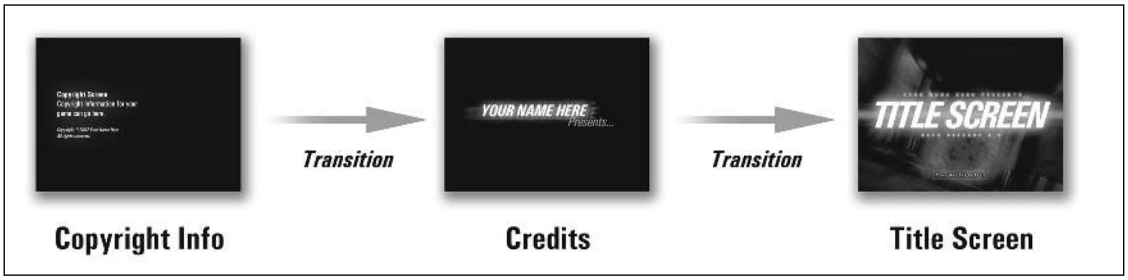


Figure 3.7

The intro sequence will be composed of three full-screen images, each of which is separated by a transition.

First a copyright screen is displayed, followed by a credits screen, followed by the game's title screen. To go from one screen to the next, I've chosen one of the simplest visual transitions I could think of. It's sort of a "double wipe," or "fold" as I call it, wherein either the two horizontal or vertical edges of the screen move inward, covering the image with two expanding black borders until the entire screen is cleared. Figure 3.8 illustrates how both of these work.

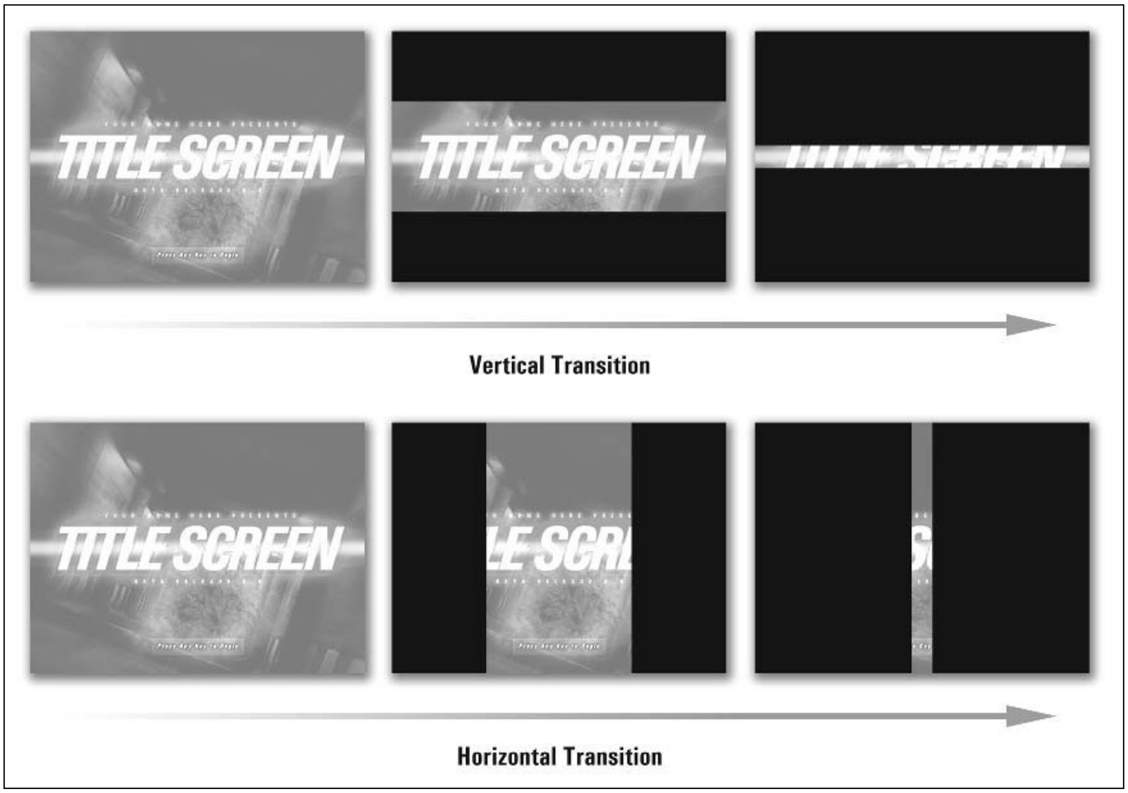


Figure 3.8

Horizontal and vertical folding transitions. Simple but effective.

The Language

In addition to displaying these images and performing transitions, the intro program plays sounds as well. Table 3.3 lists each of the commands the language will offer to facilitate everything you need.

I just added an `Exit` command on a whim here; it doesn't really serve a direct purpose because the script will end anyway upon the execution of the file line. You'll also notice the addition of `Pause`, which will allow each graphic in the intro to remain on-screen, undisturbed, for a brief period before moving to the next.

Table 3.3 Intro Sequence Commands

Command	Parameters	Description
`DrawBitmap`	`String`	Draws the specified .BMP file on the screen.
`PlaySound`	`String`	Plays the specified .WAV file.
`Pause`	`Integer`	Pauses the intro for the specified duration.
`WaitForKeyPress`	*None*	Pauses the intro until a key is pressed.
`FoldCloseEffectX`	*None*	Performs a horizontal "fold close" effect.
`FoldCloseEffectY`	*None*	Performs a vertical "fold close" effect.
`Exit`	*None*	Causes the program to terminate.

The Script

You know what you want the intro to look like, roughly at least, so you can now write the script:

```
DrawBitmap "gfx/copyright.bmp"
PlaySound "sound/ambient.wav"
Pause 3000
PlaySound "sound/wipe.wav"
FoldCloseEffectY
DrawBitmap "gfx/ynh_presents.bmp"
PlaySound "sound/ambient.wav"
Pause 3000
PlaySound "sound/wipe.wav"
FoldCloseEffectX
DrawBitmap "gfx/title.bmp"
PlaySound "sound/title.wav"
WaitForKeyPress
PlaySound "sound/wipe.wav"
FoldCloseEffectY
Exit
```

If you follow along carefully, you should be able to visualize exactly how it will play out. Each screen is displayed, along with an ambient sound effect of some sort, and allowed to remain on-screen for a few seconds thanks to `Pause`. `FoldCloseEffect` transitions to the next screen, along with a transition sound effect. Finally, the title screen (which plays a different effect) is displayed and remains on-screen until a key is pressed.

It may be simple, but this is the same idea behind just about any game intro sequence. Add some commands for playing .MPEG or .AVI movies instead of displaying bitmaps, and you can easily choreograph pro-quality introductions with nothing more than a command-based language.

The Implementation

The implementation for the commands is by no means advanced, but this is a graphical demo, which ends up making things considerably more complex. All graphics and sound code have been implemented with my simple wrapper API, so the code itself should look more or less self-explanatory.

The real difference, however, is that this program runs alongside a main program loop, which prevents `RunScript ()` from simply running until the script finishes. Because games are generally based around the concept of a main game loop, it's important that `RunScript ()` be redesigned to simply execute one instruction at a time, so that it can be called iteratively rather than once. By executing one instruction per frame, your scripts can effectively run concurrently with your game engine. Figure 3.9 illustrates this concept.

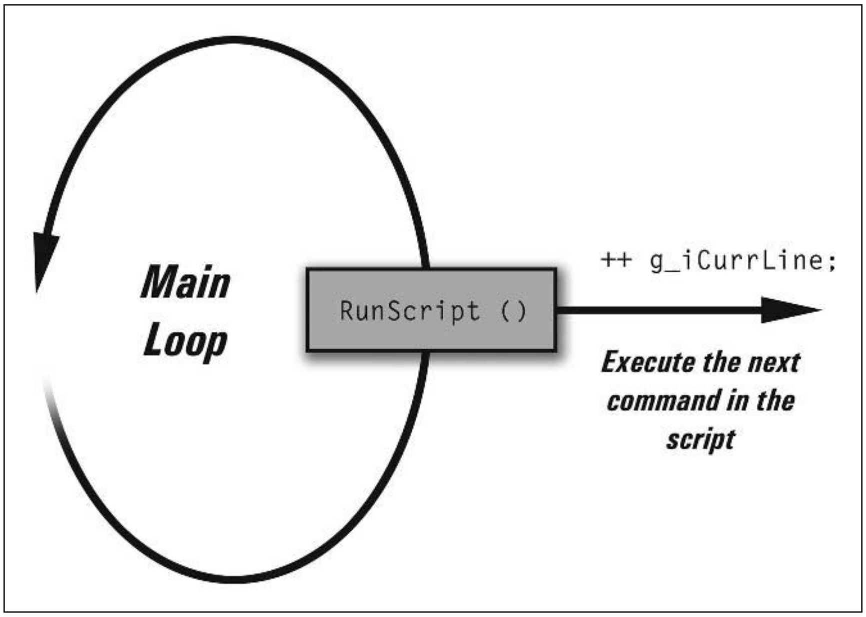


Figure 3.9

Running the script alongside the game engine.

The actual demo code is rather cluttered with calls to my wrapper API, so I've chosen to leave it out here, rather than risk the confusion it might cause. I strongly encourage you to check it out on the CD, however, although you can rest assured that the implementation of each command is simple either way. Here's the code to the new version of `RunScript ()` with the command handlers left out:

```
void RunScript ()
{
    // Make sure we aren't beyond the end of the script
    if ( g_iCurrScriptLine > g_iScriptSize )
       return;

    // Allocate some space for parsing substrings
    char pstrCommand [ MAX_COMMAND_SIZE ];
    char pstrStringParam [ MAX_PARAM_SIZE ];

    // ---- Process the current line

    // Reset the current character
    g_iCurrScriptLineChar = 0;

    // Read the command
    GetCommand ( pstrCommand );

    // ---- Execute the command

    // Move to the next line
    ++ g_iCurrScriptLine;
}
```

As you can see, the `for` loop is gone. Because the function is now only expected to execute one command per call, the function now manually increments the current line before returning, and always checks it against the end of the script just after being called.

Scripting an RPG Character's Behavior

The game intro was an interesting application for command-based scripting, but it's time to set your sights on something a bit more game-like. As you learned in the last chapter, and as was mentioned earlier in this chapter, RPGs have a number of non-player characters, called NPCs, that need to be automated in some way so they appear to move around in a lifelike fashion. This is accomplished, as you might imagine, with scripts. Specifically, however, command-based scripts can be used with great results, because NPCs, at least some of the less pivotal ones, generally move in predictable, static patterns that don't change over time. Figure 3.10 illustrates this.

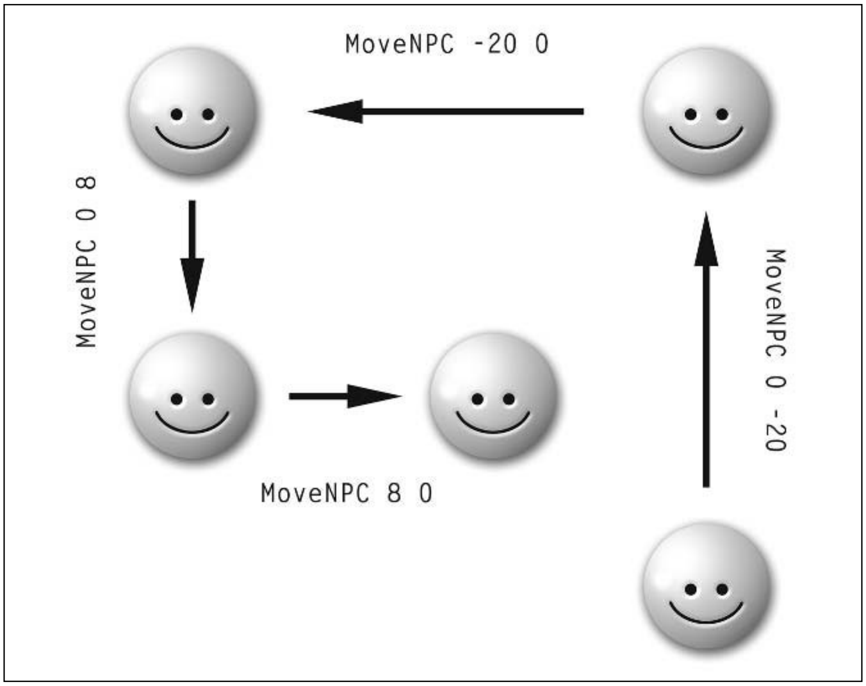


Figure 3.10

NPCs often move in static, unchanging patterns, which naturally lend themselves to command-based scripting.

The Language

This means you can now actually implement a version of the commands listed earlier when discussing RPG scripting. Table 3.4 lists these commands.

Table 3.4 RPG Commands

Command	Parameters	Description
`MoveChar`	`Integer, Integer`	Moves the character the specified X and Y distances.
`SetCharLoc`	`Integer, Integer`	Moves the character to the specified X,Y location.
`SetCharDir`	`String`	Sets the direction the character is facing.
`ShowTextBox`	`String`	Displays the specified string of text in the text box.
`HideTextBox`	*None*	Hides the text box.
`Pause`	`Integer`	Halts the script for the specified duration.

Using these commands, you can move the character around in all directions, change the direction the player's facing, display text in a text box to simulate dialogue, and cause the player to stand still for arbitrary periods. All of these abilities come together to form a lifelike character that seems to be functioning entirely under his or her own control (and in a manner of speaking, actually is).

Improving the Syntax

Before continuing, I should mention a slight alteration I made to the script interpreter used by this demo. Currently, the syntax of this language prevents some of the more helpful aspects of free-form code, like vertical whitespace and comments. These are usually used to help make code more readable and descriptive, but have been unsupported by this system until now.

The addition of both of these syntax features is quite simple. Let's look at an example of a script with both vertical whitespace and a familiar syntax for comments:

```
// Do something
ShowTextBox "This is something."
PlaySound "Explosion.wav"
```

```
// Do something else
ShowTextBox "This is something else."
PlaySound "Buzzer.wav"
```

Much nicer, eh? And all it takes is the following addition to `RunScript ()`, which is added to the beginning of the function just before the command is read with `GetCommand ()`:

```
if ( strlen ( g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ] ) == 0 ||
     ( g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ][ 0 ] == '/' &&
       g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ][ 1 ] == '/' ) )
{
    // Move to the next line
    ++ g_NPC.iCurrScriptLine;

    // Exit the function
    return;
}
```

First, the length of the line is checked. If it's zero, meaning it's an empty string, you know you're dealing with vertical whitespace and can move on. The first two characters are then checked, to determine whether they're both slashes. If so, you're on a comment line. In both cases, the current line is incremented and the function returns.

Managing a Game Character

The last thing you need to worry about before moving on to the script is how the NPC will be stored internally. Now obviously, because this is only a demo as opposed to a full game, all you really need is the bare minimum.

Because the extent of this language's control of the NPC is really just moving him around, all his internal structure needs to represent is his current location. Of course, you also need to know what direction he's facing, so add that to the list as well. That's not everything though, because there's the issue of *how* he'll move exactly.

The `MoveChar` command moves the character in pixel increments, but you certainly don't want the NPC to simply disappear at one X, Y location and appear at another. Rather, he should smoothly "walk" from his current location to the specified destination, pixel by pixel. The only problem is that `RunScripts ()` can't simply enter a loop to move the character then and there, because it would cause the rest of the game loop to stall until the loop completed. This wouldn't matter much in the demo, but it would ruin a real game—imagine the sheer un-playability of a game in which every NPC's movement caused the rest of the game loop to freeze.

So, you'll instead give the NPC two fields within his structure that define his current movement along the X and Y movements. For example, if you want the NPC to move north 20 pixels, you set his Y-movement to 20. At each iteration of the game loop, the NPC's Y-movement would be evaluated. If it was greater than zero, he would move up one pixel, and the Y-movement field would be decremented. This would allow the character to move in any direction, for any distance, without losing sync with the rest of the game loop.

So, with all of that out of the way, take a look at the structure.

```
typedef struct _NPC
{
    // Character

    int iDir;                    // The direction the character is
                                 // facing
    int iX,                      // X location
        iY;                      // Y location
    int iMoveX,                  // X-axis movement
        iMoveY;                  // Y-axis movement

    // Script

    char ** ppstrScript;         // Pointer to the current script
    int iScriptSize;             // The size of the current script
    int iCurrScriptLine;         // The current line in the script
    int iCurrScriptLineChar;     // The current character in the current
                                 // line

    int iIsPaused;               // Is the script currently paused?
    unsigned int iPauseEndTime;  // If so, when will it elapse?
}
    NPC;
```

Wait a sec, what's with the stuff under the `// Script` comment? I've decided to directly include the NPC's script within its structure. This is a bit more reflective of how an actual game implementation would work, because in an environment where 200 NPCs are active at one time, it helps to make each individual character as self-contained as possible. This way, the script is directly bound to the NPC himself. Also, you'll notice the `iIsPaused` and `iPauseEndTime` fields. `iIsPaused` is a flag that determines whether the script is currently paused, and `iPauseEndTime` is the time, expressed in milliseconds, at which the script will become active again. Again, because the script must remain synchronous with the game loop, the `Pause` command can't simply enter an empty loop within `RunScript ()` until the duration elapses. Rather, `RunScript ()`will check the script's pause status and end times each time it's called. This way, the script can pause arbitrarily without stalling the rest of the game loop.

The Script

The script for the character is pretty straightforward, but is considerably longer than anything you've seen before, and is the first to use lines that consist of comments or vertical whitespace. Take a look:

```
// RPG NPC Script
// A Command-Based Language Demo
// Written by Alex Varanese

// ---- Backing up
ShowTextBox "WELCOME TO THIS DEMO."
Pause 2400
ShowTextBox "THIS DEMO WILL CONTROL THE ONSCREEN NPC."
Pause 2400
ShowTextBox "LET'S START BY BACKING UP SLOWLY..."
Pause 2400
HideTextBox
Pause 800
MoveChar 0 -48
Pause 800

// ---- Walking in a square pattern
ShowTextBox "THAT WAS SIMPLE ENOUGH."
Pause 2400
ShowTextBox "NOW LET'S WALK IN A SQUARE PATTERN."
Pause 2400
HideTextBox
Pause 800
SetCharDir "Right"
MoveChar 40 0
MoveChar 8 8
SetCharDir "Down"
MoveChar 0 80
MoveChar -8 8
```

```
SetCharDir "Left"
MoveChar -80 0
MoveChar -8 -8
SetCharDir "Up"
MoveChar 0 -80
MoveChar 8 -8
SetCharDir "Right"
MoveChar 40 0
Pause 800

// Random movement with text box
ShowTextBox "WE CAN EVEN MOVE AROUND WITH THE TEXT BOX ACTIVE!"
Pause 2400
ShowTextBox "WHEEEEEEEEEEE!!!"
Pause 800
SetCharDir "Down"
MoveChar 12, 38
SetCharDir "Left"
MoveChar -40, 10
SetCharDir "Up"
MoveChar 7, 0
SetCharDir "Right"
MoveChar -28, -9
MoveChar 12, -8
SetCharDir "Down"
MoveChar 4, 37

MoveChar 12, 4

// Transition back to the start of the demo
ShowTextBox "THIS DEMO WILL RESTART MOMENTARILY..."
Pause 2400
SetCharLoc 296 208
SetCharDir "Down"
```

Who says command-based scripts can't be complex, huh? As you'll see in the demo included on the CD, this little guy is capable of quite a bit. You can find the scripted RPG NPC demo on the CD in the Programs/Chapter 3/Scripted RPG NPC/ folder.

The Implementation

The demo requires two major resources to run—the castle background image and the NPCs animation frames. Figure 3.11 displays some of these.

These of course come together to form a basic but convincing scene, as shown in Figure 3.12.

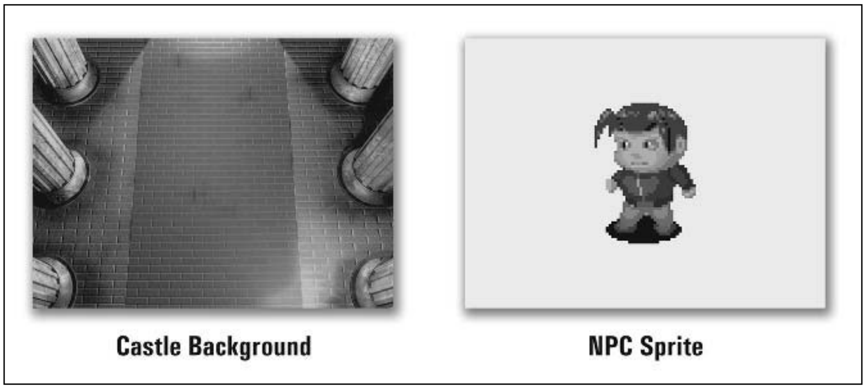


Figure 3.11

Resources used by the NPC demo.

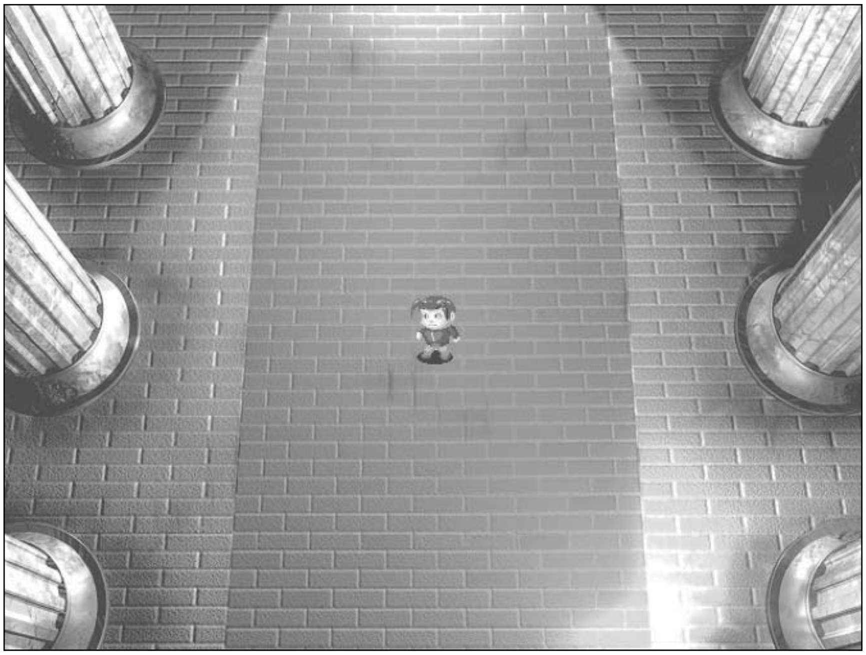


Figure 3.12

The running NPC demo.

Of course, the real changes lie in `RunScript ()`. In addition to the new command handlers, which should be pretty much no-brainers, there are some other general changes as well. Here's the function, with the command handlers this time (notice I left them in this time because the graphics-intensive code has been offloaded to the main loop):

```
void RunScript ()
{
    // Only perform the next line of code if the player has stopped moving
    if ( g_NPC.iMoveX || g_NPC.iMoveY )
        return;

    // Return if the script is currently paused
    if ( g_NPC.iIsPaused )
        if ( W_GetTickCount () > g_NPC.iPauseEndTime )
            g_NPC.iIsPaused = TRUE;
        else
            return;

    // If the script is finished, loop back to the start
    if ( g_NPC.iCurrScriptLine >= g_NPC.iScriptSize )
        g_NPC.iCurrScriptLine = 0;

    // Allocate some space for parsing substrings
    char pstrCommand [ MAX_COMMAND_SIZE ];
    char pstrStringParam [ MAX_PARAM_SIZE ];

    // ---- Process the current line

    // Skip it if it's whitespace or a comment
    if ( strlen ( g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ] ) == 0 ||
         ( g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ][ 0 ] == '/' &&
           g_NPC.ppstrScript [ g_NPC.iCurrScriptLine ][ 1 ] == '/' ) )
    {
        // Move to the next line
        ++ g_NPC.iCurrScriptLine;

        // Exit the function
        return;
    }
```

```
// Reset the current character
g_NPC.iCurrScriptLineChar = 0;

// Read the command
GetCommand ( pstrCommand );

// ---- Execute the command

// MoveChar
if ( stricmp ( pstrCommand, COMMAND_MOVECHAR ) == 0 )
{
    // Move the player to the specified X, Y location

    g_NPC.iMoveX = GetIntParam ();
    g_NPC.iMoveY = GetIntParam ();
}

// SetCharLoc
if ( stricmp ( pstrCommand, COMMAND_SETCHARLOC ) == 0 )
{
    // Read the specified X, Y target location
    int iX = GetIntParam (),
        iY = GetIntParam ();

    // Calculate the distance to this location
    int iXDist = iX - g_NPC.iX,
        iYDist = iY - g_NPC.iY;

    // Set the player along this path
    g_NPC.iMoveX = iXDist;
    g_NPC.iMoveY = iYDist;
}

// SetCharDir
else if ( stricmp ( pstrCommand, COMMAND_SETCHARDIR ) == 0 )
{
    // Read a single string parameter, which is the direction
    // the character should face
    GetStringParam ( pstrStringParam );
```

```
        if ( stricmp ( pstrStringParam, "Up" ) == 0 )
            g_NPC.iDir  = UP;
        if ( stricmp ( pstrStringParam, "Down" ) == 0 )
            g_NPC.iDir  = DOWN;
        if ( stricmp ( pstrStringParam, "Left" ) == 0 )
            g_NPC.iDir  = LEFT;
        if ( stricmp ( pstrStringParam, "Right" ) == 0 )
            g_NPC.iDir  = RIGHT;
    }

    // ShowTextBox
    else if ( stricmp ( pstrCommand, COMMAND_SHOWTEXTBOX ) == 0 )
    {
        // Read the string and copy it into the text box message
        GetStringParam ( pstrStringParam );
        strcpy ( g_pstrTextBoxMssg, pstrStringParam );

        // Activate the text box
        g_iIsTextBoxActive = TRUE;
    }

    // HideTextBox
    else if ( stricmp ( pstrCommand, COMMAND_HIDETEXTBOX ) == 0 )
    {
        // Deactivate the text box
        g_iIsTextBoxActive = FALSE;
    }

    // Pause

    else if ( stricmp ( pstrCommand, COMMAND_PAUSE ) == 0 )
    {
        // Read a single integer parameter for the duration
        int iPauseDur = GetIntParam ();

        // Calculate the pause end time
        unsigned int iPauseEndTime = W_GetTickCount () + iPauseDur;
```

```
            // Activate the pause
            g_NPC.iIsPaused = TRUE;
            g_NPC.iPauseEndTime = iPauseEndTime;
        }

        // Move to the next line
        ++ g_NPC.iCurrScriptLine;
    }
```

The function begins by checking the NPC's X and Y movement. If he's currently in motion, the function returns without evaluating the line or incrementing the line counter. This allows the character to complete his current task without the rest of the script getting out of sync. The status of the script's pause flag is then determined. If the script is currently paused, the end time is compared to the current time to determine whether it's time to activate again. If so, the script is activated and the next line is executed. Otherwise, the function returns. The current line is then compared to the last line in the script, and is looped back to zero if necessary. This allows the NPC to continue his behavior until the user ends the demo.

The typical script-handling logic is up next, along with the newly added code for handling vertical whitespace and comments. The actual command-handlers should be pretty self-explanatory. Commands for NPC movement set the movement fields with the appropriate values, the direction-setting command sets the NPC's `iDir` field, and so on. Notice, however, that the commands for hiding and showing the text box don't actually blit the text box graphic to the screen or print the string. Rather, they simply set a global flag called `g_iIsTextBoxActive` to `TRUE` or `FALSE`, and copy the specified string parameter into a global string called `g_pstrTextBoxMssg` (in the case of `ShowTextBox`, that is). This is because the game loop is solely responsible for managing the demo's visuals. All `RunScript ()` cares about is setting the proper flags, resting assured that the next iteration of the main loop will immediately translate those flag updates to the screen. The next section, then, discusses how this loop works.

The Demo's Main Loop

It's generally good practice to design the main loop of your game in such a way that it's primarily responsible for the physical output of graphics and sound. That way, the actual game logic (which will presumably be carried out by separate functions) can focus on flags and other global variables that only indirectly control such things.

This demo does exactly that. At each frame, it does a number of things:

- Calls `RunScript ()` to execute the next line of code in the NPC's script.
- Draws the background image of the castle hall.

- Updates the current frame of animation, so the character always appears to be walking (even when he's standing still, heh).
- Sets the direction the character is facing, in case it was changed within the last frame by `RunScript ()`.
- Blits the appropriate character animation sprite based on the direction he's facing and the current frame.
- Draws the text box if it's currently active, as well as the current text box message (which is centered within the box).
- Blits the entire completed frame to the screen.
- Moves the character along his current path, assuming he's in motion.
- Checks the status of the keyboard and exits if a key has been pressed.

Just to bring it all home, here's the inner-most code from the game's main loop. Try to follow along, keeping the previous bulleted list in mind:

```
// Execute the next command
RunScript ();

// Draw the background
W_BlitImage ( g_hBG, 0, 0 );

// Update the animation frame if necessary
if ( W_GetTimerState ( g_hAnimTimer ) )
    if ( iCurrAnimFrame )
        iCurrAnimFrame = 0;
    else
        iCurrAnimFrame = 1;

// Draw the character depending on the direction he's facing
switch ( g_NPC.iDir )
{
    case UP:
        if ( iCurrAnimFrame )
            phCurrFrame = & g_hCharUp0;
        else
            phCurrFrame = & g_hCharUp1;
        break;

    case DOWN:
        if ( iCurrAnimFrame )
            phCurrFrame = & g_hCharDown0;
        else
            phCurrFrame = & g_hCharDown1;
        break;

    case LEFT:
        if ( iCurrAnimFrame )
            phCurrFrame = & g_hCharLeft0;
        else
            phCurrFrame = & g_hCharLeft1;
        break;

    case RIGHT:
        if ( iCurrAnimFrame )
            phCurrFrame = & g_hCharRight0;
        else
            phCurrFrame = & g_hCharRight1;
        break;
}

W_BlitImage ( * phCurrFrame, g_NPC.iX, g_NPC.iY );

// Draw the text box if active
if ( g_iIsTextBoxActive )
{
    // Draw the text box background image
    W_BlitImage ( g_hTextBox, 26, 360 );

    // Determine where the text string should start within the box
    int iX = 319 - ( W_GetStringPixelLength ( g_pstrTextBoxMssg ) / 2 );

    // Draw the string
    W_DrawTextString ( g_pstrTextBoxMssg, iX, 399 );
}

// Blit the framebuffer to the screen
W_BlitFrame ();

// Move the character if necessary
if ( W_GetTimerState ( g_hMoveTimer ) )
{
    // Handle X-axis movement
    if ( g_NPC.iMoveX > 0 )
    {
        ++ g_NPC.iX;
        -- g_NPC.iMoveX;
    }
    if ( g_NPC.iMoveX < 0 )
    {
        -- g_NPC.iX;
        ++ g_NPC.iMoveX;
    }

    // Handle Y-axis movement
    if ( g_NPC.iMoveY > 0 )
    {
        ++ g_NPC.iY;
        -- g_NPC.iMoveY;
    }
    if ( g_NPC.iMoveY < 0 )
    {
        -- g_NPC.iY;
        ++ g_NPC.iMoveY;
    }
}

// If a key was pressed, exit
if ( g_iExitApp || W_GetAnyKeyState () )
    break;
```

So that wraps up the NPC demo. Not bad, eh? Imagine creating an entire town, bustling with the lively actions of tens or even hundreds of NPCs running on command-based scripts. They could carry on conversations when spoken to, walk around and animate on their own, and seem convincingly alive in general. That does bring up an important issue that hasn't been addressed yet, however—how exactly do you get more than one script running at once?

NOTE

Notice that rather than animate the character only while he's moving, the NPC is constantly in an animated state, even when standing still. I did this as a subtle nod to the old *Dragon Warrior* games for the Nintendo and the Japanese *Super Famicom*, which did the same thing. I find it strangely cute.

Concurrent Script Execution

Unless your game has some sort of *Twilight Zone*-like premise in which your character and one NPC are the only humans left on the planet, you're probably going to want more than one game entity active at once. The problem with this is that so far, this scripting system has been designed with a single script in mind.

Fortunately, command-based scripting is simple enough to make the concurrent execution of multiple scripts yet another reasonably easy addition. The key is noting that the current system executes the next line of the script at each iteration of the main loop. All that's necessary to facilitate the execution of multiple scripts is to execute the next line of *each* of those scripts, in sequence, rather than just one. By altering `RunScripts ()` just slightly to accept an index parameter that tells it which NPC's script to execute, this can be done easily. This is demonstrated in Figure 3.13.

The only major change that needs to be made involves using an array to store NPCs instead of a single global instance of the `NPC` structure. Of course, in order to properly handle the possibility of multiple scripts, each script-related function must be changed to accept a parameter that helps it index the proper script, which means that `LoadScript ()`, `UnloadScript ()`, `RunScript ()`, `GetCommand ()`, `GetIntParam ()`, and `GetStringParam ()` need to be altered to accept such a parameter.

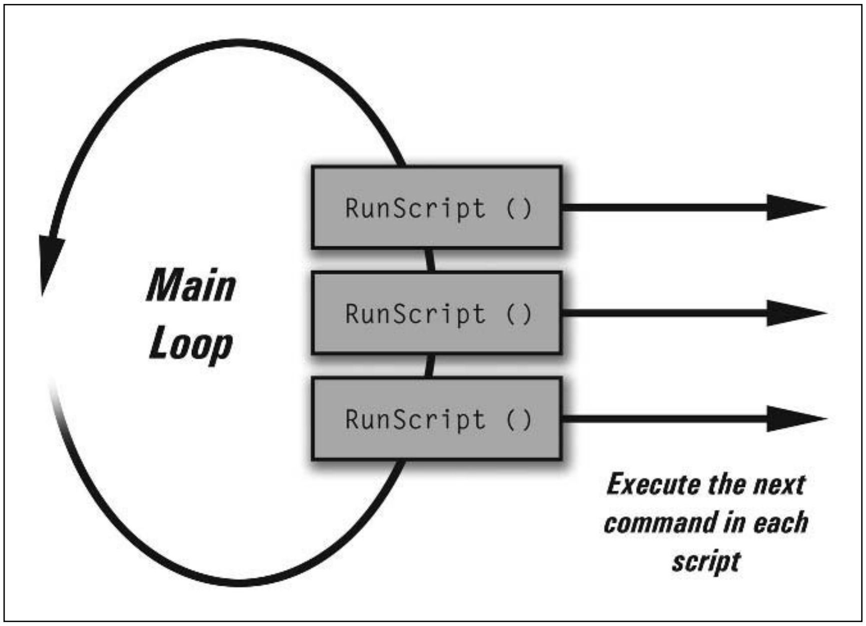


Figure 3.13

Executing a single instruction from each script.

Once these changes have been made (which you can see for yourself on the demo included on the CD), it becomes possible to create any number of NPCs, all of which will seem to move around simultaneously. Check out Figure 3.14.

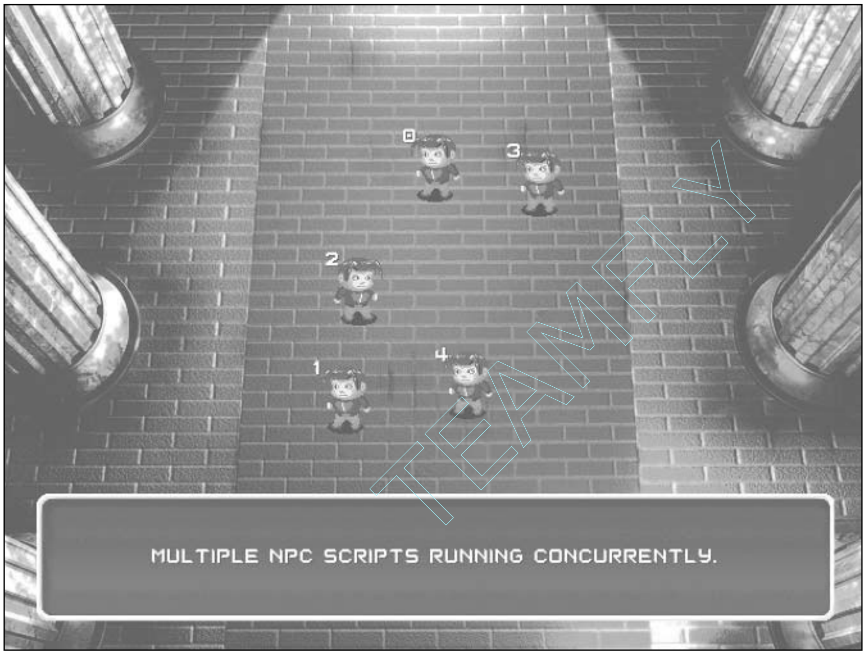


Figure 3.14

The multiple NPC demo.

Summary

You must admit; this is pretty cool. You're only just getting warmed up, and you've already got some basic game scripting going! The last demo even got you as far as the concurrent execution of multiple character scripts, which should definitely help you understand the true potential of command-based scripting. Simplistic or not, command-based scripts can pack enough power to bring moderately detailed game worlds to life.

In the next chapter, you're going to cover a lot of ground as you take a mainly theoretical tour of the countless improvements that can be made on the scripting system built in this chapter. Along the way, the fundamental concepts presented will form a foundation for the more advanced material covered in the book's later chapters, which means that the next chapter is an important one.

Overall, command-based languages are a lot of fun to play with. They can be implemented extremely quickly, and once up and running, can be used to solve a reasonable amount of basic scripting problems. After the next chapter, you'll have command-based languages behind you and can move on to designing and implementing a C-style language and truly becoming a game scripting master.

How much harder can it be, right?

On the CD

The CD contains the four demos created in this chapter, available in both source and executable form. All demos except the first, the console text output demo, require a Win32/DirectX platform to run and therefore must be compiled as such. Check out the `Read Me!.txt` file in their respective directories for compilation information.

The demos for this chapter can be found on the accompanying CD-ROM in `Programs/Chapter 3/`. The following is a breakdown of this folder's contents:

- **Console CBL Demo**/. A simple demo that demonstrates the functionality of a command-based scripting language by printing text to the console.
- **Scripted Intro**/. This demo makes things a bit more interesting by applying a command-based language to the scripting of a game intro sequence.
- **Scripted RPG NPC**/. In our first taste of the scripting of dynamic game entities, this next demo uses a command-based script to control the movement of a role playing game (RPG) non-player character (NPC).
- **Multiple NPCs**/. The chapter's final demo builds on the last by introducing an entire group of concurrently moving NPCs that seem to function entirely in parallel.

Each demo comes in both source and executable forms, in appropriately named `Source/` and `Executable/` directories. I recommend starting with the executables, as they can be tested right away to get a quick idea of what's going on.

Challenges

- *Easy:* Add and implement new commands for controlling the characters in the RPG NPC demos.
- *Intermediate:* Rework the script interpreter so it can handle whitespace more flexibly. Try allowing commands and parameters to be separated from one another by any arbitrary amount of spaces and tabs, in turn enabling you to be more free-form about your code.

- *Intermediate:* Add escape sequences that allow the double-quote symbol (") to appear within string literals without messing up the interpreter. Naturally, this can be important when scripting dialogue sequences.
- *Difficult:* Implement anything from the next chapter (after reading it, of course).

CHAPTER 4

Advanced Command-Based Scripting

"We gotta take it up a notch or shut it down for good."
——*Tyler Durden,* Fight Club

The last chapter introduced command-based scripting, and was a gentle introduction to the process of writing code in a custom-designed language and executing it from within the game engine. Although this form of scripting is among the simplest possible solutions, it has proven quite capable of handling basic scripting problems, like the details of a game's intro sequence or the autonomous behavior of non-player characters.

Ultimately, you need to write scripts in a C/C++-style language featuring everything you are used to as a programmer, including variables, arrays, loops, conditional logic, and functions. In addition, it would be nice to be able to compile this code down to a lower-level format that is not only faster to execute within the game engine, but much safer from the prying eyes of malicious gamers who would otherwise hack and possibly even break the game's scripts. You'll get there soon enough, but you don't have to abandon command-based languages entirely. You can still improve the system considerably, perhaps even to the point that it remains useful for certain specialized tasks regardless of how powerful other scripting solutions may be.

This chapter discusses topics that bring the simple command-based language closer and closer to the high-level procedural languages you're used to coding in. Although the language won't attain such flexibility and power entirely, along the way you'll be introduced to many of the concepts that will form the groundwork for the more advanced material presented later in the book. For this reason, I strongly suggest you read this chapter carefully. Even if you think command-based scripting is a joke, you'll still learn a lot about general scripting concepts and issues here.

This chapter is largely theoretical, introducing you to the concepts and basic implementation details of some advanced command-based language enhancements. The final implementation of these concepts isn't covered here , because most of it will intrude on the material presented by later chapters and disrupt the flow of the book. Fortunately, most of what's discussed here should be easy to get working for at least intermediate-level coders, so you're encouraged to give it a shot on your own. Anything that doesn't make sense now, however, will certainly become clear as you progress through the rest of the book.

In this chapter, you're going to learn about

- New data types
- Symbolic constants
- Simple iterative and conditional logic
- Event-based scripting
- Compiling command-based scripts to a binary format
- Basic script preprocessing

New Data Types

The current command-based scripting system is decidedly simple in its support for data types. Parameters can be integers or strings, with no real middle ground. You can simulate symbolic constants in a brute-force sort of manner using descriptive string literals, like `"Up"` and `"Down"`, for example, but this is obviously a messy way to solve the problem.

Furthermore, any sort of 3D game is going to need floating-point support; moving characters around in a top-down 2D game engine is one thing, because screen coordinates map directly to integers. 3D space, however, is generally independent of any specific resolution (within reason) and as such, needs floating-point precision to prevent character movements from being jerky and erratic.

Boolean Constants

Before moving into general-purpose symbolic constants, you can start small by adding a built-in Boolean data type. Boolean data, of course, is always either true or false, which means the addition of such a type is a simple matter of creating a new function, perhaps called `GetBoolParam ()`, that returns 1 or 0 if the parameter string it extracts is equal to `TRUE` or `FALSE`, respectively. This doesn't require any major additions to syntax, minus the two keywords, and is a fast-and-easy improvement that prevents you from having to use `1` or `0` or string literals. Figure 4.1 illustrates this concept.

TIP

Unless you like the idea of making an explicit separation between integer and Boolean parameters (which is understandable), there's an even easier way to support Booleans without making a significant change to your existing code base. Rather than writing a separate function called `GetBoolParam ()`, you can just rewrite `GetIntParam ()` to automatically detect the `TRUE` and `FALSE` keywords, and return 1 or 0 to the caller. This would allow your existing commands to keep functioning the way they do, and make the addition of such keywords virtually transparent to the rest of the system.

Floating-Point Support

Floating-point support is, fortunately, extremely easy to add. All it really comes down to is a function just like `GetIntParam ()`, called `GetFloatParam ()`, which passes the extracted parameter string to `atof ()` instead of `atoi ()`. This function converts a string to a floating-point value automatically, immediately making floating-point parameters possible. Check out Figure 4.2.

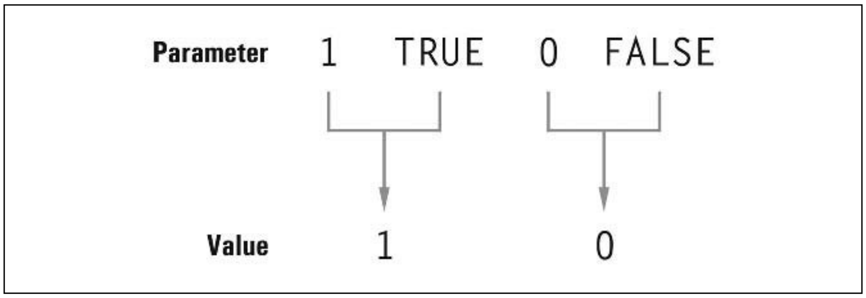


Figure 4.1

The Boolean `TRUE` and `FALSE` keywords map directly to integer values 1 and 0.

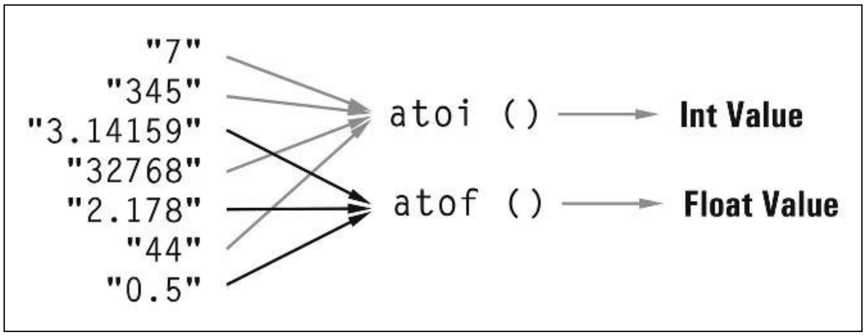


Figure 4.2

Routing the parameter string to the proper numeric-conversion function allows floating-point and integer data to be supported.

General-Purpose Symbolic Constants

Having built-in `TRUE` and `FALSE` constants is great, but there will be times when an enumeration of arbitrary symbolic constants will be necessary. You've already seen an example of this in the last chapter, when you were forced to use the string literal values `"Up"`, `"Down"`, `"Left"`, and `"Right"` to represent the cardinal directions. It would be much cleaner to be able to define constants `UP`, `DOWN`, `LEFT`, and `RIGHT` as symbols that mapped to the integer values 0-3 (or any four unique integer values, for that matter).

Interpreting these constants as parameters is very simple—you've already seen how this works with the `GetBoolParam ()` function proposed in the last section. The problem, however, is the actual mapping of the constant identifier to its value. Much like higher-level languages like C/C++, you need to define a constant's value if you want it to actually mean anything to the runtime interpreter.

A clean and simple solution is to define a new command called `DefConst` (Define Constant) that accepts two parameters—a constant identifier and an integer value. When this command is executed, the interpreter will make a record of the constant name and value, and use the value in place of any reference to the name it finds in subsequent commands. `DefConst` is a special command in that it's not part of any specific domain—any command-based language, whether it's for a puzzle game or a flight simulator, can use it in the same way (as illustrated in Figure 4.3). Here's an example:

```
DefConst UP 0
DefConst DOWN 1
DefConst LEFT 2
DefConst RIGHT 3
```

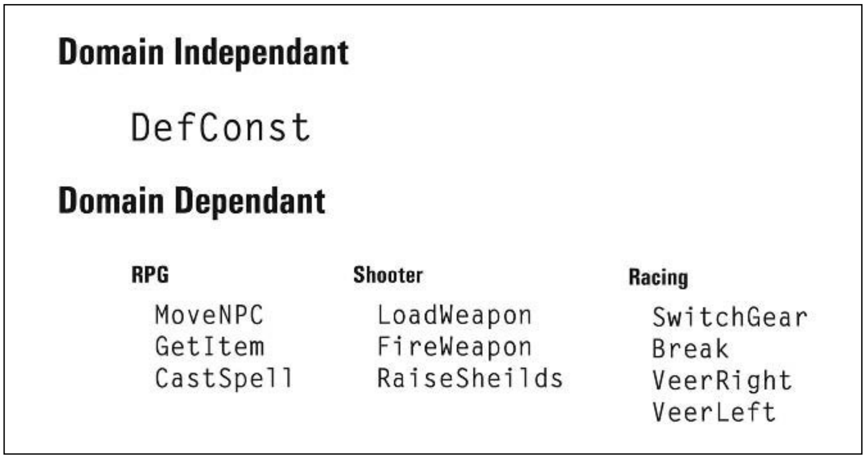


Figure 4.3

`DefConst` is a domain-independent command.

An Internal Constant List

The question is, how does the interpreter "make a record" of the constant? The easiest approach is to implement a simple linked list wherein each node maintains two values—a constant identifier string (like `"UP"`, `"DOWN"`, or `"PLAYER_ANIM_JUMP"`) and an integer value. When a `DefConst` command is executed, the first parameter will contain the constant's identifier, and the second will be its value. A new node is then created in the list and these two pieces of data are saved there. Check out Figure 4.4.

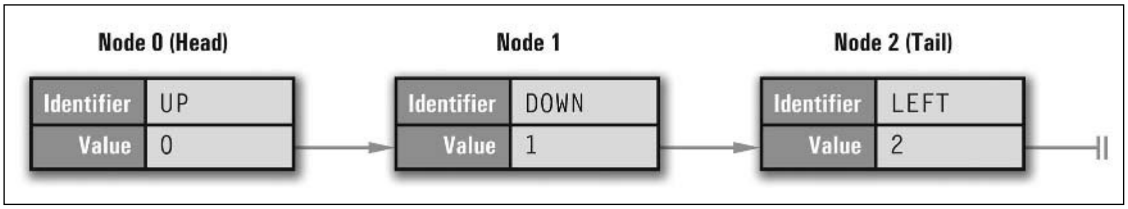


Figure 4.4

A script's constants can be stored in a linked list called the constant list.

From this point on, whenever a command is executed, constants can be accepted in the place of integer parameters. In these cases, the specified identifier is used as a key to search the constant list and find its associated value. In fact, a slick way to add constants to your existing commands without changing them is to simply rewrite `GetIntParam ()` to transparently replace constants with their respective values. Whenever the function reads a new parameter, it determines whether the first letter of the string is a letter or an underscore—because valid identifiers are generally sequences of numbers, letters, and underscores with a leading character that is never a number, this simple test tells you whether you're dealing with a constant. If not, you pass it to `atoi ()` to convert it to an integer just like always. Otherwise, you search the constant list until its matching record is found and return its associated integer value instead. If the constant is not found, the script is referencing an undefined identifier and an error should be reported. This process is illustrated in Figure 4.5.

NOTE

Of course, constants can store more than just integer values. You can probably find uses for both floating-point and string values as well; I'm sticking to integers here, however, because they're simpler. Another reason they're generally more useful than anything else, however, is that the real goal of using this sort of constants isn't so much to represent data symbolically, but rather simulate enumerations. Individual constants like character names aren't as important as groups of constants, wherein the values of the constants don't matter as long as each is unique.

This brings up an important issue, however. The implementation of `DefConst` will have to be more intelligent than simply dumping the specified identifier into the list. One of two cases could prevent the constant from functioning properly and should be checked for before the command executes. First and foremost, the constant's identifier must be valid. Due to the simplistic nature of the language's syntax, this really just means making sure the constant doesn't start with a number. Second, the identifier specified can't already exist in the list. If it does, the script is attempting to redefine an existing constant, which is illegal. Figure 4.6 illustrates the process of adding a new constant to the list.

TIP

Linked lists, although simple to implement, actually aren't the best way to store the constant list. Remember, every time a command executes that specifies a constant for one or more parameters, `GetIntParam ()` has to perform a full search of each node in the list. This can begin to take its toll on the script's performance, as string comparisons aren't exactly the fastest operation in the world and slow down more and more depending on the size of the list. Among the most efficient implementations is using the hash table, which can search huge lists of strings in nearly linear time, making it almost as fast as an array.

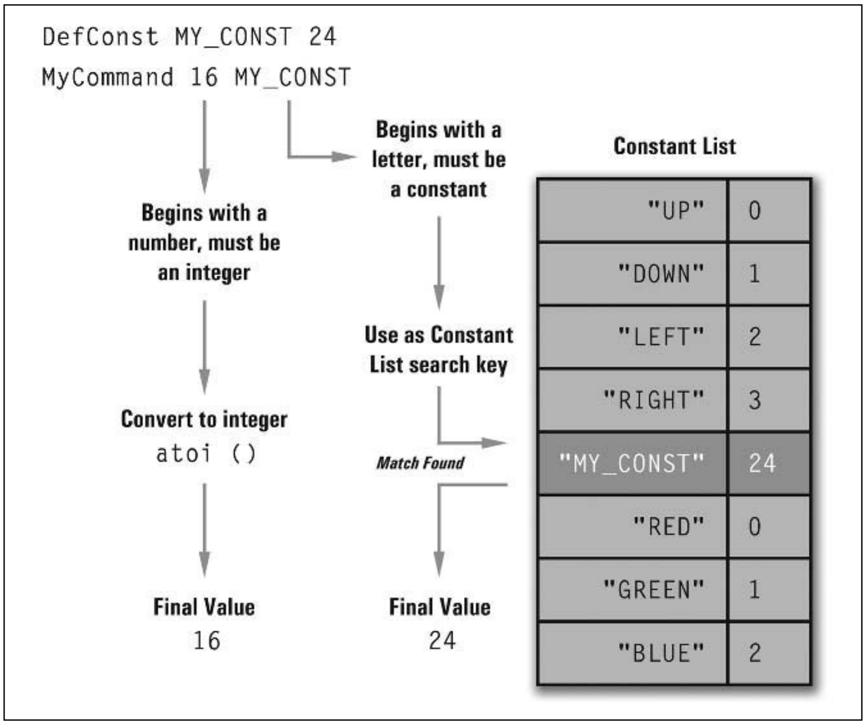


Figure 4.5

Handling constant parameters.

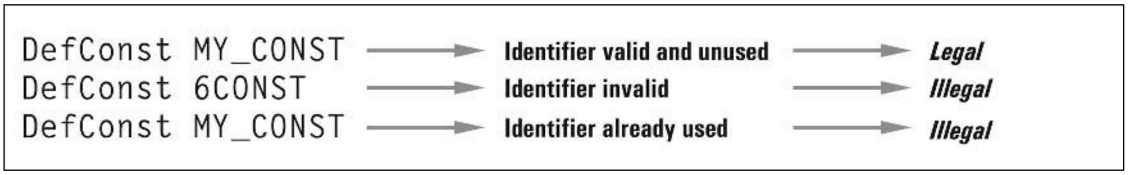

DefConst MY_CONST → Identifier valid and unused → Legal
DefConst 6CONST → Identifier invalid → Illegal
DefConst MY_CONST → Identifier already used → Illegal

Figure 4.6

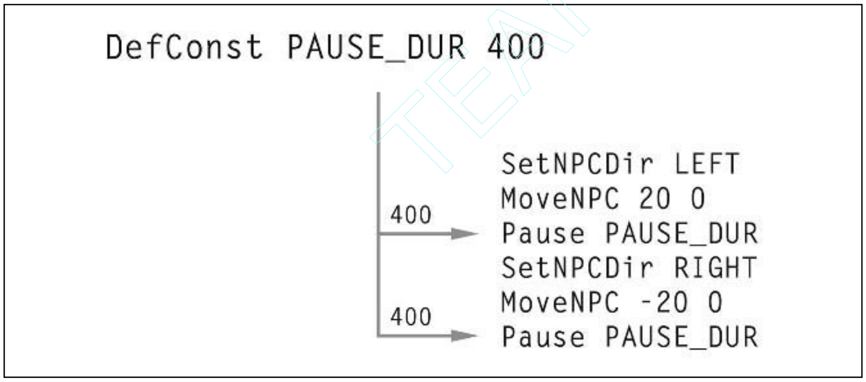
Adding a new constant to the constant list.

So, to summarize, the implementation of constants is twofold. First, `DefConst` must be used to define the constant by assigning it an integer value. This value is added to the constant list and ready to go. Then, `GetIntParam ()` is rewritten to transparently handle constant references, which allows existing commands to keep functioning without even having to know such constants exist. Here's a simple example of using constants:

```
// Define some directional constants
DefConst LEFT 0
DefConst RIGHT 1
DefConst PAUSE_DUR 400
```

```
// Cause an NPC to pace back and forth
SetNPCDir LEFT
MoveNPC 20 0
Pause PAUSE_DUR
SetNPCDir RIGHT
MoveNPC -20 0
Pause PAUSE_DUR
```

Cool, huh? Now the NPC can be moved around using actual directional constants, and the duration at which he rests after each movement can even be stored in a constant. This will come in particularly handy if you want to use the same pause duration everywhere in the script but find yourself constantly tweaking the value. Using a constant allows you to automatically update the duration of every pause using that constant with a single change, as illustrated in Figure 4.7.



```
SetNPCDir LEFT
MoveNPC 20 0
Pause PAUSE_DUR
SetNPCDir RIGHT
MoveNPC -20 0
Pause PAUSE_DUR
```

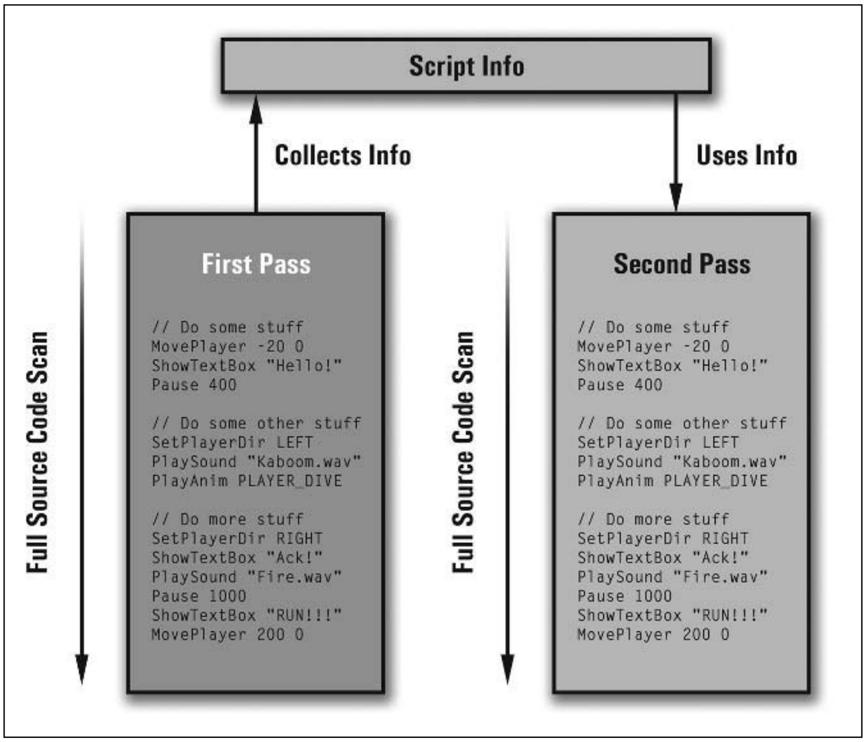
Figure 4.7

Constants allow multiple references to a single value to be changed easily.

A Two-Pass Approach

The approach to implementing the previous constants is simple, straightforward, and robust. There are numerous other ways to achieve the same results, however, some of which provide additional flexibility and functionality. One of these alternatives borrows some of the techniques used to code assemblers and compilers, and involves making two separate passes over the script—the first of which collects information regarding each of its constants, the second of which actually executes the commands. Check out Figure 4.8.

Despite the added complexity, there are definite advantages to this approach. First of all, remember that, as you saw in the last chapter, it's often desirable for scripts to loop indefinitely (or at least more than once). This comes in particularly handy when creating autonomous game entities like the NPCs in Chapter 3's multiple NPC demo. However, this means that all `DefConst` commands will be executed multiple times as well, causing immediate constant redefinition errors.



```
// Do some stuff
MovePlayer -20 0
ShowTextBox "Hello!"
Pause 400

// Do some other stuff
SetPlayerDir LEFT
PlaySound "Kaboom.wav"
PlayAnim PLAYER_DIVE

// Do more stuff
SetPlayerDir RIGHT
ShowTextBox "Ack!"
PlaySound "Fire.wav"
Pause 1000
ShowTextBox "RUN!!!"
MovePlayer 200 0
```

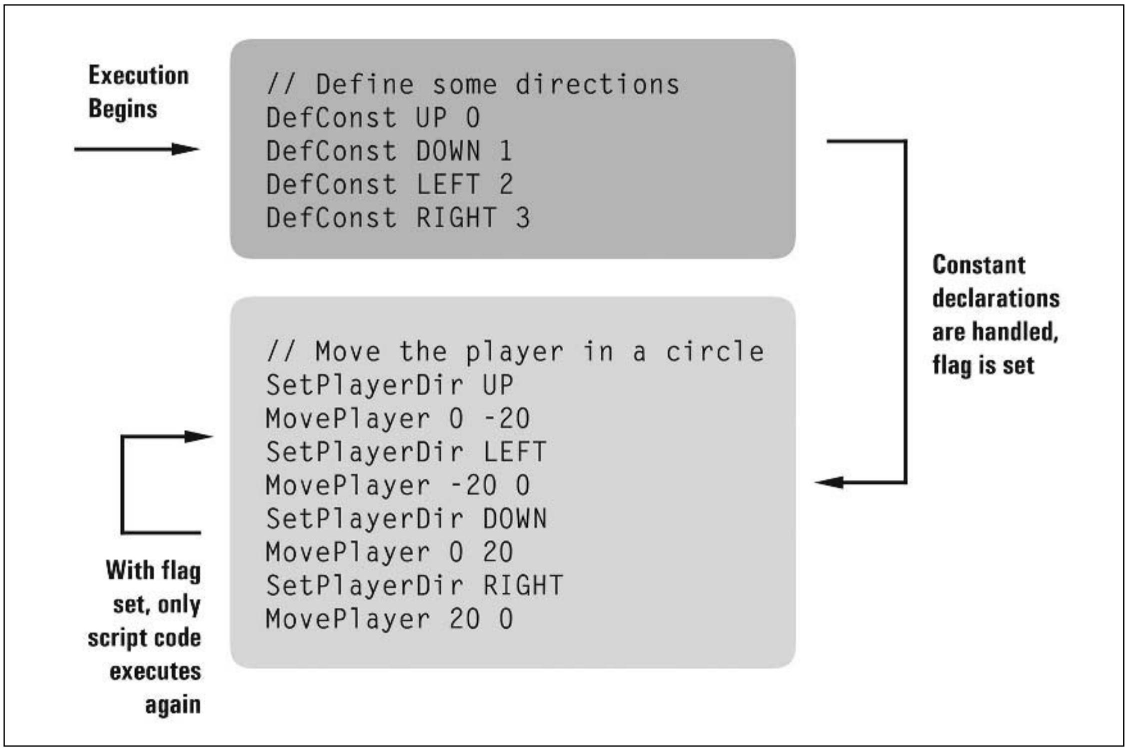
Figure 4.8

In a two-pass interpreter, initial information about the script is assessed in the first pass, whereas the second pass deals with the actual execution.

One easy way around this is to maintain a flag that monitors whether the script is in its first iteration; if so, constant declarations are handled; if not, they're ignored because the constant list has already been built. Check out Figure 4.9.

This is a reasonable solution, and will be necessary if you stick to a single-pass approach. However, the two-pass approach allows you to solve the problem in a more elegant way. Remember, even if the `DefConst` commands are ignored in subsequent iterations of the script, there's still the small overhead of reading each command string from the script buffer and determining whether it's a constant declaration. This in itself takes time, and although individual instances will seem instantaneous, if you have 20 constant declarations per script, and have 50 script-controlled characters running around, you're looking at quite a bit of useless string comparisons.

The two-pass method lets you define your constants ahead of time, and then immediately dispose of all instances of `DefConst` so that they won't bog you down later. Remember, even though this method operates in two passes, the first pass is only performed once—looping the script only means repeating the second pass (execution). If the first pass over the script builds up the constant list by handling each `DefConst` command, there's no need to hold on to the actual code in which these constants are defined any longer. On the most basic level, you can simply free each


Execution Begins
Constant declarations are handled, flag is set
With flag set, only script code executes again

```
// Define some directions
DefConst UP 0
DefConst DOWN 1
DefConst LEFT 2
DefConst RIGHT 3
```

```
// Move the player in a circle
SetPlayerDir UP
MovePlayer 0 -20
SetPlayerDir LEFT
MovePlayer -20 0
SetPlayerDir DOWN
MovePlayer 0 20
SetPlayerDir RIGHT
MovePlayer 20 0
```

Figure 4.9

A flag can be maintained to prevent constant declarations to be executed multiple times.

string in the script array that contains a `DefConst` command, and tell the interpreter to check for and ignore null pointers. Now, the comparison of each line's command to `DefConst` can be eliminated entirely, saving time when large numbers of scripts are running concurrently.

So one benefit of the two-pass approach is that it alleviates a small string comparison overhead. Granted, this is mostly a theoretical advantage, but it's worth

TIP

An even better way to handle the initial disposal of `DefConst` lines from the script is to store the script's code in a linked list, rather than a static array. This way, nodes containing `DefConst` lines can be removed from the list entirely, further saving you from having to check for a null pointer every time a line of code is executed. Because removing a node from a linked list automatically causes the pointers in the previous and next nodes to link directly to each other, the script will execute at maximum speed, completely oblivious to the fact that it contained constant declarations in the first place.

mentioning nonetheless. A real application of two-pass execution, however, is eliminating the idea of constants altogether at runtime.

If you think about it, constants don't provide any additional functionality that wasn't available before as far as actual script execution goes. For example, consider the following script fragment:

```
DefConst MY_CONST 20
MyCommand MY_CONST
```

This could be rewritten in the following manner and have absolutely no impact on the script's ultimate behavior whatsoever:

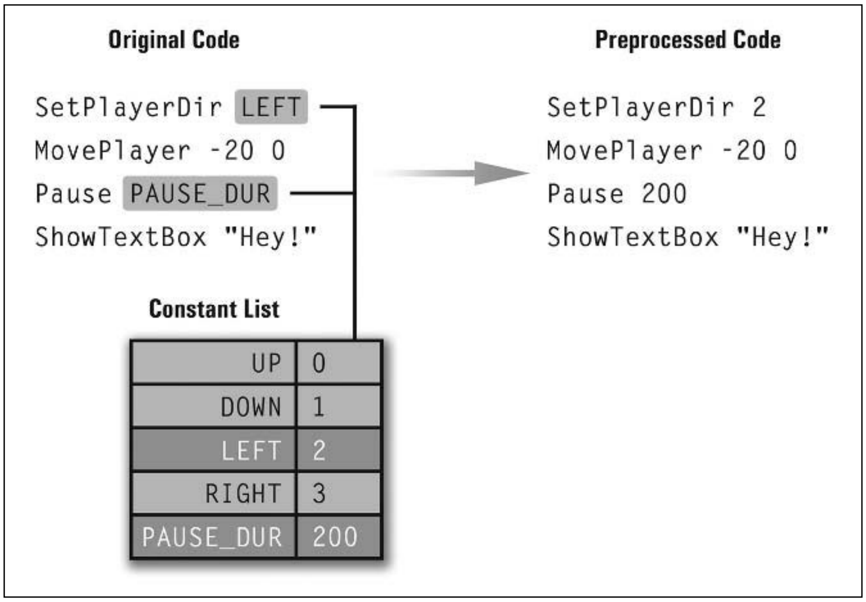
```
MyCommand 20
```

In fact, the previous line of code would run faster, because the `DefConst` line would never have to be executed and the constant list would never have to be searched in order to convert `MY_CONST` to the integer literal value of 20. When you get right down to it, constants are just a human luxury—all they do is let programmers think in more natural, tangible terms (it's easier to remember `UP`, `DOWN`, `LEFT`, and `RIGHT` than it is to remember `0`, `1`, `2`, and `3`). Furthermore, they let you use the same value over and over within scripts without worrying about needing to change each instance individually later. Although these are indeed useful benefits, they don't help the script accomplish anything new that it couldn't before. And as you've seen, they add an overhead to the execution that, although often negligible, does exist.

NOTE

Constants defined with C's `#define` directive don't actually persist until runtime— the compiler (or rather, the preprocessor) replaces all instances of the constant's name with its value. This allows the coder to deal with the symbol, whereas the processor is just fed raw data as it likes it.

The two-pass approach lets you enjoy the best of both worlds, however, because it gives you the ability to eliminate constants entirely from the runtime aspect of the script. This is done through some basic *preprocessing* of the script, which means you actually make changes to the script code before attempting to execute it. Specifically, as the first pass is being performed, each parameter of each command is analyzed to determine whether it's a constant. If so, it's replaced with the integer value found in its corresponding node in the constant list. This can be done a number of ways, but the easiest is to create a new string about the same size as the existing line of code, copy everything in the old line up until the first character of the constant, write the integer value, and then write everything from just after the last character in the constant to the end of the line. This will produce a new line of code wherein the constant reference has been replaced entirely with its integer value. This can even be applied to the otherwise built-in `TRUE` and `FALSE` keywords for the same reasons. Check out Figure 4.10 to see this in action.



Original code:

```
SetPlayerDir LEFT
MovePlayer -20 0
Pause PAUSE_DUR
ShowTextBox "Hey!"
```

Preprocessed code:

```
SetPlayerDir 2
MovePlayer -20 0
Pause 200
ShowTextBox "Hey!"
```

UP	0
DOWN	1
LEFT	2
RIGHT	3
PAUSE_DUR	200

Figure 4.10

Directly replacing constant references with their values improves runtime performance.

Now, with the preprocessed code entirely devoid of constant references, the constant list can be disposed of entirely and any extra code written into `GetIntParam ()` for handling constants can be removed. The finished script will now appear to the interpreter as if it were written entirely by hand, and execute just as fast. How cool is that?

Loading Before Executing

Aside from the added complexity of the two-pass method, there is one downside. Especially in the case of constant preprocessing, a two-pass interpreter will be performing a considerable amount of string processing and manipulation in its first pass, which means steps should be taken to ensure that only the second pass is performed at runtime.

TIP

In addition to loading all scripts up front, another way to improve overall performance is to implement a caching mechanism that orders scripts based on how recently they were active. This way, scripts can slowly be phased out of the system. A script that hasn't been used recently is less likely to be reused than a script that has just finished executing. Once a script reaches the end of the cache, it can be unloaded from memory entirely. This is an efficient method of memory organization that helps intelligently optimize the space spent on in-memory scripts.

Just as graphics and sound are always loaded from the disk long before they're actually used, scripts should be both loaded and preprocessed before running. This allows the first of the two passes to take as much time as it needs without intruding on the script's overall runtime performance. What this does mean, however, is that your engine should be designed specifically to determine all of the scripts it will need for a specific level, town, or whatever, and make sure to load all of them up front.

Once in memory, a preprocessed script can be run once or looped with no additional performance penalty. This allows the game engine to invoke and terminate scripts at will, with the assurance that all scripts have been loaded and prepped in full already.

Simple Iterative and Conditional Logic

It goes without saying that, just as in traditional programming, iterative and conditional logic play a huge role in scripting. Of course, simple command-based languages are designed specifically to avoid these concepts, as they're generally difficult to implement and require a number of other features to be added as well (for example, its hard to use both looping and branching without variables and expressions).

However, applications for both loops and branching logic abound when scripting games, so you should at least investigate the possibilities. For example, consider the NPC behavior you scripted in the last chapter. NPCs are a great example of the power of command-based scripting, because they can often get by with simple, predictable, static movement and speech. However, especially in the case of RPGs, with the turbulent nature of their always-changing game worlds, even non-pivotal NPCs help create a far more immersive world if they can manage to react to specific events and conditions (Figure 4.11 illustrates this).

Conditional Logic and Game Flags

For example, imagine a simple villager in an RPG. The player can talk to this character, invoking a script that defines his reaction to the player's presence via both speech and movement. The character talks about the weather, or whatever global plague you're in the process of valiantly defeating, and seems pretty lifelike in general. The problem arises when you talk to him more than one time and receive the same canned response every time. Also, imagine returning to town after your quest is complete and hearing him make continual references to the villain you've already destroyed! The player won't appreciate going to the trouble of saving the world if none of its inhabitants is intelligent enough to know the difference.

The common thread between both repeatedly talking to the character, as well as talking to him or her again after completing a large task, is that the conditions of the world are slightly different. In the first case, nothing has really changed, aside from the fact that this particular NPC has

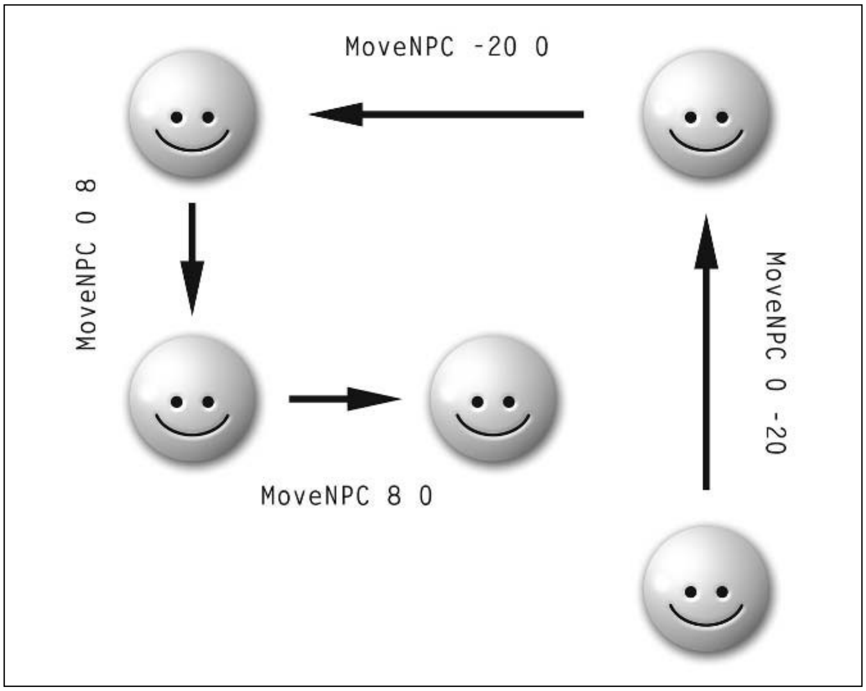


Figure 4.11

Command-based scripts are good for predictable, "canned" NPC movement.

been talked to already. In the second case, the NPC now lives in a world no longer threatened by "the ultimate evil," and can probably react in a much cheerier manner. As discussed in Chapter 2, these are all examples of *game flags*.

Game flags are set and cleared as various events transpire, and persist throughout the lifespan of the game. Each flag corresponds to a specific and individual event, ranging from mundane details like whether you've talked to Ed on the corner, all the way up to huge accomplishments like defusing the nuke embedded in the planet's central fusion reactor. Check out Figures 4.12 and 4.13.

In both cases, the change was binary. You've talked to Ed or you haven't. You've defused the bomb or you haven't. You have enough money to buy a sword or you don't. Because all of these conditions are either on or off, you can add very simple conditional logic to your scripts that does nothing more than perform one of two possible actions depending on the status of the specified flag.

NOTE

Of course, game flags don't *have* to be binary. They can also reside within a range of values or states, but for simplicity's sake. this chapter uses off and on for now.

Because the game's flags are probably going to be stored in an array or something along those lines, each flag can likely be referenced with an integer index. This means a conditional logic structure would only need the integer of the flag the script wants to check, which is even easier to implement.

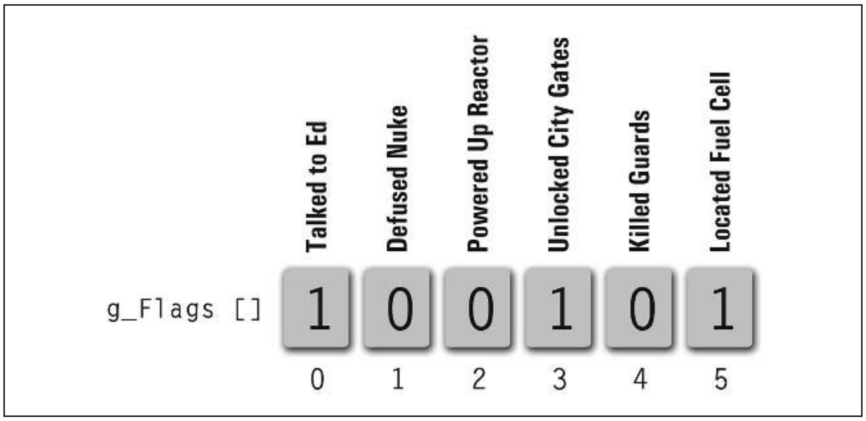


Figure 4.12

Game flags maintain a list of the status of the game's major challenges and milestones.

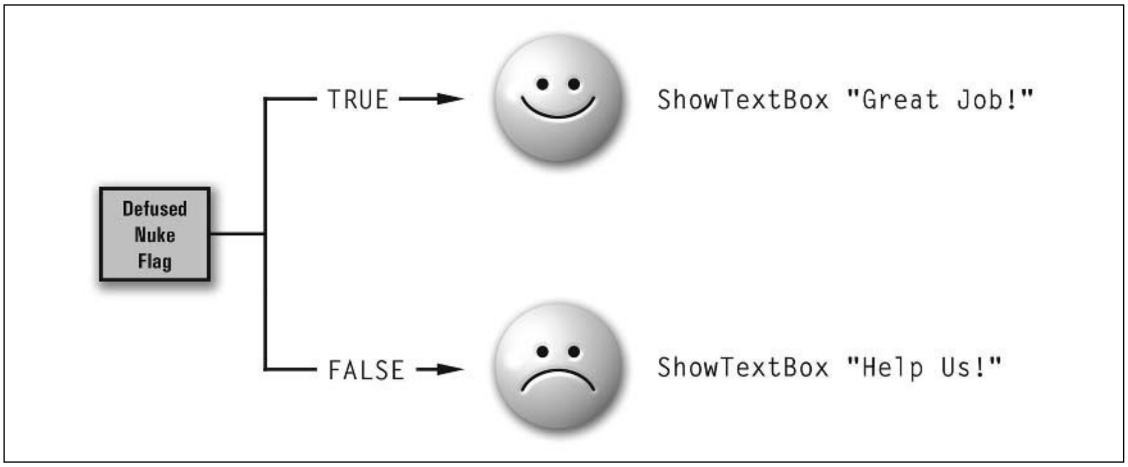


Figure 4.13

Using game flags to alter the behavior of NPCs based on the player's actions.

Furthermore, you can use the symbolic constants described in the previous section to give each flag a descriptive name such as `ED_TALKED_TO` or `NUKE_DEFUSED`.

Specifying a flag with either an integer parameter or constant is easy. The real issue is determining how to group code in such a way that the interpreter knows it's part of a specific condition. One solution is to take the easy way out and place a restriction on scripts that only allows individual commands to be executed for true and false conditions. This might look like this:

```
If NUKE_DEFUSED
ShowTextBox "You did it! Congrats!"
ShowTextBox "Help! There's a nuke in the reactor!"
```

In this simple example, the new `If` command works as follows. First, its single integer parameter (which, of course, can also be a constant) is evaluated. The following two lines of code provide both the true and false actions. If the flag is set, the first of these two lines is executed and the second is skipped. Otherwise, the reverse takes place. This is extremely easy to implement, but it's highly restrictive and doesn't let you do a whole lot in reaction to various flag states. If you want to do more than one thing as a the result of a flag evaluation, you have to precede each command with the same `If NUKE_DEFUSED` line, which will obviously result in a huge mess.

Grouping Code with Blocks

An easier and more flexible solution is to allow the script to encapsulate specific chunks of its code with *blocks*. A block of script code is just like a block of C/C++ code, and even more like a C/C++ function—it wraps a sequential series of commands and assigns it a single name by which it can be referenced. In this way, the commands can be thought of by the rest of the script as a singular unit. Here's an example of a block definition:

```
// If the nuke has been defused
Block NukeDefused
{
// The NPC should congratulate the player
ShowTextBox "You did it! Congrats!"
Pause 400

// Then he should jump up and down
PlayNPCAnim JUMP_UP_AND_DOWN

// If the nuke is still primed to detonate
Block NukePrimed
{
// The NPC should seem worried
ShowTextBox "Help! There's a nuke in the reactor!"
Pause 400

// So worried, in fact, that he runs in a circle
SetNPCDir LEFT
MoveNPC -24 0
SetNPCDir DOWN
MoveNPC 0 24
SetNPCDir RIGHT
MoveNPC 24 0
SetNPCDir UP
MoveNPC 0 -24
}
```

These blocks provide much fuller reactions to each condition, and can be referred to with a single name. Now, if the `If` command is rewritten to instead accept three parameters—an integer flag index and two block names—you could rewrite the previous code like this:

```
If NUKE_DEFUSED NukeDefused NukePrimed
```

Slick, eh? Now, with one line of code, you can easily reference arbitrarily sized blocks that can fully handle any condition. Of course, you can still only handle binary situations, but that should be more than enough for the purposes of a command-based language. Check out Figure 4.14.

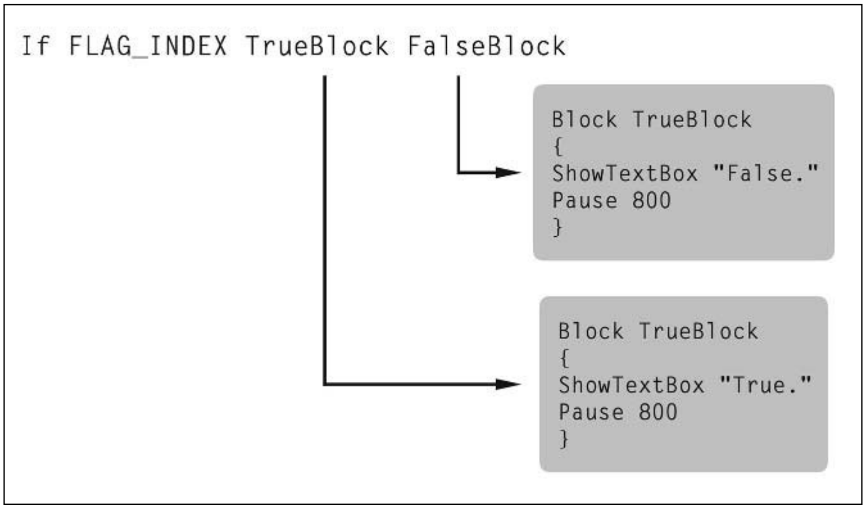

If FLAG_INDEX TrueBlock FalseBlock
Block TrueBlock { ShowTextBox "False." Pause 800 }
Block TrueBlock { ShowTextBox "True." Pause 800 }

Figure 4.14

Using blocks to encapsulate script code and refer to it easily.

Of course, this only a conceptual overview. The real issue is actually routing the flow of execution from the `If` command to the first command of either of the blocks, and then returning when finished. The first and most important piece of information is where the block resides within the script. Naturally, without knowing this, you have no way to actually invoke the proper block after evaluating the flag. In addition, you need to know when each block ends, so you know how many commands to execute before returning the flow of the script back to the `If`.

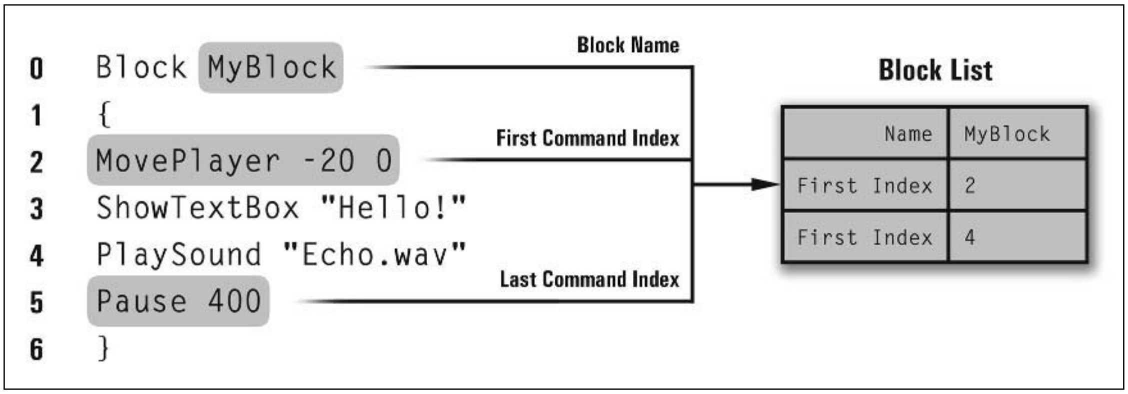
The Block List

This information can be gathered in the same way the constant list was pieced together in the first pass of the two-pass approach discussed earlier. In fact, blocks almost *require* an initial pass to be performed after loading the script, because attempting to collect information about a script's blocks *while* executing that same script is tricky and error-prone at best.

Naturally, you'll store this information in another linked list called the *block list*. This list will contain the names of each block, as well as the indexes of the first and last commands (or, if you prefer, the amount of commands in the block, although either method will work). Therefore, in addition to scouting for `DefConst` lines, the first pass also keeps an eye out for lines that begin with the `Block` command. Once this is found, the following process is performed:

- The block name, which follows the `Block` command just as the constant identifier followed `DefConst`, is read.
- The name of the block is verified to ensure that it's a valid name, and the block list is searched to ensure that no other block is already using the name.
- The next line is read, which should contain an open brace only.
- The next line contains the block's first command; this index is saved into the block list.
- Each subsequent command is read until a closing brace is found. This is the final command of the block and is also saved to the table.

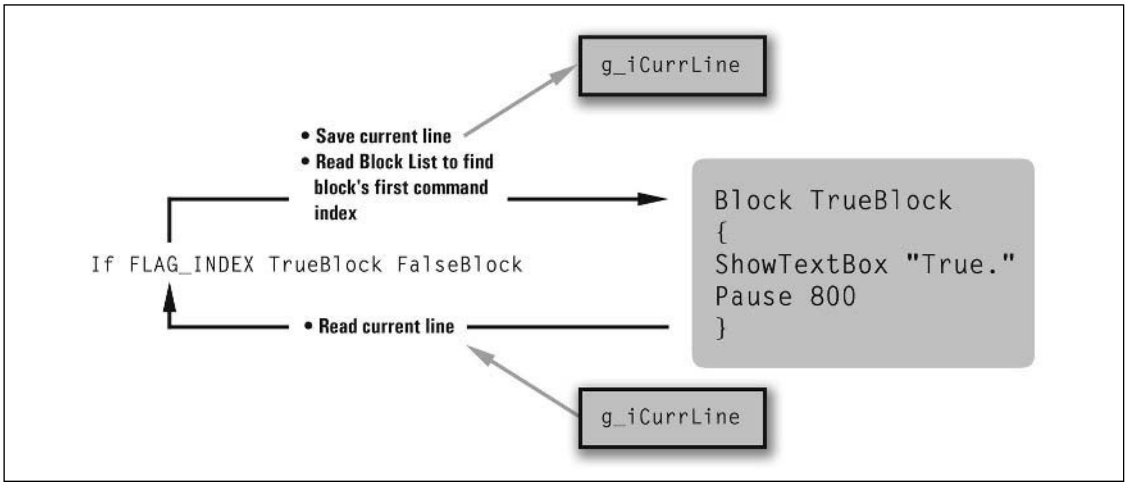
Check out Figure 4.15 to see this process graphically. With the block list fully assembled, the execution phase can begin and the `If` commands can vector to blocks easily. Of course, there's one final issue, and that's how the `If` command is returned to once the block completes. An easy solution consists simply of saving the current line of code into a variable before entering the block. Once the block is complete, this line of code is used to return to the `If` (or rather, the command immediately following it), and execution continues. As you'll see later in the book, this process is very similar to the way function calls are facilitated in higher-level languages. Figure 4.16 illustrates the process.


```
0   Block MyBlock
1   {
2   MovePlayer -20 0
3   ShowTextBox "Hello!"
4   PlaySound "Echo.wav"
5   Pause 400
6   }
```

Name	MyBlock
First Index	2
First Index	4

Figure 4.15

Saving a block's info in the block list.


g_iCurrLine
• Save current line
• Read Block List to find block's first command index
If FLAG_INDEX TrueBlock FalseBlock
• Read current line
g_iCurrLine

```
Block TrueBlock
{
ShowTextBox "True."
Pause 800
}
```

Figure 4.16

Saving the current line of code before vectoring to a block allows the block to return.

TIP

Earlier in the chapter I discussed directly replacing constants within the script's code with their respective values in a preprocessing step that allowed the script to execute faster and without the need for a separate constant list. This idea can be applied to blocks as well; rather than forcing `If` commands to look up the block's entry in the block list in order to find the index of its first command, that index can be used to directly replace the block name.

Iterative Logic

Getting back to the original topic, there's the separate issue of looping and iteration. Much like the `If` command, a command for looping needs the capability to stop at a certain point, in response to some event. Because this simple scripting system is designed only to have access to binary game flags, these will have to do.

Looping can be implemented with a new command, named `While` because it most closely matches the functionality of C/C++'s `while` loop. `While` takes two parameters, a flag index and a block name. For example, if you wanted an NPC to run to the east (away from the reactor), stopping to yell and scream periodically, until the nuke was defused, you might write a script like this:

```
Block RunLikeHell
{
// Run to the left/east, away from the reactor
MoveNPC 80 0
// Stop for a moment to scream bloody murder
ShowTextBox "WE'RE ALL GONNA DIE!!!"
Pause 300
// Keep moving!
MoveNPC 80 0
// Scream some more
ShowTextBox "SERIOUSLY! IT'S ALL OVER!!!"
Pause 300
// As long as the loop runs, this block will be executed over and over
}

// If the nuke is still primed, keep our poor NPC moving
While NUKE_PRIMED RunLikeHell
```

The cool thing is, the syntax of `While` almost gives it an English-like feel to it: "While the nuke is primed, run like hell!" Check out Figure 4.17 for a visual idea of how this works.

You may have noticed, however, that you're now using a flag called `NUKE_PRIMED` instead of `NUKE_DEFUSED`, like you were earlier. This is because, so far, there's no way to test for the opposite of a flag's status, whether it be set or cleared. You can alleviate this problem by adding the possibility for a C/C++-style negation operator to precede the flag index in a `While` command, which would look like this:

```
While ! NUKE_DEFUSED RunLikeHell
```

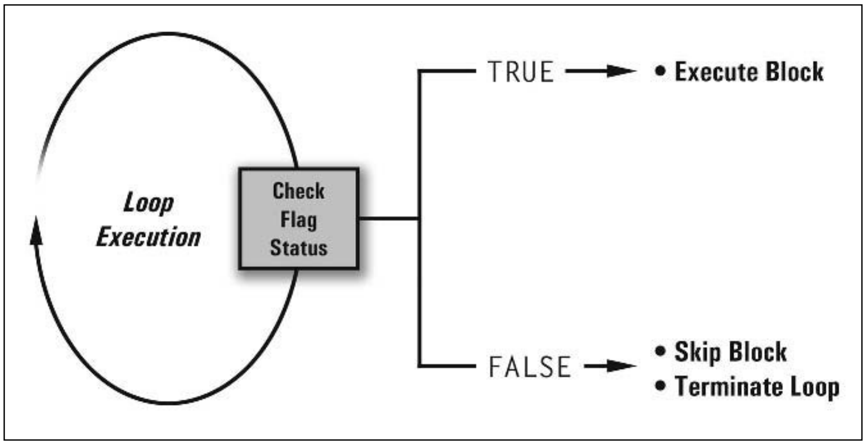


Figure 4.17

Looping the same block until the specified flag is cleared.

This is a decent solution, but it's a bit complex; you now have to test for optional parameters, which is more logic than you're used to. Instead, it's easier to just add another looping command, one that will provide the converse of `While`:

```
Until NUKE_DEFUSED RunLikeHell
```

Simple, huh? Instead of looping *while* a flag is set, `Until` loops *until* a flag is set. This allows you to use the same techniques you're used to. Of course, there's no need to actually implement two separate loop commands in the actual interpreter's code. `While` and `Until` can be handled by the same code; `Until` just needs to perform an automatic negation of the flag's value.

The looping commands of course use the same the block list gathered to support `If`, so overall, once `If` is implemented, `While` and `Until` will be trivial additions. Also, just as `If` saves the current line of code before invoking a block, the looping commands will have to do so as well so sequential execution can resume when the loop terminates.

Nesting

The addition of looping and branching commands inadvertently exposed you to the concepts of grouping common code in blocks, and invoking those blocks by name. Because this concept so closely mirrors the concept of functions, you may be wondering how nesting would work. In other words, could a `Block` contain an `If` or `While` command of its own?

Given the current state of the runtime interpreter, the answer is no. Remember, the only reason you can safely invoke a block in the first place is because you save the line of script to which it will have to return in a variable. If you were to call another block from *within* this block, it would permanently overwrite that variable with a new index, thus robbing the first block of the ability to return to the command that invoked it.

The best way to support nesting is to implement an *invocation stack* that maintains each of the indexes that blocks will need to return, in the order in which the blocks were invoked. For example, consider the following code:

```
While FLAG_X BlockX

Block BlockX
{
ShowTextBox "Block X called."
Pause 400
While FLAG_Y BlockY
}
```

```
Block BlockY
{
ShowTextBox "Block Y called."
Pause 400
While FLAG_Z BlockZ
}

Block BlockZ
{
ShowTextBox "Block Z called."
Pause 400
}
```

First `BlockX` is called, which will push the index of the first `While` line onto the stack. Then, `BlockY` is called, which pushes the index of `BlockX`'s `While` line onto the stack. The same is done for `BlockY` and its `While` command, which finally calls `BlockZ`. `BlockZ` immediately returns after displaying the text box and pausing, which pops the top value off of the stack and uses it as the index to return to. Execution then returns to `BlockY`, which pops the new top value off the stack and uses it to return to `BlockX`. `BlockX`, which is also returning, pops the final value off the stack, leaving the stack once again empty, and uses that value to return to the initial `While` command. Figure 4.18 illustrates an invocation stack in action.

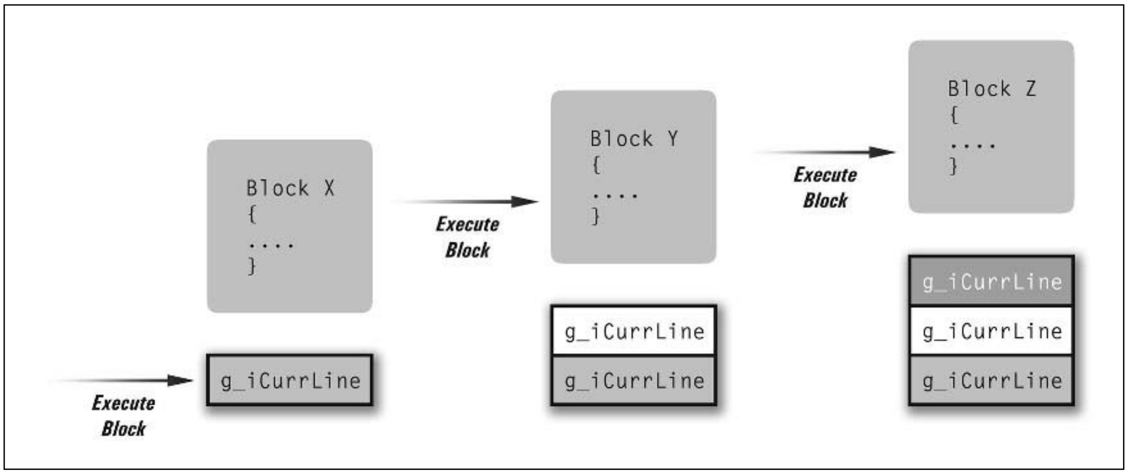


Figure 4.18

An invocation stack allows nested iterative and conditional logic.

As you can see, support for nested block invocation is not a trivial matter, so I won't discuss it past this. Besides, as the book progresses, you'll get into *real* functions and function calls, and learn all about how this process works for serious scripting languages. Until then, nesting is a luxury that isn't necessary for the basic scripting that command-based languages are meant to provide.

Event-Based Scripting

Games are really nothing more than a sequence of events, which naturally plays an important role in scripting. Events are triggered in response to both the actions of the player and non-player entities, and must be handled in order to create a cohesive and responsive game environment. Because scripts are often used to encapsulate portions of the game's logic, it helps to be able to bind scripts to specific events, so that the game engine will automatically invoke the script upon the triggering of the event.

You can already do this, because your scripts are stored in memory and can be run at any time (if you recall, the final demo of the last chapter stored a script within each NPCs structure, which could be invoked individually by passing an index parameter to `RunScript ()`). All that's necessary is to let the game engine know the index into your array of currently loaded scripts of the specific script you'd like to see run when a certain event happens, and the engine's event handler should take care of the rest.

Events, like many things, however, come in varying levels. There are very high-level events, such as the defusing of the nuke. There are then lower-level events, like talking to a specific NPC in a specific town. Events can be even of a lower-level than that. That individual NPC alone may be able to respond to a handful of its own events. In this regard, events often form a hierarchy, much like a computer's file system. Figure 4.19 illustrates an event hierarchy.

As it stands now, your system only deals with scripts on the file level. Each file maps directly to one script, which, in turn, can be used to react to one event. This is fine in many cases, but when

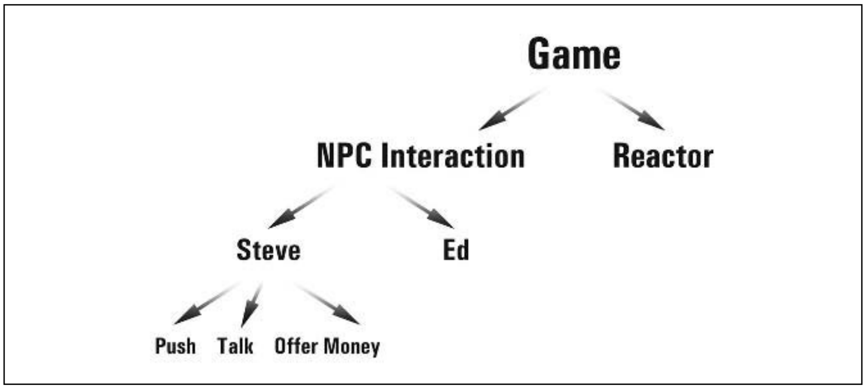


Figure 4.19

Game events form a hierarchy.

you start getting lower and lower on the heirarchy, and events become more and more specific, it gets cumbersome to implement each of these events' scripts in separate files. For example, if an NPC named Steve can react to three events—being talked to, being pushed, and being offered money—your current system would force you to write the following scripts:

```
steve_talk.cbl
steve_push.cbl
steve_offer_money.cbl
```

After a while, creating a new file for each event will get ridiculous. It won't be long before you reach this point:

```
steve_approach_while_holding_red_sword.cbl
```

It would be much nicer to be able to store Steve's entire event handling scripts in a single file called `steve.cbl`. You already have a system for defining blocks with symbolic names, so all you really need to do is allow the game engine to request a specific block to run, rather than an entire script. For example, imagine rewriting `RunScript ()` to accept a script index as well as a block name. You could then use it like this:

```
RunScript ( SCRIPT_NPC_STEVE, "Talk" );
```

This allows script files and blocks to map more naturally to levels of the event hierarchy, as shown in Figure 4.20. Inside the function, `RunScript ()` would then simply reposition the current line of the script to the first function of the block, using the block list in the same way `If`, `While`, and `Until` did. This is actually even easier, because there's no return index to worry about; once the block is finished, the `RunScript ()` function just returns to its caller.

NOTE

One important issue regarding the invocation of specific script blocks is that it *will* disrupt execution if that script is already running. Because of this, it's best to write certain scripts for the purpose of running concurrently in the background with the game engine (*synchronously*), whereas other scripts are designed specifically to provide a number of blocks to be invoked on a non-looping basis in reaction to events (*asynchronously*). Therefore, Steve may instead be implemented with two files: `steve_sync.cbl`, which runs in the background indefinitely like the NPC scripts of the last chapter, and `steve_async.cbl`, which solely exists to provide blocks the game engine can invoke to handle Steve-specific events.

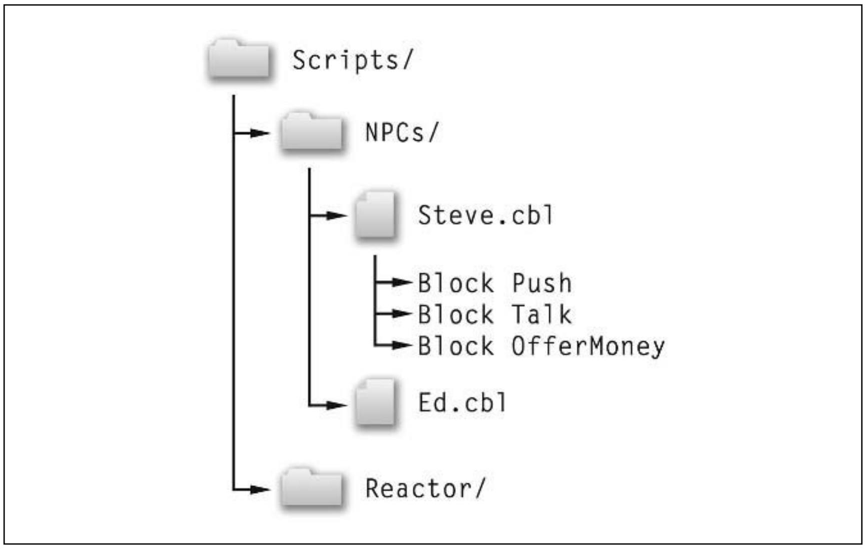


Figure 4.20

Mapping scripts' file/directory structure to the game's event hierarchy.

Compiling Scripts to a Binary Format

Thus far you've seen a number of ways to enhance a script's power and flexibility, but what about the script data itself? You're currently subjecting your poor real-time game engine to a lot of string processing that, at least when compared to dealing strictly with integer values, is slow. Just as you learned in Chapter 1, interpreting a script on a source-code level is *considerably* slower than executing a compiled script expressed in some binary format, yet that's exactly what you're doing.

Fortunately, it would be relatively easy to write a "compiler" that would translate human-readable script files to a binary format, and there are a number of important reasons why you would want to do this, as discussed in the following sections.

Increased Execution Speed

First and foremost, scripts *always* run faster in a compiled form than they do in source code form. It's just a simple matter of logic—if processing human-readable source code is more complex and taxing on the processor than processing a binary format, the binary format will obviously execute much faster.

Think about it—currently, every time a command is executed, the following has to be done:

- The command is read with a call to `GetCommand ()`. This involves reading each character from the line until a space is found and placing these characters in a separate string buffer.

- The string buffer containing the command is then compared to each possible command name, which is another operation that requires traversing each character in the string. Each character is read from the string buffer and compared to the corresponding character in the specified command name to make sure the strings match overall.
- Once a command has been matched, its handler is invoked which performs even more string processing. `GetStringParam ()` and `GetIntParam ()` are used to read string and integer parameters from the source line, performing more or less the same operation performed by `GetCommand ()`.
- `GetIntParam ()` might not have to traverse the constant list, depending on whether a preprocessing phase was applied to the script upon its loading.
- The `If`, `While`, and `Until` commands will have to search the block list in order to find the first command of the destination block, again, unless the script was preprocessed to replace all block names with such information.

Yuck! That's a *lot* of work just to execute a single command. Now multiply that by the number of commands in your script, and further multiply *that* by the number of scripts you have running concurrently, and you have a considerable load of string processing bearing down on the CPU (and that says nothing of any script blocks that may be called by the game engine asynchronously in response to events, which of course add more overhead).

Fortunately, compilation provides a much faster alternative. When all of this extraneous string data is replaced with numeric data that expresses the same overall script, scripts will execute exponentially faster. Check out Figure 4.21.

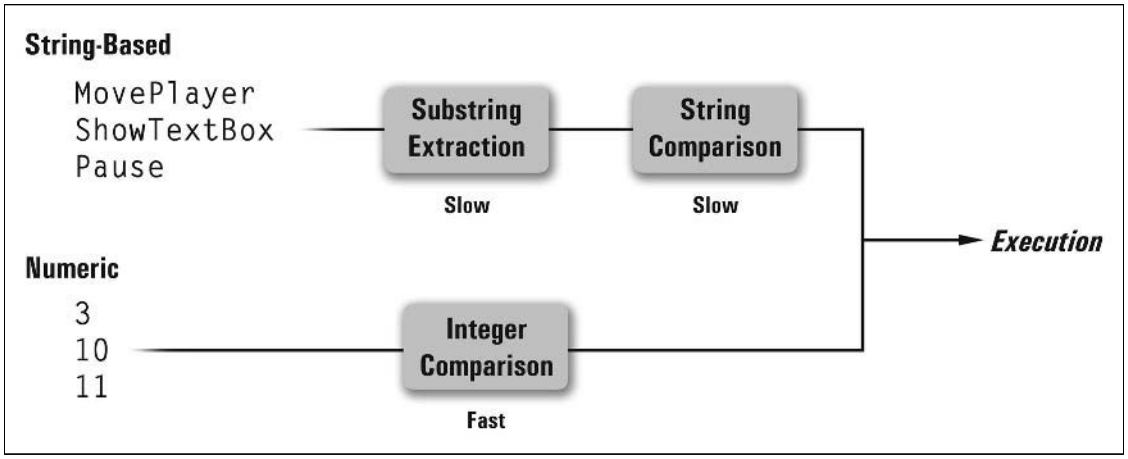


Figure 4.21

Numeric data executes much faster than string data.

Detecting Compile-Time Errors

The fastest script format in the world doesn't matter if it has errors that cause everything to choke and die at runtime. Despite the simplicity of a command-based language, there's still plenty of room for error, both logic errors that simply cause unexpected behavior, and more serious errors that bring everything to a screeching halt. For example, how easy is it to misspell a command and not know it? The current implementation would simply ignore something like "`MuveNPC`", causing your NPC to inexplicably do nothing. Of course, parameters are a serious source of potential errors as well. Parameters of the wrong type can cause serious errors as well—providing an integer when a string is expected will cause `GetStringParam ()` to scan through the entire line looking for a non-existent double-quote terminator. Simply not providing enough parameters can lead to runtime quirks, from simple logic errors to string boundary violations.

A compiler can detect all of this long before the script ever has to execute, allowing you to make your changes ahead of time. A compiler simply won't produce a binary version of the script until all errors have been dealt with, allowing you to run your scripts with confidence. Also, less potential for runtime errors means less runtime error checking is needed, contributing yet another small performance boost.

Malicious Script Hacking

Lastly, and in many ways most importantly, is the issue of what malicious players can do when a script is in an easily readable and editable form. For example, the `While` and `Until` loops practically read like broken English, which just *screams* "hack me!" to anyone who happens to load them into a text editor.

When scripts are that easily modifiable, every line of dialog, every NPC movement, and every otherwise cinematic moment in your game is at the mercy of the player. In the case of single player games, this a marginally serious issue, but when multiplayer games come into play, true havoc can be wreaked. With a single player game, it's really only your artistic vision that's at stake, and the possibility of the player either cheating or screwing up their personal version of the game. Obviously this isn't ideal, but it's nothing to get worked up over because it won't affect anyone other than the hacker.

Script hackers can ruin multiplayer games, however, which often rely on client-side scripts to control certain aspects of the game's logic. Like all client-side cheats, such hacks may result in one player having an unfair advantage over the rest of the players. For example, if one of your scripts causes the players character to slow down and lose accuracy when he's hit with a poison dart, a quick change to `poison_dart.cbl` can give that player an unconditional immunity that puts everyone else at a disadvantage.

Compiled scripts are not in a format that's easily readable by humans, nor are they even easily opened in a text editor in the first place. Unless the player is willing to crack them open in a hex editor and understands your compiled script format, you can sleep tight knowing that your game is safe and all is well.

How a CBL Compiler Works

A command-based language is easily compiled. Really, all you need to do is assign each command a unique integer value, and write a program that will convert each command from a string to this value. This compiled data is then written sequentially to a separate, binary file, and a new run-time environment is created to load and support the new format.

For example, imagine your game's particular language is composed of the commands listed in Table 4.1.

Of course, it also supports the more generic, domain-independent commands, listed in Table 4.2.

These commands can each be assigned a unique integer value, which could be called a *command code*, as listed in Table 4.3.

Table 4.1 Example Language Commands

Command	Description
`MovePlayer`	Moves the player to a specified X,Y location.
`GetItem`	Adds the specified item to the player's inventory.
`PlayPlayerAnim`	Plays a player animation.
`MoveNPC`	Moves the specified NPC to the specified X,Y location.
`PlayNPCAnim`	Plays an NPC animation.
`PlaySound`	Plays a sound.
`PlayMovie`	Plays a full-screen movie.
`ShowTextBox`	Displays a string of text in the text box.
`Pause`	Pauses execution of the script for the specified duration.

Table 4.2 Domain-Independent Commands

Command	Description
`DefConst`	Defines a constant and assigns it the specified integer value.
`If`	Evaluates the specified flag and executes one of the two specified blocks based on the result.
`While`	Executes the specified block until the specified flag is cleared.
`Until`	Executes the specified block until the specified flag is set.

Table 4.3 Command Codes

Command	Code
`DefConst`	0
`If`	1
`While`	2
`Until`	3
`MovePlayer`	4
`GetItem`	5
`PlayPlayerAnim`	6
`MoveNPC`	7
`PlayNPCAnim`	8
`PlaySound`	9
`PlayMovie`	10
`ShowTextBox`	11
`Pause`	12

This means that, if the compiler were fed a script that consisted of the following sequence of commands (ignore parameters for now):

```
DefConst
DefConst
MovePlayer
MoveNPC
PlaySound
MovePlayer
GetItem
PlaySound
```

The compiler would translate this to the following numeric sequence (see for yourself by comparing it to the previous table):

```
0 0 4 7 9 4 5 9
```

As long as you keep ignoring parameters for just a moment, you can turn this into a fully descriptive, compiled script by simply preceding this data with another integer value that tells the script loader how many instructions there are to load:

```
8 0 0 4 7 9 4 5 9
```

The script loader then reads this first integer value, uses it to determine how many instructions the file contains, and reads them into an array.

Executing Compiled Scripts

Once this file is loaded into memory, it can be executed easily—a lot more easily than source code can be interpreted. Instead of reading the command string from the current source line, you can just read the value of the array index that corresponds to the current line and enter a `switch` block that routes control to the proper handler. For example:

```
// Read the command
int iCurrCommand = g_Script [ iCurrLine ];

// Route control to the proper command handler
switch ( iCurrCommand )
{
    case COMMAND_DEFCONST:
        // DefConst handler
        break;
    case COMMAND_MOVEPLAYER:
        // MovePlayer handler
        break;

    case COMMAND_PAUSE:
        // Pause handler
        break;
}
```

These new numeric "command codes" make everything *much* faster, smaller, easier, and more robust. Of course, you are skipping one major advantage that you can easily take advantage of when compiling.

Compile-Time Preprocessing

You've already seen the advantage of preprocessing the `DefConst` command, as well as references to constants to block names. Of course, you had to do this when the script was loaded, in the game engine, which meant more room for error as the game is initializing and running. Offloading this process to the compiler makes the game engine's code much simpler and, as always, reduces the chances of runtime errors.

Preprocessing Constants

Because of this, `DefConst` doesn't even need to be compiled to a command code; rather, it can simply be preprocessed out of the script at compile-time, thus shifting all of the codes down by one. The language's new codes are listed in Table 4.4.

This means the compiler will now be responsible for generating the constant list and using it to replace constant references with their values. Scripts can now be executed with no preprocessing step and without the need to maintain or consult a constant list.

Block Reference Preprocessing

The block list can, for the most part, be handled by the compiler as well. In the compiler's first pass over the source, the block list described earlier will be built up and used to replace all references to block names with the block's index into the list so the string component can be discarded. At runtime, this index will be used to find the block's information when executing `If`, `While`, and `Until` instructions. Of course, the block list still has to persist until runtime, because the game engine will need to know where each block begins and ends.

Each entry in the block list can therefore be written out to the compiled script file as two integer values, the locations of the block's beginning and terminating commands. In addition, this list

Table 4.4 Revised Command Codes

Command	Code
`If`	0
`While`	1
`Until`	2
`MovePlayer`	3
`GetItem`	4
`PlayPlayerAnim`	5
`MoveNPC`	6
`PlayNPCAnim`	7
`PlaySound`	8
`PlayMovie`	9
`ShowTextBox`	10
`Pause`	11

will be preceded with the number of entries it contains, just like you did with the command list itself. For example, imagine a script has two blocks. The first block begins at the seventh command and ends at the twelfth, and the second begins at the 22nd and ends at the 34th. The block list would then be written out like this:

```
2 7 12 22 34
```

The leading `2` tells you how many blocks are in the list, whereas the following values are the starting and ending commands. The runtime environment can then load this into an in-memory array and be ready to roll.

Parameters

Last is the issue of compiling parameters. Parameters are a bit more complex than commands, because they come in a number of different forms. Fortunately, however, by the time preprocessing is through, you'll only have integers and strings to deal with. Naturally, integers are extremely simple to compile, because they're already in an irreducible format. Strings, although more complex, really can't be compiled much either, aside from attempting to perform some sort of compression (but then, that's not compiling, it's just compressing).

The first and most important step when compiling parameters is ensuring that the command has been supplied with both the right number of parameters, as well as parameters of the proper data type. Once this is taken care of, the next step is to write them out to the file, immediately following the command code. Because each command has a fixed number of parameters, the loader can tell how many instructions to read based on the command code alone. The loader then knows to read this number of parameters before expecting the next command code. Integers can be written out as-is, as long as the script loader knows to always read four bytes. Strings can be written out in their typical null-terminated form, as long as the loader knows this as well. Figure 4.22 illustrates the storage of commands and parameters in a compiled script file.

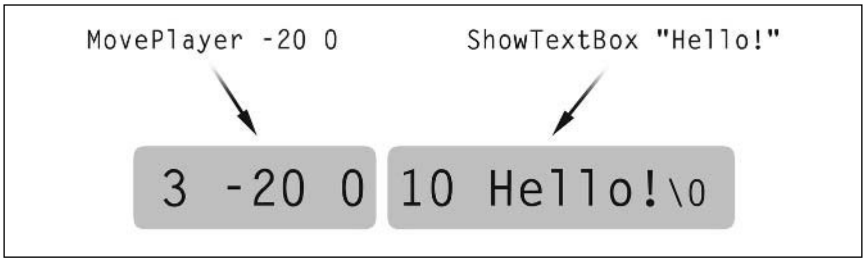


Figure 4.22

Commands and parameters are stored in a tightly packed format in a compiled script.

The real issue is what to do with them in memory. Because parameters add a whole new dimension of data to deal with, you can no longer simply store the compiled script in an integer array. Rather, each element of this array must be a structure that contains the command code and the parameters. For simplicity's sake, you can just give each element the capability to store a fixed number of parameters, so you can pick some maximum that you know you'll never exceed. Eight should be more than enough.

However, because a parameter can be either a string or an integer, you need a way to allow either of these possibilities to exist at any of the array's indexes. This can be easily accomplished with the following `union`:

```
union Param                           // A parameter
{
    int iIntLiteral;                  // An integer value
    char * pstrStringLiteral;         // A string value
}
```

NOTE

On most 32-bit platforms, the size of an integer is usually indicative of the size of a far/long pointer as well, which means that the total size of the `Param` union will most often be four bytes, because the integer and string pointer will perfectly overlap with one another.

These parameters can then be stored in a static array, which is itself part of a larger structure that represents a compiled command:

```
typedef struct Command                  // A compiled command
{
    int iCommandCode;                     // The command code
    Param ParamList [ MAX_PARAM_COUNT ]; // The parameter list
}
```

Remember, `MAX_PARAM_COUNT` is set to some number that is most likely to support any command, like 8 or 16 (both of which are total overkill). Lastly, within each command handler, you can now easily access parameters simply by referencing its `ParamList []` array. There's no dire need for specific `GetIntParam ()` or `GetStringParam ()` functions, but it is always a good idea to wrap array access in such functions to help abstract things. Figure 4.23 illustrates the in-memory command array.

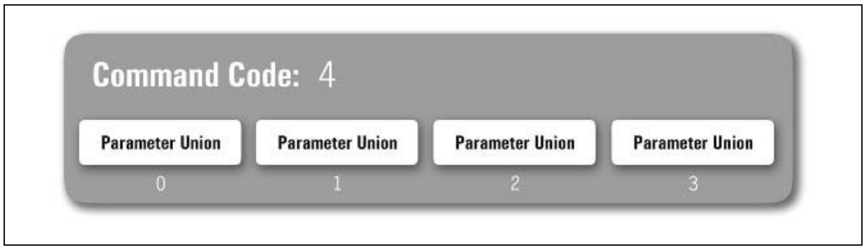


Figure 4.23

Storing commands and parameters in a single structure.

Basic Script Preprocessing

The last subject I'd like to mention is the preprocessing of scripts as they're compiled. You've already seen some basic examples of preprocessing—both the compiler and an earlier version of the script loader made multiple passes over the source code to replace constant and block references with direct numeric values. In a lot of ways, this process is analogous to the `#define` directive of C/C++'s preprocessor. For example, the following script:

```
DefConst MY_CONST 256
MyCommand MY_CONST
```

Is basically doing the same thing as the small C/C++ code fragment:

```
#define MY_CONST 256
MyCommand ( MY_CONST );
```

`DefConst` can therefore be viewed as a way to define simple macros, especially because the compiler will literally perform the same macro expansion that C/C++'s `#define` does. Of course, there's one other extremely useful preprocessor directive in C/C++ that everyone uses: `#include`.

Why would such simplistic command-based scripts need to include other files within themselves? Well, under normal circumstances they wouldn't, but with the introduction of the `DefConst` command, it's possible for scripts to define large quantities of constants that are useful all across the board. Without the capability to include scripts within other scripts, these constants would have to be re-declared in each script that wanted to use them. This would be bad enough for reasons of redundancy, but it can really cause problems when one or two of those constants need to be changed, and 20 files have to be updated to fully reflect it.

For example, any decent RPG will have countless NPCs, all of which need to move around on the map. As you've seen, the cardinal directions play an important part in this, which is why `DefConst` proved so useful. So, imagine that you have 200 NPCs in your game, all of which need `UP`, `DOWN`, `LEFT`, and `RIGHT` constants. Declaring them in all 200 files would be insanity.

The solution is a new command, `IncludeFile`, that includes files with the main script. For example, let's look at a file called `directions.cbl` that declares constants for the cardinal directions:

```
// The cardinal directions
DefConst UP 0
DefConst DOWN 1
DefConst LEFT 2
DefConst RIGHT 3
```

Note the file doesn't even have any code in it; all it does is declare constants. Now, let's look at an NPC script file:

```
// Load the direction file
IncludeFile "directions.cbl"
// Use the directions in the code
SetPlayerDir UP
MovePlayer 0, -40
```

Directions and other miscellaneous constants are one thing, but the real attraction here are game flags. Remember, games may have hundreds or even *thousands* of flags, the constants for which need to be available to *all* scripts. Declaring all of your flags in a single file means every script can easily reference various events and states. For example, here's a file called `flags.cbl`:

```
// Game flags
DefConst NUKE_DEFUSED 0
DefConst REACTOR_POWERED_DOWN 1
DefConst TOWN_DESTROYED 2
DefConst STEVE_TALKED_TO 3
```

TIP

The game flag example brings up an interesting point—not only can constant declarations be included, but entire blocks can be as well.

And here's a sample script that uses it:

```
// Include the game's flags
IncludeFile "flags.cbl"

Until TOWN_DESTROYED MoveNPCs
```

Assuming this file also declares a block called `MoveNPCs`, this script will cause the town's NPCs to move around until it's destroyed. Check out Figure 4.24 for a graphical view of file inclusion.

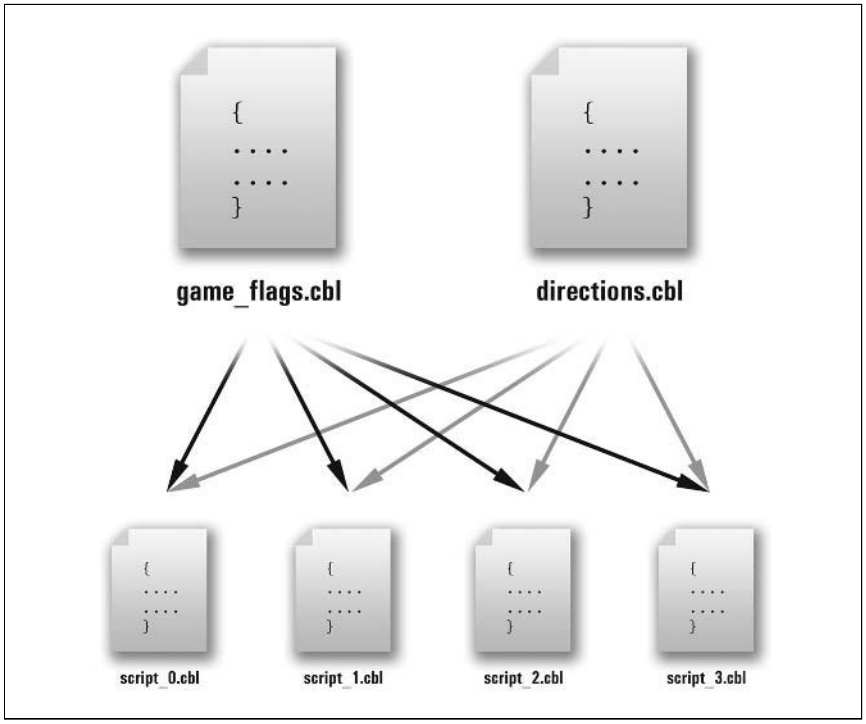


Figure 4.24

Storing game flags and other common constants in a single file that all scripts can access is an intelligent way to organize data.

File-Inclusion Implementation

A file-inclusion preprocessor command is simple to implement, at least on a basic level. The idea is that, whenever an `IncludeFile` command is found, that particular line of code is removed from the script and replaced with the contents of the file it specifies. This means that a single line of code can be expanded to N lines, which in turn means that you'll have to make a change to the way the compiler stores the source code internally. Assuming the compiler loads script source files just as the examples from Chapter 3 did, it's going to have everything locked up in a static array. This is fine until a file needs to be loaded into the script at the position of an `IncludeFile` command, at which point a large number of extra lines will need to be inserted into the array.

For this reason, the compiler should store the source in a linked list. This allows entire files to be inserted at will.

The only real *caveat* to the file-inclusion command is that included files can in turn include files of their own. Because of this, the inclusion macro must be recursive—after a file is loaded into the source code linked list, each of the nodes it added must be searched to determine whether they too include files. If so, the process completes until a file is loaded that doesn't include any files of its own.

> **CAUTION**
>
> **Because it's entirely possible that two files will attempt to include each other, there's always the potential for such files to catch themselves in an infinitely recursive loop. To prevent this, you should maintain an list of filenames referenced by `IncludeFile` commands, and ignore any instances of `IncludeFile` that reference filenames already in this list. This will prevent any file from being loaded more than once, as well as any recursive nightmares from emerging.**

Remember, the inclusion command doesn't perform any syntax checking or compiling on its own—all it does is load into the raw text data. The compiler then deals with everything as if it were one big file; it has no idea that the contents of the source code linked list were ever spread out among multiple files. For example, the previous game flag example would ultimately appear to the compiler like this:

```
// Include the game's flags
// Game flags
DefConst NUKE_DEFUSED 0
DefConst REACTOR_POWERED_DOWN 1
DefConst TOWN_DESTROYED 2
DefConst STEVE_TALKED_TO 3

Until TOWN_DESTROYED MoveNPCs
```

As you can see, even the comments were included, but of course, that doesn't matter to the compiler. The contents of the source code linked list after every file has been included would most likely appear cluttered and disorganized if you were to print it, but of course, the compiler couldn't care less as long as the code is syntactically valid. Check out Figure 4.25.

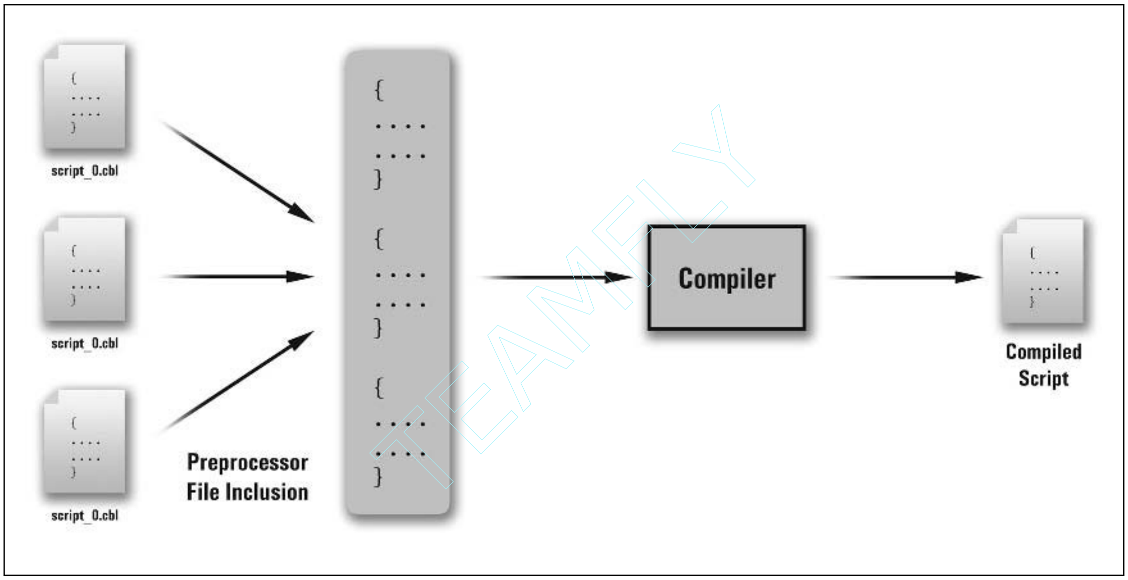


Figure 4.25

The preprocessor simply loads each file into a large script linked list as if they have always been one large unit.

Summary

Phew! This chapter has covered a lot of ground, even if it was largely theoretical. Remember, this chapter wasn't designed to help you literally implement the topics covered here. Rather, I just wanted to introduce a number of possible improvements to the system created in the last chapter, as well as lay the groundwork for some of the fundamental concepts you'll be exploring later in the book.

Issues such as preprocessing, macro and file expansion, managing constants, and grouping code into blocks all overlap heavily with the real compiler theory you'll be learning as you progress through the following chapters. Although everything discussed here was highly simplified and watered down, the underlying ideas are all there and will hopefully put you in a better frame of mind for tackling them in their true, real-life forms later. I personally find difficult stuff much easier to master when I've had a chance to think about it on a more simplistic level beforehand. That was the idea of this chapter—whether you try to implement any of this stuff or not, it will hopefully get the gears turning in your head a bit, so by the time you reach *real* compiler issues, the light bulbs will already be flashing and you'll find yourself saying "Hey! That's almost exactly how I thought it would work!"

Like I said, everything presented here is to be taken as theory, because I've hardly given you enough details to outline a full implementation. However, you'll notice that every concept I used to explain the conceptual implementation of these features was intermediate at best: string processing, textbook data structures like linked lists and hash tables, and so on. Although this chapter alone isn't going to help a total beginner get anywhere, any coder with a decent grasp on basic computer science should have no trouble getting virtually everything covered in this chapter to work in a command-based scripting system.

In the end, my goal is to help you understand that even simple scripting can be extremely useful if it's applied properly, and maybe given some help with the sort of boosted feature set we discussed here. Actually implementing everything this chapter covered would be a lot of work, but it would solve the vast majority of the scripting problems presented by mid-range games. Granted, the triple-A titles out there on the market will need something more sophisticated, but what luck! That's exactly what the following pages will cover.

This page intentionally left blank

Part Three

Introduction to Procedural Scripting Languages

This page intentionally left blank

CHAPTER 5

Introduction to Procedural Scripting Systems

"Well, when all else fails, fresh tactics!"
——Castor Troy, Face/Off

In the last section, you took your first steps towards developing your own scripting system by designing and implementing a command-based language from the ground up. Although the finished product was rather modest, many of the concepts behind basic script execution were illustrated first hand. The following chapters take things to the next level, however. In fact, it'd probably be more appropriate to refer to what's ahead as a entire paradigm shift—the sheer complexity and depth of the components involved with the finished scripting system will require not only a significant amount of structure and foresight, but a marathon runner's endurance as well.

You'll learn how compilers, assemblers, and runtime environments work together to simulate a basic CPU running inside your game, turning your engine into a virtual machine capable of running extremely powerful compiled scripts. No detail will be spared, so you probably won't be surprised that this topic will comprise the largest portion of the book—four sections to be exact. The system you're going to build over the course of these sections, called XtremeScript, will be capable of handling virtually any task you can think of. If you can do it with C/C++, you can more than likely do it with XtremeScript.

But before you get hip-deep in the nitty gritties, the first and most important step is to become fully acquainted with this type of scripting system as a whole. A clear view of the big picture will be more helpful in getting you started than anything else, so it's first on the list of things to cover.

If you're ready, let's get started. This chapter will cover

- The compilation of high-level code.
- The assembly of low-level code.
- The basic layout of a virtual machine.
- The design and arrangement of the XtremeScript system, which we'll build throughout the remainder of this book.

Overall Scripting Architecture

The overall architecture of a system like XtremeScript involves many interconnected components, which themselves can be broken down considerably, as most of them are complex individual systems in their own right. On the most basic level, however, you have the layout illustrated in Figure 5.1.

As you can see, there are really only three major components when you pan out far enough. All three were briefly introduced in Chapter 1, but this time we're going to dig a little deeper.

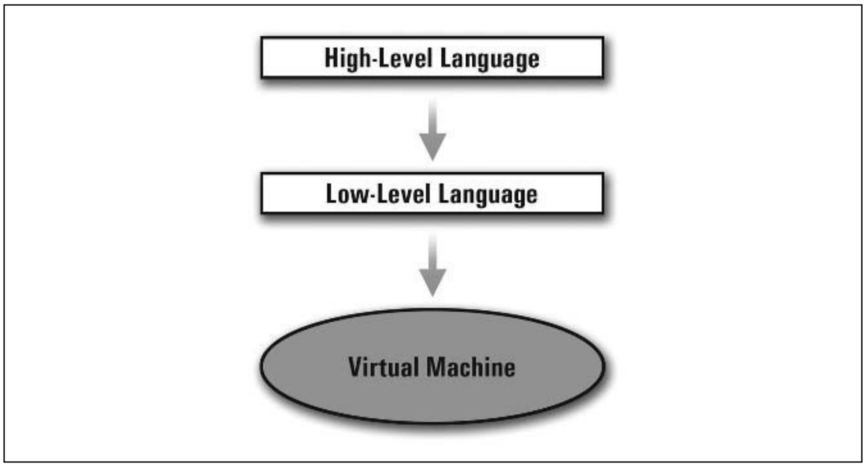


Figure 5.1

The high-level language, low-level language, and virtual machine can be considered the three most basic parts of the XtremeScript system.

High-Level Code

High-level code is the most widely recognized part of a scripting system. Because it's what scripts are written with in the first place, it's the human interface to the script module and perhaps the system's most useful component. High-level languages (HLLs), which include examples such as C, C++, Pascal and Java, were created so that problems could be described in an abstract, English-like manner. This makes HLLs extremely versatile and directly applicable to countless fields, but it's in fact due to this human-friendly nature that they're extremely inefficient when read directly by a CPU.

Humans think in high-level terms; our minds are almost entirely based on the concept of multiple levels of abstraction. This unfortunately separates us from our silicon-based friends, who prefer to see things in much finer, absolute terms; in other words, they speak a low-level language of their own. Naturally, high-level code must eventually be reduced to low-level code in order for a CPU to execute it, so you use a program called a compiler to handle this translation. The end result is the same program, differing only in the way it's described.

> **NOTE**
>
> **Technically, XtremeScript isn't exactly a C subset; in addition to implementing a smaller portion of the C language, it also introduces a few of its own constructs and features, and makes subtle changes to some of C's existing aspects. Either way, the language is clearly influenced heavily by C, so we might as well use the term.**

XtremeScript, while also the name of our future scripting system as a whole, is more precisely the name of the high-level language that the system is based around. XtremeScript is what's known as a *Csubset language,* meaning it implements the majority of the C language you already use (but not quite all). This is great news because it means you can write your script code in almost the same language you'd use to write a game engine itself. The downside, however, is that C is a complex language, and writing a program that compiles C code anything but a trivial task. The extra effort involved, however, will be more than worth it in the end. In many ways, XtremeScript is also very similar to other scripting languages like JavaScript and PHP. If you have experience with either of these, you'll feel right at home.

In short, high-level code is what you write scripts with. A compiler translates it to a low-level code, which can then be easily executed.

Low-Level Code

Low-level code, which most commonly refers to assembly language and machine code, is a way to directly control a processor such as your central processing unit, floating-point processing unit, or virtual machine (which is what you're interested in). In order to maximize speed and minimize memory requirements, low-level code consists of very simple instructions that, although of limited use on their own, can be combined to solve problems of limitless complexity. For an example of what low-level code is like, check out the following example.

Here's some C code to execute a simple assignment expression:

```
A = ( B + C ) * 8 / 5;
```

Here's the same line of code after being reduced to a generic assembly language:

```
mov Tmp, B
add Tmp, C
mul Tmp, 8
div Tmp, 5
mov A, Tmp
```

Notice that the assembly version is, to put it in rather primitive terms, only doing "one thing" per line. Although the C version can handle not only the entire expression but also the assignment with only a single line, the assembly version requires five. To briefly explain what's actually going on here, assume that `Tmp` is a temporary storage location of some sort (often called a *register*). First `B` is moved into `T` (notice that this notation places the destination (`Tmp`) *before* the source (`B`)). `C` is then added to `Tmp`, so the temporary location now holds the sum of `B` and `C`. This sum is then multiplied by 8 and divided by 5. With the expression completed, `Tmp` now holds the final result, which is assigned to `A` with another `mov` ("move") instruction.

Assembly language isn't particularly difficult to code with once you're used to it, but it should now be easy to understand why C is the preferred choice in most cases. The good news is that, for the most part, all of your scripting will be done in XtremeScript rather than assembly. Although

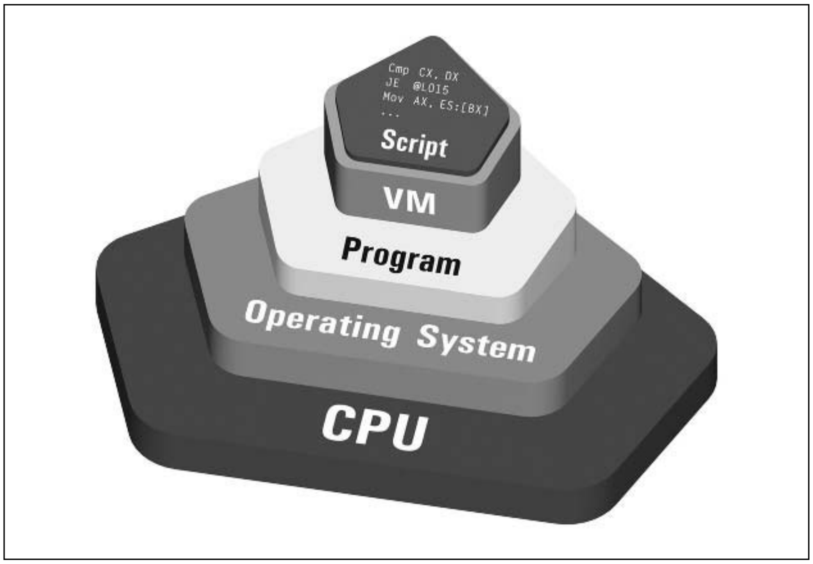
PC developers often turn to assembly language coding for an extra speed boost when maximum performance is required (such as in the case of graphics routines), scripts stand to gain little from it by comparison.

In accordance with my continuing theme of borrowing syntax from popular languages to make your script system as familiar and easy-to-use as possible, the assembly language of the XtremeScript system will be loosely based on the Intel 80X86 syntax that you might already be familiar with. We'll indeed take a number of creative liberties, but the Intel syntax will be preserved whenever possible. Once again, this eases the transition from writing engine code to writing script code in a game project and helps keeps things uniform and consistent.

Lastly, low-level code designed specifically to run on a virtual machine is often referred to as *bytecode*; this is an important term, so keep it in mind.

The Virtual Machine

With the two major languages involved in your scripting system accounted for, the last piece of the puzzle is the runtime environment. The virtual machine ultimately makes your scripts usable because XtremeScript code isn't compiled to the machine code of a physical processor such as the 80X86. To reiterate what you learned in Chapter 1, recall that the general purpose of a VM is to run code "on top" of the hardware CPU. It allows scripts to control the game engine just as the interpreter component of your command-based script module did, albeit in a far more sophisticated manner. See Figure 5.2.



```
Cmp  CX, DX
JE   @L015
Mov  AX, ES:[BX]
...
```

Figure 5.2

When virtual machine code (bytecode) runs inside the VM, it's said to be running on top of the CPU, rather than inside it. This once again refers to the "levels" that you use to describe languages; just as C is a higher-level language than assembly, XtremeScript bytecode is a higher level language than 80X86 machine code.

The XtremeScript virtual machine closely mirrors a hardware-based computer in many ways. For example, it provides its own threading system to allow multiple scripts to run simultaneously; it manages protected memory and other resources required by a running script; it allows scripts to communicate with one another via a message system; and perhaps most importantly, it provides an interface between scripts and the host application (the game itself), allowing the two to communicate easily. Figure 5.3 is a diagram of the VM's general layout.

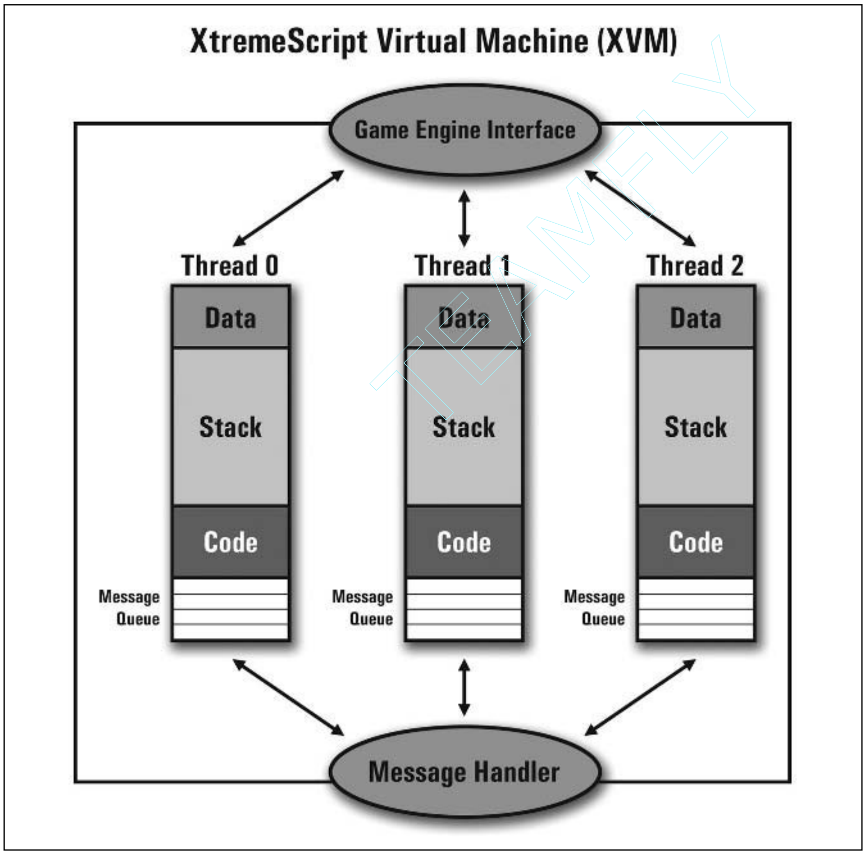


Figure 5.3

The basic layout of the XtremeScript virtual machine.

Because the VM is designed to run inside a host application rather than directly on the CPU, it makes the scripts themselves completely platform independent. For instance, if you create and release a game for Windows, and later decide to port it to Linux, the game's scripts will run without modification once the game engine and virtual machine have been rewritten for the new platform. This is also how Java achieves its platform independence—the JVM (Java Virtual

Machine) has been written for a vast number of systems, allowing Java code to run on any of them without rewriting a single line.

The XtremeScript Virtual Machine, referred to as the XVM, will be implemented as a static library that can be dropped into any game project with minimal setup. It will be highly portable from one project to the next, making it an invaluable tool in your game development arsenal.

A Deeper Look at XtremeScript

Now that you understand the most fundamental layout of the XtremeScript system, let's look a bit closer. As mentioned, a scripting engine such as the one you're going to build is naturally a highly complex piece of software, so the best way to learn how it works is to take a "top-down" approach, wherein you start with the basics and slowly work your way towards the specifics. In the last section, you learned that the XtremeScript system is based on three major entities: the high-level language that scripts are written in, the low-level language that scripts are translated into by the compiler, and the virtual machine that executes the low-level language version and manages communication with the host application (the game). The next level of detail will bring into focus two new topics—what these basic components are themselves made of, and specifically how they interact with each other.

Each of these elements is of course covered extensively in their own dedicated set of chapters later in the book, but before you get there, you're going to learn how they interact with each other and why they're individually important. In order to do that, we'll now look at the complete process of turning a text-based script into a compiled, binary version running inside the VM. Along the way you'll see why each component is necessary and what each is composed of.

The basic process, as you might have already gathered, is as follows:

1. Write the script using the XtremeScript language in a plain text file.
2. Compile the script with the XtremeScript compiler. This will produce a new text file containing the assembly language (low-level) equivalent of the original high-level script.
3. Assemble the low-level script with the XtremeScript assembler. This will produce a binary version of the low-level script in XVM machine code.
4. Link the XVM static library into your game engine.
5. At runtime, load the binary script file. The XVM will now process the machine code and the script will execute.

Figure 5.4 illustrates this process in a bit more detail.

That's definitely more complicated! But before your head explodes, let's get right down to what's going on in this diagram.

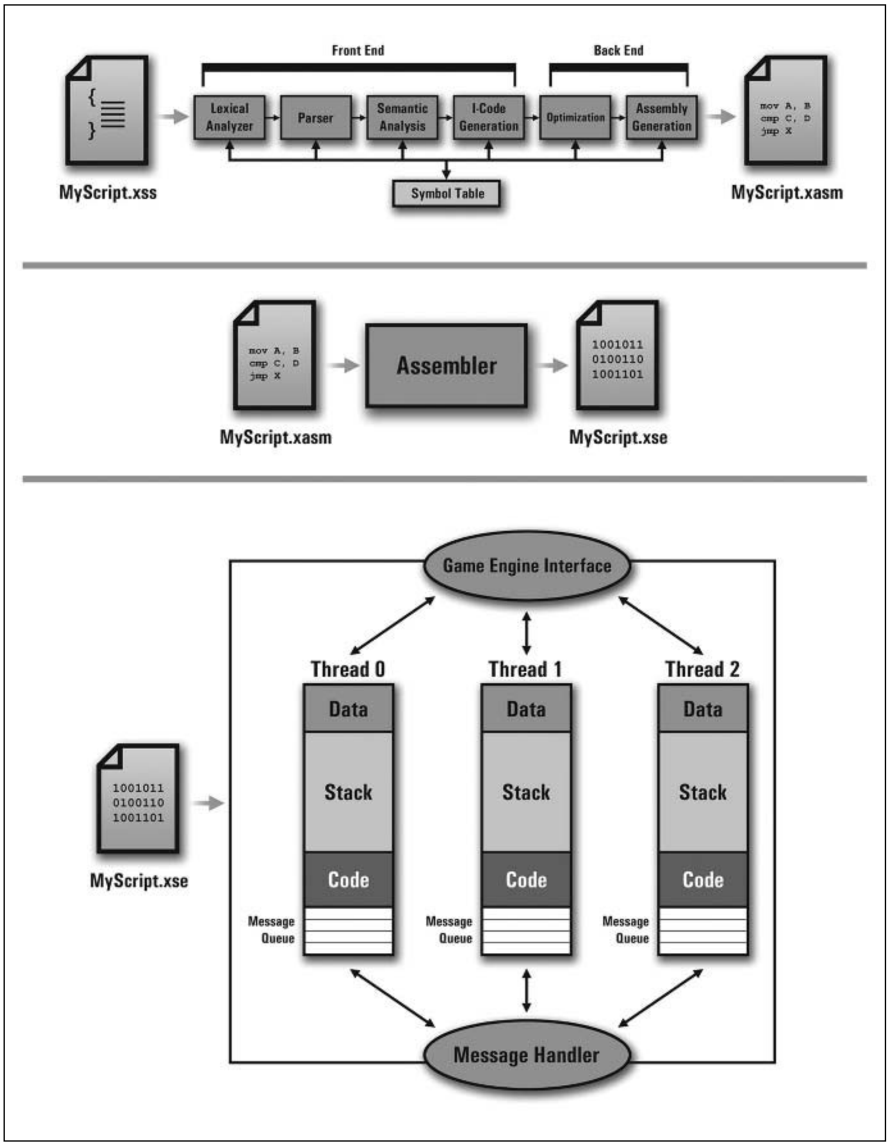


Figure 5.4

A slightly more complex look at the lifespan of a script in the XtremeScript system.

High-Level Code/Compilation

Once again, you can start with the high-level code. This is without a doubt the most profoundly convoluted step in the entire process of passing a script through the XtremeScript system, and that's no coincidence. In all of computer science, the most difficult problems faced by software engineers are often the ones that deal with the complexities of the interface between humans and computers. Natural language synthesis, image recognition, and artificial intelligence are but a few of the fields of study that have puzzled programmers for decades. Not surprisingly, the area of scripting that involves understanding and translating a human-readable language like C (or a derivative of that language like XtremeScript) is significantly more complex than understanding the lower-level steps, which operate entirely on computer-friendly code and data. The complexity of this step is proportional to its significance, however; the purpose of building a system like this in the first place is to the convenience and flexibility of scripting with high-level code. Without this first step, you probably wouldn't waste your time building the rest.

There are two major entities in the high-level portion of your scripting system. First you have the XtremeScript language itself, and second, the compiler that understands it and translates it to assembly. Designing the language will be a relatively easy job; all you really have to do is pick and choose the features you like from C, add a few of your own, and put this together in a formal language specification that you can refer to later. The compiler, on the other hand, is orders of magnitude more difficult to implement. In order to build it, you have to delve into the complex and esoteric world of *compiler theory*, the field of computer science that deals specifically with translating high-level languages. Compiler theory has earned something of a bad reputation over the years; many programmers simply look at the complexities of a language like C or C++ and immediately assume that the idea of writing software that would understand it is a virtually insurmountable task.

Make no mistake—compiler theory is hard stuff, and you're going to learn that fact first hand. But it's not *that* hard. In fact, as long as a compiler project is approached with a great deal of planning, meticulously structured code, and a little patience, anyone can do it. So, to get your feet wet and shed the first rays of light on this shadowy and mysterious topic, let's look at the basic breakdown of a compiler. You know the compiler accepts a text file containing source code, and spits out a new file containing either assembly language or machine code (which is almost the same thing), but what's going on between those two ends of the pipeline? Figure 5.5 shows an excerpt of Figure 5.4, this time focusing on the steps the compiler takes.

NOTE

This chapter explores a third component in the high-level world as well, but it is mostly lumped together with general compiler theory. It's the *preprocessor*, an incredibly useful utility introduced in the last chapter, and one you no doubt have extensive experience with as a C programmer. You'll most likely be taking advantage of a few of the more common preprocessor directives, such as `#include` for combing separate source files at compile time, and `#define` for creating constants and macros.

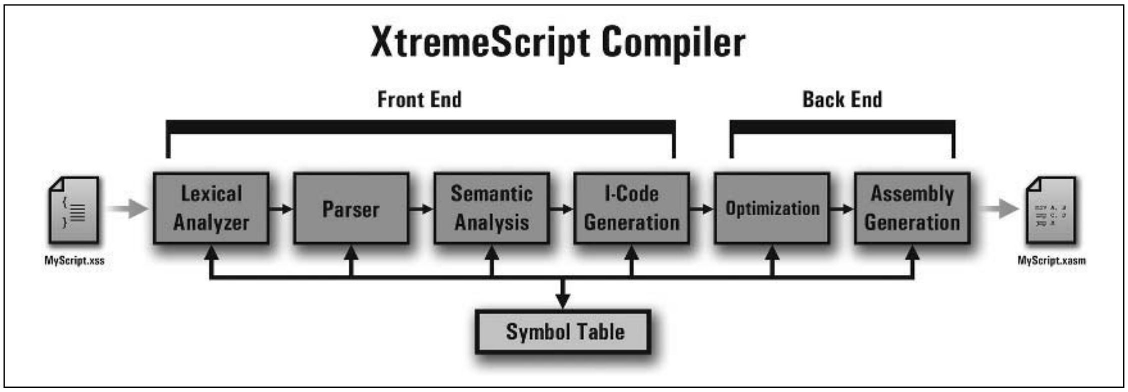


Figure 5.5

The basic steps taken by a compiler in order to translate high-level code into assembly language or machine code.

Lexical Analysis

The first and most basic operation the compiler performs is breaking the source file into meaningful chunks called *tokens*. Tokens are the fine-grained components that languages are based on. Examples include reserved words like C's `if`, `while`, `else`, and `void`. Tokens also include arithmetic and logic operators, structure symbols like commas and parentheses, as well as identifiers like `PlayerAmmo` and immediate values like `63` or `"Hello, world!"`. Lexical analysis, not surprisingly, is performed by a component of the compiler called the *lexical analyzer*, or *lexer* for short. In addition to recognizing and extracting tokens, the lexer strips away any unnecessary or extraneous content like comments and whitespace. The final output of the lexer is a more structured version of the original source code.

Parsing/Syntactic Analysis

With the source code now reduced to a collection of tokens, the compiler invokes the *parsing* phase, which analyzes the syntax of token strings. Token strings are sequences of tokens that form meaningful language constructs, like statements and expressions. For example, consider the following line of code:

```
if = ( void + ) ;-; 96 X
```

This would pass through the parser without a problem because it's composed entirely of valid tokens. However, as is clearly visible just by looking at it, it's not even close to following the rules of syntax. Parsing is one of the most complex parts of compiler construction, and can be approached in a number of ways. The parser often outputs what is known as an AST, or *Abstract*

Syntax Tree. The AST is a convenient way to internally represent source code, and allows for more structured analysis later.

Semantic Analysis

Although the syntax of a language tells you what valid source code looks like, the *semantics* of a language is focused on what that code means. Let's look at another example line of code:

```
int Q = "Hello" + 3.14159;
```

The syntax here is correct, and thus the parser won't have a problem with it. According to pure syntax, all you're doing is adding two values and assigning them to a newly declared identifier. The semantics behind this line of code, however, are invalid; you're trying to "add" a string value to a floating-point value and assign the "result" to an integer. Obviously, this doesn't make any sense and the source file needs to be rejected. After the semantic analysis phase, the internal representation of the source code is guaranteed to be correct, so you're ready to get started with the actual translation. Be assured that at this point, a lot of the really hard stuff is over with.

Intermediate Code Generation

Now that you have a fully validated internal representation of the source code, you can take the first step towards reducing it to a lower-level language. Instead of directly converting it to a specific assembly language, however, you're going to translate it to what's known as *intermediate code*, or *I-code*. I-code is something of a conversion halfway between the source language (XtremeScript in this case) and the target language (XVM assembly). I-code lets you work with a version of the source code that is very similar to assembly, and might be almost identical in this case, but is still not necessarily tied to any target machine, like the XVM. You can instead save all of your machine-specific alterations to the code for later steps.

Optimization

One of the final phases of compilation is an optional but extremely important one. Hand-written assembly from an experienced low-level coder usually yields the highest performance and requires the least amount of space. Common algorithms and operations, especially when part of a loop, usually end up being somewhat redundant because of their high-level, abstract nature. When the low-level code that performs these tasks is written directly by the programmer, these patterns are easily noticed, and can be truncated or rewritten to achieve the same result with less code. Compilers, however, have a much harder time recognizing these patterns and usually produce code that isn't quite as efficient as their hand-written equivalent. As a result, compilers are expected to optimize the code they produce whenever possible. The study of compiler-driven optimization has been expanding for decades, and today's compilers can often produce code that performs at virtually the same level as the code written by a human (or better). In this case, optimization is far less important, however. The speed overhead associated with scripts is so great (relative to native machine code like 80X86, that is) that the difference between optimized and unoptimzed script code is usually unnoticeable. Regardless, it's still a topic worth exploring.

Assembly Language Generation

The final step, of course, is converting optimized I-code to assembly language. In the case of scripts running on a virtual machine, this is really a rather simple step. I-code instructions usually have a nearly one-to-one mapping with the compiler's target code, so this phase is pretty simple. Once this is done, compilation is finished and a high-level script has been reduced to a low-level one.

The Symbol Table

Throughout the process of compilation, a data structure called the *symbol table* is used extensively. The symbol table stores information about the script's identifiers; function names, variable names, and so on. In addition to the identifier's name, its value, data type, and scope are also recorded (among many other things). The symbol table is an extremely important part of the compiler, which should be evident by its widespread use among the compiler's various phases.

The Front End versus the Back End

The phases of compilation can be separated into two extremely important groups. These are the *front end* and the *back end*, and are separated by the generation of intermediate code. The purpose of the front end is to translate a high-level source language to I-code, whereas the purpose of the back end is to reduce that I-code to a low-level target language. The beauty of this approach is that the source and target languages can be changed simply by swapping their respective ends. For example, if you wanted your compiler to accept Pascal source rather than XtremeScript, you'd simply rewrite the front end to lex and parse Pascal. If you wanted to generate code for the Intel 80X86 rather than the XVM, you'd rewrite the back end. This is why I-code is designed to have such a generic structure.

This wraps up the look at the high-level world of XtremeScript. To reiterate, the compiler and its associated language are the two most complex aspects of virtually any scripting system, but are also the most useful. Although the remaining elements are by no means trivial, few would disagree that they pale in comparison to the difficulty involved in implementing the high-level entities.

At this stage, you can compile XtremeScript code, but the output is an ASCII assembly language file. This will once again have to be translated to a lower-level language in order to create the executable scripts you're after, so let's move on to the next step in the process.

Low-Level Code/Assembly

Turning an ASCII-formatted assembly language source file into a binary, machine-code version is far simpler than compiling high-level code, but it's still a reasonably involved process. This process is called *assembly*, and is naturally handled by a program called an *assembler*.

The Assembler

Assembly language is significantly simpler than higher-level code for obvious reasons. One of the major differences is that low-level code doesn't perform iteration through abstract structures like `while` and `for` loops. Rather, basic comparisons are made involving two operands and the results determine whether a *jump* is made to a given line label. Jumps in assembly language are analogous to the frowned-upon `goto` keyword in C. `goto` might be considered poor programming practice in higher-level contexts, but it's the very foundation of low-level branching and iteration.

Jumps also provide the only real complexity in the assembly process. Assemblers spend most of their time simply reading each instruction and converting them to their numeric equivalent (called an *opcode*). The size of opcodes varies, however, depending primarily on the number and types of parameters they accept. Because of this, the size of a given block of instructions can be hard to determine until after the assembly process. In order to translate a jump, however, the distance from the jump instruction to its target instruction must be known. As a result, many assemblers employ a *two-pass* approach. The first pass reduces every instruction to an opcode, whereas the second pass finalizes jumps by calculating the distance to their target instructions.

The Disassembler

Disassemblers are nifty little utilities that can reverse the process of an assembler. By mapping numeric opcodes to their instruction mnemonics, rather than the other way around, an assembled binary script can be converted back to its human-readable, assembly language, equivalent. Disassemblers are commonly used for reverse engineering, hacking compiled programs, and other less-than-mainstream activities. It might not come as a surprise, but they'll be of very little use in this scenario. There's really no need to reverse engineer a system you've built yourself (unless a sharp blow to the head leaves you with a bad case of amnesia), and it's unlikely that you'll ever have to "hack" into your own scripts. Because of this, you're left to implement a disassembler on your own if you're interested (which you'll be more than capable of doing after chapter 9).

The Debugger

Bugs are often considered the crux of a programmer's existence (especially mine). Due primarily to our error-prone nature as humans, as well as the complexity of computer systems, bugs play a pivotal and recurring role in the development of software. Although programmers still usually spend far more time debugging a program than they do writing it, many tools have been invented to help ease and accelerate the process of hunting bugs down and squashing them. These tools are called *debuggers*.

In the low-level world, debuggers usually work by loading an assembly language program into memory and letting the user step through it, instruction by instruction. As each instruction is executed, its operands are listed and the state of the virtual machine is presented in an organized manner. For example, memory maps can be displayed to let the users monitor how and where memory is being manipulated, or the contents of the stack can be illustrated in a literal stack format to allow the users to watch the stack grow and shrink and take note of incoming and outgoing values.

Debuggers are similar to virtual machines in the sense that they provide a runtime environment for scripts. The main differences are of course that debuggers are meant to be used for development purposes only; they generally don't provide the intended output of the script, but rather present a visual representation of its existence in memory at runtime. They're also far less performance-critical, because debugging is usually a slow process that's meant to be taken one step at a time (no horrific pun intended).

Lastly, there exist a number of popular variations on the simple debugger discussed here. For example, many compilers can optionally output a *debug version* of the executable containing extra information that can be specifically utilized by debugging programs. This can include line numbers from the original source code, identifier names, comments, or anything else that the compiler normally discards somewhere along the way but that might prove useful while analyzing the code within the confines of a debugger. Many development packages take this a step further by displaying the original high-level code in between blocks of assembly to provide the most accurate depiction of how source code behaves at runtime.

With both the compiler and assembler in place, you can produce binary, executable scripts from text-based source files. This is the brunt of the work involved in building a scripting system, but you still need something to actually execute those scripts with.

The Virtual Machine

The final piece of the puzzle is, as always, the virtual machine. The VM, like the command-based script module from the last two chapters, is a fully embeddable software component that can be easily dropped into a game project with little to no modification. It's implemented in this book as a static library, but a dynamically linked library would certainly have its benefits.

Although you've already learned about the XVM for the most part, there are a few things that could use some elaboration. For instance, w haven't really decided on how exactly a script will communicate with the host application. You know that one of the primary features of a VM is its interface with the game engine, but how this will actually work is still something of a mystery.

In almost any software system, an *interface* between two entities is usually embodied by a collection of exposed functions. By calling one of these functions, you're in essence "sending a message" to the entity that exposes it. For instance, if the script wants to know how much ammo the player has, it requests that information by calling a function exposed by the game engine called `GetPlayerAmmo ()`. It's equally likely that the game will need to call one of the script's functions as well. This is very important in the case of event-based scripting, in which case the script might provide a function pointer to the game engine that would then be used to tell the script when a given event has taken place. As an example, the script for an enemy character might give the game engine a pointer to a function called `HandleDamage ()` that would then be called every time the enemy is shot or otherwise damaged. This is called a *callback*, because the runtime environment is calling one of the script's functions "back" after previously having a pointer to it. The collection of functions the game engine exposes is called it's API, or *Application Programming Interface*.

Another serious issue in the case of virtual machines is security. As was mentioned briefly in the first chapter, scripts can wreak some pretty serious havoc when left unchecked. Buggy code can just flip out and lock the game up by overwriting the wrong memory areas or losing itself in an endless loop, whereas malicious code can intentionally cause problems in the same manner. If a script crashes and the virtual machine isn't there to handle the situation, the game engine can often go down with it. This is an undesirable situation, so a number of measures should be taken to prevent it whenever possible. This can include "loop timeouts" that attempt to provide a timely end to otherwise infinite loops by imposing a limit on the number of iterations they can cycle through, and of course memory protection such as monitoring the reading and writing of a given script to make sure it stays within its allocated address space.

Recursion can also quickly spiral out of control, so stack space should be carefully monitored. In the event that something does go wrong, the virtual machine will at least have a good idea of what it was and where it happened, allowing a graceful cleanup or exit.

The XtremeScript System

You now have a good idea of how this script system is going to work. You've looked at the high-level and low-level languages and utilities, the virtual machine, and the details regarding the interface between scripts and the game engine. The following summary outlines the major features and primary details of the XtremeScript system. This will be the starting point in the process of implementing it.

High-Level

The high-level aspect of XtremeScript can be summarized with the following points:

- Based around XtremeScript, a C-subset language our scripts will be written in. The language will be designed to resemble C and C++ as much as possible, in order to keep the environment familiar to the programmer.
- High-level code will be compiled with the XtremeScript compiler and translated to an ASCII-formatted assembly source file ready to be assembled.
- A preprocessor will be included to deliver many of the popular directives C programmers are accustomed to.
- High-level code will provide the human interface to the underlying script system.

Low-Level

Below the high-level components of the system lies the lower-level:

- Based around a simple assembly language with Intel 80X86-style syntax. Once again, a similar syntax is intended to keep things uniform and consistent.
- Assembly language is assembled into binary, executable scripts composed of bytecode with the XtremeScript assembler.
- Additional utilities include a disassembler that converts executable scripts back to ASCII-formatted assembly source files, and a simple debugger that provides a controlled and interactive runtime environment for compiled scripts.

Runtime

Lastly, the system is rounded out by its run-time presence:

- Scripts are executed at runtime inside the XtremeScript Virtual Machine, or XVM.
- The XVM is an embeddable component, packaged in a static library that can be easily linked to a game project.
- The XVM provides an interface between running scripts and the game engine through an API consisting of game engine functions that scripts can call. Scripts can expose functions of their own, allowing the game engine to perform callbacks. This is primarily useful for trapping events.
- Multiple scripts can be loaded and run simultaneously.
- Scripts can communicate with one another via a message system. This can be useful in the case of multiple enemy scripts that need to coordinate themselves with one another, for instance.

- Each running script is given a protected environment with its own block of memory, code, stack space, and message queue. Scripts cannot read or write outside of their own address space, ensuring a higher-level of stability.
- Other general security schemes can be put in place, such as loop timeout limits.

That pretty much wraps things up. This list, although superficial, will provide an adequate road map for the coming chapters. These components really are significantly more complex than what's listed here, but this should be enough to get you started with the general order of things.

Summary

This chapter has practically sent you through a time warp. Only a few pages ago you were applying the finishing touches to your modest, charming little command-based script module, and already you've taken your first major step towards designing and implementing a true scripting system with a C-based high-level language and numerous components and utilities.

The remainder of this section of the book focuses on the more general topics of procedural scripting systems. In the next chapter you're going to be introduced to a few of the most popular scripting systems in use today and learn how to integrate them with your own programs. You might even pick up an idea or two for XtremeScript.

After that, you're going to take a look at C, C++, and a number of other high-level languages. As you look through their design and function, you'll start to nail down the features you need and don't need in order to script games. From this list, you'll be able to draft up a formal language specification for XtremeScript. You'll also add a few of your own ideas, and the end result will be a detailed blueprint that will come in very handy when the compiler theory section rolls around.

If nothing else, the one thing you should have picked up in this chapter is that you have a long road ahead. Fortunately, you're going to learn so much along the way that every last step will be more than worth it. And, as you've learned throughout this chapter, the end result will be a powerful, versatile system that will prove useful in countless future projects.

You're encouraged to read this chapter more than once if even the slightest detail seems a bit fuzzy. Remember, you can sweat most of the details you've covered so far; you obviously can't be expected to truly understand the phases of compilation or the details of the XVM architecture just yet. I included it all to give you an idea of the complexity behind what you're doing. What you do need to know, however, is how these general pieces fit together. That's the most important thing.

Aside from that, roll up your sleeves—the real fun is just getting started!

This page intentionally left blank

CHAPTER 6

Integration: Using Existing Scripting Systems

"This will feel... a little weird."
——*Morpheus,* The Matrix

The last chapter introduced you to scripting in a more technical manner through a general overview of how the pieces fit together, with a focus on exactly how they do so in XtremeScript. Armed with this information, you're now ready for your first hands-on encounter with "real" scripting, which will be the integration of some of the more popular existing scripting systems with a graphical C program.

In this chapter, you're going to:

- Learn about the concept of integration and the use of abstraction layers to facilitate communication between separate entities.
- Take a tour of three popular scripting languages—Lua, Python, and Tcl—and learn enough about them to write reasonably powerful scripts.
- Learn how these scripting systems are integrated with C programs and, combined with your knowledge of their respective languages, use them to control a small, graphical host application.

Integration

Before getting into the details of how to use these existing scripting systems, you need to master the concept that underlies the use of all of them— *integration.* Integration, to put it simply, is the process of taking two or more separate, often unrelated entities and making them communicate and work together for some common goal. You can see examples of integration and its importance all throughout the software world—3D rendering and modeling packages often extend their functionality through the use of plug-ins; Sun's *Java Connector Architecture* allows modern, Java-based application servers to talk to legacy enterprise information systems to make corporate transaction records and inventory catalogs available on the Web; and of course, game engines communicate with scripting systems to allow game designers and players to provide game content and modifications in an external and modular fashion. See Figure 6.1.

Generally, the biggest challenge involved in integrating two things is establishing some sort of channel through which they can easily and reliably communicate. This provides the foundation for everything else, as virtually any facet of an integration project will ultimately rely on the capability for entity X to talk to entity Y and receive a response.

The solution to this problem lies in an age-old software-engineering concept known as the *abstraction layer.* An abstraction layer, also known as an *interface,* is any software component that sits

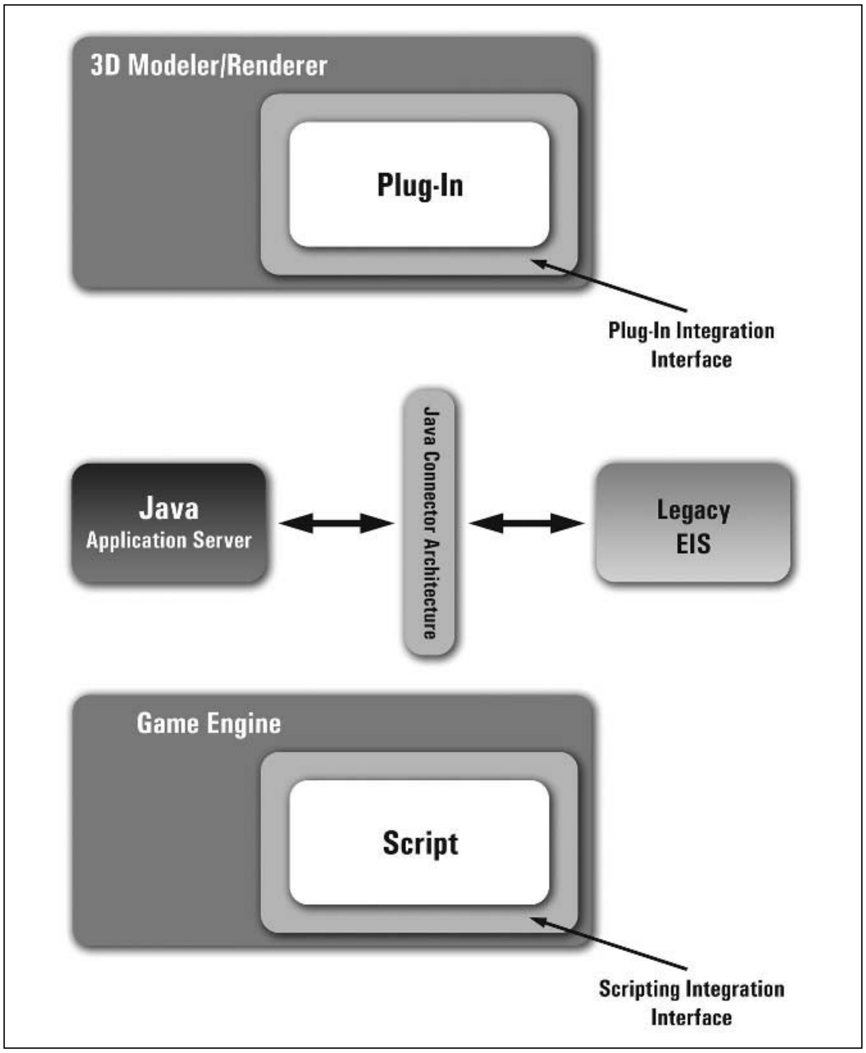

3D Modeler/Renderer
Plug-In
Plug-In Integration Interface
Java Application Server
Java Connector Architecture
Legacy EIS
Game Engine
Script
Scripting Integration Interface

Figure 6.1

Examples of integration.

between two or more entities, interpreting and routing their input and output instead of letting them communicate directly (which may not even be possible). To understand this concept better, consider the analogy of a human translator. A translator for English and Japanese, for example, is someone who is fluent in both languages and allows English-only speakers to communicate with Japanese-only speakers by listening to what the first party has to say in one language, and repeating it to the second party in the other. The process works both ways, and the end result is that the two parties can easily communicating despite an otherwise impenetrable language barrier. This process is illustrated in Figure 6.2.

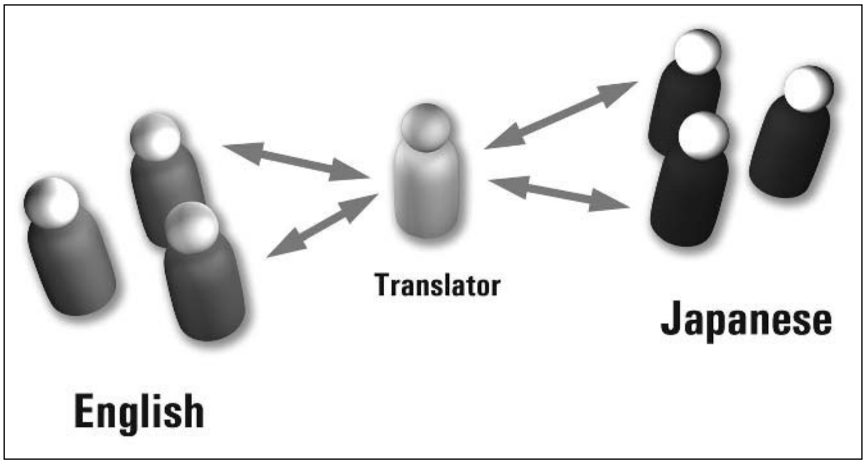


Figure 6.2

A conceptual diagram of two parties communicating through a translator.

It's called a *layer* because, for example, the translator is "wedged" in between the English and Japanese speaking parties, much like a layer of adhesive sits between two surfaces. It's considered *abstract* because neither entity knows all the details of the others; in this case, the Japanese speakers don't know English, and the gai-jin don't know Japanese. Regardless, thanks to the translator, they can communicate as if this issue didn't even exist. To either side, the process of inter-language communication has been *abstracted* to something far simpler. Rather than having to spend years upon years attaining fluency in the language of the other party, both parties can carry on in almost the exact same manner they usually would, while still getting the job done.

Bringing this example back to the context of game scripting, the reason you need an integrating layer of abstraction between a script and the game engine is because neither the scripting language nor C has built-in facilities for "talking" to the other. In computer science terms, phrases like "talking to" and "sending messages between" software entities generally mean calling functions. In other words, if you have two programs in memory, each of which has a number of functions for receiving input and producing output, these two programs can communicate rather easily by simply calling each other's functions. Anyone who's done any reasonable amount of Windows programming should have plenty of experience with this (think callbacks). Check out Figure 6.3 for a more visual explanation.

When Program X calls one of Program Y's functions, it's talking to it. When Program Y returns a value, or calls one of Program X's functions, it's talking back. So, it seems that in order for a script written in, say, Python, to communicate with the game engine written in C, all they need to do is call each other's functions and everything will work out. The problem is, there are no built-in provisions for doing this. Even if you define a function in your Python script called `MovePlayer`

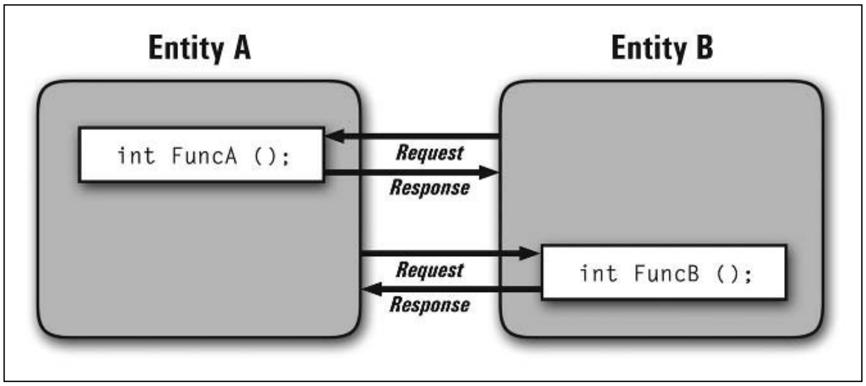


Figure 6.3

Software entities communicate with each other by calling functions.

`()`, which accepts two numeric values for moving the player along the X- and Y-axes, the following code certainly won't compile in C:

```
Int X = 16,
    Y = 32;
MovePlayer ( X, Y );
```

Why not? Because from the perspective of your C compiler, `MovePlayer ()` doesn't exist. More importantly, even if the compiler knew about the function, how would the function be called? Python and XtremeScript, like all scripting languages, are not compiled to machine code. Unlike the C functions, there is no block of native assembly language in memory that implements the logic behind the `MovePlayer ()` function. Rather, this function is represented as a different, assembly-*like* format that exists in and can be executed by Python's runtime environment and nothing else. Your poor C compiler wouldn't know what to do with the function call either way. Figure 6.4 illustrates this.

Likewise, how is your Python script going to talk to C? Just as your compiled C program runs directly on the machine and expects the functions it calls to exist in the physical "world" of, for

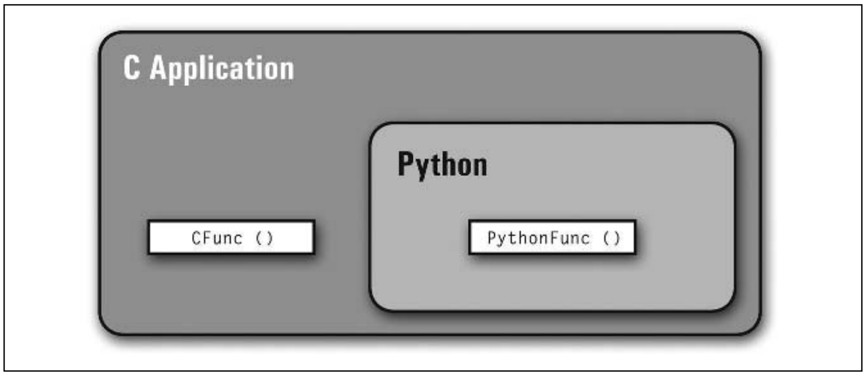


Figure 6.4

The problem: C and Python (or any scripting language) exist in separate runtime environments, and therefore have no way of directly talking to one another.

example, 80x86 machine code, Python expects just the opposite and deals only with other Python scripts, which are far more high-level and "virtual" because they run inside the Python runtime environment. The problem is that these two languages exist in "parallel dimensions" so to speak, and therefore have no intrinsic methods of communication.

If you're in the mood for a fairly out-there example, consider the following. Many scientists in the quantum mechanics and physics communities believe that the universe exists in a larger *multiverse*; a "collection" of presumably infinite separate, parallel universes. This means that while you may live on earth, a person just like you may also live on a "different" earth—one that resides in another universe. As it stands now, there's no way for you to talk to your alter-ego in this dimension, just like C can't communicate with Python. However, if we can find a way to reach out of, or *transcend*, or own universe, we might be able to establish a means by which multiple universes can communicate with each other. Although I've admittedly taken more than a little dramatic license here, this is in essence the same thing you're trying to do with C and the scripting system of choice. Of course, the integration of scripting systems is probably a lot less likely to make its way into an episode of the *Twilight Zone*.

Coming back down to earth, this is where the handy translator comes back into the picture. It may no longer be a problem of English vs. Japanese, but as you've seen, any time two or more software components are having trouble communicating, an abstraction layer can solve the problem by providing a common ground of some sort. The problem, to put it specifically, is that you need the scripting system to call C functions and vice versa, but have no way of doing so.

To figure this out, let's look more carefully at exactly what the translator does. When the English party says something to the translator, the spoken phrase is recognized and understood by the translator's brain, and then converted to its corresponding equivalent in Japanese. These new Japanese words are then spoken by the translator, and are subsequently understood by the Japanese party. The reason I've phrased this in such detail is that it's almost an exact analogy for the abstraction of inter-language function calls. The key to remember here is that the exact sound waves that are produced in English are *not* the same waves that the Japanese party ultimately understands. Likewise, the Python system will not receive the exact same function call that was sent out by the C program when it comes time for the two to communicate. Rather, it will receive a *translated* function call that was sent by the abstraction layer. The same is true conversely.

To put it simply, the abstraction layer will be assigned the job of sitting in between C and Python. This layer is capable of understanding function calls from both C and Python, and likewise, is capable of issuing them as well. So, when Python wants to talk to C, it instead calls the *abstraction layer's* functions for sending a message. The abstraction layer will then make a new function call of its own, but one that conveys the same message, to the C program. This new function call will be understandable by C, and the message will have traveled from the script to the game engine. Naturally, the process is reversed when C wants to talk to Python. Have a look at Figure 6.5.

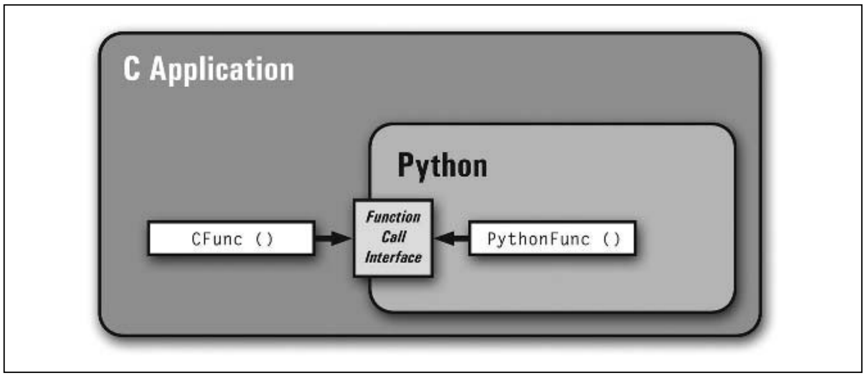


Figure 6.5

Python and C can communicate thanks to an abstraction layer that receives and translates function calls.

Again, this is an *abstraction* because Python and C still haven't learned how to talk to each other. Rather, they've simply learned how to talk to a translator, which in turn is capable of talking to the other party for them.

IMPLEMENTATION OF SCRIPTING SYSTEMS

Generally, a scripting system is implemented in the form of a static library or something similar, although a dynamic library like a Windows DLL would work just as well and in roughly the same way. This library usually contains two crucially important components, both of which are necessary to fully enable the scripting process. The first and most obvious component is the runtime environment (also known as a *virtual machine*, a term you should be familiar with by now), which is capable of loading scripts in the system's language, such as Python or Tcl. Once loaded, the runtime environment either automatically begins execution of the script, or waits for the host application to give it the green light. The other component is the interface that allows it to talk to the host application and vice versa. This is of course the abstraction layer. The host application is then linked with this library, and the resulting executable is capable of being externally controlled by scripts. When a scripting system is encapsulated in this way for easy integration with host applications, it's an *embeddable* scripting system, because it "embeds" itself into the host in the same way a 3D graphics card is "embedded" into your computer, or a pacemaker is "embedded" into your body.

Scripting *languages* vary in their details quite a bit from one to the next, but scripting *systems* themselves are almost invariably written in C or C++. This means that the runtime environment that runs the Python script, as well as the interface that allows it to talk to the game engine, are both written in a language that the engine is directly compatible with. Because a C program can easily talk to a C library, that's one side of the C-Python interface taken care of already. The other half of the puzzle is also easily solved because the Python library not only physically contains the Python script, but has records of all of its relevant information—including data about what sort of functions the script defines as well as how to call them. This information, coupled with the fact that it already has an intrinsic connection to the C host application, explains exactly how function calls can be translated back and forth from the script to the host.

In other words, both the C program and the Python script can now break up their function calls into two groups. First are traditional calls that work within their respective environment; C calls to C functions, and Python calls to Python functions. These are called *intra-language* function calls. The second group consists of calls from the host that are intended for Python and calls from Python that are intended for the host (*inter-language* function calls). Because neither of these function calls go directly from Python to C or vice versa, they all really just boil down to calling the Python library and requesting it to translate the message. Check out Figure 6.6 to see this in action.

The API provided by the typical scripting system library are pretty much what you would expect; functions for loading and unloading scripts, functions that tell a given script to start or stop

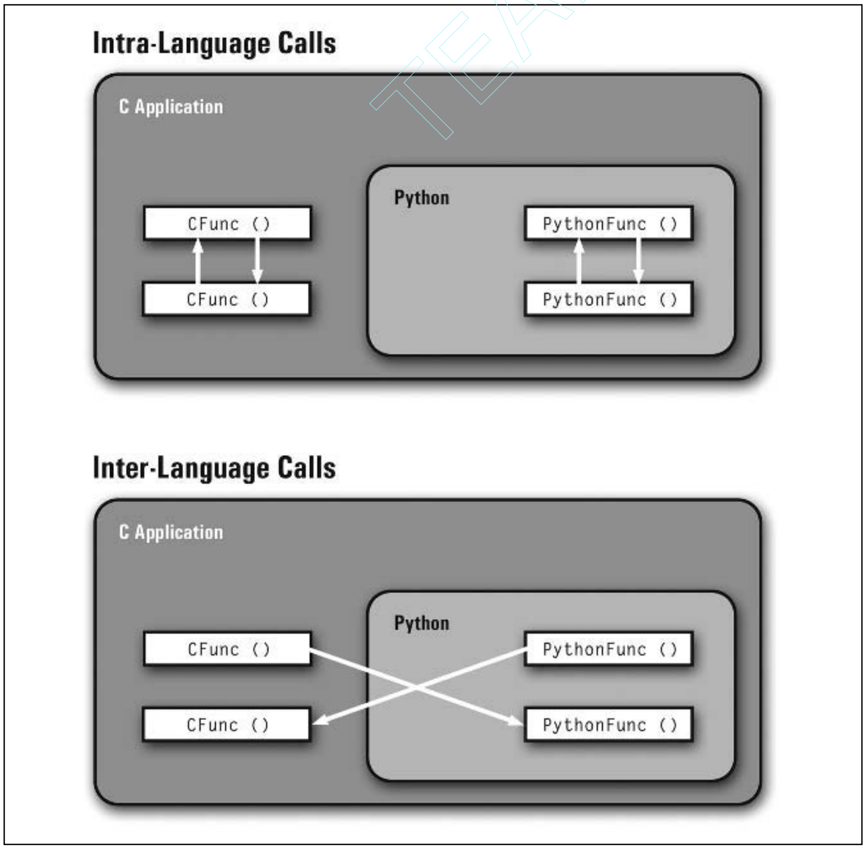


Figure 6.6

There are now two types of function calls to consider; those that exist within a given runtime environment, and those that are meant to cross the boundaries between Python and C.

running, perhaps a few general functions for initializing and shutting down the runtime environment itself, and of course, functions for calling other functions defined by the script. If you write a script called my_script.scr, for example, that consists of three functions, DoThing0 (), DoThing1 (), and DoThing2 (), the pseudocode for a small C program that loads and interacts with the script through the scripting system library might look like this:

```
InitRuntime ();                  // Initialize the runtime environment
LoadScript ( "my_script.scr" );  // Load the script
CallFunction ( "DoThing0" );     // Call DoThing0 ()
CallFunction ( "DoThing1" );     // Call DoThing1 ()
CallFunction ( "DoThing2" );     // Call DoThing2 ()
FreeScript ();                   // Free the script
ShutDownRuntime ();              // Shut the environment down again
```

Pretty straightforward, huh? The one detail I haven't really covered is how you pass parameters to these functions, but this still illustrates the overall process pretty well. I also haven't talked about how the scripting system library knows which C functions correspond to incoming function calls from the script, so let's just scrap the theoretical talk and get your hands dirty with some real scripting action and answer these questions in practice.

The Bouncing Head Demo

In order to try out these scripting systems, the first thing you'll need is a host application to script. Obviously it would be a bit ridiculous for me to wheel out a full game just for use in this chapter, so instead you're going to start small and script a simple bouncing sprite demo.

The demo is decidedly basic; it displays a background image, loads a few frames of a rotating alien head, and bounces them around the screen while looping through the alien's animation. The background image is a little composition of some of my hi-res texture art and some random junk strewn over it, all of which is given a dark, hazy purplish tint. It has the kind of look to it that reflects the amount of Crystal Method and BT I listen to while doing this sort of thing. You can see the demo running in Figure 6.7, or run the included Demo 6.1 on the CD and see it for yourself.

The goal here is to get familiar with the scripting systems this chapter covers by recoding the logic behind the demo with scripts, so your first step is to walk through everything the demo does in a reasonable level of detail. After doing this, you should be able to pick and choose the elements that should be offloaded to scripts, and which should remain hardcoded in C.

Figure 6.7

A screenshot of the bouncing head demo. It's trip-hoptastic!

In a nutshell, the demo is composed of three phases: initialization, the main loop, and shutdown. Let's first look at the steps taken by the initialization phase:

- The Wrappuh API is initialized, which provides the program with simple access to DirectX for graphics, sound, and input.
- The video mode is set. In this case, 640x480 is used with 16-bit color.
- The random number generator is seeded.
- Each of the separate on-screen alien head sprites is initialized with random locations, velocities, and directions.
- The background image is loaded.
- Each frame in the spinning alien head animation is loaded, one by one.
- The current frame of the animation is set to 0.
- Two timers are initialized—one that will tell you when to advance the animation to the next frame, and one that will tell you when to move the sprites along their path.
- The `while` loop that will be the main loop of the program is started and runs until the Escape key is pressed.

Initializing such a simple demo may have seemed trivial at first, but when you actually analyze things like this, they usually turn out to be just a bit more complex than you originally anticipated. The lesson here is that when scripting, don't overestimate or underestimate your requirements. Depending on the situation, your scripting language of choice might not even be capable of handling a small detail you've overlooked, and as a result, you'll end up finding out that your language of choice was inappropriate halfway into the process of writing the actual scripts. This certainly isn't a fun revelation, so plan ahead.

Now that you've nailed down exactly what the initialization phase can do (and what the other two phases will do in a moment), you can tell for sure whether a given language will be capable of handling the job. Moving on, let's look at the guts of the main loop. At each frame of the demo, you'll have to:

- Blit the full screen background image, mainly to display the image itself, but also to overwrite the previous frame.
- Loop through each unique on-screen sprite and draw it at its current location, with the current frame of the spinning head animation. Each head has the ability to spin in the opposite direction, so you may need to invert the current frame number to simulate the other direction.
- Blit the newly finished frame to the screen.
- Check the status of the Escape key, and exit the program if it's been pressed.
- Check the animation timer and update the animation if necessary.
- Check the movement timer and, if necessary, loop through each on-screen sprite and move along its current path at its current velocity. Once the sprite has been moved, you must check its location against each of the four boundaries of the screen and adjust its direction in the event of a collision to simulate a bounce.

Lastly, let's look at what's required to shut the demo down after the main loop has been terminated by pressing Escape:

- Free the background image.
- Free each frame of the animation, one by one.
- Shut down the Wrappuh API.

As is usually the case, the shutdown phase is the simplest. So, now that you know exactly what the demo needs to do, you can decide which parts will remain in C, and which parts will be removed to be re-implemented with scripts. Naturally, you aren't going to redo the *entire* demo in a scripting language, because that would pretty much defeat the whole purpose of scripting in the first place. So, let's get the list of things that should remain in C out of the way:

- The first and last steps of the initialization phase should stay in C simply because they're so basic. The first step is the initialization of Wrappuh— it happens only once and involves nothing more than calling a function, so there's no need to script that. The last step is starting the `while` loop, which is a bit more serious. If you actually move the loop itself into the scripts, your C program will do virtually nothing in the next version of the demo— it passes control to the script, which will run until the user exits, and the C side of things will be inactive. A better design is to keep the actual main program loop running in C and give the script only a small portion of each loop iteration to keep the sprites bouncing around. Also, the random number generator can be seeded in C. This is another operation that's done only once and is so basic and obscure that there's no need for the script to worry about it.

- The C host will load the images.
- The C host will set the video mode.
- Just about everything the main loop needs to do will be scripted, so you can forget about C here. The C program will check for the user pressing Escape, however (although this could be done in either language).
- Just like the initialization phase, there's no need to make the script worry about shutting down the Wrappuh API, so you can leave that where it is.

As you can see, the C version will barely do anything; aside from the most basic initialization and shut down tasks, the only thing C is really responsible for is providing the main loop itself. In this regard, the C program can now be considered a "shell" or "skeleton" that just sets the stage for the scripts to do the real work. So, let's think about what you'll need to recode with scripts:

- The scripts will handle setting all of the initial sprite information, like their location and direction.
- Once in the loop, the scripts will be in charge of almost everything. They'll move the sprites around, they'll check for collisions, and they'll even make the calls to the blitter in order to physically get the graphics on the screen.
- The script won't really have any hand in the shut down process.

Once you have this logic re-implemented in scripts, you can test their *true* power, which is the capability to change this functionality even after the C program has been compiled. This will enable you to alter the bouncing effect or really any other aspect of the scripted program on a whim.

You're ready to roll at this point. The host application is written, your goals for the scripts are clear, so all that's left is to jump in and learn about your first scripting language.

CAUTION

There is one thing I must make absolutely clear before continuing, however. Whether you plan on using Lua or not, I *strongly* recommend you read the section on it in full. This is because all three scripting systems and languages are fundamentally similar in many ways, and describing these common concepts three separate times for each language would be a huge waste of pages. As a result, these concepts are introduced once in the Lua section and then simply referred to in the other two. Make sure you understand everything in this section before attempting to read the other two.

Lua (and Basic Scripting Concepts)

The first stop on your scripting language tour is the quaint little town of Lua. Lua is a simple, easy-to-use language and scripting system designed to extend any sort of program by giving it the capability to load and execute optionally compiled scripts (which, really, is the goal of virtually any scripting system). Lua the language is paradoxically characterized by both its basic and straightforward syntax, as well its understated but powerful capability to be expanded significantly by the only non-primitive data structure it supports, the *table*. Don't let its mild-mannered appearance fool you, however; Lua's been put to good use in such commercial games as *MDK2* and *Balder's Gate*. It can definitely pull its weight when it has to. Lua the scripting system is equally clean and easy to use; it comes as a single static library coded in pure C and ready to be dropped into any host application for some hot, steamy scripting action.

Before getting into the details of how to write scripts in the Lua language, have a look at the components that the Lua system provides.

The Lua System at a Glance

I think the real beauty of the Lua scripting system is its simplicity. When you initially download the package, you won't find billions of scattered files and executables. Instead, you'll find the include files and libraries needed to link Lua into your host application, as well as a small handful of utilities. That's all you need, and that's all you get. Of course, you can find Lua on the included CD under the `Scripting Systems/Lua/` directory.

The Lua Library

The Lua library is composed mainly of two files: `lua.lib` and `lua.h`. The library in most respects follows the archetypical outline in that it provides a clean API for initializing itself and shutting down, as well as functions for loading scripts, executing them, and building the function call interface that will let them talk back and forth with your host application. I'll get back to the details of how to use this library later.

The `luac` Compiler

Lua comes with an easy-to-use command-line driven compiler called `luac`. Typing `luac` at the command prompt will display the program's usage info. To compile a script, simply type:

```
luac <Filename>
```

where `Filename` is the name of the script. The script will be compiled into a file called `luac.out` by default, but this can be changed with the `-o` switch. For example, if you have a script called `test.lua` that you want compiled to a file with the same name, you type this:

```
luac -o test.out test.lua
```

What may surprise you about all this, however, is that you don't ever actually need to use the `luac` compiler in order to use the scripting system. Scripts written in Lua can be loaded directly by the Lua library and will be compiled on-the-fly, at the time they're loaded. This is a nice feature because it allows you to immediately see the results of your script code; you don't have to waste any time on an intermediate compiling step, and you don't have to manage two filenames. The downsides, however, include the fact that you won't get particularly meaningful compile-time errors when your compiling is done at runtime. Because your game (or whatever the host application may be) will be in control of the screen at the time, Lua won't be able to print out a list of syntax errors, for example. The other problem is that loading scripts will now be somewhat slower, as Lua will have to spend the extra time compiling it then and there.

So, `luac` is generally a good program to have around. Not only does it let you compile your scripts ahead of time for much faster loading at runtime, but it also provides you with the same level of compile-time error information that you'd expect from any other compiler. Another advantage is that you won't have to distribute the source to your scripts with your game; instead, you can just release the compiled binaries, which aren't particularly easy for malicious gamers to hack, and also take up less space. In other words, you don't *have* to use the compiler, but you will most likely want to (and definitely should anyway).

The `lua` Interactive Interpreter

Another utility that comes with Lua is the interactive interpreter. This useful little program, also accessible from the command prompt, simply displays the following upon invocation:

```
>
```

Although the interface is about as friendly as the infamous `DEBUG` utility that ships with MS-DOS, the program lets you immediately test out blocks of Lua code by typing them directly into the interpreter and seeing the results in real time (hence the "interactivity"). I haven't discussed the syntax of Lua yet, but the following should be pretty self-explanatory. For example, if you were to type the following:

```
> X = 32
> Y = 64
> print ( X + Y )
```

You'd see the following output:

```
96
```

The last piece of information regarding the `lua` interactive interpreter worth mentioning is that it can also be used to immediately run simple scripts without the need to embed the `lua.lib` runtime environment into a C program. Simply call `lua` with a filename as the single command-line parameter, like so:

```
lua my_script.lua
```

TIP

You'll notice that the interpreter seems to evaluate your statements as soon as you press Enter, even if they're supposed to be part of a larger construct such as an if block. To enter a full block of code without immediately executing it as it's typed, simply follow each line in the block with a backslash (\), much like a multi-line #define macro in C. All of the code will be executed at once after the first non-backslash-terminated line is entered.

and it will attempt to execute and print the output of the script. In addition, `lua` will provide the same level of detail in compile-time errors as `luac` will, which can be useful. Lastly, scripts running inside the `lua` interpreter are automatically given a special `print ()` function, which can be used to print values to the screen, much like `printf ()` in C. Even though I haven't discussed Lua syntax yet, the following should be pretty self-explanatory:

```
print ( "Hello, world!" );
```

Running this in `lua`, strangely enough, produces the following output:

```
Hello, world!
```

Keep this function in mind as you read through the following sections.

The Lua Language

Lua as a language is simple and straightforward. It won't take long to learn the syntax and semantics behind it, and once you have them down, you'll find it elegant and easy to use. The syntax somewhat resembles a mixture of C, BASIC, and Pascal, resulting in a no-frills look and feel that, although not a perfect C clone, should still be an easy transition to make when switching from game engine code to script code. This chapter refers to Lua 4.0, the latest official release at the time of this writing.

The interactive interpreter I mentioned in the last section will be extremely useful during the next few pages; if you really want to follow along, start it up and play with some of the language examples that are discussed. It's the best and fastest way to get really familiar with how Lua works. I highly recommend it.

Comments

I like to introduce comment syntax first when describing a language, because it generally shows up in the code examples anyway. Lua's single comment type is denoted with a double-dash:

```
-- This is a comment.
```

Just like the // comment in C++, Lua's comments cause everything from the double-dashes to the end of the line to be ignored by the compiler. Lua has no provisions for block comments, so multi-line comments must be broken into single lines manually:

```
-- This is the first line of a comment,
-- which is continued down here,
-- and finished here.
```

It's a bit of a hassle, but oh well. :)

Variables

Like most scripting languages, Lua is *typeless.* This means that any variable can hold any value of any type at any time, as opposed to languages like C, which force you to declare a variable of a given type and stick to that type throughout the variable's lifespan. Also unlike C, Lua variables need not be officially declared. Rather, a variable is brought into existence at the time of its first assignment. However, as you'll see, this initial assignment is restricted to some extent in many cases and is often considered a somewhat "implicit" declaration. More on this later.

CAUTION

Avoid creating identifiers that consist of an underscore followed by an all-caps string, such as _IDENTIFIER. This convention is used internally by Lua for its own use, and the possibility of a future version of the language defining the same identifier you've used in your scripts may potentially break your code. Besides, they're ugly anyway.

Identifiers in Lua follow the same rules that exist in C—valid identifiers are sequences of letters, numbers, and underscores that begin with a non-numeric character (meaning a letter or underscore). Identifiers are also case-sensitive, so myvar, myVar, MyVar, and MYVAR are all considered different variable names.

Because variables need only be assigned to be declared, the following block of code would declare and initialize two variables, X and Y:

```
X = 4096            -- Declare X and set its value to 4096
Y = "Hello, world!" -- Declare Y as a string containing "Hello, world!"
```

This little example also illustrates another quirk of Lua's syntax: that semicolons aren't required to terminate lines. However, the semicolon can still be used and is still required in the case of statements that span multiple lines. Consider the following:

```
MyVar0 = 128     -- Valid statement; semicolons are optional.

MyVar1 = 256;      -- Also valid; semicolons can be used if preferred.

print (
"This is a long line!"
);            -- Valid, multi-line statements are allowed as long
              -- as the semicolon is present.
print (
"So is this!"
)             -- Invalid, multi-line statements must end with ';'.
```

TIP

Even though it's optional in most cases, I suggest using semicolons to terminate all statements in Lua anyway. Not only does it make the language seem that much more C/C++ like, but it also makes your code clearer and more robust. If you find that a given statement is getting too long and want to break it into multiple lines, having a semicolon already in place will make sure you don't forget to add it afterwards and wind up with a compile-time error. It's just a good rule of thumb to stick with. As a C and/or C++ programmer, it will be a reflex anyway.

Even though variables only need to be assigned to be declared, they still can't actually be used as arithmetic expressions without being given some sort of initial value. This is because all variables are assigned `nil` before their first assignment, which doesn't make sense in the case of math operations. For example:

```
U = 1024;
V = 2048;
print ( U + V );
print ( U + V + W );
```

This would produce the following:

```
3072
error: attempt to perform arithmetic on global 'W' (a nil value)
stack traceback:
       1:      main of string "print ( U + V ); ..." at line 4
```

The first line of the output is the sum `3072`, just like you would expect, but the following lines are an error message letting you know that `W` cannot be used to perform arithmetic. I'll discuss `nil` in more detail in the following section.

The last issue of variables to cover now is the concept of *multiple assignment*, which Lua supports. Multiple assignment allows you to put more than one variable on the left side of the assignment operator, like so:

```
X, Y, Z = 2, 4, 8;
```

After this line executes, X will equal 2, Y will equal 4, and Z will equal 8. This left-to-right order allows you to tell which identifier will receive which value. Multiple assignment works for any sort of assignment, so you can use it to move the value of one set of variables into another as well:

```
U, V, W = X, Y, Z;
Print ( U, V, W );
```

Which will produce the following (assuming you're using the same X, Y, and Z you initialized in the last example):

```
2       4       8
```

If you're anything like me, the first thought you had when you saw this form of assignment notation was "what happens if you don't provide an equal number of variables and values on both sides of the assignment operator?" Fortunately, in another example of Lua's robust nature, this is handled automatically. In the first case, if you don't provide enough values on the right side to assign to all of the variables left side, the extra variables will be assigned nil:

```
X, Y, Z = 16, 32;
```

This will assign X 16 and Y 32, but Z will be set to nil. This even works in cases when the extra variable has already been initialized. For example:

```
U, V, W = 256, 512, 1024;
print ( U, V, W );
U, V, W = 2048, 4096;
print ( U, V, W );
```

Even though W was assigned a value in the first assignment, which will be visible in the output of the first print () call, the second assignment will replace it with nil:

```
256     512     1024
2048    4096    nil
```

In the second case, where there aren't enough variables on the right side to receive all of the values on the left, the unused values will simply be ignored, so a line like this:

```
X, Y = 8192, 16384, 32768, 65536;
```

is perfectly legal and will only assign X and Y the first two values. The last two variables will simply vanish without a trace, much like Paulie Shore's career.

Overall, multiple assignment is a convenient shorthand but definitely has potential to make your code less-than-readable. Only use it in cases when you're sure that the code is clearly understandable, and try not to do it for too many variables at once. Don't try to get cute and impress your friends with huge tangles of multiple assignment; it will only result in error-prone code. One good use of the technique; however, is swapping two values in one line easily:

```
X = 16;                    -- Declare some variables
Y = 32;
print ( "Unswapped:", X, Y ); -- Print them out
X, Y = Y, X;               -- Swap them with multiple assignment
print ( "Swapped:", X, Y );   -- Print the swapped values
```

This will produce the following:

```
Unswapped:   16        32
Swapped:     32        16
```

Data Types

Now that you can declare and use variables, you're probably interested in knowing what you can stuff into them. Lua supports six data types:

- **Numeric.** Integer and floating-point values. Unlike C, these two types of numeric values are considered the same data type.
- **String.** A string of characters.
- **Function.** A reference to a formally declared function, much like a function pointer in C (but simpler to use and more discreet).
- **Table.** Lua's most complex and powerful data type; tables can be as simple as associative arrays and as complex as the basis for more advanced data structures like linked lists and classes.
- **Userdata.** A slightly more obscure data type that allows C pointers to be stored in Lua variables for a more tight integration into the host application. Userdata pointers correspond to the `void *` pointer type in C. I won't be covering this data type.
- **nil.** The simplest data type by far, `nil`'s only job is to be different from every other value the language supports. This means it makes a good flag value, especially when you want to mark something as uninitialized or invalid. In fact, any reference to a variable that hasn't been directly assigned a value will equal `nil`. `nil` is also the only concept of "falsehood" the language supports. In other words, `nil` is like a more robust version of C's `NULL`. This is consistent with what you saw in the last section when you tried adding a `nil` value to two integers, which is illegal in Lua. This is an important lesson: `nil` is *false*, but it is *not* equal to zero in a numeric or arithmetic sense. This is why arithmetic expressions involving `nil` variables don't make sense and result in a runtime error.

If you happen to have the Lua interpreter open at the time, try using the `type ()` function to examine various identifiers. The `type ()` function returns a string describing the data type of whatever identifier is passed to it, so consider the following:

```
print ( type ( 256 ) ); \
print ( type ( 3.14159 ) ); \
print ( type ( "It's a trap!" ) );
```

Upon pressing Enter, you should see the following output:

```
number
number
string
```

Right off the bat, the numeric and string types should be a snap, and even the function type is pretty simple when you think about it. `nil` is easy to grasp as well, and the `Userdata` type is beyond the scope of this book so I won't be discussing it any further. That leaves you with tables, which is good because they deserve the most explanation.

NOTE

Although I'm sure you've picked up on this already, I'd just like to make sure that you're clear on the `print ()` function. `print ()` will print any value passed to it, as well as the contents of any identifier. This is a special function built in to the version of Lua running in the interpreter to allow immediate feedback while coding interactively. The function also allows you to pass it comma-delimited lists, the output of which will be aligned with tab stops. You'll see more of this later.

Before moving on, however, I'd just like to quickly mention one last aspect of Lua's data types: *coercion*. Coercion is when one data type is cast, or *coerced* into another for the sake of executing an expression. For example, numeric values and strings can be used interchangeably in a number of expressions, like so:

```
print ( 16 + 32 );
print ( "16" + 32 );
print ( 16 + "32" );
print ( "16" + "32" );
```

Each of these `print ()` calls will output the numeric value `48`. This is because whenever a string was encountered in the arithmetic expression, it was coerced into its numeric form. Lua recognizes strings that can be converted meaningfully to numbers, like the previous ones. However, the following statement would cause an error:

```
print ( 16 + "32" + "Alex" );
```

The first two values, `16` and `"32"`, are valid. `16` is already an integer value and `"32"` can be coerced into one and still make sense. When the last string value (`"Alex"`) is reached, however, Lua will attempt to convert it to a number and find that it has no numeric equivalent, thus stopping execution to report the error of attempting to use a string in an arithmetic expression:

```
error: attempt to perform arithmetic on a string value
```

Tables

Tables in Lua are, first and foremost, associative arrays not unlike the ones found in other scripting languages like Perl and PHP. Associative arrays are also comparable to the hash table structure provided in the standard libraries for languages like Java and C++.

Tables are indexed with the same syntax as a C array, and are initialized in much the same way. For example, consider the following table declarations that mimic C string and integer arrays:

```
IntArray = { 16, 32, 64, 128 };
StringArray = { "Aho", "Sethi", "Ullman" };
```

Although you didn't have to specify a data type for the table, or even its size, you do use the traditional C-style `{ … }` notation for initialization. Once the tables have their values, they can be accessed much like you'd expect, but with one major difference: the initialized values start at index 1, not zero:

```
print ( IntArray [ 1 ] );
print ( StringArray [ 2 ] );
```

This code will produce the following output:

```
16
Sethi
```

Of course, even though an initialization set is automatically indexed from 1, it doesn't mean index zero can't be used:

```
IntArray [ 0 ] = 8;
print ( IntArray [ 0 ], IntArray [ 1 ], IntArray [ 2 ] );
```

will produce the following output:

```
8      16       32
```

Although it's important to note that index zero is perfectly valid as long as you manually give it a value, the real lesson in the preceding example is your ability to add new elements to a table whenever you need to. Notice that the set of values that initialized the table included only indexes 1 through 4, but you can still expand the array to cover 0 through 4 by simply assigning a value to the desired index. Lua will automatically expand the array to accommodate the new values. In fact, virtually any index you can imagine will already be accessible the moment you create a new table. For example:

```
print ( IntArray [ 0 ] );
print ( IntArray [ 2 ] );
print ( IntArray [ 24 ] );
print ( IntArray [ 512 ] );
```

Even though indexes 24 and 512 are far from the initialization set, check out the output:

```
8
32
nil
nil
```

Neat, huh? Lua automatically created and initialized indexes 24 and 512, allowing you to access them without any sort of out-of-bounds or access-violation errors. In this regard, table indexes are much like typical Lua variables in that they are created only when they are first assigned (or when you initialize them with the `{ … }` notation), but will contain `nil` until then.

The next important aspect of Lua tables is that they are *heterogeneous*, which means that not all indexes must contain the same type of value. For example:

```
MyTable [ 0 ] = 256;          -- Assign an integer to index 0
MyTable [ 1 ] = 3.14159;      -- Assign a float to index 1
MyTable [ 2 ] = "Yahtzee!";   -- Assign a string to index 2
```

The three indexes of this table contain three different data types, further illustrating a table's flexibility. In addition to being able to hold any sort of primitive value, table indexes can also hold references to other tables, which opens the door to endless possibilities. Most obviously, this lets you simulate multi-dimensional arrays, like so:

```
MultiTable = {};
MultiTable [ 0 ] = { "ABC", "DEF", "GHI" };
MultiTable [ 1 ] = { "JKL", "MNO", "PQR" };
MultiTable [ 2 ] = { "STU", "VWX", "YZ" };
print ( MultiTable [ 0 ][ 1 ] );
print ( MultiTable [ 1 ][ 2 ] );
print ( MultiTable [ 2 ][ 3 ] );
```

Which will output the following:

```
ABC
MNO
YZ
```

NOTE

Even though I indexed `MutliTable []` from 0 to 2, each of the other three-index tables that were directly initialized at `MultiTable [ 0 ]`, `MultiTable [ 1 ]`, and so on, are indexed automatically 1 to 3 because of Lua's one-index convention. I automatically use zero-indexing out of habit, but it's definitely important to keep Lua's style in mind. Forgetting this detail can lead to some nasty logic errors.

It's important to know exactly how things are working under the hood when working with tables that contain tables, however. When working with Lua, don't think of tables as values, but rather as *references*. Any time you access a table index or assign a table to another table index, you're actually dealing with the references Lua maintains for these tables, *not* the values themselves. For example, the output of the following code snippet could represent some serious logic errors if you aren't aware of what's happening:

```
X = {};                           -- Declare a table
X [ 0 ] = 16;                     -- Give it three indexes
X [ 1 ] = 32;
X [ 2 ] = 64;
print ( "X: ", X [ 1 ] );         -- Print out index 1
Y = {};                           -- Declare a new table
Y [ 0 ] = X;                      -- Give it one index, containing X
Y [ 0 ][ 1 ] = "String";          -- Set the index 1 of index 0 to a string
print ( "Y: ", Y [ 0 ][ 1 ] );    -- Print out index 1 of index 0 of Y
print ( "X: ", X [ 1 ] );         -- Print out index 1 of X
```

As you can see, the assigning of `X` to `Y [ 0 ]` didn't copy the `X` table and all of its values. Rather, `Y [ 0 ]` was simply given a *reference* to `X`, which means that any subsequent changes made to the table located at `Y [ 0 ]` will also affect `X`, as can be seen in the output. This is a lot like pointers in C, but I'll keep the pointer analogies to a minium because this topic can be confusing enough as it is. Refer to Figure 6.8 for an illustration

Moving on, the next major aspect of Lua tables to discuss is their associative nature. In other words, instead of being forced to use integer indexes to index your array, you can use values of any type. In this regard, tables work on the principal of *key : value pairs*, which let you associate values with other values, called keys, for more intuitive indexing. Consider the following example:

```
Enemy = {};
Enemy [ "Name" ] = "Security Droid";
Enemy [ "HP" ] = 200;
```

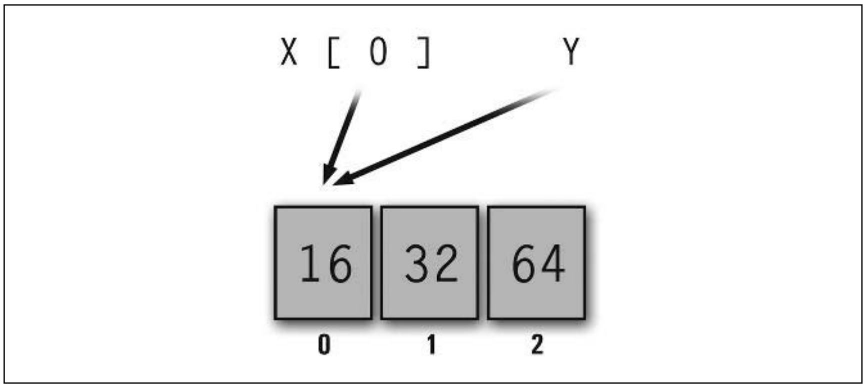


Figure 6.8

Both X and Y are referring to the same physical data; as a result, any changes to either reference will appear to affect the other.

```
Enemy [ "Weapon" ] = "Pulse Cannon";
Enemy [ "Sprite" ] = "../gfx/enemies/security_droid.bmp";
print ( "Enemy Profile:" );
print ( "\n  Type:", Enemy [ "Name" ],
        "\n    HP:", Enemy [ "HP" ],
        "\nWeapon:", Enemy [ "Weapon" ] );
```

Which will print out the following:

```
Enemy Profile:

  Type: Security Droid
    HP: 200
Weapon: Pulse Cannon
```

As you can see, each of table's elements was indexed with strings as opposed to numbers. To use the previous terminology, `"Name"`, `"HP"`, `"Weapon"`, and `"Sprite"` were the table's *keys*. The keys were associated with values, which appeared on the right side of the assignment operator. For instance, `"Name"` was the key to the value `"Security Droid"`. This example also introduced you to the `\n` escape code for newlines, which functions just as it does in C. You'll see the rest of Lua's escape codes later.

Any literal data type can be used as a key, so integers, floating-point values, and of course strings, are all valid. Lua also provides an extra notational convenience for instances where the string key is also a valid identifier. For example, consider the following rewrite of the previous example:

```
Enemy = {};
Enemy.Name = "Security Droid";
Enemy.HP = 200;
Enemy.Weapon = "Pulse Cannon";
Enemy.Sprite = "../gfx/enemies/security_droid.bmp";
print ( "Enemy Profile:" );
print ( "\n  Type:", Enemy.Name,
        "\n    HP:", Enemy.HP,
        "\nWeapon:", Enemy.Weapon );
```

As you can see, the string keys are now being used as if they were fields of a `struct`-like structure. In this case, that's exactly what they are. Lua automatically adds these identifiers to the table, allowing them to be accessed in this way. This technique is completely interchangeable with string keys, so the following code:

```
Table = {};
Table.X = 16;
Table [ "Y" ] = 32;
print ( Table [ "X" ], Table.Y );
```

will output:

```
16    32
```

as if everything was declared using the same method. Internally, Lua doesn't care, so `Table [ "Key" ]` is always equivalent to `Table.Key`, provided that `"Key"` is a string containing a valid identifier.

Advanced String Features

You've seen how basic string syntax works in Lua, but there are a few slightly more advanced topics worth covering before moving on. The first is *escape sequences*, which are special character codes preceded by a backslash (`\`) and direct the compiler to replace certain parts of the string before compilation instead of taking them literally. As an example of when escape sequences are necessary, imagine wanting to use a double quote in a string, such as in the following example:

```
Quote = ""Welcome to the real world", she said to me, condescendingly.";
```

The problem is that the compiler will think the string ends immediately after the second double quote (which is really just supposed to denote the beginning of the quotation), which is in reality the first character in the string. Everything following this will be considered erroneous. Escape sequences help you alleviate this problem by giving the compiler a heads-up that certain quotes are not meant to begin or end the string, but are just characters *within* a larger string. The escape sequence `\"` (backslash-double quote) is used to do just this. With escape sequences, you can rewrite the previous line and compile it without problems:

```
Quote = "\"Welcome to the real world\", she said to me, condescendingly.";
```

There are a number of escape sequences supported by Lua in addition to the previous one, but most are related to text formatting and are therefore not particularly useful when scripting games. However, I personally find the following useful: `\\` (Backslash), `\'` (Single Quote), and `\XXX`, where `XXX` is a three-digit decimal value that corresponds to the ASCII code of the character that should replace the escape sequence.

Using the `\"` escape sequence can be a pain, however, when dealing with strings that contain a lot of double quotes. Because this is a possibility when scripting games (because many scripts will contain heavy amounts of dialog that possibly require double quotes), you may want to avoid the problem altogether by using single-quotes to enclose your strings, which Lua also supports. For example, consider the following:

```
PrintQuote ( 'You run into the room. "No!" you scream, as you notice your gun is
missing.' );
```

The previous string is equivalent to the following line, but easier to write (and more readable):

```
PrintQuote ( "You run into the room. \"No!\" you scream, as you notice your gun is
missing." );
```

Of course, if for some reason you need to use a large number of single quotes, you can just stick to the double-quoted string.

Lastly, Lua supports a third method of enclosing strings that is by far the most powerful. Enclosing your string with double brackets, such as the following line, allows you to insert physical line breaks directly into the string value without causing a compile-time error:

```
MyString = [[This is a
multi-line
string.]];
print ( MyString );
```

This will produce the following output:

```
This is a
multi-line
string.
```

Expressions

Expressions in Lua are a bit more like Pascal than they are like C, in that they offer a more limited set of operators and use text mnemonics for certain operators instead of symbols. Lua's many operators are organized in Tables 6.1 through 6.3.

Table 6.1 Lua Arithmetic Operators

Operator	Function
+	Add
-	Subtract
*	Multiply
/	Divide
^	Exponent
-	Unary negation
..	Concatenate (strings)

Table 6.2 Lua Relational Operators

Operator	Function
==	Equal
~=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Table 6.3 Lua Logical Operators

Operator	Function
`and`	And
`or`	Or
`not`	Not

Major differences from C worth noting are as follows: the != (Not Equal) operator is replaced with the equivalent ~= operator, and the logical operators are now mnemonics instead of symbols (and instead of &&). These are important to remember, as it's easy to forget details like this and have a "C lapse". :)

Conditional Logic

Now that you have a handle on statements, expressions, and values, you can start structuring that code with conditional logic. Like C and indeed most high-level languages, Lua uses the tried-and-true if statement, although its syntax is most similar to BASIC:

```
if <Expression> then
       Block;
elseif <Expression> then
       Block;
end
```

Unlike C, the expression does not have to be enclosed in parentheses, but you can certainly add them if you want. Expressions can contain parentheses even when they aren't necessary. Here's an example of using if:

```
X = 16;
Y = 32;
if X > Y then
       print ( "X is greater." );
else
       print ( "Y is greater." );
end
```

Lua does not support an analog to C's switch construct, so you can instead use a series of elseif clauses to simulate this (and indeed, this is done in C at times as well). For example, imagine you have a variable called Item that keeps track of an item the player is carrying and implements its behavior when used. Normally one might use a switch to handle each possible value, but you have to use an if-elseif-else chain instead.

```
if Item == "Sword" then
       -- Handle sword behavior
elseif Item == "Morning Star" then
       -- Handle morning star behavior
elseif Item == "Nunchaku" then
       -- Handle nunchaku behavior
```

```
else
       -- Unknown item
end
```

As you can see, the final `else` clause mimics C's `default` case for `switch` blocks. As a gentle reminder, remember that the logical operators in Lua follow a different syntax from C:

```
X = 1;
Y = nil;
if X ~= Y then
       print ( "X does not equal Y." );
end
if X and Y then
       print ( "Both X and Y are true." );
end
if X or Y then
       print ( "Either X or Y is true." );
end
if not ( X or Y ) then
       print ( "Neither X nor Y is true." );
end
```

Iteration

The last control structures to consider when discussing Lua are its iterative structures (in other words, its loops). Lua supports a number of familiar loop types: `while`, `for`, and `repeat`. `while` and `for` should make C programmers feel at home, and Pascal users will appreciate the inclusion of `repeat`. All of the structures have a fairly predictable syntax, so take a look at all of them:

```
while <Expression> do
       -- Block
end

for <Index> = <Start>, <Stop>, <Step> do
       -- Block
end

repeat
       -- Block
until <expression>
```

That should all look pretty reasonable, although the exact syntax of the `for` loop might be a bit confusing. Unlike C, which allows you to use entire statements (or even multiple statements) to define the loop's starting condition, stopping condition, and iterator, Lua allows only simple numeric values (in this regard, it's a lot like BASIC). The step value is also optional, and omitting it will cause the loop to default to a step of 1. Take a look at some examples:

```
for X = 0, 3 do
       print ( "Iteration:", X );
end
```

This code will produce:

```
Iteration:   0
Iteration:   1
Iteration:   2
Iteration:   3
```

As you can see, the step value was left out and the loop counting from 0 to 3 in steps of 1. Here's an example with the step included:

```
for X = 0, 7, 2 do
       print ( "Iteration:", X );
end
```

It produces:

```
Iteration:   0
Iteration:   2
Iteration:   4
Iteration:   6
```

Before moving on, I should mention an alternative form of the `for` loop that you might find useful. This version is specifically designed for traversing tables, and looks like this:

```
for <Key>, <Value> in <Table> do
       -- Block
end
```

This form of the loop traverses through each key : value pair of `Table`, and sets `Key` and `Value` appropriately at each iteration. `Key` and `Value` can then be accessed within the loop. For example:

```
MyTable = {};
MyTable [ "Key0" ] = "Value0";
MyTable [ "Key1" ] = "Value1";
MyTable [ "Key2" ] = "Value2";
for MyKey, MyValue in MyTable do
       print ( MyKey, MyValue );
end
```

produces the following output:

```
Key0   Value0
Key2   Value2
Key1   Value1
```

NOTE

Notice that in the first example for the table-traversing form of the `for` loop, the values seem to have been printed out of order. The key : value pair `"Key2"`, "`Value2`" came before `"Key1"`, "`Value1"`. This is because associative arrays don't have the same numeric order that integer-indexed tables do, so the order at which elements are added is not necessarily the element in which they are stored.

Functions

Functions in Lua follow a pattern similar to that of most languages, in that they're defined with an initial declaration line, containing an identifier and a parameter list, followed by a code block that implements the function. Here's an example of a simple function that adds two numbers and returns the sum:

```
function Add ( X, Y )
       return X + Y;
end
print ( Add ( 16, 32 ) );
```

The output, of course, is `48`. The only real nuance regarding functions is that unlike most languages, all variables referenced or created in a function are in the global scope by default. So, for example, imagine changing the previous code so that it looks like this:

```
function Add ( X, Y )
       return X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

Now, instead of printing the return value of the `Add ()` function, you print the uninitialized `GlobalVar`. Not surprisingly, the output is simply `nil`. However, when you add another line:

```
function Add ( X, Y )
       GlobalVar = X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

You once again get the proper output of 48. This is because GlobalVar is automatically created in the global scope, and therefore is visible even after Add () returns. To suppress this and create local variables, the local keyword is used. So, if you simply add one instance of local to the previous example:

```
function Add ( X, Y )
       local GlobalVar = X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

The output of the script is once again nil, as it would be in most other languages. This is because GlobalVar is created only within the Add () function's scope (so you should probably consider renaming it "LocalVar"), and is therefore invisible once it returns.

The last thing to mention about functions is that they too can be assigned to variables and even table elements. Imagine two variables called Add () and Sub (), which each perform their respective arithmetic operation:

```
function Add ( X, Y )
       return X + Y;
end

function Sub ( X, Y )
       return X - Y;
end
```

You could assign either of these functions to a variable called MathOp, like this:

```
MathOp = Add;
```

And could then call the Add () function indirectly by "calling" MathOp instead:

```
print ( MathOp ( 16, 32 ) );
```

The output will be 48. The interesting thing, however, is what happens when all you change is the function that you assign to MathOp:

```
MathOp = Sub;
print ( MathOp ( 16, 32 ) );
```

Because MathOp now refers to the Sub () function, your output will be -16. As mentioned previously, this capability to "assign" functions to variables is like a somewhat simplified version of C's function pointers. Use it wisely, my friend.

One last detail; because functions can be assigned to table elements, you can take advantage of the same notational shorthands. For example:

```
function PrintHello ()
       print ( "Hello, World!" );
end
MyTable = {};
MyTable [ "Greeting" ] = PrintHello;
```

At this point, the `"Greeting"` element of `MyTable` contains a reference to `PrintHello ()`, which can now be called in two ways:

```
MyTable [ "Greeting" ] ();
MyTable.Greeting ();
```

Both are valid and considered equivalent as far as Lua is concerned, but I personally prefer the latter version because it looks more natural.

NOTE

Again, if you're anything like me, a gear or two may have started to turn when you saw the last example. "Functions? Stored in tables and accessible just like methods in a class? Hmmmm..." Yes, my friends, this is a small part of the puzzle of how Lua can emulate object-orientation. I won't be covering that in this book, but it's certainly an interesting topic to investigate. See if you can figure out the rest!

Integrating Lua with C

Now that you understand the Lua language enough to get around, it's time for the *real* fun to begin. In a moment, you'll return to the bouncing alien head demo and recode the majority of its core logic with Lua as an example of true script integration. But before you go that far, you need to first get your feet wet by getting Lua to run inside and interact with a simple console application to make sure you understand the basics.

The first goal is decidedly simple; write one or two basic scripts, load them in a simple console application, and print some basic output to the screen that illustrates the interactions between the C program and Lua.

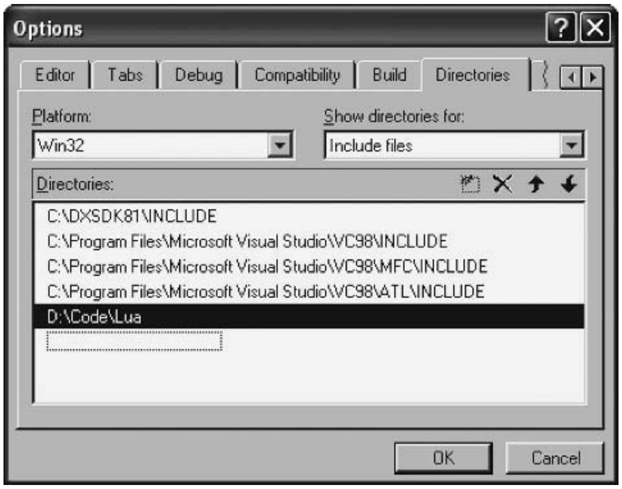
Specifically, this program illustrates the following techniques:

- Loading Lua script files and executing them.
- Exporting a C function so that it can be called from Lua scripts.
- Importing Lua functions from scripts so that they can be called from C.
- Passing parameters and returning values in a number of data types to and from both C and Lua.
- Reading and writing global variables in Lua scripts.

Compiling a Lua Project

Understanding how to compile a Lua project is the first and most important thing to understand for obvious reasons. Not surprisingly, the first step is to include `lua.h` in your main source file and make sure the compiler knows where to find the `lua.lib` library.

In the case of Microsoft Visual C++ users, this is a simple matter of selecting Options under the Tools menu and activating the Directories tab. Once there, set the Show Directories For pop-up menu to Include Files. Click the new directory button (the document icon with the sparkle in the upper-left corner) and enter the path to your Lua installation folder (which should contain `lua.h`). Next, set the Show Directories For pop-up to Library Files and repeat what you did for the include files (as long as that same directory also includes `lua.lib`). Figure 6.9 shows the Options dialog box.



Platform: Win32
Show directories for: Include files
Directories:

```
C:\DXSDK81\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
D:\Code\Lua
```

Figure 6.9

The Visual C++ Options dialog box.

Once these settings are complete, make sure to physically include `lua.lib` in your project. I like to put mine under a Libraries folder within the project.

Including the header file is simple enough, but there is one snag. Lua is a *pure-C* library. That may not mean much these days, when popular compilers pretty much blur the difference between C and C++ programs, but unless you're using a pure C programming environment, your linker will have some issues with it if you don't explicitly mention this fact. So, make sure to include `lua.h` like this:

```
extern "C"
{
        #include <lua.h>
}
```

Remember, this will work only if you properly set your path as described previously.

NOTE

In case you're not familiar with it, `extern` is a *directive* that informs the linker that the identifiers (namely functions) defined within its braces follow the conventions of another language and should be treated as such. In this case, because most people are using the C++ linker that ships with Microsoft Visual C++, you need to make sure it's prepared for a C library that uses slightly different conventions when declaring functions and the like.

Initializing Lua

Lua works on the concept of *states*. A Lua state is essentially a structure that contains information regarding a specific instance of the runtime environment. Each state can contain one script at any time, which is loaded into memory for use. To load and execute multiple scripts concurrently, one needs only to initialize multiple states.

Think about states in the same way you'd think about two instances of the same program in memory. Imagine starting Photoshop (if you don't own Photoshop, imagine owning it as well). Now imagine loading Photoshop again, thus creating two instances of the program at once. Each instance exists in its own "space," and is unrelated to and unaffected by the other. You can open a photo of your dog in one instance, and while doing post-production work on a 3D rendering in the other. Both instances of Photoshop, although essentially the same program with the same functionality, are doing different things at the same time without any knowledge of each other.

From the perspective of the host application, a Lua state is simply a pointer to `lua_State` structure. Once you've declared such a pointer, you can call `lua_open ()` to intialize the state. The only parameter required by `lua_open ()` is the stack size that this particular state will require. Don't worry too much about this; stack size will really only affect the state's ability to handle excessive nesting of function calls, so unless you're going to be hip deep in recursive algorithms, just set it to something like 1024 and forget about it (even this is overkill, but memory is cheap these days so go nuts!). In the relatively unlikely event that you run into stack-overflow errors, just increase it. Here's an example:

NOTE

You can also pass zero to `lua_open ()`, which will cause the stack size to default to 1024 elements.

```
lua_State * pLuaState = lua_open ( 1024 );
```

This example creates a new state called `pLuaState` that refers to an instance of the runtime environment with a stack of 1024 elements. This state is now valid, and is capable of loading and executing scripts.

Of course, no initialization function is complete without its corresponding shut down function. Once you're done with your Lua state, be sure to close it with `lua_close`:

```
lua_close ( lua_State * pLuaState );
```

Loading Scripts

Loading scripts is just as easy as initializing the Lua state. All that's necessary is calling `lua_dofile ()` and passing it the appropriate filename of the script, as well as the state pointer you just initialized. `lua_dofile ()` has the following signature:

```
int lua_dofile ( lua_state * pLuaState, const char * pstrFilename );
```

To execute a script stored in the file `"my_script.lua"`, you enter the following:

```
iErrorCode = lua_dofile ( pLuaState, "my_script.lua" );
```

The `pLuaState` instance of the runtime environment will now load, verify, and immediately execute the file. Keep in mind that `lua_dofile ()` will load both compiled and uncompiled scripts transparently; you can pass it either type of file and it will automatically detect and handle it properly. However, because uncompiled scripts will need to be compiled before they can be executed, they will take slightly longer to load. Also, uncompiled scripts are not necessarily valid and may contain syntactic or semantic errors that a compiler would normally not allow. In this case, the call to `lua_dofile ()` will not succeed, so let's discuss its potential error codes. Refer to Table 6.4 for a complete listing.

Once the script is loaded, it is immediately executed. This isn't always what you want; many times, you'll want to load a script ahead of time and execute it later, or even better, execute different parts of it at different times. I'll cover this in a moment. For now, let's just focus on simply loading and running scripts.

NOTE

As you can see, the only shred of compile-time error information `lua_dofile ()` will give you is `LUA_ERRSYNTAX`, which is pretty much one step above nothing at all. Let this be another example of how useful the `luac` compiler is, which gives you a rundown of compile-time errors in detail beforehand. Don't be lazy! Use it!

You can load scripts, but how will you actually know if they're doing anything? You don't have any way to print text from the Lua script to your console application, so even if the script works, you have no way to observe it. This means that even before you write and execute a Lua script,

Table 6.4 `lua_dofile ()` Error Codes

Code	Description
0	Success.
`LUA_ERRRUN`	An error occurred while running the script.
`LUA_ERRSYNTAX`	A syntax error was encountered while pre-compiling the script.
`LUA_ERRMEM`	The required memory could not be allocated.
`LUA_ERRERR`	An error occurred with the error alert mechanism. Kind of embarrassing, huh?. :)
`LUA_ERRFILE`	An error occurred while attempting to open or read from the file.

you have to learn how to call C functions from Lua. Once you can do this, you just wrap a function that wraps `printf ()` or something along those lines, and you can print the output of your scripts to the console and actually watch it run.

As such, pretty much everything following this point deals with how Lua and C are integrated, starting with the all-important Lua stack.

The Lua Stack

Lua communicates with C primarily through a stack structure that can be used to pass everything from the values of global variables to function references to parameters to return values. Lua uses this stack internally for a number of tasks, but all you care about is how you can use it to talk to Lua scripts and interpret their responses.

Let's first take a look at some of the generic stack-manipulation functions and macros that Lua provides. It might not make total sense just yet as to how these are used or why, but rest assured it will all make sense soon. You should come to understand the basics of these functions before learning how to apply them.

Much like tables, Lua stacks are indexed starting from 1. This is important to know because the stack does not have to be accessed in a typical stack fashion at all times. The traditional "push-and-pop" stack interface is always available, but you can refer to specific elements of the stack much like you do an array when necessary.

At any time, the index of the stack's top element will be equal to stack's overall size. This is because Lua indexes the stack starting from 1; therefore, a stack of one element can be indexed from 1-1, a stack of 16 elements can be indexed from 1-16, and so on. This is a stark contrast from C and most other languages, in which arrays and other aggregate structures begin indexing from 0. In these cases, the "top" or "last" element in the structure is always equal to the size *minus one*. Figure 6.10 shows you the Lua stack visually.

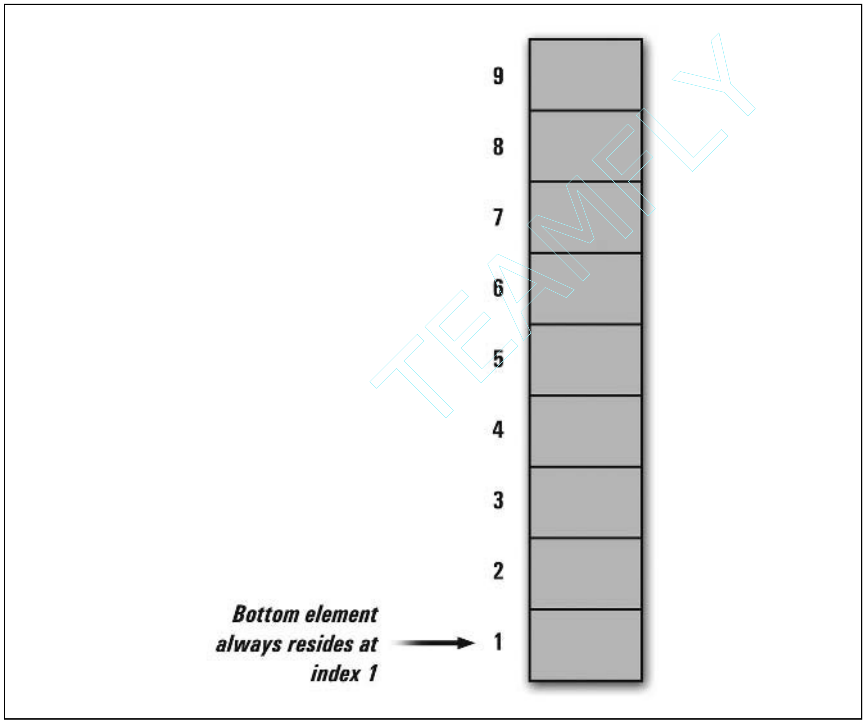


Figure 6.10

The Lua stack.

A program's stack is a turbulent data structure; as functions are called and expressions are evaluated, it grows and shrinks in an erratic pattern. Because of this, stacks are usually accessed in relative terms. For example, when a given function is active, it usually works with its own local portion of the stack, the offset of which is usually passed by the runtime environment.

In the case of Lua, you'll generally be accessing the stack to do one of two things: to write a C function that your scripts can call, or to access your script's global variables. In both cases, the Lua stack will be presented to your program such that the indexes begin at 1. In essence, Lua "protects" the rest of the stack that your program isn't accessing, much like memory-protected operating systems like Windows and Linux protect the memory of your computer from a program if it lies outside of its address space. This makes your job a lot easier, because you can always pretend your chunk of the stack begins at 1. Take a look at Figure 6.11, which illustrates this.

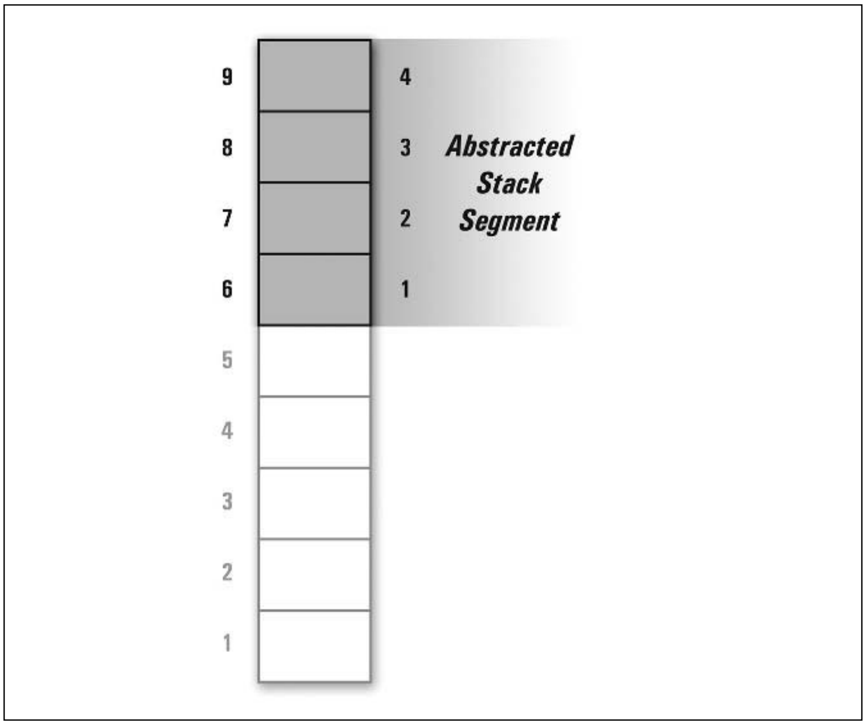


Figure 6.11

Regardless of the size of the stack, Lua will always present what appears to be an empty stack starting from 1 when it is accessed from C.

So to sum things up, Lua will virtually always appear to portray an empty stack starting from 1 when you attempt to access it from C. That being said, let's look at the functions that actually provide the stack interface. Lua features a rich collection of stack-related functions, but the majority of them won't be particularly useful for your purpose and as such, I'll be focusing only on the major ones.

First off, there's `lua_gettop ()`, which gives you the index of the top of the stack:

```
int lua_gettop ( lua_State * pLuaState );
```

As you learned when you took a look at `lua_open ()`, each Lua state has its own stack size, and thus, its own stack. This means all stack functions (as well as the rest of Lua's functions for that matter) require a pointer to a specific state. Getting back to the topic at hand, this function will return the index of the top element `int`. As you learned, this is also equal to the size of the stack.

Up next is `lua_stackspace ()`, which returns the number of stack elements still available in the stack. So, if the stack size is 1024, and 24 bytes have been used at the time this function is called, 1000 will be returned. This function is especially important because the host application, *not* Lua, is responsible for preventing stack overflow. In other words, if your program is rampantly pushing value after value onto the stack, you run the risk of an overflow error because Lua won't stop or even alert you until it's too late. `lua_stackspace ()` should be used in any case where large numbers of values will be pushed onto the stack, especially when the pushing will be done inside loops, which are especially prone to overflow errors.

The next set of functions you will read about is one of the most important. It provides the classic push/pop interface that stacks are usually associated with. Despite the fact that Lua is typeless, C and C++ certainly aren't, and as such you'll need a number of functions for pushing different data types:

```
void lua_pushnumber ( lua_State * pLuaState, double dValue );
void lua_pushstring ( lua_State * pLuaState, char * pstrValue );
void lua_pushnil ( lua_State * pLuaState );
```

These are three of Lua's `lua_push* ()` functions, but they're the only ones you really have a need for (the rest deal with more obscure, Lua-oriented data types). lua_`pushnumber ()` accepts a double-precision float value, which is a superset of all numeric data types Lua supports (integers, single- and double-precision floating-point). This means that both `int`s and `float`s need to be passed with this function as well. Next is `lua_pushstring ()`, which predictably accepts a single `char *` that points to a typical null-terminated string. The last function worth mentioning is `lua_pushnil ()`, which doesn't require any value, as it simply pushes Lua's `nil` value onto the stack (which, if you remember, is conceptually similar to C's `NULL`, except that it's not equal to zero).

Popping values off the stack is a somewhat different story. Rather than provide a collection of `lua_pop* ()` functions to match the push functions, Lua simply provides a single macro called `lua_pop ()`, which looks like this:

```
lua_pop ( lua_State * pLuaState, int iElementCount );
```

This macro does nothing more than pops `iElementCount` elements off the stack. They don't actually go anywhere when you pop them, so this function can only be used to remove the values, not extract them. To actually receive the values and store them in C variables, you must use one of the following functions *before* calling `lua_pop ()`:

```
double lua_tonumber ( lua_State * pLuaState, int iIndex );
const char * lua_tostring ( lua_State * pLuaState, int iIndex );
```

Again, the functions should be pretty easy to understand just by looking at them. Give either function an index into the stack, and it will return its value (but will *not* pop or remove that value). In the case of numeric values, you'll always receive a `double` (whether you want an integer or not), and in the case of strings, you'll of course be returned a `char` pointer. Because neither of these functions actually removes the value after returning them, I'll just reiterate that you need to use `lua_pop ()` afterwards if you actually want the value taken off the stack afterwards. Otherwise, these functions can be used to read from anywhere in Lua's stack. To reliably read from the top of the stack every time with these functions, remember to use `lua_gettop ()` to provide the index.

Actually, because Lua doesn't provide a particularly convenient way to directly pop a value off the stack in the traditional context of the stack interface, let's write some macros to do it now. Using the existing Lua functions, you have to do three things in order to simulate a stack pop:

- Get the index of the stack's top element using `lua_gettop ()`.
- Use one of the `lua_to* ()` functions to convert the element at the index returned in the first step to a C variable.
- Use `lua_pop ()` to pop a single element off the top of the stack.

Because this would be a fairly bulky chunk of code to slap into your program every time you want to do this, a nice little macro that wraps this all up into a single call would be great. Here's one that will pop integers off the stack in one fell swoop:

```
#define PopLuaInt( pLuaState, iDest ) \
{                                     \
    iDest = ( int ) lua_tonumber ( pLuaState, lua_gettop
        ( pLuaState ) ); \
    lua_pop ( pLuaState, 1 ); \
}
```

Just pass the macro a valid Lua state and an integer and it will be filled with the proper value. Here's a small code example (assume that `pLuaState` has already been created with `lua_open ()`):

```
int X, Y;
X = 0;
Y = 32;
lua_pushnumber ( pLuaState, Y );
printf ( "X: %d, Y: %d\n", X, Y );
PopLuaInt ( pLuaState, X );
printf ( "X: %d, Y: %d\n", X, Y );
```

The output will be:

```
X: 0, Y: 32
X: 32, Y: 32
```

Try writing similar versions of the macro for floating-point numerics and strings. Be the first kid on your block to collect all three!

So at this point, you can do some basic querying of stack information, and you can push and pop stack values of any data type, as well as perform random access to arbitrary stack indexes (thereby treating it like an array). That's pretty much everything you'll need, but there are a few remaining stack issues to discuss.

First of all, because you now have the ability to read from anywhere in the stack, you should read a bit more about what a valid stack index is. Remember that the Lua stack *always* starts from 1.

Because of this, 0 is never a valid index (unlike tables) and should not be used. Past that, valid indexes run from 1 to the size of the stack. So, if you have a stack of four elements, 1, 2, 3, and 4 are all valid indexes.

One interesting facet of Lua stack access, however, is using a negative number. At first this may seem strange, but using a negative has the effect of accessing the stack "in reverse," so to speak. Index 1 always points to the bottom of the stack, whereas -1 always points to the top. Going back to the example of a four-element stack, consider the following. If index 1 points to the bottom, so does index -4. If index 4 points to the top, so does -1. The same goes for the other elements: element 2 can be indexed with either 2 or -3, whereas element 3 can be accessed with either 3 or -2. Basically, you can always access the stack either relative to the top or relative to the bottom, depending on which is most convenient. Figure 6.12 helps illustrate this concept.

Lastly, let's take a look at a few extra functions Lua provides for determining the type of a given stack element without removing or copying it into a variable first.

```
void lua_type ( lua_State * pLuaState, int iIndex );
void lua_isnil ( lua_State * pLuaState, int iIndex );
void lua_isnumber ( lua_State * pLuaState, int iIndex );
void lua_isstring ( lua_State * pLuaState, int iIndex );
```

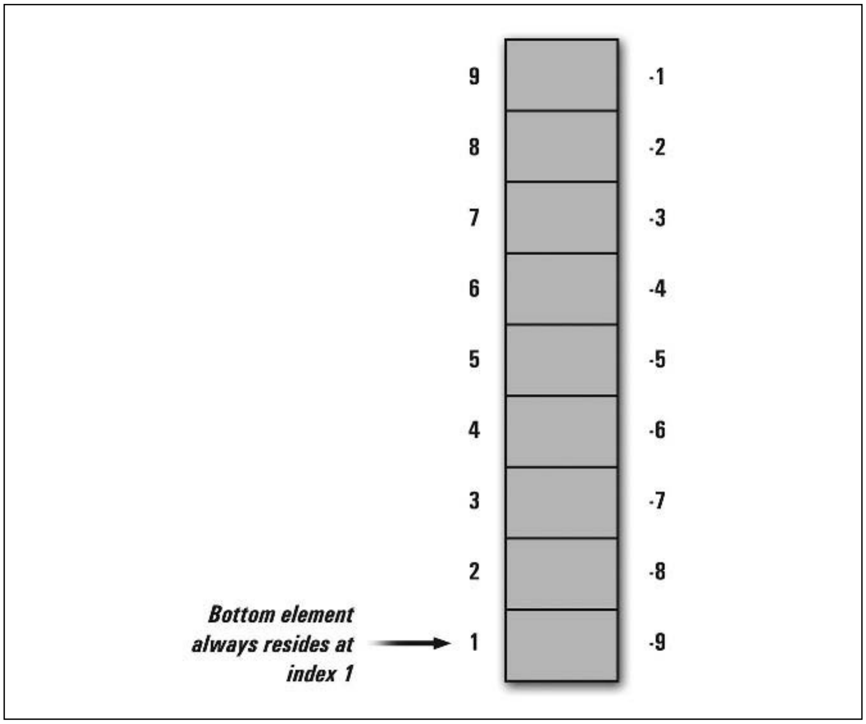


Figure 6.12

Stacks can be accessed relative to either the top or bottom element, depending on the sign of the index. Positive indexes work from the bottom up, whereas negatives work from the top down.

The first function, lua_type (), returns one of a number of constants referring to the type of the element at the given index. These constants are shown with a description of their meanings in Table 6.5.

Table 6.5 lua_type () Return Constants

Constant	Description
LUA_TNIL	nil
LUA_TNUMBER	Numeric: int, long, float, or double.
LUA_TSTRING	String
LUA_TNONE	Returned when the specified index is invalid. Nice job, slick!

The other lua_is* () functions work in the same way, but simply return 1 (true) or 0 (false) if the specified index is compatible with the given type. So for example, calling lua_isnumber (pLuaState, 8), will return 1 if the element at index 8 is numeric, and 0 otherwise. As you'll learn later in this section, Lua passes parameters to C functions on the stack; when writing a C function that Lua can call, these functions can be useful when attempting to determine whether the parameters passed are of the proper types.

Exporting C Functions to Lua

The process of making a function of the host application callable from Lua (or any scripting system, for that matter) is called *exporting*. To export a function from C to Lua, you simply need to pass a function pointer to the Lua runtime environment, as well as a string containing a name the function should be known by inside the scripts. Lua provides a simple function for this (actually, it's a macro), as follows:

```
lua_register ( lua_State * pLuaState, const char *
pstrFuncName, lua_CFunction pFunc );
```

Given a function name string, the actual function pointer (I'll cover the lua_CFunction structure in a second) and the specific Lua state to which this function should be exported, lua_register (), will *register* the function, which allows scripts to refer to it just like any other function. For example, the following script is considered valid if a C function called CFunc () is exported to the state in which it runs:

```
function MyFunc0 ( X, Y )
	-- ...
end
function MyFunc1 ( Z )
	-- ...
end
MyFunc0 ( 16, 32 );
MyFunc1 ( "String Parameter" );
CFunc ( 2, 4.8, "String Parameter" );
```

Of course, if `CFunc ()` is not exported, this will produce a runtime error. Notice, however, that the syntax for calling the C function is identical to any other Lua function, including parameter passing. Speaking of parameters, one detail to remember is that exported C functions do not have well-defined signatures. You can pass any number of parameters of any primitive data type and Lua won't complain. It's the C function's responsibility to sort out the incoming parameters.

To get a feel for how this actually works in practice, let's create that text-printing function discussed earlier, so your subsequent scripts can communicate with you through the console.

The first step, of course, is to write the function. The first attempt at a `printf ()` wrapper might look like this:

```
void PrintString ( char * pstrString )
{
	printf ( pstrString );
	printf ( "\n" );
}
```

This simple wrapper does nothing more than pass `pstrString` to `printf ()` and follow it up with a newline. This is fine as a general-purpose `printf ()` wrapper, but it's not going to work with Lua. Lua requires any C-defined functions to follow a specific function signature, so it can easily maintain a list of function pointers. The prototype of a Lua-compatible C function must look like this:

```
int FuncName ( lua_State * pLuaState );
```

Not only is this signature quite a bit different than the `PrintString ()` wrapper, it looks like it would work only for a function that doesn't require any parameters (aside from the Lua state) and always returns an integer, doesn't it? The reason all functions can follow this same format is because parameters from Lua and return values to Lua are not handled in the same way as they are in C. Both incoming parameters and outgoing results are pushed onto the Lua stack.

Because all incoming parameters are on the stack, you can use Lua's stack interface functions to read them. Remember, at the time your function is called, Lua will make it seem as if the stack is currently empty (whether it is or not), so all of your stack accessing will be relative to element index 1. At the beginning of your C function, the stack will be entirely empty except for any parameters that the Lua caller may have passed. Because of this, the size of the stack is always synonymous with the number of parameters the caller passed, and thus, you can use `lua_gettop ()`.

Once you know how many parameters have been passed, you can read them using Lua's `lua_to* ()` functions, although you'll need to know what data type you're looking for ahead of time. So, if you wrote a function whose parameter list looked like this:

```
( integer X, float Y, string Z )
```

You could read these three parameters like this:

```
int X = ( int ) lua_tonumber ( pLuaState, 1 );
float Y = lua_tonumber ( pLuaState, 2 );
char * Z = lua_tostring ( pLuaState, 3 );
```

Notice that parameter `X` was at index 1, `Y` was at index 2, and `Z` was at index 3. Lua always pushes its parameters onto the stack in the order they're passed.

Values can be returned in the opposite manner, by pushing them onto the stack before the C function returns. Like passed parameters, return values are pushed onto the stack in the order in which they should be received. Remember, Lua supports multiple assignment and thus multiple return values from functions. If this hypothetical function were to return three more numeric values, the code would look something like this:

```
lua_pushnumber ( pLuaState, 16 );
lua_pushnumber ( pLuaState, 32 );
lua_pushnumber ( pLuaState, 64 );
return 3;
```

TIP

Remember, you can always use the `lua_is* ()` functions to validate the data type of the passed parameters. This is especially important because Lua won't force the caller of a host API function to follow a specific prototype, and you have no other way of knowing for sure that the passed parameters are valid.

Notice that the function returns an integer value corresponding to the number of result values the function should return to Lua (3 in this case). This is very important, as it helps Lua clean up the stack properly afterwards, and can lead to stack corruption errors if this number is not correct. Let's imagine this C function is exported under the name `CFunc ()`. If it's called from Lua in order to return three values, the variables in the following code:

```
U, V, W = CFunc ( X, Y, Z );
```

would be filled in the same order you pushed the values. So, `U` would be set to 16, `V` to 32, and `W` to 64.

So you're now capable of registering a C function with Lua, as well as receiving parameters and returning results. That's pretty much everything you need, so let's have a go at implementing that `printf ()` wrapper mentioned earlier. I'll just show you the code up front and I'll dissect it afterwards:

```
int PrintStringList ( lua_State * pLuaState )
{
    // Get the number of strings
    int iStringCount = lua_gettop ( pLuaState );
    // Loop through each string and print it, followed by a newline
    for ( int iCurrStringIndex = 1; iCurrStringIndex <=
        iStringCount; ++ iCurrStringIndex )
    {
        // First make sure that the current parameter on the
        // stack is a string
        if ( ! lua_isstring ( pLuaState, 1 ) )
        {
            // If not, print an error
            lua_error ( pLuaState, "Invalid string." );
        }
        else
        {
      // Otherwise, print a tab, the string, and finally a newline
      printf ( "\t" );
      printf ( lua_tostring ( pLuaState, iCurrStringIndex ) );
      printf ( "\n" );
        }
    }
    // Return zero, as this function does not return any results
    return 0;
}
```

As you can see the function is now called `PrintStringList ()` and accepts a variable number of string parameters, which are then printed, indented by one tab, and followed by a newline. The function starts with a call to `lua_gettop ()`, which, as you remember, can be used to get the number of parameters when writing host API functions. This value is put in `iStringCount`, and a `for` loop begins in which each string is read from the stack and then printed to the screen. `lua_isstring ()` is used to validate each string. If the parameter is of a non-string type, `lua_error ()` is called. You haven't seen this function before, so I'll take a moment to explain it. Designed for use in console applications, `lua_error ()` accepts a Lua state and a string parameter and halts the current script just before printing the supplied message. Here's the prototype, just for reference:

```
void lua_error ( lua_State * pLuaState, char * pstrMssg );
```

Getting back on track, the rest of the loop deals with reading the string from the stack using `lua_tostring ()` and printing it to the screen (in between the tab and newline characters). The function is finished when the loop ends, and it returns 0 because there were no results to be returned to the Lua caller. Notice also that the parameters passed on the stack are not popped off by the function; this is handled automatically by the Lua runtime environment.

NOTE

When writing host API functions, it helps to be aware that Lua will always ensure that there is at least a minimum number of stack elements available. This number is stored in the `lua.h` constant `LUA_MINSTACK` (which is set to 16, by default). This means that no matter what, your function will always have at least `LUA_MINSTACK` stack elements to work with, although it's always good practice to make sure of this with `lua_stackspace ()`.

Executing Lua Scripts

Now that you have your `PrintStringList ()` written and exported, you're ready to write your first Lua script and watch it execute from within your C host. This first script will be decidedly simple; all you need to do right now is print out a few strings to make sure everything is working right. Once you know you have set everything up correctly, you can accomplish more complex tasks.

This first script will pretty much just do some variable assignment and pass some strings to `PrintStringList ()` to display the results. Let's check it out:

```
-- Create a full name string
FirstName = "Alex";
LastName = "Varanese";
FullName = "Name: " .. FirstName .. " " .. LastName;

-- Now put the floating point value of pi into a string
Pi = 3.14159;
PiString = "Pi: " .. Pi;       -- Numeric values can be automatically coerced to
strings

-- Test some logic
X = 0;                         -- Try setting this to nil instead of zero
```

```
if X then
   Logic = "X is true.";   -- Remember, only nil is considered false in Lua
else
   Logic = "X is false.";
end

-- Now call your exported C function for printing the strings
PrintStringList ( "Random Strings:", "" ); -- The extra empty
                                           -- string is just to
                                           -- create a blank line
PrintStringList ( FullName, PiString, Logic );
```

The first part of the script, called `test_0.lua`, creates two string variables, `FirstName` and `LastName`, and uses the `..` string concatenation operator to combine them into `FullName`. The next part uses a floating-point value to create a string containing the first few digits of pi. Notice that Lua automatically casts, or *coerces*, the floating-point value into a valid string. Next, you create the last string, `Logic`, by setting it to one of two different values depending on whether the variable `X` evaluates to true. This illustrates Lua's definition of truth as any non-`nil` value.

Lastly, with all three strings ready (`FullName`, `PiString`, and `Logic`), you make two calls to `PrintStringList ()` to display them on the console provided by the host C program. Once again, note that the syntax for calling the exported C function was typical Lua syntax, which allows your C functions to blend seamlessly into your Lua-defined functions (even though this script didn't have any).

Returning to the C side of things, your host application's `main ()` function starts with this:

```
// Initialize a Lua state and set the stack size to 1024
lua_State * pLuaState = lua_open ( 1024 );

// Register your simple function with the Lua state for
// printing text strings
lua_register ( pLuaState, "PrintStringList", PrintStringList );

// Print the title
printf ( "Lua Integration Example\n\n" );

// Execute your first test script, which just prints
// random strings
printf ( "Executing Script test_0.lua:\n\n" );
lua_dofile ( pLuaState, "test_0.lua" );
```

All that's necessary to run this script is to initialize Lua with a call to `lua_open ()`, register the `PrintStringList ()` function with `lua_register ()`, and finally load and execute the script in one fell swoop with `lua_dofile ()`. The output of this program will look like this:

```
Lua Integration Example

Executing Script test_0.lua:

        Random Strings:

        Name: Alex Varanese
        Pi: 3.14159
        X is true.
```

Thanks to `PrintStringList ()`, you can be sure that everything went smoothly because the results are right there on the console. Now that you have a simple framework built up for executing Lua, you can try your hand at a more sophisticated example.

Importing Lua Functions

You're probably not too surprised to learn that the opposite of exporting a function from C is importing one from Lua. Naturally, *importing* a function is the process of making that function callable from C, which means that Lua can not only take advantage of C functions you've already written, but your host application can capitalize on any useful functions you may have written in your scripts.

The next script will be primarily focused on demonstrating this concept. To begin, you're going to write a new script, one that defines two functions. The first function will be called `Exponent ()`, and, given two parameters `X` and `Y`, will return `X ^ Y`. The second function, `MultiplyString ()`, will *multiply* a string, which basically just means repeating a string a specified number of times. In other words, `"Hello"` multiplied by four produces the following:

```
HelloHelloHelloHello
```

Although these two functions are indeed simple, they prove educational; between the two of them, they will demonstrate:

- How a Lua function is called from C.
- How both numeric and string parameters are passed to a Lua function from a C host.
- How both numeric and string results can be returned to the C host from Lua functions.

Which is just about everything you need to know about function importing.

Let's get this new script started, which is called `test_1.lua`, with the `Exponent ()` function:

```
-- Manually computes exponents in the form of X ^ Y
function Exponent ( X, Y )
       -- First, let's just print out the parameters
       PrintStringList ( "Calculating " .. X ..
            " to the power of " .. Y );
       -- Now manually compute the result
       Exponent = 1;
       if Y < 0 then
              Exponent = -1;      -- Just return -1
                                        -- for all negative exponents
       elseif Y ~= 0 then
              for Power = 1, Y do
                    Exponent = Exponent * X;
              end
       end
       -- Return the final value to C
       return Exponent;
end
```

To make the function more substantial, I've chosen to implement the exponent function with a manual loop that multiplies 1 value by itself `Y` times. Of course, Lua provides a built-in exponent operator with `^`, so there'll be no need for you to do this in practice. Regardless, it works by first setting `Exponent` to 1 and immediately checking for some alternative cases. The first case is a negative power; which isn't supported by the function. Instead, -1 is returned in all such cases. Next, you check to make sure you aren't raising `X` to the power of zero. If so, you only need to return `Exponent` as is, because raising anything to zero yields 1. Lastly, you handle a valid exponent with the loop described previously. The function concludes with the `return` keyword, which returns the final exponent value to C.

You'll notice I start the function with a call to `PrintStringList ()` that prints a brief message. I do this just to keep some variety going in the C/Lua interaction. Without a simple call to this function, the script would consist entirely of Lua calls, which doesn't illustrate real-world scripting quite as well.

The other function `test_1.lua` will provide is `MultiplyString ()`:

```
-- "Multiplies" a string; in other words, repeats a string
-- a number of times
function MultiplyString ( String, Factor )
       -- As with the above function, print out the parameters
```

```
        PrintStringList ( "Multiplying string \""
             .. String .. "\" by " .. Factor );
        -- Multiply the string
        NewString = "";
        for X = 1, Factor do
                NewString = NewString .. String;
        end
        -- Return the multiplied string to C
        return NewString;
end
```

This function is even simpler than Exponent. All it does is create a variable called NewString and assign it the empty string. NewString will contain the multipled string and is what you'll return to C. You then enter a simple for loop which repeatedly appends String to NewString, once again using the .. operator.

With these two functions saved in test_1.lua, you can return to your C host program and add the new code necessary to test it.

TIP

Remember, you can always optionally compile your scripts. Generally, it's easier to skip the compilation step while you're initially coding and debugging them, but once they're finished, don't forget to run them through luac. lua_dofile () is capable of loading both compiled and uncompiled scripts, so you won't have to change your C host (except to change the filename to refer to the compiled version, if it's different). Recall that compiled scripts load faster, are less error-prone, and are much less vulnerable to hacking.

The C side of things will get a little more complicated than it's been so far, but it's still nothing you can't handle. The first thing to understand is that lua_dofile () will no longer immediately execute anything when test_1.lua is loaded. This is because, unlike your previous script, there isn't any code in the global scope. It's like writing a C program without main (). Because all code resides in functions, the Lua runtime environment won't run anything until those functions are called. Because the script never calls any of these functions, in the global scope, nothing ever executes. lua_dofile () has now effectively become a pure script loader, at least conceptually (it will still attempt to run the script, even though nothing will happen).

Once the script is in memory, you can freely call any of its functions at will. Lua doesn't have a particularly high-level mechanism for calling functions, so you'll have to do things fairly manually using the stack. Fortunately, it's still a pretty straightforward process. Have a look.

In Lua, functions can be thought of as *globals*, just as much as global variables can be thought of as globals. This doesn't mean they're any more like variables than C functions are, but they can be referred to this way. The first thing you need to do when calling a function is push a reference to the function onto the stack. Because functions are simply another global, you can use `lua_getglobal ()` to do the job:

```
lua_getglobal ( pLuaState, "FuncName" );
```

Where `FuncName` is a string value that corresponds to the name of the function within the script. Once the function reference is on the stack, you need to push its parameters on as well. Parameters are pushed onto the stack in left-to-right order. If `FuncName` looks like this:

```
function FuncName ( IntParam, StringParam )
```

And we want to essentially call it like this:

```
FuncName ( 256, "Hello!" );
```

The parameters would be pushed onto the stack like this:

```
lua_pushnumber ( pLuaState, 256 );
lua_pushstring ( pLuaState, "Hello!" );
```

Simple, eh? Now that the function call is represented on the stack in its entirety, you deliver the coup-de-grace by calling `lua_call ()`, which looks like this:

```
lua_call ( lua_State * pLuaState, int ParamCount, int ResultCount );
```

This function will call whatever function was most recently pushed onto the stack, passing `ParamCount` parameters and expecting `ResultCount` results. Remember, due to the multiple assignment capabilities of Lua, functions can return multiple values. If `FuncName ()` accepts the two parameters listed previously and returns one result, the call to `lua_call ()` would look like this:

```
lua_call ( pLuaState, 2, 1 );
```

Lastly, you need to know how to retrieve the result. The result (or result*s*, depending on how many the function returns) will be left on the stack. In your case, assuming `FuncName ()` returned a single integer result, you can use the following code to read it:

```
int iResult = ( int ) lua_tonumber ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
```

You use `lua_tonumber ()` to convert the element at index 1 of the stack to a double-precision floating-point value, and then cast it to an integer to store in the receiving variable. You know the return value is at index 1 because the function only returns one value. The stack is then cleaned up using `lua_pop ()` to remove the return value and bring balance to the force.

That's everything there is to know about basic Lua function calls from the host application. Now that you know what you're doing, let's go back to `test_1.lua` and try calling your `Exponent ()` and `MultiplyString ()` functions.

```
printf ( "\nLoading Script test_1.lua:\n\n" );
lua_dofile ( pLuaState, "test_1.lua" );

// Call the exponent function
// Call lua_getglobal () to push the Exponent ()
// function onto the stack
lua_getglobal ( pLuaState, "Exponent" );
// Push two numeric parameters
lua_pushnumber ( pLuaState, 2 );
lua_pushnumber ( pLuaState, 13 );
// Call the function with 2 parameters and 1 result
lua_call ( pLuaState, 2, 1 );
// Pop the numeric result from the stack and print it
int iResult = ( int ) lua_tonumber ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
printf ( "\tResult: %d\n\n", iResult );

// Call the string multiplication function
// Push the MultiplyString () function onto the stack
lua_getglobal ( pLuaState, "MultiplyString" );
// Push a string parameter and the numeric factor
lua_pushstring ( pLuaState, "Location" );
lua_pushnumber ( pLuaState, 3 );
// Call the function with 2 parameters and 1 result
lua_call ( pLuaState, 2, 1 );
// Get the multiplied string and print it
const char * pstrResult;
pstrResult = lua_tostring ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
printf ( "\tResult: \"%s\"", pstrResult );
```

Everything should pretty much speak for itself; all I've done here is directly applied the technique for calling Lua functions described previously.

At this point, you've learned quite a bit; once you have the ability to call functions from both the host application and the running script, along with parameters and return values, you're pretty much prepared for anything. Most of the interaction between these two entities will lie in function calls. Because you've learned the language as well, you should be familiar enough with Lua in general to get started with your own experiments and exploration. Of course, you still need to get back to the bouncing alien head demo, but before that, there's one last detail of interaction I'd like to show you.

Manipulating Global Lua Variables from C

The last real piece of the C/Lua integration puzzle I'm going to cover is the manipulation of a script's global variables from C. Because globals are often used to control the program on a high level, there are times when you can direct and manipulate the general behavior of your scripts with nothing more than the reading and writing of globals. I personally prefer to keep everything function-based. Rather than directly editing a global variable, I like to assign that global a pair of "setter and getter" functions, which allow me to alter the global's value indirectly and subsequently more safely. However, you're ultimately the one who has to decide how your game's scripts will work, so here's an extra technique for your arsenal in case you personally consider it a better way to go.

As you've seen to some extent, the `lua_getglobal ()` and `lua_setglobal ()` functions can be used to read and write globals indirectly through the stack. Calling `lua_getglobal ()` causes the value of the specified global variable to be pushed onto the stack, whereas `lua_setglobal ()` will pop the value off the top of the stack into the specified global. So, for example, if you wanted to set the value of an integer global called `X`, you simply do the following:

```
lua_pushnumber ( pLuaState, 256 );  -- Push 256 onto the stack
lua_setglobal ( pLuaState, "X" );      -- Move the top stack value into X
```

It's simply a matter of pushing the desired value onto the stack and using lua_`setglobal ()` to move it into place. Likewise, the integer value of `X` could be read with the following code:

```
lua_getglobal ( pLuaState, X );    -- Push X's value onto the stack
int X = ( int ) lua_tonumber ( pLuaState, 1 );  -- Grab the top stack value
```

All you need to do is push the given global's value onto the stack and then convert the value at that index to an integer to store in a C variable. Once again, you're assuming that the stack is empty at the time of the call to `lua_getglobal ()`, which means the value will be placed at index 1. Because this may not always be the case, be sure to use `lua_gettop ()` in practice to get the proper index of the stack's top value. Also, remember to clear the stack off when you're done; calls to `lua_getglobal ()` should generally be followed by a call to `lua_pop ()`.

Let's finish `test_1.lua` by adding some global variables to manipulate. Before the definition of your two functions, let's add the following:

```
GlobalInt = 256;
GlobalFloat = 2.71828;
GlobalString = "I'm an obtuse man...";
```

This gives you three globals to work with, all of differing types. To get things started, let's just try reading their values and printing them from C:

```
// Read some global variables
printf ( "\n\tReading global variables...\n\n" );

// Read an integer global by pushing it onto the stack
lua_getglobal ( pLuaState, "GlobalInt" );
printf ( "\t\tGlobalInt: %d\n", ( int )
    lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Read a float global
lua_getglobal ( pLuaState, "GlobalFloat" );
printf ( "\t\tGlobalFloat: %f\n", lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Read a string global
lua_getglobal ( pLuaState, "GlobalString" );
printf ( "\t\tGlobalString: \"%s\"\n", lua_tostring
    ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
```

Let's expand the example just a bit to write new values to the globals. Of course, you'll re-read them as well to make sure the writes worked:

```
// Write the global variables and re-read them
printf ( "\n\tWriting and re-reading the global variables...\n\n" );

// Write and read the integer global
lua_pushnumber ( pLuaState, 512 );
lua_setglobal ( pLuaState, "GlobalInt" );
lua_getglobal ( pLuaState, "GlobalInt" );
printf ( "\t\tGlobalInt: %d\n", ( int ) lua_tonumber
    ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
```

```
// Write and read the float global
lua_pushnumber ( pLuaState, 3.14159 );
lua_setglobal ( pLuaState, "GlobalFloat" );
lua_getglobal ( pLuaState, "GlobalFloat" );
printf ( "\t\tGlobalFloat: %f\n", lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Write and read the string global
lua_pushstring ( pLuaState, "...so I'll try to be oblique." );
lua_setglobal ( pLuaState, "GlobalString" );
lua_getglobal ( pLuaState, "GlobalString" );
printf ( "\t\tGlobalString: \"%s\"\n", lua_tostring ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
```

Done and done. The last thing to add to your C host is a call to `lua_close ()` to clean everything up:

```
lua_close ( pLuaState );
```

Re-coding the Alien Demo in Lua

Aside from Vader, one last challenge remains. As I mentioned earlier, one of your exercises as you learn each language will be to recode the bouncing alien head demo I showed you at the beginning of the chapter.

Initial Evaluations

As I mentioned earlier, all you really want to do with Lua is set the initial location, velocity, and spin direction of each sprite with the script, as well as produce each frame of the demo by moving the sprites around the screen and handling collisions.

The first thing you need to do is decide exactly what the script will be in charge of. Once you know this, you can establish an appropriate host API— a set of functions that will give the script the capabilities it needs to carry out its tasks.

Because your script will first be responsible for initializing the sprites, let's break down exactly what this entails:

- Set the initial X, Y coordinates to a random on-screen location.
- Set the initial X, Y velocity to random values.
- Set the initial spin direction to a random value (0 or 1).
- Store these values in a script-defined table, just as the original C version stored them in an array.

In short, you need to create a table within the script that will hold all of your bouncing alien heads; each element of the array needs to describe its corresponding alien head in the same way that the `Alien struct` did in the hardcoded version. Obviously, table manipulation is built in to Lua, so you don't need to provide any functionality for that from the host app. What you do need to provide, however, is a function that can generate random numbers.

Once initialization is complete, your script won't be called again until the main loop of the application has begun. Once this takes place, the script will be called once per frame. At each frame, the script will be in charge of the following tasks:

- Blit the background image.
- Loop through each alien in the table and draw it at its current location.
- Blit the completed frame to the screen.
- Update the current frame of animation when the animation timer is active.
- Loop through each alien in the table once again to move it along its current path, and handle collisions as they occur when the movement timer is active.

As you can see, the per-frame part of the script will be required to do a lot more things that Lua isn't directly capable of, so the bulk of your host API will be geared towards these needs. Now that you know what you need, let's lay these functions out.

The Host API

As you've seen, your primary requirements will be generating random numbers, blitting various bitmapped images, and checking the status of timers. With these needs in mind, your host API will look like this:

```
int HAPI_GetRandomNumber ( lua_State * pLuaState );
int HAPI_BlitBG ( lua_State * pLuaState );
int HAPI_BlitSprite ( lua_State * pLuaState );
int HAPI_BlitFrame ( lua_State * pLuaState );
int HAPI_GetTimerState ( lua_State * pLuaState );
```

Notice that I've preceded each of the function names with `HAPI_` (which of course stands for "**H**ost **API**"). This ensures that your host API functions and C-only functions are kept separate. This is just good practice in general when scripting with any language.

As for the functions, they should be fairly self-explanatory, but I'll go over them just in case there's any ambiguity:

- `HAPI_GetRandomNumber ()` accepts two numeric parameters; minimum and maximum values that define a range from which a random number should be chosen and returned to the caller.

- HAPI_BlitBG () is a simple function that causes the background image to be blitted to the framebuffer. No parameters are necessary.
- HAPI_BlitSprite () accepts parameters referring to an X, Y location and an index into the array of frames of the spinning alien head animation.
- HAPI_BlitFrame () is another simple function that blits the framebuffer to the screen. Like HAPI_BlitBG (), no parameters are needed.
- HAPI_GetTimerState () this function accepts a single numeric parameter containing an index that refers to a specific timer. The state of that timer (1 for active, 0 for inactive) is returned to the caller.

With the host API laid out, let's take a look at the new structure of the host application.

The New Host Application

The landscape of the C side of things is quite a bit different now that you're offloading a good portion of the demo's functionality to Lua. Gone is much of the original code, and in its place you find the host API and a number of calls to the Lua system. Speaking of the host API, its one of the biggest changes (or additions, I should say). Have a look at the definitions for a few of the host API functions:

```
int HAPI_GetRandomNumber ( lua_State * pLuaState )
{
        // Read in parameters
        int iMin = GetIntParam ( 1 );
        int iMax = GetIntParam ( 2 );
        // Return a random number between iMin and iMax
        ReturnNumber ( ( rand () % ( iMax + 1 - iMin ) ) + iMin );
        return 1;
}
```

HAPI_GetRandomNumber () does its job in two phases; first the parameters are read in, and then the result is sent out. You start by declaring two integer variables, iMin and iMax, and initialize them with the values returned from GetIntParam (). Wait a second, "GetIntParam ()"? What was that?

Throughout the process of rewriting the alien head demo with Lua, there appeared a number of places where macros that wrapped the calls to the actual Lua functions made things a lot cleaner. For example, when a host API function wants to read in an integer parameter, it has to do something like this:

```
int iParam = ( int ) lua_tonumber ( pLuaState, iIndex );
```

First of all, the function lua_tonumber () itself isn't the most intuitive name, at least in this context. What the function is really doing is reading the stack element at iIndex and returning it as a numeric value. At least, that's how things are working internally. All you need to worry about, however, is that the function is returning a parameter. So right off the bat, wrapping it in a macro that provides a more descriptive name will result in improved code readability. Second, you have to cast the value the function returns to an `int` because Lua works only with floating-point numerics. Having this cast clog up your code everywhere is just going to make things messier, so the following macro:

```
#define GetIntParam( Index ) \
       ( int ) lua_tonumber ( g_pLuaState, Index );
```

just makes everything cleaner, more descriptive, and more concise. This is a trend that you'll find continues throughout this section, so be prepared for a few more macros along these lines.

Where were we? Oh right, `HAPI_GetRandomNumber ()`. Anyway, once you read in the `iMin` and `iMax` parameters, you use another macro, `ReturnNumer ()`, to return the result of a call to the standard C `rand ()` function. `ReturnNumer ()` is very similar to `GetIntParam ()`, except that it of course automates the process of returning a numeric. Let's look at the code:

```
#define ReturnNumer( Num ) \
       lua_pushnumber ( g_pLuaState, Num );
```

Much nicer, eh? Another plus to these macros is that they save you from having to manually pass that Lua state every time you make a Lua call as well. Of course, if you find yourself writing programs that maintain multiple states (which you most likely will, because that's how you implement multiple scripts running at once), you'll lose this luxury.

Overall, `HAPI_GetRandomNumber ()` illustrates an important point when discussing host APIs, because all it really boiled down to was a simple wrapper for `rand ()`. You may find that a large portion of your host API functions don't provide any unique functionality of their own. Rather, they'll usually just wrap existing functions to make the same functions your C program uses accessible to your scripts. Don't worry if you find yourself doing a lot of this. At first it may seem like a lot of extra coding for nothing, but it's the only way to provide your scripts with the functions they're ultimately going to need to be useful.

Let's check out one more host API function, and then I'll move on:

```
int HAPI_BlitSprite ( lua_State * pLuaState )
{
       // Read in parameters
       int iIndex = GetIntParam ( 1 );
       int iX = GetIntParam ( 2 );
       int iY = GetIntParam ( 3 );
```

```
        // Blit sprite
        W_BlitImage ( g_AlienAnim [ iIndex ], iX, iY );
        // Return nothing
        return 0;
}
```

Again, you see a similar process. First you read in three integer parameters with your handy `GetIntParam ()` macro. You then pass those parameters directly to the Wrappuh function `W_BlitImage ()`, which performs the blit. Unlike `HAPI_GetRandomNumber ()`, this function does not return anything to Lua, hence the `return 0`.

Moving along, I've created two helper functions for initializing and shutting down Lua in its entirety. `InitLua ()` allows you to open the Lua state and register all of the functions in your host API in one call:

```
void InitLua ()
{
        // Open a new Lua state
        g_pLuaState = lua_open ( LUA_STACK_SIZE );
        // Register your host API with Lua
        lua_register ( g_pLuaState, "GetRandomNumber",
             HAPI_GetRandomNumber );
        lua_register ( g_pLuaState, "BlitBG", HAPI_BlitBG );
        lua_register ( g_pLuaState, "BlitSprite", HAPI_BlitSprite );
        lua_register ( g_pLuaState, "BlitFrame", HAPI_BlitFrame );
        lua_register ( g_pLuaState, "GetTimerState", HAPI_GetTimerState );
}
```

Notice that the host API functions are not exposed to Lua scripts with the `HAPI_` prefix. I did this because there are so few functions in the script (as you'll soon see), that there's no need to differentiate. Of course, for large script projects you may find it useful to precede your function names with `HAPI_` on both the C and Lua sides of things.

`LUA_STACK_SIZE` is just a constant I've set to 1024. Nothing new.

`InitLua ()` of course has a matching `ShutDownLua ()`, although this function is a bit of a waste, because it only encapsulates one line:

```
void ShutDownLua ()
{
        // Close Lua state
        lua_close ( g_pLuaState );
}
```

What can I say? I'm a bit of a neat-freak, so `InitLua ()` had to have a matching `ShutDown ()` function, whether it was necessary or not. :) It would just seem lopsided without one!

After the call to `InitLua ()`, you'll have a valid Lua state and your host API will be locked and loaded. It's here where the scripting really begins. After all of your C-side initialization is done, you can initialize your alien head sprites with one call:

```
CallLuaFunc ( "Init", 0, 0 );
```

That's right, another macro has reared its head. This one, aptly entitled `CallLuaFunc ()`, calls Lua functions. (Honestly, sometimes I wish my function names were less descriptive—it makes the explanations of what they mean seem so anticlimactic.) Normally, because a Lua function call involves using `lua_getglobal ()` to put the function reference onto the stack, and then calling `lua_call ()`, this macro helps you out a bit by reducing everything to a single line:

```
#define CallLuaFunc( FuncName, Params, Results ) \
{ \
      lua_getglobal ( g_pLuaState, FuncName ); \
      lua_call ( g_pLuaState, Params, Results ); \
}
```

Just pass it a string containing the function name, the number of parameters, and the number of results.

Anyway, the call to the Lua script was in reference to a function called `Init ()`, as you can see. Because I haven't covered the contents of the script yet, just take this on faith.

Immediately following the call to your script's `Init ()` function, the main loop of the demo begins, which is now rather minimalist because its guts have been transferred to Lua:

```
// Start the main loop
MainLoop
{
       // Start the current loop iteration
       HandleLoop
       {
            // Let Lua handle the frame
            CallLuaFunc ( "HandleFrame", 0, 0 );
            // Check for the Escape key and exit if it's down
            if ( W_GetKeyState ( W_KEY_ESC ) )
                   W_Exit ();
       }
}
```

Another call to `CallLuaFunc ()`, and another script function you haven't yet seen. This one is called `HandleFrame ()`, and naturally, handles the current frame by moving the sprites around. Once again, you'll see these two functions in the next section.

That's it! In summary, the new host application works by first defining a series of functions that collectively form the host API, and then initializes Lua by using `lua_open ()` to create a Lua state and register the host API's functions. At this point, the Lua system is all ready to go, and the script's two functions are called. First `Init ()` is called to initialize the sprites, and `HandleLoop ()` is called once per frame to move them around. Because you're done with the C stuff, you can now move on and actually see these two functions (among other things).

The Lua Script

The Lua script, which I've given the almost frighteningly creative filename `script.lua`, is the only one you'll need for this demo. In it, there are four main elements, as follows:

- An area for declaring constants.
- An area for declaring global variables.
- The first function, `Init ()`.
- The second (and last) function, `HandleFrame ()`.

As you can see, a script is structured in the same way a program is, something you'll discover in more and more depth as your mastery of scripting unfolds. Although scripts and programs are indeed fundamentally and technically different things; they're conceptually the same in most respects.

As I said, your script will consist mostly of a constant declaration section, a global variable declaration section, and two functions. Notice again that there is no code in the global scope—in other words, code that resides outside the functions—because it would be automatically executed by `lua_dofile ()` and you don't necessarily want anything to be run at that time. Rather, you'd like Lua to simply load the file into memory for you and let it sit for you to reference later through function calls when you need to.

TIP

Even though this script example has no code in the global scope, and thus no code that automatically runs after the call to `lua_dofile ()`, this isn't always something to avoid. If your script has a block of initialization code that you know you're only going to call once at the time the script is loaded, you might as well put this code in the global scope so `lua_dofile ()` automatically executes it for you. To put it in C++ terms, think of it as a "constructor" for your script.

Remember, loading a script involves a decent amount of hard drive access, format validation, and possibly even an entire compilation of the script (if your script is still in source code form). Scripts are no different than bitmaps or sounds in this respect; their loading phase is costly and should only be done outside of speed-critical code (i.e., outside of your main loop). Calling `lua_dofile ()` to execute a script on a per-frame basis would be frame rate homicide (which is only legal in Texas).

Getting back to the topic at hand, let's look at the script's constant declaration section:

```
ALIEN_COUNT       = 12;
MIN_VEL      = 2;
MAX_VEL      = 8;
ALIEN_WIDTH      = 128;
ALIEN_HEIGHT      = 128;
HALF_ALIEN_WIDTH   = ALIEN_WIDTH / 2;
HALF_ALIEN_HEIGHT   = ALIEN_HEIGHT / 2;
ALIEN_FRAME_COUNT   = 32;
ALIEN_MAX_FRAME   = ALIEN_FRAME_COUNT - 1;
ANIM_TIMER_INDEX   = 0;
MOVE_TIMER_INDEX   = 1;
```

The trick here is that Lua doesn't actually support constants. The best you can do is just pretend that it does by declaring your constant values as global variables that are written out with typical `CONSTANT_NOTATION` (like that). Lua just considers them typical globals, but at least your code will look the way you want it to. If you compare this block of code to the original hardcoded C version, you'll find that I've pretty much just copied the constant declarations and pasted them right into the Lua source.

Next up, let's have a look at your global variables

```
Aliens = {};
CurrAnimFrame = 0;
```

Only two declarations needed here. First you create a table called `Aliens` that will keep track of all of your bouncing heads. Next, you create a simple numeric called `CurrAnimFrame`, which keeps track of the current frame of the alien head animation.

With your constants and globals out of the way, you have all the data you need. Now it's time for some code. Let's have a look at the first of two functions this script will provide, `Init ()`:

```
function Init ()
       -- Initialize the alien sprites
       -- Loop through each alien in the table and initialize it
       for CurrAlienIndex = 1, ALIEN_COUNT do
              -- Create a new table to hold all of the alien's fields
              local CurrAlien = {};
```

```
            -- Set the X, Y location
            CurrAlien.X = GetRandomNumber ( 0, 639 - ALIEN_WIDTH );
            CurrAlien.Y = GetRandomNumber ( 0, 479 - ALIEN_HEIGHT );
            -- Set the X, Y velocity
            CurrAlien.XVel = GetRandomNumber ( MIN_VEL, MAX_VEL );
            CurrAlien.YVel = GetRandomNumber ( MIN_VEL, MAX_VEL );
            -- Set the spin direction
            CurrAlien.SpinDir = GetRandomNumber ( 0, 2 );
            -- Copy the reference to the new alien into the table
            Aliens [ CurrAlienIndex ] = CurrAlien;
    end
end
```

As you should remember, this is the function that's called by the following line back in the host application:

```
CallLuaFunc ( "Init", 0, 0 );
```

So, as soon as this line of code is hit, the `Init ()` function listed previously will be run.

The function really just has one job: initialize the array of bouncing alien heads. Just like in the original pure C version, this means giving each head a random location on-screen, a random velocity, and a random spin direction. Naturally, this is facilitated by a `for` loop.

To actually store the alien head demo, you need to store a smaller table at each index of the `Aliens` table. This is because there are a number of pieces of information that each head has to keep track of. To put this another way, think of it like a multidimensional array, or an array of `structs` in C. Each index of the table has another table (or rather, a *reference* to another table) that holds that particular element's information, like its X, Y location and its velocity. Check out Figure 6.13 for a visual representation of this.

All in all this is a simple concept, but there is one snag that can *really* trip you up if you're not ready for it. As I've mentioned before, it's important to think of tables in Luas references, rather than values. Because of this, assigning a table to an element of another table in a loop, like this:

```
Aliens [ CurrAlienIndex ] = CurrAlien;
```

means that `Aliens [ CurrAlienIndex ]` only receives a *reference* to the `CurrAlien` table, not the values themselves. So, at the next iteration of the loop, when you put new values into `CurrAlien` and assign it to the next index of `Aliens`, you'll find that both the current element as well as the previous element seem to suddenly have the same values. This is due to the fact that both elements have been given a reference to `CurrAlien`, so as soon as you change the values for the second element of the table in the next iteration of the loop, the first element will seem to inexplicably change along with it. Figure 6.14 illustrates this relationship.

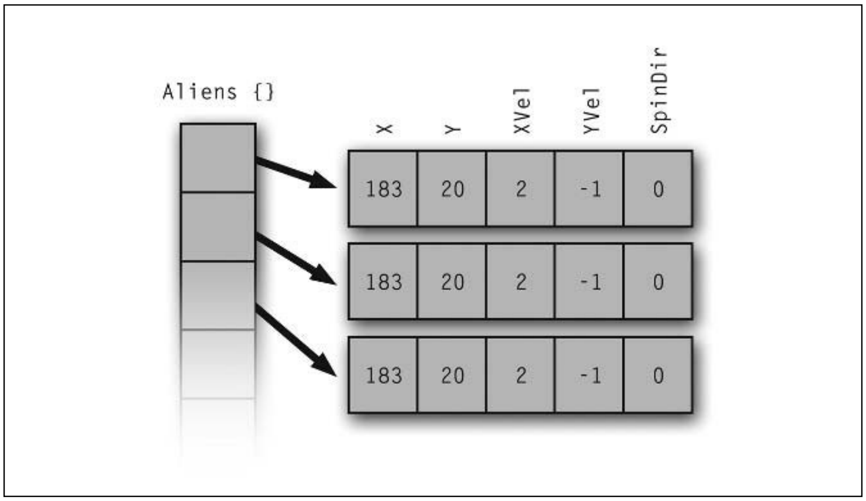


Figure 6.13

Each element of the `Aliens` table contains another table that holds that element's specific data.

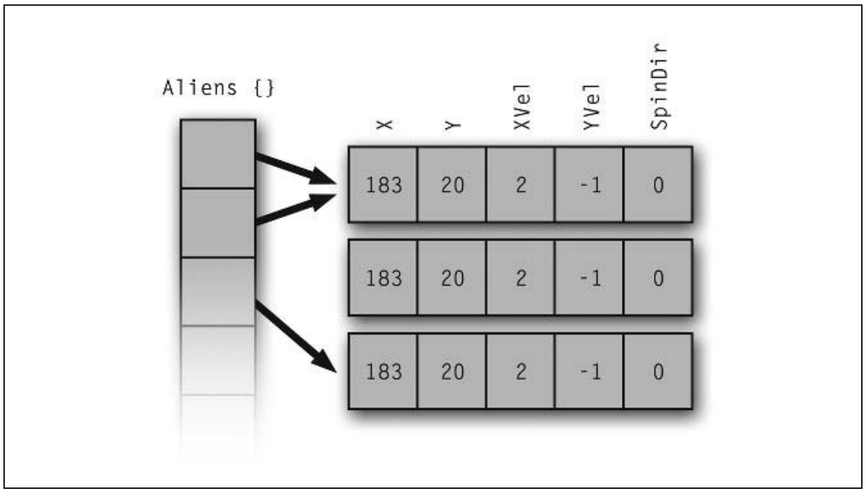


Figure 6.14

Two elements of `Aliens` point to the same table, and therefore reflect the changes made to one another.

To solve this problem, you simply start the loop with this line:

```
local CurrAlien = {};
```

Assigning `{}` to `CurrAlien` forces Lua to allocate a new table and therefore provide a fresh, unused reference. You can then fill the values of this instance of `CurrAlien` and freely assign it to the next element of `Aliens`, without worrying about overwriting the values you set in the last iteration. It's a simple problem with a simple solution, but left unchecked this little detail can cause logic errors that truly wreak havoc. :)

The rest of the alien head initialization loop is pretty much what you would expect; each element of `CurrAlien` is set to a random value, using the `GetRandomNumber ()` function that the previously discussed host API provides. Once this loop completes, `Init ()` is finished and the global `Aliens` table contains a record of every bouncing alien head.The script is now fully prepared to enter the main loop, which will call `HandleFrame ()` at each iteration. Let's have a look at this function:

```
function HandleFrame ()
       -- Blit the background image
       BlitBG ();
       -- Blit each sprite and move it along its path
       for CurrAlienIndex = 1, ALIEN_COUNT do
               -- Get the X, Y location
               local X = Aliens [ CurrAlienIndex ].X;
               local Y = Aliens [ CurrAlienIndex ].Y;
               -- Get the spin direction and determine
               -- the final frame for this sprite
              -- based on it.
              local SpinDir = Aliens [ CurrAlienIndex ].SpinDir;
              if SpinDir == 1 then
                     FinalAnimFrame = ALIEN_MAX_FRAME - CurrAnimFrame;
              else
                     FinalAnimFrame = CurrAnimFrame;
              end
              -- Blit the sprite
              BlitSprite ( FinalAnimFrame, X, Y );
       end
       -- Blit the completed frame to the screen
       BlitFrame ();
       -- Increment the current frame in the animation
       if GetTimerState ( ANIM_TIMER_INDEX ) == 1 then
              CurrAnimFrame = CurrAnimFrame + 1;
              if CurrAnimFrame >= ALIEN_FRAME_COUNT then
                     CurrAnimFrame = 0;
              end
       end
       -- Move the sprites along their paths
       if GetTimerState ( MOVE_TIMER_INDEX ) == 1 then
              for CurrAlienIndex = 1, ALIEN_COUNT do
                     -- Get the X, Y location
                     local X = Aliens [ CurrAlienIndex ].X;
                     local Y = Aliens [ CurrAlienIndex ].Y;
```

```
                        -- Get the X, Y velocities
                        local XVel = Aliens [ CurrAlienIndex ].XVel;
                        local YVel = Aliens [ CurrAlienIndex ].YVel;
                        -- Increment the paths of the aliens
                        X = X + XVel;
                        Y = Y + YVel;
                        Aliens [ CurrAlienIndex ].X = X;
                        Aliens [ CurrAlienIndex ].Y = Y;
                        -- Check for wall collisions
                        if X > 640 - HALF_ALIEN_WIDTH or X <
                            -HALF_ALIEN_WIDTH then
                              XVel = -XVel;
                       end
                       if Y > 480 - HALF_ALIEN_WIDTH or Y <
                            -HALF_ALIEN_WIDTH then
                              YVel = -YVel;
                       end
                       Aliens [ CurrAlienIndex ].XVel = XVel;
                       Aliens [ CurrAlienIndex ].YVel = YVel;
              end
       end
end
```

Quite a bit larger than `Init ()`, eh? As you can see, there's a decent amount of logic to attend to here, so let's knock it out piece by piece.

The first step is easy; you make a single call to `BlitBG ()`, a host API function that slaps the background image into the framebuffer. This overwrites the last frame's contents and gives you a fresh slate on which to draw the new frame.

You then use a `for` loop to iterate through each alien in the bouncing alien head array, saving the X, Y location and final animation frame into local variables which are passed to host API function `BlitSprite ()` to put it on the screen. Notice that you don't necessarily use the global `CurrAnimFrame` as the frame passed to `BlitSprite ()`. This is because each head has its own spinning direction, which may be forwards or backwards. If it's forwards, you can use `CurrAnimFrame` as-is, but you must subtract `CurrAnimFrame` from `ALIEN_MAX_FRAME` if it's backwards. This lets certain sprites cycle through the animation in one direction, whereas others cycle through it the other way.

At this point, you've drawn the background image and each alien sprite. All that's left to complete this frame is to call `BlitFrame ()`, another host API function, which blasts the framebuffer to the screen. The graphical aspect of the current frame has been taken care of, but now you need to handle the logic. This means moving the alien heads along their paths and checking for collisions, among other things.

The first thing to do after blitting the new frame to the screen is update CurrAnimFrame. You do this by incrementing the variable, and resetting it to zero if the increment pushes it past ALIEN-_MAX_FRAME. Of course, you want to perpetuate the animation at a fixed speed; if you incremented CurrAnimFrame every frame, the animation might move too quickly on faster systems. So, you've synchronized the speed of the animation with a timer that was created in the host application. This timer ticks at a certain speed, which means you have to use GetTimerState () at each frame to see whether it's time to move the animation along. This ensures a more uniform speed across the board, regardless of frame rate.

This takes you to the last part of the HandleFrame () function, which is the movement of each sprite and the collision check. Like the animation, the movement of the sprites is also synched to a timer, which means you make another call to GetTimerState (). Assuming the timer has completed another tick, you start by saving the X, Y coordinates of the sprite and the X, Y velocities to local variables. You then add the velocities to the X, Y coordinates to find the next position along the path the alien should move to. You put these values back into the Aliens array and then perform the collision check. If the new location of the sprite is above or below the extents of the screen, you reverse the Y velocity to simulate the bounce. The same goes for violations of the horizontal extents of the screen, which cause a reversal of the X velocity. Once these two checks have been performed, the X and Y velocities are placed back into the Aliens table as well and the movement of the sprites is complete.

You've now completed the script, which means the only thing left to do is sit back and watch it take off. Check out the demo on the accompanying CD. On the surface it looks identical to the hard-coded version, but there are two important differences. First, you may notice a slight speed difference. This is a valuable lesson—don't forget that despite all of its advantages, scripting is still noticably slower than native executable code in most situations. Second, and more obviously, remember that even though you've compiled the host application, the script itself can be updated and changed as much as you want without recompiling the executable. Because this is the whole reason you perhaps got into this crazy scripting business in the first place, I suggest you take the time to try changing the general behavior of the

NOTE

Remember, compiling your scripts with luac is always recommended. Now that you've finished working on the Lua demo, you might as well compile script.lua for future use. As I've said, lua_dofile () just needs the filename of the compiled version, and will handle the rest transparently. It costs you nothing, and in return you get faster script load times (although it's highly unlikely that you'll notice a difference in this particular example). Either way, it's a good habit to start early.

script and watch the executable change with it. As a challenge, try adding a gravity constant to the bouncing movement of the heads; perhaps something that will slowly cause them to fall to the ground. Once they're all at the bottom of the screen, reverse the polarity and watch them "fall" back up. This shouldn't take too much effort to implement given what you've done so far, and it will be a great way to experience first-hand the power scripts can have over their compiled host applications. Maybe you can create some trig functions in the host API and use them to move the gravity constant along a sinusoid.

Advanced Lua Topics

I've covered the core of the language as well as most of the details you'll need for integration. This should be more than sufficient for most of your game scripting needs, but if you're anything like me, you can't sleep at night until you've learned *everything*. And if you're anything like I am tonight, you won't sleep at all because you're all hopped up on *Red Bull* and are too busy running laps on the roof. So, allow me to discuss a few advanced topics that enhance Lua's power but are beyond the scope of this book:

- **Tag Methods**. One of Lua's defining features is the capability for it to extend itself. This is implemented partially through a feature called *tag methods*, which are functions defined by the script that are assigned to key points during execution of Lua code. Because these functions are called automatically by the Lua runtime, the programmer can use them to extend or alter the behavior of said code.
- **Complex Data Structures.** Lua only directly supports the table structure, but as you've seen, tables can not only contain any value, but can also contain references to other tables as well as functions. You can probably imagine how these capabilities lend themselves to the construction of higher-level data structures.
- **Object-Oriented Programming.** This is almost an extension of the last topic, but Lua is capable of implementing classes and objects through clever use of tables. Remember, tables can include function references, which gives them the capability to simulate constructors, destructors, and methods. Because functions can return table references as well, constructor functions can create tables to certain specifications automatically. Oh, the possibilities!
- **The Lua Standard Library.** Lua also comes with a useful standard library, much like the one that comes with C. This library is broken into APIs for string manipulation, I/O, math, and more. Becoming familiar with this library can greatly expand the power and flexibility of your scripts, so it's definitely worth looking into. Also, in case you were wondering, this is why your Lua distribution comes with `lualib.h` and `lualib.lib`. These extra files implement the standard library.

Web Links

For more general information on Lua, as well as the Lua user community, check out the following links. These are also great places to begin your investigation of the advanced topics described previously:

- **The Official Lua Web Site:** `http://www.lua.org/`. This is the official source for Lua documentation and distributions. Check here for updates on the language and system, as well as general news.
- **lua-users.org:** `http://www.lua-users.org/`. A gathering of a number of Lua users, offering a focused selection of content and resources.
- **lua-l: Lua Users Mailing List:** `http://groups.yahoo.com/group/lua-l/`. The lua-l Yahoo Group is a gathering of a number of Lua developers who discuss Lua news and ask/answer questions. It's a frequently evolving source of up-to-date Lua information and a good place to familiarize yourself with the language itself and its real-world applications.

Python

Lua was a good language to start with because it's easy to use and has a reasonably familiar syntax. Now that you've worked your way through a number of examples with the system and used it to successfully control the bouncing alien head demo, you now have some real-life scripting experience and are ready to move onto something more advanced. Enter Python.

Python is another general-purpose scripting system with a simple but powerful object-oriented side that's been employed in countless projects by programmers of all kinds over the years (including a number of commercial games). One somewhat high-profile example is Caligari's trueSpace, a 3D modeling, rendering and animation package that uses Python for user-end scripting. The syntax of the language is unique in many ways, but will ultimately prove familiar enough to most C/C++ programmers.

The Python System at a Glance

Python is available from a number of sources, two of the most popular being the ActiveState *ActivePython* distribution, available free at `www.activestate.com`, and the Python.org distribution, also free, at `www.python.org`. I went with the Python.org version, so I recommend you download that one. Linux users will most likely already have Python available on their systems as part of their OS distribution.

You can install the Python 2.2.1 distribution by running the self-extracting installer found in the directory mentioned previously. Mine was installed to `D:\Program Files\Python22`; make sure you note where yours is installed as well. Once you've found it, you're pretty much ready to get started.

Directory Structure

When the installation is complete, check out the `Python22/` directory (which should be the root of your Python installation). In it, you'll find the following subdirectories:

- **`DLLs/`.** DLLs necessary for runtime support. Nothing you need to worry about.
- **`Doc/`.** Extensive HTML-based documentation of Python and the Python system. Definitely worth your attention.
- **`include/`.** Header files necessary when linking your application with Python.
- **`Lib/`.** Support scripts written in Python that provide a large code base of general functionality.
- **`libs/`.** The Python library modules to be linked with your program.
- **`tcl/`.** A basic Tcl distribution that enables Python to use Tkinter, a Tcl/Tk wrapper that provides a GUI-building interface. You won't be working with this, as GUIs are beyond the scope of the simple game scripting in this chapter.
- **`Tools/`.** Some useful Python scripts for various tasks. Also not to be covered in this chapter.

Nothing too complicated, right? Now that you have a general idea of the roadmap, direct your attention back to the root directory of the installation. Here you'll find `python.exe`, which is a handy interactive interpreter.

The Python Interactive Interpreter

Just like Lua, Python features an interactive interpreter that allows you to input script code line-by-line and immediately observe the results. This interpreter should be found in your root Python directory and is named `python.exe`. Go ahead and get it started. You should see this:

```
Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once you're in, you should be looking at what is known as the *primary prompt*. This consists of three consecutive greater-than signs (`>>>`) and means the interpreter is ready for you to input code. Like the Lua interpreter, `python` will attempt to execute each line as it's entered; to suppress this until a certain amount of lines have been written, terminate each line with a backslash (`\`) until you're ready for the interpreter to function again.

NOTE

It's interesting to note that out of the three languages you work with here, Python has the friendliest interpreter. The other two start up and simply shove a single-character prompt in your face, whereas `python` at least provides some basic instructions. Oh well. :)

Also, similar to Lua, `python` can run entire Python scripts from text files, which is of course much easier when you want it to execute large scripts, because it would quickly become tedious to retype them over and over. It's also a good way to validate your scripts; the interpreter will flag any compile-time errors it finds in your code and provide reasonably descriptive error messages. Python files are generally saved with the `.py` extension, so get in the habit of doing this as soon as possible.

To exit `python`, press Ctrl+Z (which will produce "`^Z`" at the promt) and press Enter.

The Python Language

Python is a rich language boasting a large array of syntactic features. There are usually more than a few ways to do something, which translates to a more flexible programming environment than other, more restrictive languages. It also means that discussing basic Python is a more laborious task than discussing simpler languages, like the tour of Lua. So, rather than standing around and dissecting the situation any further, let's just dive right in and get started with the syntax and semantics of Python.

Comments

I talk about comments first because they're just going to show up in every subsequent example anyway. Python only directly supports one type of comment, denoted by a hash mark (#). Here's an example:

```
# This is a comment.
```

However, by taking clever advantage of Python's syntax, you can simulate multi-line comments like this:

```
""" This is
    a multi-line
    comment! Sorta! """
```

Just be aware right now that this code isn't *really* a comment, it just happens to act almost exactly like one. You'll find out exactly what's going on in a moment.

Variables

Like Lua, Python is typeless and thus allows any variable to be assigned any value, regardless of the type of data it currently holds. Assignment in Python looks pretty much the way it does in most languages, using the equals sign (=) as the operator. Here are some examples:

```
Int = 16                    # Set Int to 16
Float = 3.14159             # Set Float to 3.14159
String = "Hello, world!"    # Set String to "Hello, world!"
```

Note the lack of semicolons. Python does allow them, but they aren't useful in the same way they are in Lua and are rarely seen in the Python scripts you'll run across. As a result, I suggest you build the habit of omitting semicolons when working with Python. Multiple lines in Python code are instead facilitated with the now familiar backslash (`/`) terminator:

```
MyVar\
=\
"Hello!"
print MyVar
```

This code prints "`Hello!`" to the screen.

Python, like Lua, also supports multiple assignments, wherein more than one identifier is placed on the left side of the assignment operator. For example:

```
X, Y, Z = U, V, W
```

This code sets `X` to the value of `U`, `Y` to the value of `V`, and `Z` to the value of `W`. Unlike Lua, however, Python isn't quite so forgiving when it comes to an unequal number of variables on either side of the assignment. For example,

```
X, Y, Z = U, V      # Note that Z is not given a value
```

and

```
X, Y = U, V, W      # Note that W is not assigned anywhere
```

Both of these lines result in compile-time errors.

Python also supports assignment chains, like so:

```
X = Y = Z = 512 * 128
print X, Y, Z
```

When executed in the interpreter, the previous code will output the following:

```
65536 65536 65536
```

Ironically, despite support for this feature, assignments cannot appear in expressions, as you'll see later.

Python requires that variables be initialized before they appear on the right side of an assignment or in an expression. Any attempt to reference an uninitialized variable will result in a runtime error.

NOTE

As you've probably noticed, Python features a built-in `print ()` function. Unlike Lua, however its contents need not be enclosed in parentheses. Also, Python's `print ()` accepts a variable sized, comma-separated list of values, all of which will be printed and delimited with single spaces.

Data Types

Python has a rich selection of data types, even directly supporting advanced mathematical concepts like complex numbers. However, your experience with Python in the context of game scripting will be primarily limited to the following:

- **Numeric—**Integer and floating-point values are directly supported, with any necessary casting that may arise handled transparently by the runtime environment.
- **String—**A simple string of characters, although Python does support a vast selection of differing string notations and built-in functions. I'll discuss a few of them soon.
- **Lists—**Unlike numerics and strings, the Python *list* is an aggregate data structure like a C array or Lua table. As you'll see, lists share a common syntax with strings in many cases, which proves quite useful.

Numerics can be expressed in a number of ways. You've already seen simple integers and floats, like `64` and `12.3456`, but you can also express them in other ways. First of all, you should learn the difference between plain integers and long integers. Plain integers are simply strings of digits, although they cannot exceed the range of -2^31 to 2^31. Long integers, on the other hand, can be of any size as long as they're suffixed with an `L`:

```
HugeNum = 12345678901234567890L
```

NOTE

The `L` in long integers can be either upper- or lowercase, but the uppercase version is much more readable. I recommend using it exclusively.

You can also express integers in other bases, like octal and hexadecimal. These follow the same rules as most C compilers:

```
Octal = 0342      # Octal numbers are prefixed with 0
Hex = 0xF2CA4     # Hex numbers are prefixed with 0x
```

Basic Strings

As stated, Python has extensive support for strings, both in terms of their representation and the built-in operations that can be performed on them. To get things started, consider the multiple ways in which a Python string literal can be expressed. First off is the traditional double-quote syntax we all know and love:

```
MyString = "Hello, world!"
```

This code, of course, sets `"Hello, world!"` to the variable `MyString`. Next up is single-quote notation:

```
MyString = 'Hello, world!'
```

This has the exact same effect. Right off the bat, however, one advantage to this method is that double-quotes can be used in the string without tripping up the compiler. Unlike many languages, however, a string literal in Python can span multiple lines, as long as the backslash terminator is used:

```
MyString = "Hello\
, \
world!"
```

Two important notes regarding this particular notation is that it works with both single and double-quoted lines, and that the line breaks you see in the source will *not* actually translate into the string. You'll have to use the familiar `\n` (newline) code from C in order to cause a physical line break within the string. Printing the previous code from would yield

```
"Hello, world!"
```

Another type of string, however, is the triple-quoted string. This admittedly bizarre syntax allows line breaks to appear in a string literal without the backslash, because they're considered characters in the string. For example:

```
print """I stand before
you,
a broken
string!"""
```

This code prints:

```
I stand before
you,
a broken
string!
```

As you can see, it's printed to the screen just as it appeared in the code, something of a "WYSIWYG" approach to string literals.

At this point it should be clear why the aforementioned technique for simulating block comments works the way it does. Because Python (like many languages) allows isolated expressions to appear outside of a larger statement like an assignment, these "comments" are really just string literals left untouched by the compiler that don't have any effect at runtime. Triple-quoted strings can use both single- and double-quotes:

```
X = """String 0."""
Y = '''String 1.'''
```

String Manipulation

Once you've defined your strings, you can use Python's built-in string manipulation syntax to access them in any number of ways, smash them together, tear them apart, and just wreak havoc in general.

String concatenation is one of the most common string operations, and Python makes it very easy with the + operator:

```
print "String" + " " + "concatenation."
```

This code outputs:

```
String concatenation.
```

In addition, you can use the * operator for repeating, or *multiplying* strings, just like you did in the Lua script:

```
print "Hello" + "!" * 8
```

This code will enthusiastically print:

```
Hello!!!!!!!!
```

NOTE

Python.org does not necessarily condone obnoxious yelling. At least I don't.

Now that you can make your strings bigger, let's see what you can do about making them smaller; in other words, accessing substrings and individual characters. To address the first comment, strings can be accessed like arrays when individual characters need to be extracted. For example:

```
MyString = "Stringlicious!"
print "Index 4 of '" + MyString + "' is:", MyString [ 4 ]
```

Because Python strings begin indexing at zero, like C, printing index 4 of `MyString` will produce:

```
Index 4 of 'Stringlicious!' is: n
```

In addition to simple array notation, however, *slice notation* can also be used to easily extract substrings, which has this general form:

```
StringName [ StartIndex : EndIndex ]
```

Get the idea? Here's an example:

```
MyString = "Stringtastic!"
print "Slicing from index 3 to 8:", MyString [ 3 : 8 ]
```

Here's its output:

```
Slicing from index 3 to 8: ingta
```

Just provide two indexes, the starting index and ending index of the slice, and the characters between them (inclusive) will be returned as a substring.

There are also a number of shortcuts that can be performed with slice notation. Each of the forms slice notation can assume is listed in Table 6.6.

These shorthand forms for slicing to and from the extents of the set can come in pretty handy, so keep them in mind (the "set" being the characters of the string in this case). Figure 6.15 illustrates Python string slicing.

An important point to mention in regards to strings is that they cannot be changed on a substring level. In other words, you can change the entire value of a string variable, by assigning it a new string, like this:

```
MyString = "Hello"              # MyString contains "Hello"
MyString = "Goodbye"            # Now it contains "Goodbye"
```

You can also append to a string in either direction, like this:

```
MyString = "So I said '" + MyString + "!'"
```

Table 6.6 Slice Notation Forms

Notation	Meaning
`[ X : Y ]`	Slices from index `X` to index `Y`.
`[ X : ]`	Slices from index `X` to the last index in the set.
`[ : Y ]`	Slices from the first index of the set to index `Y`.
`[ : ]`	Covers the entire set.

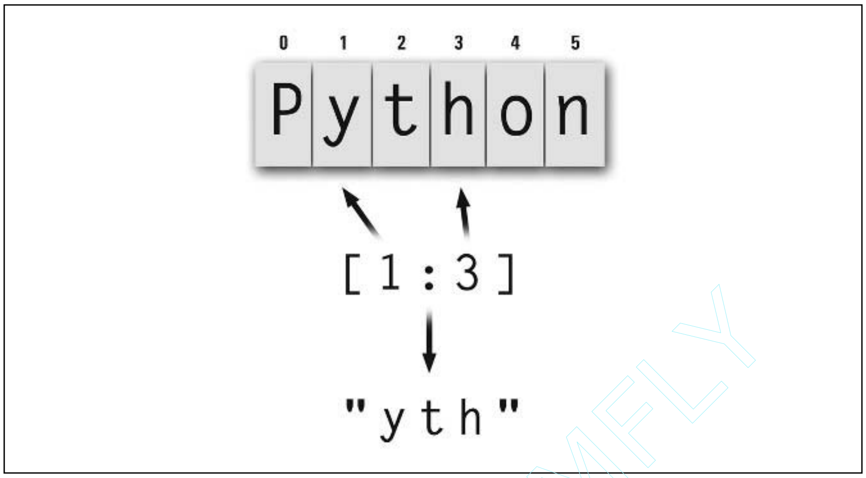


Figure 6.15

Python string slicing.

At which point `MyString` will contain "`So I said 'Goodbye!'`". What you *can't* do, however, is attempt to change individual characters or slices of a string. The compiler won't like either of these cases:

```
MyString [ 3 ] = "X"
MyString [ 0 : 2 ] = "012"
```

This sort of substring alteration must be simulated instead by creating a new string based on the old one, with the desired changes taken into account.

CAUTION

In another example of Python's slightly more strict conventions, be aware that indexing a string character outside of its boundaries will cause a "`string index out of range`" runtime error. Oddly, however, this does not apply to slices; slice indexes that are beyond the extents are simply truncated, and slices that would produce a negative range (slicing from a higher index to a lower index rather than vice-versa) are reversed, thus correcting the problem. (I suppose this particular decision was made because "clipping" a slice will generally yield more usable results than forcing a stray character index to remain in the bounds of the string. In the former case, you're simply asking for too much of the string; in the latter, all signs point to a more serious logic error.)

Lastly, check out the built-in function `len ()`, which Python provides to return the length of a given string:

```
MyString = "Plaza de toros de Mardid"
print "MyString is", len ( MyString ), "characters long."
```

This example will output:

```
MyString is 24 characters long.
```

Lists

Lists are the main aggregate data structure in Python. Something of a cross between C's array and Lua's table, lists are declared as comma-separated values that are accessible with integer indexes. Lists are created with a square-bracket notation that looks like this:

```
MyList = [ 256, 3.14159, "Alex", 0xFCA ]
```

In the previous example, `256` resides at index 0, `3.14159` is at index 1, `"Alex"` is at 2, and so on. Like Lua, Python lists are heterogeneous and can therefore contain differing data types in each element. Unlike Lua, however, list elements can *only* be accessed with integer indexes, meaning they're more like true arrays than associative arrays or hash tables. Also, new elements cannot simply be added on the fly, like this:

```
MyList [ 31 ] = "Uh-oh!"
```

Doing something like this in Lua is fine, but you'll get an "`index out of range`" error in Python. This is because index 31 does not exist in the list. One nice feature of lists, however, is that they *can* be changed on an index or slice level after their creation, unlike strings. For example:

```
MyList [ 2 ] = "Varanese"
```

Here you've changed index 2, which originally contained my first name, to now contain my last name, and Python doesn't complain.

With these few exceptions, lists are mostly treated like strings, which means all the indexing and slicing notation discussed in the last section applies to lists exactly. In fact, lists can even be printed like strings; in other words, without an index or a slice after the identifier:

```
print MyList
```

This code outputs the following:

```
[256, 3.1415899999999999, 'Alex', 4042]
```

Note that the hex value `0xFCA` was translated to its decimal equivalent when printed.

Python provides a large assortment of built-in functions for dealing with lists. I only cover a select few here, but be aware that there are many more. Consult the documentation that came with your Python distribution for more information if you're interested.

Just like strings, the `len ()` function can be used to return the number of elements in a list. Here's an example:

```
MyList = [ "Zero", "One", "Two", "Three" ]
print "There are", len ( MyList ), "elements in MyList."
```

Running this script would produce the following output:

```
There are 4 elements in MyList.
```

The next group of functions I'm going to discuss can be called directly from a given list, much like a method is called from an object of a class. In other words, they'll follow this general form:

```
List.Function ( ParameterList );
```

Earlier I mentioned that you can't just randomly add elements to a list. Although you still can't add an element to any arbitrary index, you can *append* new elements to the end of a list using `append ()`, which accepts a single parameter of any type:

```
MyList.append ( "Four" );
MyList.append ( "Five" );
MyList.append ( "Six" );
MyList.append ( "Seven" );
print "There are now", len ( MyList ), "elements in MyList."
```

This will produce:

```
There are now 8 elements in MyList.
```

As you can see, four integer elements were appended to the end of the list, giving you eight total indexes (0-7).

In addition to appending single elements, you can append an entire separate list as well with the `extend ()` function. This parameter takes a single list as its parameter.

```
List0 = [ 0, 1, 2, 3 ]
print List0;
List1 = [ 4, 5, 6, 7 ]
print List1;
List0.extend ( List1 )
print List0
```

This example produces the following output:

```
[0, 1, 2, 3]
[4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Lastly, let's take a look at `insert ()`. This function allows a new element to be inserted into the list at a specific index, pushing everything beyond that index over by one to make room.

```
MyList = [ "Game", "Mastery." ]
print MyList
MyList.insert ( 1, "Scripting" )
print MyList
```

The output for this example would be:

```
['Game', 'Mastery']
['Game', 'Scripting', 'Mastery']
```

It's all pretty straightforward stuff, but as you can see, they make lists a great deal more flexible.

The last thing I want to mention before moving on is that lists, as you might imagine, can be nested in a number of ways. Among other things, this can be used to simulate multi-dimensional arrays. Here's an example:

```
SuperList = [ "Super0", "Super1", "Super2" ]
SubList0 = [ "Sub0", "Sub1", "Sub2" ]
SubList1 = [ "Sub0", "Sub1", "Sub2" ]
SubList2 = [ "Sub0", "Sub1", "Sub2" ]
SuperList [ 1 ] = SubList1
print SuperList
print SuperList [ 1 ]
print SuperList [ 1 ][ 1 ]
```

When executed, this example produces the following output:

```
['Super0', ['Sub0', 'Sub1', 'Sub2'], 'Super2']
['Sub0', 'Sub1', 'Sub2']
Sub1
```

Notice how the first line of the output shows `SubList1` literally nested inside `SuperList`. Also notice that there are three different levels of indexing; printing out `SuperList` in its entirety, printing `SubList1` in its entirety as `SuperList [ 1 ]`, and printing out `SubList [ X ]` individually as `SuperList [ 1 ][ X ]`.

Of course, just as you saw in Lua, the issue of references rears its ugly head again. After assigning SubList1 to SuperList [1] in the last example, check out what happens when I make a change to SubList 1:

```
print "SubList1:       ", SubList1
print "SuperList [ 1 ]:", SuperList [ 1 ]
SubList1 [ 1 ] = "XYZ";
print "SubList1:       ", SubList1
print "SuperList [ 1 ]:", SuperList [ 1 ]
```

Here's the output:

```
SubList1:        ['Sub0', 'Sub1', 'Sub2']
SuperList [ 1 ]: ['Sub0', 'Sub1', 'Sub2']
SubList1:        ['Sub0', 'XYZ', 'Sub2']
SuperList [ 1 ]: ['Sub0', 'XYZ', 'Sub2']
```

Ah-ha! Changes made to SubList1 affected the contents of SuperList [1], because they're both pointing to the same data. As always, be very careful when dealing with references in this manner. I am talking about logic errors you'll have flashbacks of 20 years from now. Tread lightly, soldier!

Expressions

Python's expressions work in a way that's quite similar to C, Lua, and most of the other languages you're probably used to. Tables 6.7 through 6.10 contain the primary operators you have to work with.

Table 6.7 Python Arithmetic Operators

Operator	Function
+	Add/concatenate (strings)
-	Subtract
*	Multiply/multiply (strings)
/	Divide
%	Modulus
**	Exponent
-	Unary negation

Table 6.8 Python Bitwise Operators

Operator	Function
`<<`	Shift left
`>>`	Shift right
`&`	And
`^`	Xor
`|`	Or
`~`	Unary not

Table 6.9 Python Relational Operators

Operator	Function
`<`	Less than
`>`	Greater than
`<=`	Less than or equal
`>=`	Less than or equal
`!=`, `<>`	Not equal (<> is obsolete)
`==`	Equal

Table 6.10 Python Logical Operators

Operator	Function
`and`	And
`or`	Or
`not`	Not

Here are a few general-purpose notes to keep in mind when dealing with Python expressions:

- Like Lua, Python's logical operators are spelled out as short mnemonics, rather than symbols. For example, logical *and* is `and` rather than `&&`.
- Assignments cannot occur in expressions. Python has removed this because of its significant probability of leading to logic errors, as it often does in C. With Python there's no possibility of confusing == with =, because = won't compile if it's found in an expression.
- Zero is always regarded as false, whereas any nonzero value is true.
- Strings and numerics shouldn't appear in arithmetic expressions together. Python won't convert either value to the data type of the other, and a runtime error will result.

Conditional Logic

Now that you've had a taste of Python's expression syntax, you can put it to use with some conditional logic. Python relies on one major conditional structure. Not surprisingly, it's the good ol' `if`. Here's an example:

```
Switch = "Blue"
Access = 0
print "Evaluating security..."
if Switch == "Blue":
   print "Clearance Code Blue - File Access Granted."
   Access = 1
elif Switch == "Green":
   print "Clearance Code Green - Satellite Access Granted."
   Access = 2
else:
   print "Clearance Code Red - Weapons Access Granted."
   Access = 3
print "...done."
```

The output from this example, by the way, will look like this:

```
Evaluating security...
Clearance Code Blue - File Access Granted.
...done.
```

There's a lot to learn about Python from this example alone, so let's take it from the top. The first thing you see is the general form of the `if` statements themselves. Instead of C's form, which looks like this:

```
if ( Expression )
```

Python's form looks like this:

```
if Expression:
```

Also, `else if` has been replaced with `elif`, a more compact version of the same thing. Make sure to note that all clauses; the initial `if`, the zero or more `elif`'s, and the optional `else`; all must end with a colon (`:`).

The other important lesson to learn here is how a code block is denoted in Python. In C, you rely on curly braces, so an `if` statement can look like any of the following and still be considered valid:

```
if ( X < 0 ) { X = 0; Y = 1; }

if ( X < 0 ) {
X = 0; Y = 1;
}

if ( X < 0 )
{
   X = 0;
   Y = 1;
}

if ( X < 0 )
{ X = 0;
    Y = 1;
}
```

In other words, C is a highly free-form language. The placement of elements within the source file is irrelevant as long as the order is valid. So, as long as `if` is followed by a parenthesized expression, which is in turn followed by an opening curly brace, a code block, and a closing curly brace, you can insert any configuration of arbitrary whitespace and line breaks.

Python is significantly different in this regard. Although the language overall is still relatively free-form, it does impose some important restrictions on indentation for the purpose of code blocks, because that's how a code block's nesting level and grouping is defined. There aren't any curly braces, no `BEGIN` and `END` pairs, just lines of code that can be grouped and nested based on how many tabs inward they reside.

Remember, there's no `switch` equivalent to be found; such a construct is instead simulated with `if…elif` sequences (which is done in C at times as well).

Here are a few more examples to help the paint dry:

```
X = 0
Y = 1

if X > 0:
    print "X is greater than zero."

if X <= 0 or Y != 1:
    print "X is less than or equal to zero."

if X or Y:
    print "Between X and Y, one, the other, or both are true."

Z = "Quantum Foam"
if ( X + Y ) and Z:
    print "X + Y and Z are both true."
```

And the output:

```
X is less than or equal to zero.
Between X and Y, one, the other, or both are true.
X + Y and Z are both true.
```

Iteration

Moving right along, the next stop on the route is iteration. Python provides two common looping structures, `while` and `for`. Despite the Python-esque syntax changes, `while` operates just like its C counterpart, so let's have a look at it:

```
Iteration = 0
while Iteration < 16:
    print "Loop Iteration:", Iteration
    Iteration = Iteration + 1
```

When run, this script will produce the following:

```
Loop Iteration: 0
Loop Iteration: 1
Loop Iteration: 2
Loop Iteration: 3
Loop Iteration: 4
Loop Iteration: 5
Loop Iteration: 6
Loop Iteration: 7
Loop Iteration: 8
Loop Iteration: 9
Loop Iteration: 10
Loop Iteration: 11
Loop Iteration: 12
Loop Iteration: 13
Loop Iteration: 14
Loop Iteration: 15
```

While I am on the topic of loops, I should cover some of the required loop-handling statements that most languages provide. Like C, Python gives you `break` and `continue`, and they function just like you'd expect them to. `break` causes the flow of the program to immediately jump to just outside the loop, effectively avoiding the rest of the loop's lifespan. `continue` causes the current iteration of the loop to terminate prematurely and the next one to begin.

Another statement worth mentioning when discussing Python loops is `else`.What is `else` doing in a discussion of loops you ask? Well, Python allows loops to provide an `else` clause that is guaranteed to execute if the loop terminates for any reason *other* than a `break` statement. So, if a loop is set to run 32 times, the `else` clause will execute after the 32nd iteration. However, if the loop prematurely `break`s for whatever reason, the `else` clause will be ignored. Here's an example:

```
print "First Loop – No Break"
Iteration = 0
while Iteration < 8:
   print "Loop Iteration:", Iteration
   Iteration = Iteration + 1
else:
   print "Else clause activated."

print
print "Second Loop – With Break"
Iteration = 0
while Iteration < 8:
   print "Loop Iteration:", Iteration
   Iteration = Iteration + 1
   if Iteration == 3:
      break;
else:
   print "Else clause activated."
```

And here's the output:

```
First Loop - No Break
Loop Iteration: 0
Loop Iteration: 1
Loop Iteration: 2
Loop Iteration: 3
Loop Iteration: 4
Loop Iteration: 5
Loop Iteration: 6
Loop Iteration: 7
Else clause activated.

Second Loop - With Break
Loop Iteration: 0
Loop Iteration: 1
Loop Iteration: 2
```

Next up are `for` loops, which work slightly differently than they do in C. In Python, a `for` loop is given just two things— an iterator variable and a list (yes, the data structure discussed earlier). The iterator then traverses the list, one by one, executing the body of the loop each time. So, for example:

```
for X in [ 0, 1, 2, 3 ]:
   print X
```

This code produces the following output:

```
0
1
2
3
```

This may seem a bit odd to you. If you have to explicitly declare a list for every range of numbers you want to iterate through, how on earth would you do any sort of complex or large-scale loops? How long is it going to take to type the list declaration for a loop that needs to iterate 100,000 times? Also, what about trickier progressions such as a loop that skips every 17 numbers from 33 to 261? It all seems far too complex to be serious. Besides, any explicitly defined range can't be changed at runtime, which imposes yet another huge limitation.

Fortunately, Python includes a function that allows you to easily generate procedural lists (*procedural* meaning the list is generated based on some predefined formula or procedure rather than a human hardcoding each value individually). For example, say you want to loop through a list 1024 times. Rather than type out all 1024 comma separated list elements, you simply do this:

```
for X in range ( 0, 1023 ):
   print X
```

(You'll have to run this yourself, my editors wouldn't appreciate a dump of 1024 lines. :)

`range ()` automatically generates and returns a list consisting of each digit from 0 to 1023, and the loop works. You can also define a *step*, along which the iterator should progress as it moves from one end of the range to the other. So if you want to modify the last example to count from 0 to 1023 but skip every four numbers along the way, you can do this:

```
for X in range ( 0, 1023, 4 ):
   print X
```

Easy as pi. Just for reference, here's a pseudo-prototype for the `range ()` function:

```
list range ( Start, End, Step );
```

Remember that `Step` is optional, and defaults to 1.

Functions

The last piece of the puzzle in understanding the basics of the Python language is the function. Like any good language, Python lets you create user-defined functions you can call by name, pass parameters to, and receive values from. Here's an example of a simple Python function for determining the maximum of two numbers:

```
def GetMax ( X, Y ):
   print "GetMax () Parameters:", X, Y
   if X > Y:
      return X
   else:
      return Y

print GetMax ( 16, 24 )
```

The output for this would be:

```
GetMax () Parameters: 16 24
24
```

This simple example uses the `def` keyword (short for *define*) to create a new function called `GetMax ()`. This function accepts two parameters, `X` and `Y`. As you can see, parameters need only be listed; the typeless nature of Python means you don't have to declare them with data types or anything like that. As for the function body itself, it follows the same form that loops and the `if` construct have. The `def` declaration line is terminated with a colon, and every line underneath it that composes the function body is indented by one tab.

Once inside the function, parameters can be referenced just like any other local variable, and the `return` keyword functions just like in C, immediately exiting the function and optionally sending a return value back to the caller.

As you can see, functions are pretty straightforward in Python. The only real snag to worry about is global variables. Local variables are created within the function just like any other variable, so there's nothing to worry about there. Globals, however, are slightly different. Globals can be *referenced* within a function and retain their global value, but if they're *assigned* a new value, that value will reset to its original global value when the function returns. The only way to permanently alter a global's value from within a function is to *import* it into the function's scope using the `global` keyword. Here's an example:

```
GlobalInt = 256
GlobalString = "Hello!"

def MyFunc ():
   print "Inside MyFunc ()"
   GlobalInt = 128
   global GlobalString
   GlobalString = "Goodbye!"
   print GlobalInt, GlobalString

MyFunc ()

print
print "Outside MyFunc ()"
print GlobalInt, GlobalString
```

When you run the script, you'll see this:

```
Inside MyFunc ()
128 Goodbye!

Outside MyFunc ()
256 Goodbye!
```

When `MyFunc ()` is entered, it gives both global variables new values. It then prints them out, and you can see that both variables are indeed different. However, when the function returns and you print the globals again from within their native global scope, you find that `GlobalInt` has seemingly gone from 128, the value `MyFunc ()` set it to, back to 256. `GlobalString`, on the other hand, seems to have permanently changed from `"Hello!"` to `"Goodbye!`". This is because it's the only one that was imported beforehand with `global`.

At this point, you've learned quite a bit about the basic Python language. You understand variables, data types, and expressions, as well as list structures, conditional logic, iteration, and functions. Armed with this information, it's time to set your sights on integration.

Integrating Python with C

Integrating Python with C is not particularly difficult, but there are a number of details to keep track of along the way. This is due to the fact that the API provided by Python for interfacing its runtime environment with a host application is somewhat fine grained. Rather than provide a small set of features that allow you to simply and easily perform basic tasks like loading scripts, calling functions, and so on, you're forced to do these things "manually" by fashioning this higher-level logic from a sequence of lower-level calls.

Fortunately, it's still a pretty easy job overall, and as long as you follow the next few pages closely, you shouldn't have any troubles. This section will cover the following topics:

- How to load and execute Python scripts in C.
- How to call Python functions from C, with parameters and return values.
- How to export C functions so they can be called from within Python scripts.

Just like you did when studying Lua, you'll first practice these skills by testing them with some simple test scripts, and then apply them to the bouncing alien head demo that was originally coded in C.

Compiling a Python Project

The first step in compiling a Python project is making sure that your compiler's paths for include and library files are set to the Python installation's `include/` and `libs/` paths. You can then use the `#include` directive to include the main Python header file, `Python.h`:

```
#include <Python.h>
```

The last step is including `Python22.lib` with your project. From here, you've done everything you need get started with Python. At least, *in theory*.

The Debug Library

In practice, there's a slight issue with the Python.org 2.2 distribution; the `python22_d.lib` file is missing, at least in its compiled form. You can download the source and build it yourself, but for now, running any Python program will result in the following linker error:

```
LINK : fatal error LNK1104: cannot open file "python22_d.lib"
```

The reason for this error is that `python22_d.lib` is the *debug* version of the library, with extra debug-specific features. When you compile your project in debug mode, special flags in the Python library's header files will attempt to use this particular .LIB file, which won't be available and thus result in the error. Rather than waste your time compiling anything, however, it's a lot easier to resolve this situation by simply forcing Python to use the non-debug version in all cases.

To do this, open up `pyconfig.h` in the Python installations `include/` directory. Go to line 335, which should be the first in this block of code:

```
#ifdef _DEBUG
#pragma comment(lib,"python22_d.lib")
#else
#pragma comment(lib,"python22.lib")
#endif
#endif /* USE_DL_EXPORT */
```

The first change to make is on the second line in this block. Change `python22_d`.lib to `python22.lib`, and you should be left with this:

```
#ifdef _DEBUG
#pragma comment(lib,"python22.lib")
#else
#pragma comment(lib,"python22.lib")
#endif
#endif /* USE_DL_EXPORT */
```

The next and final change to make is right below on line 342:

```
#ifdef _DEBUG
#define Py_DEBUG
#endif
```

Just comment these three lines out entirely, so they look like this:

```
/*
#ifdef _DEBUG
```

```
#define Py_DEBUG
#endif
*/
```

That's everything, so save `pyconfig.h` with the changes and the Python library will use the non-debug version of `python22.lib` in all cases. Everything should run smoothly from here on out.

Initializing Python

Within your program, the initialization and shut down of Python is quite simple. Just call `Py_Initialize ()` at the outset, and `Py_Finalize ()` before shutting down. Within these two calls, the Python system will be activated and ready to use. Notice that there's no "instance" of the Python runtime environment; you simply initialize it once and use it as-is throughout the lifespan of your program:

```
Py_Initialize ();
... Python application logic ...
Py_Finalize ();
```

NOTE

From here on out, I'll be taking a somewhat superficial look at how Python is integrated with C. The reason for this is that Python overall is a fairly complex system, and a full explanation would detract heavily from the rest of the book— especially the coverage of Lua and Tcl. What you'll get here is enough understanding to actually make everything work, with a reasonable level of understanding. Overall, it should be more than enough to get you started with game Python scripting.

With the simplest possible Python application skeleton in place, you're ready to get started with an actual project. To test your Python integration capabilities, let's start by writing some scripts that demonstrate common integration tasks, like loading scripts, calling functions, and stuff like that.

Python Objects

One of the most important parts in understanding how Python integration works is understanding *Python objects*. A Python object is a structure that represents some peice of Python-related data. It may be an integer or string value residing somewhere within a script, a script's function, or even an entire script. Virtually everything you'll do as you embed Python in your application will involve these objects, so it's important to comfortably understand them as soon as possible.

Python objects are just C structures, but you always deal with *pointers* to the objects, never the objects themselves. Here's a sample declaration of some Python objects:

```
PyObject * pMyObject;
PyObject * pMyOtherObject;
```

The actual objects are created by functions in the Python integration API, so you don't have to worry about that just yet.

Reference Counting

Python objects are vital to the overal scripting system, and as such, are often used in a number of places at once. Because of this, you can't safely free a Python object arbitrarily, because you have no idea whether something else is using it. To solve this problem, Python objects have a *reference count*, which keeps track of how many entities are using the object at any given time. The reference count of a non-existent or unused object is always zero, and every time a new copy of that objects pointer is made for some new purpose, it's the job of the code responsible to increment the reference count.

Because of this, you'll never explicitly free Python objects yourself. Rather, you'll simply decrement them to let the scripting system know that you're done with them. Once an object's reference count reaches zero, the system will know it's safe to get rid of it. To decrement a Python object's reference count, we use `Py_XDECREF ()`:

```
Py_XDECREF ( pMyOtherObject );
Py_XDECREF ( pMyObject );
```

Notice that I decrement the reference counts in the reverse of the order the objects were declared (or more specifically, as you'll see, the order in which they're used). This ensures that any possible interconnections between the objects elsewhere in the system are "untangled" in the proper order.

So in a nutshell, Python objects will form the basis for virtually every peice of data you use to interact with the system, and it's important to decrement their reference counts when you're done using them. Figure 6.16 demonstrates the idea of Python objects and reference counts.

Loading a Script

Python scripts are loaded into C with a function called `PyImport_Import ()`. Because it's going to take a bit of explanation, let's just look at the code first:

```
PyObject * pName = PyString_FromString ( "test_0" );
PyObject * pModule = PyImport_Import ( pName );
if ( ! pModule )
{
    printf ( "Could not open script.\n" );
    return 0;
}
```

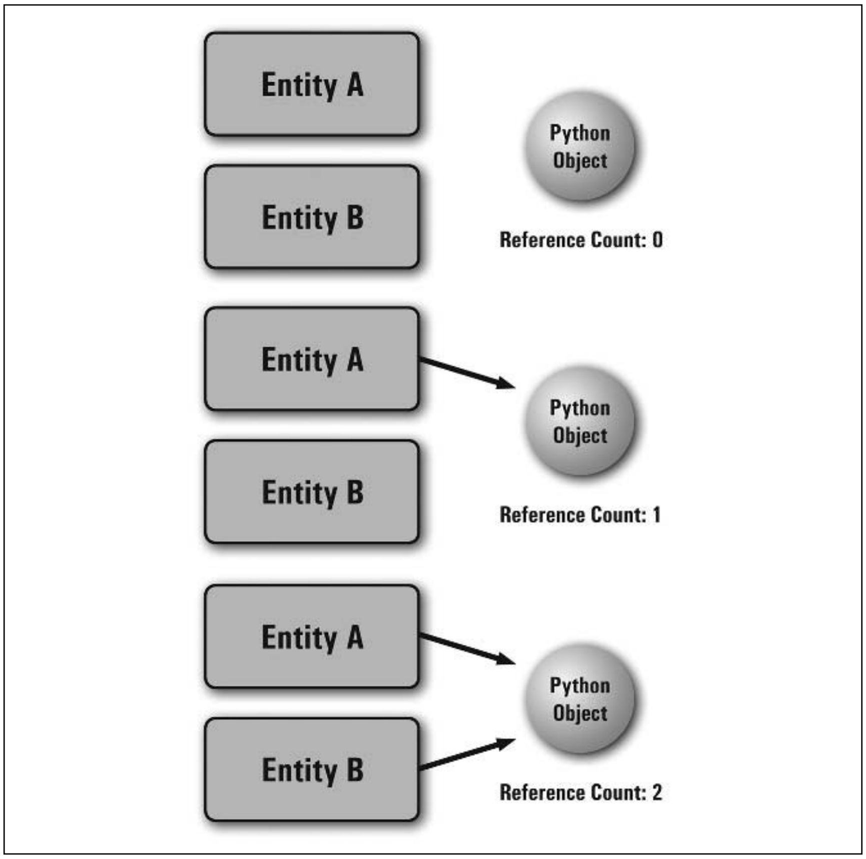


Figure 6.16

Python objects and reference counts.

Simply put, this code loads a script called `test_0.py` into the `pModule` object. What's all this extra junk, though? The first thing you'll notice is that you're creating a Python object called `pName`. It's created in a function called `PyString_FromString ()`, which takes a C-string and creates a Python object around it. This allows the string to be accessed and manipulated within the script, which will be necessary in the next line down. Note also that the file extension was omitted from the filename.

Once you've created the `pName` string object, it's passed to `PyImport_Import ()`, which loads the script into memory and returns a pointer in the form of the `pModule` pointer. What you've done here is *import* a *module*. A "module" in Python terms is a powerful grouping mechanism that resembles the package system in Java. All you really need to know, however, is that the module you've just imported contains your script.

Like Lua, any code in the global scope is automatically executed upon the loading of a script. To test this, let's write a simple script and run it with the previous code. Here's `test_0.py`:

```
IntVar = 256
FloatVar = 3.14159
StringVar = "Python String"

# Test out some conditional logic
X = 0
Logic = ""
if X:
    Logic = "X is true"
else:
    Logic = "X is false"

# Print the variables out to make sure everything is working
print "Random Stuff:"
print "\tInteger:", IntVar
print "\t  Float:", FloatVar
print "\t String: " + '"' + StringVar + '"'
print "\t  Logic: " + Logic
```

By saving this as `test_0.py` and loading it with the `PyImport_Import ()` routine, you'll see the following results printed to the console:

```
Random Stuff:
        Integer: 256
          Float: 3.14159
         String: "Python String"
          Logic: X is false
```

Calling Script-Defined Functions

Executing an entire script at load-time is fine, but real control comes from the ability to call specific functions at arbitrary times. To get things started, let's create a new script, this one called `test_1.py`, and add a function to it:

```
def GetMax ( X, Y ):

    # Print out the command name and parameters
    print "\tGetMax was called from the host with", X, "and", Y

    # Perform the maximum check
    if X > Y:
        return X
    else:
        return Y
```

The `GetMax ()` function accepts two integer parameters and returns whichever value is greater. The question is: how can this function be called from C?

The Module Dictionary

To understand the solution to this problem, you need to understand a script module's *dictionary*. The dictionary of a module is a data structure that maps all of the script's identifiers to their respective code or data. By searching the dictionary with a specific identifier string, a Python object wrapping that identifier's associated code or data will be returned. In this case, you want to use the script's dictionary to get a Python object containing the `GetMax ()` function, and you'd like to use the string `"GetMax"` to do so.

Fortunately, the Python/C integration API makes this pretty easy. The first thing you need to do is declare a new Python object that will store the dictionary or the module. Here's the code for doing so, along with the code for loading the new `test_1.py` script:

```
// Load a more complicated script
printf ( "Loading Script test_1.py...\n\n" );
pName = PyString_FromString ( "test_1" );
pModule = PyImport_Import ( pName );
if ( ! pModule )
{
    printf ( "Could not open script.\n" );
    return 0;
}

// Get the script module's dictionary
PyObject * pDict = PyModule_GetDict ( pModule );
```

After calling `PyModule_GetDict ()` with the pModule pointer that contains the script, `pDict` will point to the module's dictionary and give you access to all the identifier mappings you'll ever need. With the dictionary in hand, you can use the `PyDict_GetItemString ()` function to return a Python object corresponding to whatever identifier you specify. Here's how you can get the `GetMax ()` function object:

```
PyObject * pFunc = PyDict_GetItemString ( pDict, "GetMax" );
```

You have the function, so now what? Now, you need to worry about parameters. You know `GetMax ()` accepts two of them, but how are you going to pass them? You'll see how in just a moment, when you learn how to call the function, but for now, you need to focus on how the parameters are stored during this process. For this, I'll briefly cover another Python aggregate data structure, similar to the list, called the *tuple*.

Passing Parameters

Without getting into too much detail, tuples are used by Python to pass parameters around in inter-langauge function calls. At least, that's all you need to know about them. For the time being, just think of tuples as a list- or array-like structure. Simply put, you need to declare a new tuple, fill it with the parameters you want to send, and pass the tuple's parameter to the right places. Let's start by creating a tuple and adding the two integer parameters `GetMax ()` accepts, using the `PyTuple_New ()` function:

```
PyObject * pParams = PyTuple_New ( 2 );
```

`pParams` now points to a two-element tuple. Note, of course, that the code requested a tuple of two elements because that's the number of parameters you want to pass. To set the values of each of the two elements, you use the `PyTuple_SetItem ()` functions. Of course, you can only add Python objects to the tuple, so you'll use the `PyInt_FromLong ()` function to convert an integer literal value into a valid object. Check it out:

```
PyObject * pCurrParam;
pCurrParam = PyInt_FromLong ( 16 );
PyTuple_SetItem ( pParams, 0, pCurrParam );
pCurrParam = PyInt_FromLong ( 32 );
PyTuple_SetItem ( pParams, 1, pCurrParam );
```

The `pCurrParam` object pointer is first declared as temporary storage for each new integer object you create. `PyInt_FromLong ()` is then used to convert the specified integer value (16, in this case) to a Python object, the pointer to which is stored in `pCurrParam`. `PyTuple_SetItem ()` is then called.

The first parameter this function accepts is the tuple, so you pass `pParams`. The next is the index into the tuple to which you'd like to add the item, so 0 is passed. Finally, `pCurrParam` is the actual object whose value you'd like to add. So, this call tells the function to add `pCurrParam` to element zero of the `pParams` tuple. The function is repeated for index one, at which point the tuple contains 16 and 32. These are the parameters you'd like to send `GetMax ()`.

Calling the Function and Receiving a Return Value

The last step is of course to call the function and grab the return value it produces. This can be done in two lines. The first line actually calls the function and stores the return value in a locally defined Python object pointer. The second call extracts the raw value from this object. Check it out:

```
PyObject * pMax = PyObject_CallObject ( pFunc, pParams );
int iMax = PyInt_AsLong ( pMax );

printf ( "\tResult from call to GetMax ( 16, 32 ): %d\n\n", iMax );
```

`PyObject_CallObject ()` is the call to make when invoking a script-defined function, provided you have a Python object that wraps the desired function. Fortunately you do, so you pass `pFunc`. You also pass the `pParams` tuple, giving the function its parameters. `PyObject_CallObject ()` also returns a Python object of its own, containing the return value. Because you're expecting an integer, you use the `PyInt_AsLong ()` function to read it. When this code executes, you'll see the following results:

```
GetMax was called from the host with 16 and 32
Result from call to GetMax ( 16, 32 ): 32
```

Out of 16 and 32, the function returned 32 as the larger of the two, just as it should have.

Exporting C Functions

There's a lot you can do with the capability to call script-defined functions. Indeed, this process forms the very backbone of game scripting; if, at any time, the game engine can call a specific script-defined function, it can make the script do anything it needs it to do, exactly when necessary. This is only one side of the coin, however. In order to really get work done, the script needs to be able to call C-defined functions as well.

Defining the Function

In order to to do this, you first need to properly define a host API function. To keep things simple, I'll use the same host API function example created for the Lua demo; a function that prints a string a specified number of times. The logic to such a function is obviously trivial, but as you'd expect, the real issue is defining the function in such a way that it's "compatible" with Python. Let's start with the code:

```
PyObject * RepeatString ( PyObject * pSelf, PyObject * pParams )
{
    printf ( "\tRepeatString was called from Python:\n" );

    char * pstrString;
    int iRepCount;
    // Read in the string and integer parameters
    if ( ! PyArg_ParseTuple ( pParams, "si", & pstrString, & iRepCount ) )
    {
        printf ( "Unable to parse parameter tuple.\n" );
        exit ( 0 );
    }

    // Print out the string repetitions
    for ( int iCurrStringRep = 0;
          iCurrStringRep < iRepCount;
          ++ iCurrStringRep )
        printf ( "\t\t%d: %s\n", iCurrStringRep, pstrString );

    // Return the repetition count
    return PyInt_FromLong ( iRepCount );
}
```

Let's start with the function's signature. `RepeatString ()` accepts two parameters; a `PyObject` pointer called `pSelf`, and a second object pointer called `pParams`. `pSelf` won't be necessary for these purposes, so forget about it. `pParams`, on the other hand, is a tuple containing the parameters that were passed to you by the script. Naturally, this is an important one. The function also returns a `PyObject` pointer, which allows the return value to be sent directly back to Python without a lot of fuss.

Once inside the function, you'll usually want to start by reading the parameters. Of course, this isn't as easy as it would be in pure C or C++, because your parameters are stuffed inside the `pParams` tuple and therefore not quite as accessible. In order to read parameters passed from Python, use the `PyArg_ParseTuple ()` function. This function accepts a tuple pointer, a format string, and a variable number of pointers to receive the parameter values. Of course, this deserves a bit more explanation.

The tuple pointer parameter is simple. You first pass `pParams` so the function knows which tuple to read from. The next parameter, however— the format string—isn't quite as intuitive at first glance. Essentially what this function does is uses a string of characters to express which parameters are to be read, and in what order. In this example, `PrintStuff ()` wants to read a string and integer, in that order, so the string `"si"` is passed. If you wanted to read an integer followed by a string, it would be `"is"`. If you wanted to read an integer, followed by two strings and another integer, it would be `"issi"`. Get it?

Following the format string are the variables that will receive the parameter values. Think of this part of the function as if it were the values you pass `printf ()` after the string. Once again, order matters, so you pass `& pstrString`, followed by `& iRepCount` to receive the values.

The last order of business within a host API function (aside from the intended logic itself) is the return value. Because you're returning Python objects, you have to send *something* back. If there's nothing you want to return, just use `PyInt_FromLong ()` to generate the integer value zero. In your case, however, you'll return the specified repetition count just for the sake of returning something. `PyInt_FromLong ()` is still used, however.

The Host API

You have your function squared away, so the next step is defining a host API in which to store it. Unlike Lua, in which separate functions are registered one at a time with the Lua state with separate function calls, the host API in Python is added in one fell swoop. In order to do this in a single call, you can prepare an array ahead of time that fully describes every function in the host API.

Each element of this array is a `PyMethodDef` structure, which consists of a string function name, a function pointer adhering to the prototype, some flags, and a descriptive string that defines the function's intended behavior. Here's some code for declaring a host API array (known in a Python terms as a *function table*):

```
PyMethodDef HostAPIFuncs [] =
{
    { "RepeatString", RepeatString, METH_VARARGS, NULL },
    { NULL, NULL, NULL, NULL }
};
```

I'm using curly brace notation to define the array within its declaration. The first `PyMethodDef` represents the `RepeatString ()` function. The first field's value is `"RepeatString"`, which is the string that Python will look for within your scripts in order to determine when the function is being called. The next is `RepeatString`, a pointer to the function. Next up is `METH_VARAGS`. What this is doing is telling Python that the function accepts a variable number of arguments. This is the best bet for all of your functions, so just get in the habit of using it. The last parameter is set to `NULL`; otherwise it would be a string describing the `RepeatString ()` function. Because this doesn't really help you much, just ignore it.

You'll also notice that a second element is defined, one in which every field is `NULL`. This is because you won't be telling Python how many functions are in this array; rather, it waits until it hits this all-`NULL` "sentinel". This is the sign to stop reading from the array.

You're now ready to do something with the host API, but what? Oddly enough, the way to make these functions accessible to your script is to create a new module, and add the functions to this new module's dictionary. This will result in an otherwise empty module with three functions, ready to be used by the script. To create a new module, call `PyImport_AddModule ()`, like so:

```
// Create a new module to hold the host API's functions

if ( ! PyImport_AddModule ( "HostAPI" ) )
    printf ( "Host API module could not be created." );
```

This function simply accepts a string containing the module's desired name. In this case, name it `HostAPI`. You already have the function table prepared, so add it to the module:

```
if ( ! Py_InitModule ( "HostAPI", HostAPIFuncs ) )
    printf ( "Host API module could not be initialized." );
```

`Py_InitModule ()` initializes a module by adding the function table specified in the second parameter to its dictionary. The `HostAPI` module now contains the functions defined in the `HostAPIFuncs []` array, which refers simply to `RepeatString ()` in this example.

Calling the Host API From Python

Within the demo program, a new module called `HostAPI` exists with a record of the `RepeatString ()` function. The question now is how this function can be called. To start things off, the script itself needs to be aware of the `HostAPI` module. In order to call its functions, the module needs to be brought into the script's scope. This is done with the `import` keyword. Let's modify `test_1.py` to include this at the top:

```
import HostAPI
```

`import` is something like the C's preprocessor's `#include` directive, but as you can see, it's not limited to working solely with files. Although most modules imported by a Python script are stored on the disk initially, your `HostAPI` module was created entirely at runtime and therefore only exists in memory. However, because the Python library was made aware of `HostAPI`'s existence with the `PyImport_AddModule ()` function, it knew not to look for a `HostAPI.py` file when it executed the `import` statement and instead simply imported the already in-memory version.

> **NOTE**
> **What `import` does specifically is bring a module into a script's *namespace*; this can be thought of conceptually as adding a list of the module's functions to the script's dictionary, which was discussed earlier.**

The only snag here is that you now have to reposition the time at which you load `test_1.py`. Currently, you're declaring and initiailizing the `HostAPI` module *after* the script is loaded, which will cause a problem with the addition of the `import` keyword. Python will execute `import` as soon as the script is loaded, and because this is taking place before you add your module, it won't be able to find anything by the name of `HostAPI` and will terminate the loading process. To remedy this, remember to define any modules you'd like your scripts to use *before* loading the scripts:

```
// Create a new module to hold the host API's functions
if ( ! PyImport_AddModule ( "HostAPI" ) )
    printf ( "Host API module could not be created." );

// Create a function table to store the host API
PyMethodDef HostAPIFuncs [] =
{
    { "RepeatString", RepeatString, METH_VARARGS, NULL },
    { NULL, NULL, NULL, NULL }
};

// Initialize the host API module with your function table
if ( ! Py_InitModule ( "HostAPI", HostAPIFuncs ) )
    printf ( "Host API module could not be initialized." );

// Load a more complicated script
printf ( "Loading Script test_1.py...\n\n" );
pName = PyString_FromString ( "test_1" );
pModule = PyImport_Import ( pName );
if ( ! pModule )
{
    printf ( "Could not open script.\n" );
    return 0;
}
```

Now, Python will have a record of `HostAPI` when `test_1.py` imports it, and everyone will be happy. Moving back to the script itself, you're now capable of calling any `HostAPI` function (of which there's still just one). To test your `RepeatString ()` function, let's write a new Python function called `PrintStuff ()` that you can call from your program to make sure everything worked:

```
def PrintStuff ():
    # Print some stuff to show we're alive
    print "\tPrintStuff was called from the host."
    # Call the host API function RepeatString () and print out its return
    # value
    RepCount = HostAPI.RepeatString ( "String repetition", 4 )
    print "\tString was printed", RepCount, "times."
```

Everything should look simple enough, but notice that in the call to `RepeatString ()`, you had to prefix it with `HostAPI`, the name of the module in which it resides, forming `HostAPI.RepeatString ()`. This is done for the same reason you prefixed the Lua host API functions in the last section with `HAPI_`—to help prevent name clashes. This way, if the script already defined a function called `RepeatString ()`, the inclusion of the `HostAPI` module wouldn't cause a problem. Python always knows exactly which module you're attempting to work with.

When this code is executed, you should see the following on your console:

```
PrintStuff was called from the host.
RepeatString was called from Python:
        0: String repetition
        1: String repetition
        2: String repetition
        3: String repetition
String was printed 4 times.
```

That's it! With the capability to call Python functions from C and vice versa, you've established a complete bridge between the two languages, giving you a full channel of communication. To really put this to the test, finish what you started and use your Python integration skills to recode the bouncing alien head demo with a Python core.

NOTE

Before moving on, however, I've just got a little public service announcement to make—try to remember at all times that Python is *extremely* strict about the indenation of a line. I've already discussed that rather than using block delimiting tokens like C's `{...}` notation, or Pascal's `BEGIN...END`, Python relies instead on the number of spaces or tabs preceding a line of code to determine its nestling level and scope. Remember—*any* line of code outside of a function *must* start the absolute start of the line; no spaces, tabs or anything. Within a function, everything in the top nesting level *must* be exactly one tab or space in. Beyond that, nested structures like `while`, `if`, and `for` add a single tab or space to the identation of any code within their blocks.

Re-coding the Alien Head Demo

You've hopefully become comfortable by now with the basic process of Python integration, so you can now try something a bit more dynamic and use Python to rewrite the central logic behind the bouncing alien head demo initially coded in C earlier in the chapter. I already covered a lot of the general theory behind how this recoding process is laid out in the Lua section, so make sure to check it out there if you haven't already.

Initial Evaluations

You adequately surveyed the landscape of this particular project in the Lua section earlier. You determined that the best part of the demo to recode was the per-frame logic; the code that moves each alien head around and checks for collisions. This means that information about each alien is maintained within the script. To this, the script needs to define two functions: `Init ()`, which initializes the alien head array before entering the main loop, and `HandleFrame ()`, which draws the next frame to the screen and handles the movement and collision checks for each sprite.

In order to do this, the host API of the program must expose functions for drawing sprites, background images, and blitting the back buffer to the screen. It also needs to be able to return random numbers, the status of timers, and other such miscellany. Again, however, if you're looking for more specific information on how the separation between the script and the host application will work, check out the Lua section, where I covered all of this in more depth. The organization of a scripting project is usually language independent, unless you're focusing on a particularly language-specific feature. Because of this, the technique covered in the Lua provides helpful perspective here.

In short, the main loop of the original pure-C demo will be gutted entirely in favor of the new Python-defined `HandleFrame ()` function.

The Host API

The host API you'll expose to Python will include the same set of functions covered in the Lua version of this demo. The code to each function is rather simple and self-explanatory, so I won't waste the page space listing them here. You're always encouraged to refer to the source on the companion CD; however, the demos for this chapter can be found in `Programs/Chapter 6/`. What are useful, however, are the function prototypes, listed here:

```
PyObject * HAPI_GetRandomNumber ( PyObject * pSelf, PyObject * pParams );
PyObject * HAPI_BlitBG ( PyObject * pSelf, PyObject * pParams );
PyObject * HAPI_BlitSprite ( PyObject * pSelf, PyObject * pParams );
PyObject * HAPI_BlitFrame ( PyObject * pSelf, PyObject * pParams );
PyObject * HAPI_GetTimerState ( PyObject * pSelf, PyObject * pParams );
```

Remember, for a host API function to be compatible with Python, it must return a PyObject pointer and accept two PyObject pointers as parameters. Also remember that you always prefix host API functions with HAPI_ to ensure that they don't clash with any of the other names in the program. Within each function, parameters are extracted using a format string and the PyArg_ParseTuple () function, as you saw earlier. Values are returned in the form of Python objects directly through C's native return keyword. Here's an example of the host API function HAPI_GetRandomNumber ():

```
PyObject * HAPI_GetRandomNumber ( PyObject * pSelf, PyObject * pParams )
{
    // Read in parameters
    int iMin,
        iMax;
    PyArg_ParseTuple ( pParams, "ii", & iMin, & iMax );

    // Return a random number between iMin and iMax
    return PyInt_FromLong ( ( rand () % ( iMax + 1 - iMin ) ) + iMin );
}
```

The "ii" format string is passed to PyArg_ParseTuple () to let it know that two integers need to be read from the parameter tuple. PyInt_FromLong () is used to convert the result of your random number calculation to a Python object on the fly, a pointer to which is returned and subsequently passed back to the caller within the script by return.

The New Host Application

The changes made to the original C demo, which is now the host application of the Python demo, are straightforward and relatively minimal. In addition to including the definitions for each host API function, it's necessary to initialize and shut down Python before entering the main loop. Furthermore, the main loop's body is removed and replaced with a call to HandleFrame (), and the loop itself is preceded by a call to Init ().

Let's start with the initialization of Python. Because this involves a call to Py_Initialize (), the initialization of the HostAPIFuncs [] array, and the creation of the HostAPI module, it's best to wrap it all in a single function, which I call InitPython ():

```
void InitPython ()
{
    // Initialize Python
    Py_Initialize ();
    // Store the host API function table
    static PyMethodDef HostAPIFuncs [] =
    {
        { "GetRandomNumber", HAPI_GetRandomNumber, METH_VARARGS, NULL },
        { "BlitBG", HAPI_BlitBG, METH_VARARGS, NULL },
        { "BlitSprite", HAPI_BlitSprite, METH_VARARGS, NULL },
        { "BlitFrame", HAPI_BlitFrame, METH_VARARGS, NULL },
        { "GetTimerState", HAPI_GetTimerState, METH_VARARGS, NULL },
        { NULL, NULL, NULL, NULL }
    };

    // Create the host API module
    if ( ! PyImport_AddModule ( "HostAPI" ) )
        W_ExitOnError ( "Could not create host API module" );

    // Add the host API function table
    if ( ! Py_InitModule ( "HostAPI", HostAPIFuncs ) )
        W_ExitOnError ( "Could not initialize host API module" );
}
```

Nothing here is new, but notice that suddenly the `HostAPIFuncs []` array is quite a bit larger than it was. Despite the now considerable function list, however, remember to append the last element with a sentinel element consisting entirely of `NULL` fields. This is how `Py_InitModule ()` knows when to stop reading from the array. Forgetting this detail will almost surely result in a crash.

Shutting down Python is of course considerably easier, but it's more than just a call to `Py_Finalize ()`. In addition, you have to remember to decrement the reference count for each Python object we initialize. Because of this, each main object used by the program is global:

```
PyObject * g_pName;                         // Module name (filename)
PyObject * g_pModule;                       // Module
PyObject * g_pDict;                         // Module dictionary
PyObject * g_pFunc;                         // Function
```

Although I haven't showed you the code that uses these modules yet, they should all look familiar; they're just global versions of the Python objects used in the last demo for managing modules, dictionaries, and functions. The point, however, is that this allows you to decrement them in the `ShutDownPython ()` function you call at the end of the program:

```
void ShutDownPython ()
{
    // Decrement object reference counts
    Py_XDECREF ( g_pFunc );
    Py_XDECREF ( g_pDict );
    Py_XDECREF ( g_pModule );
    Py_XDECREF ( g_pName );

    // Shut down Python
    Py_Finalize ();
}
```

Whether or not you'd like to keep all of your main Python objects global in a real project is up to you; I primarily chose to do it here because it helps illustrate the process of initialization and shutdown more clearly.

Within the demo's main function, after loading the necessary graphics, Python is initialized and the script is loaded. Fortunately, most of this job is done for you by the `InitPython ()` function:

```
// Initialize Python
InitPython ();

// Load your script and get a pointer to its dictionary
g_pName = PyString_FromString ( "script" );
g_pModule = PyImport_Import ( g_pName );
if ( ! g_pModule )
    W_ExitOnError ( "Could not open script.\n" );
g_pDict = PyModule_GetDict ( g_pModule );
```

As was the case in the last demo, the script is loaded by putting its filename without the extension into the `g_pName` object with `PyString_FromString ()` (the script will of course be saved as `script.py`). A pointer to the module itself is stored in `g_pModule` after the script is imported with `PyImport_Import ()`, and by making sure it's not null, you can determine whether the script was loaded properly. You finish the loading process by storing a pointer to the script module's dictionary in `g_pDict`.

Next up, the script needs to be given a chance to initialize itself. Even though you haven't seen the script or its `Init ()` function yet, here's the code to call it from the host:

```
// Let the script initialize the rest

g_pFunc = PyDict_GetItemString ( g_pDict, "Init" );
PyObject_CallObject ( g_pFunc, NULL );
```

Because `Init ()` won't take any parameters, you just pass `NULL` instead of a python object array when calling `PyObject_CallObject`. This is a flag to the function that lets it know not to look for a parameter list.

The last section of code implements the main loop and shuts down Python upon the loop's termination. It starts by reusing the `g_pFunc` pointer from the last example as a pointer to the script-defined `HandleFrame ()` function:

```
// Get a pointer to the HandleFrame () function
g_pFunc = PyDict_GetItemString ( g_pDict, "HandleFrame" );

// Start the main loop
MainLoop
{
    // Start the current loop iteration
    HandleLoop
    {
        // Let Python handle the frame
        PyObject_CallObject ( g_pFunc, NULL );

        // Check for the Escape key and exit if it's down
        if ( W_GetKeyState ( W_KEY_ESC ) )
            W_Exit ();
    }
}

// Shut down Python
ShutDownPython ();
```

As you can see, the main loop of the program is now considerably simpler. All that's necessary is a call to `PyObject_CallObject ()` to invoke your frame-handling function, and a check to make sure the Escape key hasn't been pressed to terminate the demo. Again, you pass `NULL` in place of a parameter list, because `HandleFrame ()` won't accept any parameters. Everything is tied up nicely with a call to `ShutDownPython ()` when the loop breaks.

The Python Script

The last piece of the puzzle is a Python script to drive everything. The script can be found in `script.py`, and begins with a declaration of the constants it will need:

```
ALIEN_COUNT         = 12                # Number of aliens onscreen
MIN_VEL             = 2                 # Minimum velocity
MAX_VEL             = 8                 # Maximum velocity
```

```
ALIEN_WIDTH         = 128               # Width of the alien sprite
ALIEN_HEIGHT        = 128               # Height of the alien sprite
HALF_ALIEN_WIDTH    = ALIEN_WIDTH / 2   # Half of the sprite width
HALF_ALIEN_HEIGHT   = ALIEN_HEIGHT / 2 # Half of the sprite height

ALIEN_FRAME_COUNT   = 32                # Number of frames in the animation
ALIEN_MAX_FRAME     = ALIEN_FRAME_COUNT - 1 # Maximum valid frame

ANIM_TIMER_INDEX    = 0                 # Animation timer index
MOVE_TIMER_INDEX    = 1                 # Movement timer index
```

Again, however, like Lua, Python doesn't support formal constants. As a result, you simply have to use globals that use the traditional constant naming convention to simulate them. The "constants" defined here are the same ones you saw in Lua; just enough to regulate the velocity, size, quantity, and general behavior of the bouncing sprites.

Next up are the script's globals (or at least, the ones that aren't pretending to be constants). All the script needs to maintain globally is the current frame of animation and the sprite array itself, though, so this is a decidedly short section:

```
Aliens = []         # Sprites
CurrAnimFrame = 0   # Current frame in the alien animation
```

This leaves you with the script's functions, of which there are two. The first is `Init ()`, which as you saw, is called once before entering the main loop. This gives the script a chance to initialize the sprite array. This function, therefore, is concerned primarily with giving each on-screen alien sprite a random location, velocity, and spin direction:

```
def Init ():

    # Import your "constants "
    global ALIEN_COUNT
    global ALIEN_WIDTH
    global ALIEN_HEIGHT
    global MIN_VEL
    global MAX_VEL

    # Import the Aliens list
    global Aliens

    # Loop through each alien of the list and initialize it
    CurrAlienIndex = 0
    while CurrAlienIndex < ALIEN_COUNT:

        # Set a random X, Y location
        X = HostAPI.GetRandomNumber ( 0, 639 - ALIEN_WIDTH )
        Y = HostAPI.GetRandomNumber ( 0, 479 - ALIEN_HEIGHT )

        # Set a random X, Y velocity
        XVel = HostAPI.GetRandomNumber ( MIN_VEL, MAX_VEL )
        YVel = HostAPI.GetRandomNumber ( MIN_VEL, MAX_VEL )

        # Set a random spin direction
        SpinDir = HostAPI.GetRandomNumber ( 0, 2 )

        # Add the values to a new list
        CurrAlien = [ X, Y, XVel, YVel, SpinDir ]

        # Nest the new alien within the alien list
        Aliens.append ( CurrAlien )

        # Move to the next alien
        CurrAlienIndex = CurrAlienIndex + 1
```

Lastly, there's the `HandleFrame ()` function, which draws the next frame and handles the movement and collisions of the alien sprites. It also updates the current animation frame global:

```
def HandleFrame ():

    # Import your "constants"

    global ALIEN_COUNT
    global ANIM_TIMER_INDEX
    global MOVE_TIMER_INDEX
    global ALIEN_FRAME_COUNT
    global ALIEN_MAX_FRAME
    global HALF_ALIEN_WIDTH
    global HALF_ALIEN_HEIGHT

    # Import the globals

    global Aliens
    global CurrAnimFrame
```

```
# Blit the background

HostAPI.BlitBG ()

# Update the current frame of animation

if HostAPI.GetTimerState ( ANIM_TIMER_INDEX ):
    CurrAnimFrame = CurrAnimFrame + 1
    if CurrAnimFrame > ALIEN_MAX_FRAME:
        CurrAnimFrame = 0

# Loop through each alien and draw it

CurrAlienIndex = 0
while CurrAlienIndex < ALIEN_COUNT:

    # Get the X, Y location

    X = Aliens [ CurrAlienIndex ][ 0 ]
    Y = Aliens [ CurrAlienIndex ][ 1 ]

    # Get the spin direction

    SpinDir = Aliens [ CurrAlienIndex ][ 4 ]

    # Calculate the final animation frame

    if SpinDir:
        FinalAnimFrame = ALIEN_MAX_FRAME - CurrAnimFrame
    else:
        FinalAnimFrame = CurrAnimFrame

    # Draw the alien and move to the next

    HostAPI.BlitSprite ( FinalAnimFrame, X, Y )
    CurrAlienIndex = CurrAlienIndex + 1

# Blit the completed frame to the screen

HostAPI.BlitFrame ()
```

```
# Loop through each alien and move it, checking for collisions

CurrAlienIndex = 0
while CurrAlienIndex < ALIEN_COUNT:

    # Get the X, Y location

    X = Aliens [ CurrAlienIndex ][ 0 ]
    Y = Aliens [ CurrAlienIndex ][ 1 ]

    # Get the X, Y velocity

    XVel = Aliens [ CurrAlienIndex ][ 2 ]
    YVel = Aliens [ CurrAlienIndex ][ 3 ]

    # Move the alien along its path

    X = X + XVel
    Y = Y + YVel

    # Check for collisions

    if X < 0 - HALF_ALIEN_WIDTH or X > 640 - HALF_ALIEN_WIDTH:
        XVel = -XVel

    if Y < 0 - HALF_ALIEN_WIDTH or Y > 480 - HALF_ALIEN_HEIGHT:
        YVel = -YVel

    # Update the positions

    Aliens [ CurrAlienIndex ][ 0 ] = X
    Aliens [ CurrAlienIndex ][ 1 ] = Y
    Aliens [ CurrAlienIndex ][ 2 ] = XVel
    Aliens [ CurrAlienIndex ][ 3 ] = YVel

    # Move to the next alien

    CurrAlienIndex = CurrAlienIndex + 1
```

The logic here should speak for itself, and has been covered in the Lua section anyway. Speaking of Lua, you'll notice that this was one of many references to the Lua version of this demo. If you were to compare the scripts and even the host applications of each of these demos to one another, you'd find that they're almost exactly alike. This is because, as I said, scripting can often be approached in a language-independent manner.

That's everything for the Python demo, so check it out on the CD! You can find everything covered throughout this chapter in `Programs/Chapter 6/` on the accompanying CD.

Advanced Topics

As I've a few times stated before, Python is a large language with countless features and structures. To fully teach it would require a book of its own, but here's a list of both miscellaneous topics I just didn't have time to mention here, as well as advanced concepts that would've been beyond the scope of simple game scripting:

- **List Functions.** Python provides a number of useful functions for dealing with lists. These functions range from stack-like interfaces to sorting, and can be a godsend when writing list-heavy code. Before reinventing the wheel, make sure Python doesn't already have you covered.
- **Exceptions**. Python supports exceptions, an elegant method of error handling found in languages like C++ and Java. Rather than constantly having to pass around error codes and check the validity of handles, exceptions automatically route errors to a specialized block of code designed just for handling them.
- **Packages.** Packages are a built-in feature of the Python language, also found in Java. Packages let you group scripts, functions, and objects in a directly supported way that provides greater organization and promotes code reuse.
- **Object-Orientation.** Even though I didn't cover it here, Python has serious potential as an object-oriented language. For larger games that require more meticulous organization of entities and resources, objects become invaluable.

Web Links

Check out the following links for more information about Python:

- **Python.org:** `http://www.python.org/`. The central hub on the net for Python development news and resources. Lots of great documentation, up-to-date distribution downloads, and a lot more.
- **MacPython:** `http://www.cwi.nl/~jack/macpython.html`. The official home of the Python Mac port.

- **Jython.org:** `http://www.jython.org/`. Jython is an interesting project to port Python in its entirety to the Java platform, opening Python scripting to a whole new set of applications and users.
- **ActiveState:** `http://www.activestate.com/`. Makers of the ActiveState ActivePython distribution.

Tcl

So far this chapter has been dealing with languages that bear at least a reasonable resemblance to C. Lua and Python, despite their obvious syntactic quirks, are still fairly similar to the more familiar members of the ALGOL-family. What you're about to embark on, however, is a journey into the heart of a language unlike anything you've ever seen (assuming you've never seen Tcl, of course). Tcl is a truly unique language, one whose syntax is likely to throw you through a loop at first. Rest assured, however, that if anything, Tcl is in many ways the simplest of all three languages in this chapter. The best advice I can offer you as you're learning is to go slowly and try not to assume too much. New Tcl users have the tendancy to assume something works one way just because their instinct tells them so, when it clearly works some other way upon further inspection. So pace yourself and don't race ahead just because you think you've already got it down.

Tcl, which is actually pronounced phonetically as "Tickle" instead of the letters "T C L" like you might assume, stands for "Tool Command Language". It's a small, simplistic language designed to easily integrate with a host application and allow that host to define its own "commands" (which, in essence, form the Host API, to use a familiar term). Its syntax is designed to be ambiguous and flexible enough to fit applications in virtually any domain. These qualities make it a good choice as a scripting system.

These days, Tcl is virtually never mentioned on its own. Rather, it's been almost permanently associated with a related utility, "Tk" (pronounced "Tee Kay"), which is a popular windowing toolkit used to design graphical user interfaces. Tk is actually a Tcl *extension*—a new set of commands for the Tcl language that allows it to create windows, buttons, and other common GUI elements, as well as bind each of those elements to blocks of Tcl code to give the interface functionality. Tcl and Tk work so well together that Tk is now a required part of any Tcl distribution, and together the package is referred to collectively as Tcl/Tk. However, because windowing toolkits are much less important to the subject of game scripting than the Tcl language itself, I won't be discussing Tk.

ActiveStateTcl

You'll be using the ActiveStateTcl distribution throughout the course of this chapter. ActiveStateTcl is available for Linux, Solaris, and Windows, implementing Tcl 8.3 (the latest version at the time of this writing).

You can download ActiveStateTcl for free from `www.activestate.com`.

It's a clean and easy-to-use package, which can be installed in Windows simply by executing the self-extracting archive. It's almost as easy for Linux users; just put on a Babylon 5 T-shirt, get root access by telnetting into Pine, compile your `.tar` utility, hand-assemble `vi`, dump the resulting machine code stream into a shell script, and `chmod` everything. You should be up and running in no time. :)

Tcl is designed to be a simple language that's easy and fast to use. As a result, the average Tcl distribution is going to be fairly similar from one to the next, so the following rundown of the contents of ActiveStateTcl for Windows should at least generally apply to whatever distro you may happen to have (although it's recommended you follow the book's examples with the version supplied by ActiveState).

The Distribution at a Glance

ActiveStateTcl's distribution will unpack itself into a single directory called *TCL* (or something similar, unless you changed it at install time). I installed my copy in `D:\Program Files`, so everything I'll be doing from here on out will be relative to the `D:\Program Files\TCL` directory. This will have ramifications when it comes time to compile your demos, so make sure you know where Tcl has been installed on your machine.

Inside this root directory you'll find some obligatory text files (`license.terms` is just information on the distribution's licensing agreement, and `README.txt` is some quick documentation with further information on some installation details). There are also a number of subdirectories:

- **`bin/`.** Binaries of the Tcl implementation; you'll be interested in the executable utilities mostly.
- **`demos/`.** A number of demos for the various extensions ActiveStateTcl provides, many of which focus on the Tk windowing toolkit. I'm more concenred about the pure Tcl language itself, however—these extensions are generally for non-game related scripting tasks and as such will be of little use to you.
- **`doc/`.** Documentation on the entire Tcl distribution in the form of a single `.chm` file. The Tcl language reference alone in this thing makes it quite useful. You should make a habit of referring to this thing whenever you have a syntax or usage question (of course, this book can help too.

- `include/`. The header files necessary to use both the Tcl implementation of ActiveStateTcl, as well as the extensions it provides. You'll find quite a bit of stuff in here, but the only file in this folder you really need is `tcl.h`.
- `lib/`. The compiled library (`.lib`) files necessary to use Tcl within your programs. Like `include/`, it's a crowded folder, but all you'll really need is `tcl83.lib`. Everything else will follow from that.

You'll notice that some of the Tcl files you use throughout this chapter are appended with the "83" version number. This is specific to this distro and is not necessarily what you'll find in other versions or distributions. If you're having trouble finding your specific files, just look for the filename that overall seems closest to what you're looking for. If it's simply appended by what appears to be a version number or code, it's probably the one you want. For example, I'll make a number of references to `tcl83.lib`, but your distribution might have a file called `tcl82.lib`, or maybe even just `tcl.lib`. As you can see, all three filenames share the common `tcl*.lib` form. Just keep that in mind and you should be fine.

The tclsh Interactive Interpreter

Much like Lua, Tcl comes with an interactive interpreter that allows you to directly input code and see the results. It's called `tclsh` (which is short for "Tcl Shell," but is pronounced "ticklish"), so look for `tclsh.exe` under the `bin/` directory of your ActiveStateTcl installation. Its interface is also similar to Lua; featuring a single-character prompt:

```
%
```

It may not exactly roll out the welcome wagon, but it's a hugely useful program. Try to keep it open if you can as you tour the language so you can immediately test out the examples and make sure you're getting it down. Also, like Lua's interpreter, ending a line with a \ (backslash) allows it to be continued on the next line without being interpreted (until you enter a non-backslash terminated line).

The last important feature of `tclsh` is that you can immediately run and test full Tcl script files rather than individual lines of code by passing the script's filename to `tclsh` as the first command-line parameter. For example:

```
tclsh my_script.tcl
```

This code executes `my_script.tcl`.

At any time, enter `exit` at the `%` prompt to exit `tclsh`.

NOTE

In addition to tclsh, you may notice what appears to be a similar utility called `wish`. `wish` is another `tclsh`-like shell but is compiled with the Tk extension, allowing you to immediately enter and execute script code that creates and uses Tk GUIs. Again, because Tk is beyond the scope of simple game scripting, you won't have a need for it. It's definitely fun to play with though.

What, No Compiler?

That's right, most pure versions of Tcl do not ship with a compiler, which means all scripts are loaded by the game directly as human-readable source. Because you should know by now that loading a script at runtime is not a good way to handle compile-time errors, remember to use `tclsh` to attempt to execute your file beforehand; this will help weed out compile-time errors with adequately descriptive messages, a luxury you won't have at runtime.

Tcl Extensions

As you will soon see, Tcl is a language based primarily on the concept of *commands*. Although you won't actually see what a command is in detail until the next section, commands can be thought of in a manner similar to a function call in C, although sometimes they're designed to emulate control structures like conditional branching and loops as well. All versions of Tcl support a simple set of built-in commands called the *Tcl core*. To expand the language's functionality to more specific domains, however, Tcl is designed to support *extensions*.

A Tcl extension is a compiled implementation of new commands that can be linked with the host application to provide scripts with new functionality. In a lot of ways, extensions are like C libraries; they're a specialized group of functions that provide support for a specific domain—like graphics and sound—that the language alone would not have otherwise provided. Tk is a good example of an extension; when linked with your program, your Tcl scripts can use it to invoke the GUI elements it supports.

ActiveStateTcl comes with a large number of extensions ready to use, which is why there are so many files and subdirectories in the `include/` and `lib/` directories. I know I'm beginning to sound like a broken record, but these are beyond the scope of the book and can be ignored.

NOTE

Just to make sure you're clear on why you're told to ignore these extensions, imagine if this was a book on general game programming in C++. I'd start off by introducing the C++ compiler, and would walk you through the various libraries it came with like DirectX, the Win32 API, and so on. However, I'd be sure to mention that a lot of the libraries the compiler may come with, such as database access APIs, are not specifically related to game programming and can be ignored. Of course, later you may find that your game works well with a database, and end up using the libraries anyway, so I encourage you to investigate Tcl's extensions on your own. You may find more than a few game-related uses for them.

The Tcl Language

Now that you're familiar with the Tcl distribution, you can move on to the language. Tcl can be difficult to get comfortable with, because there are some deceptively subtle differences in its fundamental nature when compared to the more conventional languages studied thus far. Ironically, Tcl's incredible simplicity and generic design end up making it especially confusing to some newcomers.

Commands—The Basis of Tcl

The major difference between Tcl and traditional C-like languages is not immediately apparent, but is by far the most important concept to understand when getting started. *There is no such thing as a statement, keyword, or construct in Tcl;* every *line of code is a command.* Recall the discussion of command-based languages in Chapter 3. You'll be surprised to find that Tcl is rather similar; instead of using keywords and constructs to form assignments, function calls, conditional logic, and iteration, *everything* in the Tcl language is done through a specific command. But what exactly *is* a command?

A Tcl *command* is just like the commands discussed in Chapter 3, albeit considerably more flexible both in terms of syntax and functionality. A Tcl command is a composition of *words.* Just like English, the Tcl language defines a word as a consecutive collection of characters. By "consecutive" I mean that there is no internal whitespace, and it should also be noted that Tcl's definition of "character" literally means just about any character, including letters, digits, and special symbols. Also like English, Tcl words are separated by whitespace which can consist of spaces or tabs. Here's an example of a Tcl command called `set`.

```
set X 256
```

CAUTION

Command names are case-sensitive, so `set` is not the same as `SET`, `Set`, or `SeT`.

The `set` command is used for setting the value of a variable (which makes it analogous to C's = assignment operator). In this example, the command consisted of three words. The first word was the name of the command itself ("`set`"). All Tcl commands must obviously identify themselves, and therefore, all Tcl commands are one or more words in length. The first word is always the command name. After this word, you find two more; `X` and `256`. `X` is the name of the variable you want to put the value into, and `256` is the value.

As you can most likely see, commands mirror the concept of function calls; the first word is like the function identifier, whereas all subsequent words provide the parameters. Because of this, the order of the words is just as important as the order of parameters when calling a function. For example, whereas the previous example would set `X` to the desired value, the following would cause an error:

```
set 256 X
```

For obvious reasons, I might add. Putting `X` "into" `256` doesn't make any more sense than the following would in C:

```
256 = X;
```

Also, like functions, commands generally return a value. Even `set` does this; it returns whatever value was set to the variable in question. Because `tclsh` prints the output of each command you issue, entering the previous line in the interpreter will result in this:

```
% set X 256
256
```

So, to summarize what you've learned so far, *every* line of a Tcl script is a command. Commands are a series of whitespace-separated words, wherein the first word is always the commands name, and the words following are the command's parameters. Commands generally return values as well.

This may seem odd at first, and you might find yourself asking questions like, "if *every* line is a command, how can you do things like expressions, conditional logic, and loops?" To understand the answer, you need to understand the next piece of the Tcl puzzle, *substitutions.*

Substitution

The next significant aspect of Tcl is that conceptually, it's a highly recursive language. This is due to the fact that commands can contain commands within themselves; in turn, those commands can further contain commands, a process that can continue indefinitely. That was an awkward sentence I know, so here's an example to help make things a bit clearer:

```
set X [ expr 256 * 256 ]
```

Here, you almost seem to be deviating from the standard practice of defining commands as a string of space-delimited words. This, however, is the Tcl syntax for embedding a command into another command (the brackets are each considered single-character words of their own). In this case, the new command `expr`, which evaluates expressions, was embedded into `set` as the third word (or second *parameter*, as I prefer to say it). A more intelligent way to think about this relationship, however, is in terms of *substitution.* Remember, most commands produce an output of some sort. In the case of `expr`, the output is obviously the result of whatever expression was fed to it. So for example, entering the `expr` statement by itself into `tclsh` would look like this:

```
% expr 256 * 256
65536
```

As you can see, the output of `expr 256 * 256` is `65536`, the product of the multiplication. When evaluating the following command:

```
set X [ expr 256 * 256 ]
```

the Tcl interpreter takes the following steps:

1. The first word is read, informing Tcl that a `set` command is being issued.
2. The second word is read, which tells the `set` command that the variable `X` is the destination of the assignment.
3. The open bracket `[` is read, which informs Tcl that a new command is beginning *in place of* `set`'s second parameter.
4. The former set operation is now on hold as the next word is read. Because you're now dealing with a new command, the word-reading process starts over after the `[`, and the next word is once again treated as the command's name. `expr` is read, telling Tcl that an expression command is now being issued.
5. Tcl reads every word following `expr` and sends it as a separate parameter. Because of this, `256`, `*`, and `256` are all sent to `expr` separately (but in the proper order of course). `expr` then analyzes these incoming words and evaluates the expression they describe. In this regard, the `expr` command is much like a calculator.
6. Tcl encounters the closing bracket `]`, and, rather than sending it as another parameter to `expr`, treats it as a sign that the second, embedded command has finished, and the `set` command can be resumed. The result of the `expr` command then *substitutes* the original `[ expr 256 * 256 ]` command.
7. The output of the `expr` expression, 65536, is sent to `set` as the second parameter (or, more specifically, the value that will be placed in `X`).
8. The `set` command is invoked, and `X` is assigned 65536.

One of the key points to realize here is that `set` never knew that `[ expr 256 * 256 ]` was ever one of its parameters, because Tcl automatically evaluated the command and substituted it with whatever output it produced. Because of this, the following two lines are equivalent and appear identical from the perspective of the `set` command:

```
set X [ expr 256 * 256 ]
set X 65536
```

To further understand this, imagine that you wrote the following function in C:

```
int Add ( int X, int Y )
{
   return X + Y;
}
```

It's just a simple function for adding two integers and returning the result. However, imagine you called it like this:

```
int Sum = Add ( 16 * 16, 128 / 4 );
```

Both parameters in this case are not immediate integer values, but are rather expressions. Rather than sending the string representation of these expressions to the `Add ()` function, the runtime environment will first evaluate them, and simply send their results as parameters. Just like the `set` command, `Add ()` will add the two values, never knowing they were the results of expressions. Besides, `Add ()` is defined with one line of code— hardly enough logic to properly parse and evaluate a mathematical expression. `set` is similar in this regard. The actual `set` command itself has no expression parsing capabilities whatsoever, which means that it, and virtually all other commands in Tcl, relies on `expr` to provide that.

This concept can and is taken to the extremes, so being able to understand this process quickly is key to mastering Tcl. Here's a slightly more complicated example, taken directly from `tclsh`:

```
% set X [ expr [ set Y 4 ] * 2 ]
8
```

As you can see, the commands are now nested two levels deep. Basically, `X` is set the result of an expression. The expression is defined as the result of the `set` command multiplied by 2. Because `set` returns whatever value it put into the specified variable, which was `4`, this evaluates to 8, which finally is `set` to `X`. Figure 6.17 illustrates this process graphically.

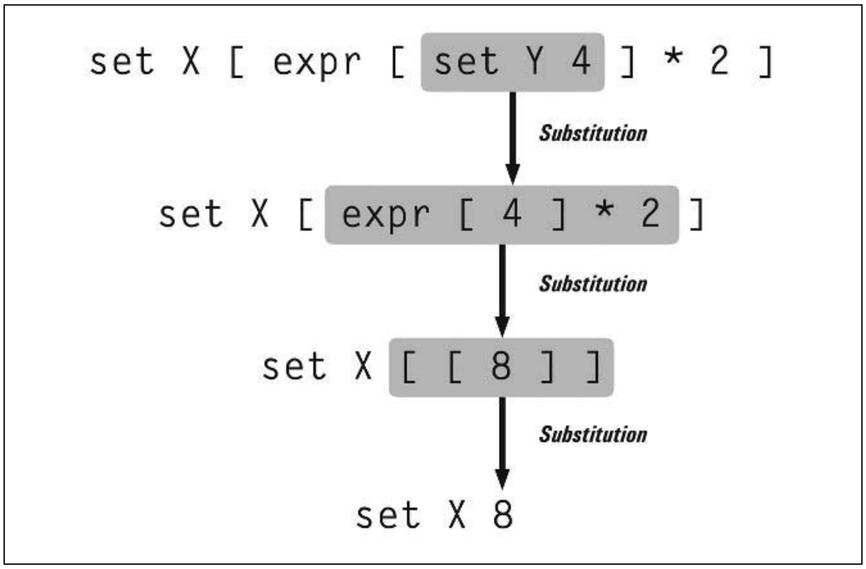


Figure 6.17

A breakdown of Tcl command substitution.

I haven't covered the details of expressions yet, but this should help you understand how complex programming can be done using a language based entirely on commands, provided those commands can be nested within one another.

NOTE

As a matter of style and convention, commands should not be nested too deeply. Just like extremely complex one-line expressions are generally not appreciated in C when they could be written more clearly with multiple lines, Tcl code is easier to read and understand when a possibly huge nest of embedded commands is broken into multiple, simpler commands instead.

What you see is known as *command substitution.* This is a useful technique and is one of the cornerstones of Tcl programming, but another equally important concept is *variable substitution.* For reasons you'll learn about later, a variable name alone can't just be dropped into an expression, like this:

```
set X [ expr Y / 8 ]
```

Attempting to run this in `tclsh` will yield the following:

```
syntax error in expression "Y / 8"
```

Furthermore, you can't simply assign one variable to another, whether an expression is involved or not. You'll inadvertently set the variable in question to a string containing the name of the second variable rather than its value. For example:

```
% set Y 256
256
% set X Y
Y
```

As you can see, the output of the first assignment was the numeric value 256, like you would expect. In the second case, however, you simply set `X` to the string "`Y`", which is not what you intended. In order to make this work, you use the dollar-sign `$` to prefix any variable whose value should be substituted in place of its identifier. For example:

```
% set Y 256
256
% set X $Y
256
```

This clearly produces the proper value. Just as the `[]` notation told Tcl to replace the command within the brackets with the command's output, the `$` tells Tcl to replace the name of the variable after the dollar sign with its value. So, this too is considered identical from the perspective of `set`:

```
set X $Y
set X 256
```

Assuming `Y` is equal to 256, of course. Lastly, let's see how this can be used to correct the first example:

```
% set X [ expr $Y / 8 ]
32
```

Presto! The expression now evaluates as intended, without error, and properly assigns `32` to `X`.

One last thing before moving on—despite the fact that most commands return a value, and that `tclsh` will always print this value immediately following the execution of the command, you can also print values of your own to the screen using the `puts` (put string) command, like this:

```
set X "Hello, world!"
puts $X
```

This will print:

```
Hello, world!
```

So, in a nutshell, Tcl lives up to its name as a "command language". Because almost everything Tcl is capable of doing is actually credited to a specific command rather than the language itself, Tcl on its own is a very simplistic, hollow entity. I personally find this to be a fascinating approach to coding, as it makes for an extremely high-level language that's just about as open-ended as it could possibly be.

Each time Tcl is used, the host application it's embedded in will invariably provide its own set of specialized commands. Again, these are conceptually identical to the host API concept. However, each instance of Tcl does indeed bring with it a small set of common commands for variable assignment, expression parsing, and the like. These basic, common commands are known as the *Tcl core* and are always present. You can almost think of them as the standard library in C, except that you don't need to manually include them.

At this point, as long as you've understood everything so far, you're out of the woods with Tcl. Being able to make sense of its substitution rules and the concept of a language based solely on commands will allow you to learn and use the rest of the language with relative ease. However, this means that if anything so far has been unclear, I *strongly* urge you to re-read it until it makes sense. You'll have significant trouble understanding anything else if you don't already have this down. It's like trying to learn trigonometry or calculus without first learning algebra—without that basis firmly in place, you won't get very far.

Anyway, with this initial Tcl philosophy out of the way, let's get on to actually examining the language (which, as I mentioned previously, is primarily just a matter of learning about the commands in the Tcl core).

Comments

Comments in Tcl are almost the same as they were in Python, and are denoted with the hash mark (#). Everything following the hash mark is considered a comment. For example:

```
# Set X to 256
set X 256
```

There's one snag to Tcl comments, though, which is a side-effect of the way Tcl interprets a command. Remember that all Tcl scripts boil down to space-delimited words. Because of this, putting a comment after a word will end up producing unwanted results. For example:

```
set X 256      # Set X to 256
```

At first glance, this looks just like the first example, the only difference being the comment following the command on the same line. Entering this in `tclsh` however will produce the following:

```
wrong # args: should be "set varName ?newValue?"
```

The problem is that Tcl broke the previous line into eight words, whereas in the first example, the `set` line was only three words. Because of this, `set` was sent seven parameters instead of two:

```
X, 256, #, Set, X, to, 256
```

When `set` noticed it was receiving extra words beyond `256`, it issued the previous error. To alleviate this problem, make sure to terminate any command that will share its line with a semicolon like this:

```
set X 256;     # Set X to 256
```

Which will work just fine. Tcl knows that the command has ended when it reaches the semicolon, so it won't attempt to send the comment as a parameter. This brings up another aspect of Tcl's syntax, however, which is that lines can optionally end with a semicolon, and that semicolons can be used to allow more than one command on a given line. For example:

```
set X 256; set Y $X
```

Will set `X` and `Y` to 256 without any trouble. Ultimately, this means that you can make a case either way for the use of semicolons in your Tcl scripts. On the one hand, I personally feel they're unnecessary because I rarely put comments on the same line as code in any language. However, many people do (including me, for that matter, when I'm declaring a constant or global) and will be forced to use them in at least some cases. Because I think consistency is important, I suggest you either don't use semicolons at all (and therefore give all of your comments their own line), or use them everywhere.

Variables

In Tcl, all values are stored internally as strings. Although Tcl does do its share of optimization to make sure that clearly numeric values are not subject to constant string manipulation overhead, you can still think of all Tcl values as conceptually being string-based. As a result, Tcl is yet another example of a typeless scripting language; a rather ubiquitous trait—if not something of an unofficial standard—in the world of scripting.

As you've seen, variables are created and initialized with the `set` command. This command accepts two parameters, an identifier and a value. If the identifier doesn't correlate to an already existing variable, a new variable of that name will be created. Here are some examples of using `set`:

```
# Create a variable with an integer value
set IntVar 256
puts $IntVar
# Create a variable with a floating-point value
set FloatVar 3.14159
puts $FloatVar
# Create a variable with a one-word string value
set ShortStringVar Hello,
puts $ShortStringVar
# Create a variable with a longer string
set LongStringVar "Hello, world!"
puts $LongStringVar
```

The output of the previous code will be the following:

```
256
3.14159
Hello,
Hello, world!
```

An interesting aspect of this example is that the third variable created, `ShortStringVar`, is assigned a string that isn't in quotes. To understand this, remember that Tcl defines a word as any sequence of characters that isn't broken up by whitespace. Because of this, the `set` command is sent that single word as the value to assign to `ShortStringVar`, which is of course `Hello,`. What this tells you is that the purpose of strings in Tcl is different than other languages. The concept of a string in Tcl is less about data and data types, and more about simply grouping words. Anything surrounded in double quotes is interpreted by Tcl to be a single word, even if it includes spaces. This is also the reason why assigning a variable to another variable like this:

```
set X Y
```

Only serves to assign the variable's name (in this case, `X` takes on the string value "`Y"`, as you saw previously).

The next variable-related command worth discussing is `unset`, which can be used to delete a variable:

```
# Create a string variable and print it
set Ellie "They're alive."
puts $Ellie

# Delete it and try printing it again
unset Ellie
puts $Ellie
```

Here's the output:

```
They're alive.
can't read "Ellie": no such variable
    while executing
"puts $Ellie"
```

As you can see, the first attempt at printing the value succeeded, but when `unset` cleared the variable from Tcl's internal records, the second attempt resulted in an error. This shows you that Tcl does require all variables to be created with the `set` command.

Next up is the `incr` command, which lets you add a single value to an integer variable, usually for the purpose of incrementing it. Because of this, `incr` defaults to a value of 1 if only the variable name is specified. Although `incr` adds whatever value you pass it, you can decrement the variable as well by passing a negative number. Here's an example:

```
# Create an integer variable and print its value
set MyInt 16
puts $MyInt

# Increment MyInt by one
incr MyInt
puts $MyInt
```

```
# Add 15 to MyInt
incr MyInt 15
puts $MyInt

# Decrement MyInt by 24
incr MyInt -24
puts $MyInt
```

Here's the example's output:

```
16
17
32
8
```

The last variable-related command I'll discuss here is `append`, which you can think of as `incr` for strings. Because `incr` only alters the value of integer variables, you'll get an error if you try passing a string or float to it. `append`, on the other hand, let's you append a variable number of values to a string. Check it out:

```
# Create a string
set Title "Tao of"
puts $Title

# Append another string to it
append Title " the"
puts $Title

# Append two more strings to it
append Title " " "Machine"
puts $Title
```

This code produces the following output:

```
Tao of
Tao of the
Tao of the Machine
```

Notice that in the second call to `append`, two strings are passed, one of which is a space. Remember, because Tcl words are delimited by spaces, the only way to pass whitespace to a command is to surround it with quotes. As a side note, passing a numeric variable to `append` will immediately (but permanently) change that variable into a string containing the string-representation of the number.

One thing about `append` is that its functionality seems redundant; after all, the following `append` example:

```
# Append a variable using the append command
set Title "Running Down "
append Title "the Way Up"
```

could be written just as easily with only the `set` command and produce the same results:

```
# Append a variable using the set command and variable substitution
set Title "Running Down"
set Title "$Title the Way Up"
```

`append`, however, is more internally efficient in cases like this, when a string needs to be built up incrementally. Besides—the syntax is clearer this way anyway, so you might as well just make a habit of doing simple string concatenation with `append` instead of `set` with substitution.

NOTE

One *extremely* important detail to master is knowing when to use a variable name as-is (`MyVar`) and when to use variable substitution (`$MyVar`). Use the variable name when a command actually expects a variable's *identifier*—such as the first parameter for `set`, `incr`, or `append`. Use variable substitution when you want the command to receive the variable's *value* instead, like `puts`, or the second parameters for `set`, `incr`, and `append`.

Arrays

The next step up from variables in Tcl is the array. Tcl arrays, like Lua tables, are actually *associative arrays* or *hash tables*. They allow keys to be mapped to values in the same way a C array maps integer indexes to values. Of course, Tcl arrays can use integer indexes in place of keys, but that's up to you—this is another product of Tcl treating all data as strings.

Tcl arrays are like variables in that they are created at the time of their initialization, and are referenced with the following form:

```
ArrayName(ElementName)
```

Note that a Tcl array index is surrounded in `()`, rather than `[]` like many other languages. Here's an example:

```
# Create an array with four indexes
set MyArray(0) 256
set MyArray(1) 512
set MyArray(2) 1024
set MyArray(3) 2048
```

This creates an array of four elements called `MyArray` and assigns values to each index. You may notice that, in a departure from my normal coding style, there aren't spaces around the parentheses and index in the array reference. Normally I'd use `MyArray ( 0 )`, rather than `MyArray(0)`. This is another example of Tcl's separation of words with spaces. If you were to attempt to run the following code:

```
set MyArray ( 0 ) 10
```

You'd get an error for sending too many parameters to `set`, because it would receive the following five words from Tcl:

```
MyArray
(
0
)
10
```

Note that even though you've only been using what appear to be integer indexes so far to enumerate the arrays, Tcl is actually interpreting them as strings. As a result, the following two lines of code are equally valid:

```
# Create an associative array
set MyArray(0) 3.14159
set MyArray(Banana) 3.14159
puts $MyArray(0)
puts $MyArray(Banana)
```

Here's the output:

```
3.14159
3.14159
```

Arrays in Tcl are pretty simple, as you've seen so far. The only other real issue I'd like to mention is multidimensional arrays. Tcl doesn't support them directly, but thanks to a clever side-effect of Tcl's variable substitution, you can simulate them with a syntax that looks as if they were actually part of the language. Check out the following, while keeping in mind that Tcl only supports a single dimension:

```
# Create a seemingly two-dimensional array
set MyArray(0,0) "This is 0, 0"
set MyArray(0,1) "This is 0, 1"
set MyArray(1,0) "This is 1, 0"
set MyArray(1,1) "This is 1, 1"

# Print two of its indexes
puts $MyArray(0,0)
puts $MyArray(1,1)

# Now print two more, using variables as indexes
set X 0
set Y 1
puts $MyArray($X,$Y)
set X 1
set Y 0
puts $MyArray($X,$Y)
```

To understand how this works, remember that `Tcl` allows any string to be used as an index. In this case, the strings you chose just happened to *look* like the syntax for multidimensional array indexes. Tcl just lumps indexes like "`0,0`" into a single string. And why shouldn't it? There aren't any spaces, so it doesn't have any reason not to. The previous array is really just a single-dimensional associative array, in which the keys are "`0,0`", "`0,1`", "`1,0`" and "`1,1`". As far as Tcl is concerned, the keys could just as well be "`Red`", "`Green`", "`Blue`" and "`Yellow`".

The real cleverness, however, is using variables to access the array. Because variable substitution occurs *before* the values of parameters are passed to a given command, you can basically construct your own variable identifier on the fly, even in the case of commands like `set` and `append`. Because of this, you're using variables to put together an index into an array at runtime. If `X` contains the string "`0`", and `Y` contains the string "`1`", you can concatenate the two strings with a comma in between them to create the final array index: "`0,1`". Tcl, however, is still oblivious to your strategy and considers it just another string index, as it would "`Banana`" constructed from "`Ban`", "`a`", and "`na`".

Expressions

The funny thing about expressions is that Tcl has absolutely no built-in support for them whatsoever. This may seem like a strange statement to make for two reasons:

- Any decent language is going to have to support expressions in order to be useful.
- You've already seen examples of expressions, albeit simple ones, earlier in this chapter.

Both of these points are correct. So what gives?

Basically, what I'm driving at is the fact that the Tcl *language* doesn't support expressions in any way. As you've seen, all Tcl really does is pass space-delimited words to commands and perform substitution with the `$` and `[]` notation. So, to provide expression-parsing support, the `expr` command was created. This seems like a trivial detail, but it's very important. The *only* reason you've been able to use expressions in the examples so far is because `expr` provides that functionality.

As has been demonstrated, `expr` is used to evaluate any expression and is generally embedded as a parameter in other commands. It always returns the final value of whatever expression was fed to it. Here's an example:

```
# Create some variables
set X 16
set Y 256
set Z 512
# Print out an arbitrary expression that uses all three
puts [ expr ( $X * $Y ) / $Z + 2 ]
```

This code outputs `10`.

From now on, even when I refer to "Tcl expressions," or "expressions in Tcl," what I am really referring to is the `expr` command specifically (or any other command that provides expression parsing functionality as well, of which there are a few). I'll use these phrases interchangeably, however.

The `expr` command supports the full set of standard operators, as you'd expect. Tables 6.11 through 6.14 list Python's operators. Note that I've added a new column for the data types that each operator supports.

Table 6.11 Tcl Arithmetic Operators

Operator	Description	Supported Data Types
+	Add	Integer, Float
-	Subtract	Integer, Float
*	Multiply/Multiply Strings	Integer, Float
/	Divide	Integer, Float
%	Modulus	Integer, Float
-	Unary Negation	Integer, Float

Table 6.12 Tcl Bitwise Operators

Operator	Description	Supported Data Types
<<	Shift Left	Integer
>>	Shift Right	Integer
&	And	Integer
^	Xor	Integer
|	Or	Integer
~	Unary Not	Integer

Table 6.13 Tcl Relational Operators

Operator	Description	Supported Data Types
<	Less Than	Integer, Float, String
>	Greater Than	Integer, Float, String
<=	Less Than or Equal	Integer, Float, String
>=	Less Than or Equal	Integer, Float, String
!=	Not Equal	Integer, Float, String
==	Equal	Integer, Float, String

Table 6.14 Python Logical Operators

Operator	Description	Supported Data Types
&&	And	Integer, Float
||	Or	Integer, Float
!	Not	Integer, Float

Something you can quickly ascertain from these tables is that string operands are only permitted when using the relational operators (<, >, <=, >=, !=, ==). Something you may be wondering, though, is why or how the data type of an operand even matters, because I've belabored the fact that Tcl sees everything as strings. This may be true, and Tcl *does* indeed see the world in terms of strings, but the `expr` command specifically is designed only to deal with numerics (except, again, in the case of the relational operators).

Remember that there's really no such thing as a variable when `expr` evaluates an expression. It, like any other Tcl command, is just being fed a series of words that it attempts to convert to either numbers or operators. What really happens when you try using string variables or literals, from the perspective of `expr`, is that suddenly all these letters and non-operator symbols begin to appear in the stream of incoming words. Understandably, this causes it to freak out. Consider the following example:

```
# Create an integer variable
set MyInt 32768
# Create a string variable
set MyString "Ack!"
# Attempt to use the two in an expression
puts [ expr $MyInt * $MyString + 2 ]
```

The initial batch of words to be sent to `expr` looks like this:

```
$MyInt * $MyString + 2
```

This looks like a valid expression, when you ignore the contents of `MyString`, at least. Now let's look at the final stream of words after Tcl performs variable substitution, which is what `expr` will see:

```
32768 * Ack! + 2
```

Doesn't make much sense, right? This should help you understand why certain data types make sense in certain places and others don't. It has nothing to do with Tcl specifically; it's simply the way the `expr` command was designed.

Conditional Logic

With expressions under your belt, you can move on to tackle conditional logic. At this point, after I've beaten the concept of Tcl commands into your head, you should be well aware that every line of a Tcl script is a command (or a comment), without exception. How then, is something like an `if` construct implemented?

Simple—`if` is a command too. Except, unlike C's `if`, which wraps itself around code blocks and only allows a certain block to be executed based on the result of some expressions, `if` accepts the expression and code blocks as parameters. Here's an example:

```
# Create a variable
set X 0

# Print different strings depending on its value
if { $X > 0 } {
   puts "X is greater than zero."
} else {
   puts "X is zero or less."
}
```

Which outputs:

```
X is zero or less.
```

What you're seeing here is a command whose parameters are chunks of Tcl code. The syntax that provides this, the `{}` notation, is actually a special type of string that allows line breaks and suppresses variable substitution. In other words, this particular type of string is much more WYSIWYG than the double-quote style. Because line breaks can be included in the script, this allows you to code in a much more natural, C-like fashion, as shown. Without this capability, the expression and code for each clause would have to be passed to `if` in a single line. In fact, here's another `if` example that uses the same syntax as above, but looks a bit more like the command that it really is:

```
# Create a variable
set Y 0
# Print different strings depending on its value
if { $Y < 0 } { set Y 0 } else { set Y 1 }
```

The parameters passed to this command are:

```
{ $Y < 0 }, { set Y 0 }, else, { set Y 1 }
```

Also supported is the `elseif` clause, which can exist zero or more times in a given `if` structure. Here's an example:

```
set Episode 5
if { $Episode == 4 } {
   puts "A New Hope"
} elseif { $Episode == 5 } {
   puts "The Empire Strikes Back"
} elseif { $Episode == 6 } {
   puts "Return of the Jedi"
} else {
   puts "Prequel"
}
```

Note also that the first parameter passed to an `if` command is an expression; like `expr`, `if` provides its own expression-evaluation capabilities.

Lastly, you may again be wondering why I've again deviated from my usual coding style by putting the opening and closing curly-braces of each code block in unusual places. This is another syntax imposition on behalf of Tcl. Remember, the only reason you're getting away with these line breaks in the first place is because `{}` strings allow them. This means that the line breaks can only occur *`within`* the braces, forcing me to make sure that each word begins on a line where a curly-brace string is beginning or ending as well. Without this, the Tcl interpreter would lose the continuity that helps it find its way from the beginning to the end of the command.

NOTE

Tcl does support a `switch` command, but to keep things simple I've decided not to cover it. Naturally, you can always use `if-elseif-else` blocks to simulate its functionality.

Iteration

Looping in Tcl is just like conditional logic; it's yet another example of commands performing tasks that you wouldn't necessarily think they're capable of. As always, you're going to get started with the trusted `while` loop:

```
set X 16
while { $X > 0 } {
   incr X -1
   puts "Iteration: $X"
}
```

Here's the output:

```
Iteration: 15
Iteration: 14
Iteration: 13
Iteration: 12
Iteration: 11
Iteration: 10
Iteration: 9
Iteration: 8
Iteration: 7
Iteration: 6
Iteration: 5
Iteration: 4
Iteration: 3
```

```
Iteration: 2
Iteration: 1
Iteration: 0
```

Almost identical to C, right? Indeed, `while` has been implemented in a familiar way. The command takes two parameters, an expression and a code block to execute as long as that expression evaluates to true (which, if you remember, is defined in Tcl as any nonzero value). Here's the `while` from the previous example rewritten in a form that helps remind you that it's just another command like anything else:

```
while { $X > 0 } { incr X -1; puts "Iteration: $X" }
```

`for` follows `while`'s lead by following a very C-like form. The `for` command accepts four parameters; the first three being the typical loop control statements you'd find in a C for loop—the initialization, the end case, and the iterator—with the fourth being the body of the loop. The following code rewrites the functionality of the `while` example:

```
for { set X 16 } { $X > 0 } { incr X -1 } {
   puts "Iteration: $X"
}
```

Which provides the expected output, of course:

```
Iteration: 16
Iteration: 15
Iteration: 14
Iteration: 13
Iteration: 12
Iteration: 11
Iteration: 10
Iteration: 9
Iteration: 8
Iteration: 7
Iteration: 6
Iteration: 5
Iteration: 4
Iteration: 3
Iteration: 2
Iteration: 1
```

Notice how closely this code mirrors its C equivalent

```
for ( int X = 16; X > 0; -- X )
   printf ( "Iteration: %d\n", X );
```

Everything is in roughly the same place, so you should feel pretty much at home.

Lastly, just like the other two languages, Tcl gives you `break` and `continue` for obvious purposes. `break` causes the loop to immediately terminate, causing program flow to resume just after the last line of the loop. `continue` causes the current iteration of the loop to terminate prematurely, causing the next one to begin immediately.

Functions (User-Defined Commands)

Tcl supports functions, but thinking of them as C functions isn't exactly appropriate. What you're really going to do in this chapter is define your own new Tcl commands. Because commands are identified with a name, are passed a list of parameters, and can return a value, they really are identical to functions in a conceptual sense. However, calling one of these "functions" follows the exact same syntax as calling a Tcl core command; as a result, it's better practice to refer to the following as user-defined commands.

Creating a Tcl command is remarkably easy. Once again, as expected, the actual syntax for creating a command is itself a command, called `proc` (short for procedure, which is yet another name you could call these things). `proc` accepts three parameters; a command name, a parameter list, and a body of Tcl code. As you'd expect, once this command finishes execution, the new user-defined command can be called by its name, passed any necessary parameters, and executed (the Tcl environment will locate and run the code you provided in the third parameter). The result, as with all other commands, then replaces its caller.

To get things started, let's look at a user-defined command example:

```
proc Add { X Y } {
    expr $X + $Y
}
puts [ Add 32 32 ]
```

Which produces the output of `64`. This example creates a new command called `Add`, which accepts two parameters, adds them, and returns the sum. Note that the second parameter to `proc`, after the name `Add`, is a space-delimited parameter list. In this case, it consists of `{ X Y }` and tells `proc` that your function should accept two parameters using these names.

TIP

What you're actually looking at in the case of the `{ X Y }` parameter list is what's known as a Tcl *list*. A list is basically a lightweight version of an array, and is somewhat awkward to use. It's fine in the case of specifying parameter lists for use with the `proc` command, but it's not all that useful in general practice—especially when you can just use associative arrays. As a result, I won't be covering lists in this book.

Because most Tcl commands return values, you probably will too at some point. Just like other languages, this is done with the `return` command. `return` causes whatever command it's called from to exit, and its single parameter is returned as the return value. For example, if you changed the custom Add command to look like this:

```
proc Add { X Y } {
   return 0
   expr $X + $Y
}
puts [ Add 32 32 ]
```

The command would always return 0, no matter what parameters you pass it.

The last issue to discuss with custom commands is that of global variables. Unlike languages like C, you can't simply refer to a global from within a command. For example, attempting to do the following will produce an error:

```
# Create a global variable
set GlobalVar "I'm global variable."

# Create a generic command
proc TestGlobal {} {
   # Create a local variable
   set LocalVar "Not me, I'm into the local scene."

   # Print out both the global and local
   puts $GlobalVar
   puts $LocalVar
}

# Call your command
TestGlobal
```

The interpreter will produce an error telling you that the variable GlobalVar hasn't been initialized when you pass it to puts. This is because globals are not automatically imported into a command's local scope. Instead, you must do so manually, using the global command like so:

```
# Create a global variable
set GlobalVar "I'm global variable."

# Create a generic command
proc TestGlobal {} {
   # Create a local variable
   set LocalVar "Not me, I'm into the local scene."
```

```
    #Import the global variable
    global GlobalVar

    # Print out both the global and local
    puts $GlobalVar
    puts $LocalVar
}

# Call your command
TestGlobal
```

The error will no longer occur, and the output will look like this:

```
I'm global variable.
Not me, I'm into the local scene.
```

CAUTION

You're free to create local variables with the same name as globals, but an error will occur if you attempt to use `global` to import a variable into the local scope after a local variable has already been initialized with its name. In other words, if you're going to be using a global variable in your function, don't create any other variables beforehand with the same name.

This works because `global` brings the specified global variable into the function's local scope until it returns.

Integrating Tcl with C

The integration of Tcl with C is rather easy, and involves much less low-level access than does Lua. Tcl does not force you to deal with an internal stack, for example; rather, high-level functions are provided for common operations like exporting functions, reading globals, and so on.

Just like you did with Lua, you'll first write a few basic scripts and then move on to recode the alien head demo. Along the way you'll learn the following:

- How to load and execute Tcl scripts from C.
- How to export C functions so that they can be called as commands from Tcl scripts.
- How to invoke both Tcl core and user-defined commands from C.
- How to pass parameters and return values to and from both C and Tcl.
- How to manipulate a Tcl script's global variables.

Compiling a Tcl Project

To get things started, let's briefly cover the details involved in compiling a Tcl application. First and foremost, just like with Lua, make sure you have the proper paths set in your compiler. I won't repeat every last detail that I mentioned in the Lua section, but in a nutshell, make sure your include file and library directories match the `include/` and `lib/` subdirectories of your Tcl installation.

Once your paths are set, include the main Tcl header:

```
#include <tcl.h>
```

Finally, physically include the `tcl83.lib` library with your project (remember, of course, that your distribution's main .LIB file might not be `tcl83.lib` exactly, unless you're using ActiveStateTcl version 8.3 like me).

At this point, you should be ready to get started.

Initializing Tcl

Just as Lua is initialized by creating a new Lua state, the Tcl library is initialized by creating a new instance of the *Tcl interpreter*. Just as you must keep track of your state in Lua, Tcl requires that you keep track of the pointer to your interpreter. To create this pointer and initialize Tcl, use the following code:

```
Tcl_Interp * pTclInterp = Tcl_CreateInterp ();
if ( ! pTclInterp )
{
    printf ( "Tcl Interpreter could not be created." );
    return 0;
}
```

As you can see, the interpreter is created with a call to `Tcl_CreateInterp ()`, which does not require any parameters. If the call fails, a `NULL` pointer will be returned.

When you're finished with the interpreter (which will usually be at the end of your program), you free the resources associated with it by calling `Tcl_DeleteInterp ()`, like so:

```
Tcl_DeleteInterp ( pTclInterp );
```

You now know how to initialize Tcl, so you can lay out your plans for your first attempt at writing a Tcl host application before trying the alien head demo. Because you should try everything at least once, the program should:

- Load an initial script that just prints random values on the screen, so you know everything's working.
- Load a second script that defines its own commands but does not execute immediately.
- Register a C function with Tcl, thereby making it accessible to the script as a command.
- Test your importing/exporting abilities by calling a user-defined Tcl command and having it call you back. You'll then call a more complicated command that requires parameters and returns a value.
- Finish up by manipulating the Tcl script's global variables, and printing the result.

Sounds like a plan, huh? Let's get to work.

Loading and Running Scripts

Just as in Lua, Tcl immediately attempts to execute scripts when they're loaded. Because most of the time, you will simply load a script once and deal with it later, the issue of code in the global scope once again becomes significant. Any code in the global scope of the script will run upon the script's loading; user-defined commands, however, will not. Therefore, any functionality written into those commands will not execute until you tell them to.

Scripts can be loaded with the `Tcl_EvalFile ()` function ("`EvalFile`" being short for Evaluate File, of course). This function accepts two parameters; a pointer to the Tcl interpreter, as well as the filename of the script to be loaded. Here's an example:

```
if ( Tcl_EvalFile ( pTclInterp, "test_0.tcl" ) == TCL_ERROR )
{
    printf ( "Error executing script." );
    return 0;
}
```

`Tcl_EvalFile ()` will return `TCL_OK` if everything went as it should've, and will return `TCL_ERROR` if the file can't be read for some reason. This can either arise due to an I/O error, or because a compile-time error occurred (yes, Tcl does perform a pre-compile step).

As stated before, any code in the script's global scope will be executed immediately. Because all you really want to do right now is make sure everything is working properly, let's write a quick little test script for just that purpose. Fortunately for us, the `puts` command is part of the Tcl core, not just the `tclsh` interpreter, which means that even scripts loaded into your program can inherently write text out to the console. In other words, you don't have to worry about exporting C functions just yet, like you did when integrating with Lua. Rather, you can get started immediately.

The script you'll load will be a simple one. It creates a few variables, performs a simple `if` block, and then prints the results. Let's save it to `test_0.tcl`, which is the file you attempted to open in the previous example snippet. Here's the code:

```
# Create some variables of varying data types
set IntVar 256
set FloatVar 3.14159
set StringVar "Tcl String"
# Test out some conditional logic
set X 0
set Logic ""
if { $X } {
   set Logic "X is true."
} else {
   set Logic "X is false."
}
# Print the variables out to make sure everything is working
puts "Random Stuff:"
puts "\tInteger: $IntVar"
puts "\t  Float: $FloatVar"
puts "\t String: \"$StringVar\""
puts "\t  Logic: $Logic"
```

Running the host application with the call to `Tcl_EvalFile ()` will produce the following output:

```
Random Stuff:
        Integer: 256
          Float: 3.14159
         String: "Tcl String"
          Logic: X is false.
```

You now know everything works. With the Tcl interpreter working properly, you can move on to a more advanced script and the concepts you'll have to master in order to implement it.

Calling Tcl Commands from C

The first advanced task will be calling a Tcl command from C. Fortunately, this is an extremely simple process, thanks to a function called `Tcl_Eval ()`. `Tcl_Eval ()` evaluates a Tcl script passed as a string, which makes it ideally suited for executing single commands from C. Here's an example:

```
Tcl_Eval ( "puts \"Hello, world!\"" );
```

This would produce the following output when run:

```
Hello, world!
```

Because you can apparently call `puts` quite easily, you should be able to call your own user-defined commands just as easily. This is how you can call specific blocks of your script at will; by wrapping these blocks in commands and using `Tcl_Eval ()` to invoke them.

As a simple example, let's create a new script file called `script_1.tcl`. Within this file you'll create a user-defined command called `PrintStuff`, whose sole purpose is to print a line of text with `puts` that tells you it's been called. You can then load this new file with `Tcl_EvalFile ()` and use `Tcl_Eval ()` to call the command. Here's the code to `PrintStuff ()`:

```
proc PrintStuff {} {
   # Print some stuff to show we're alive
   puts "\tPrintStuff was called from the host."
}
```

Remember, the `proc` command is a Tcl-core command for creating your user-defined commands (or procedures, if you want to think of them like that). Here's the code to call it:

```
Tcl_Eval ( pTclInterp, "PrintStuff" );
```

Note that `Tcl_Eval ()` requires you to pass the pointer to your interpreter as well as the command. When this program is run, the following will appear:

```
   PrintStuff was called from the host.
```

Now that you can call Tcl commands, let's see if you can get the script to call one of your functions.

Exporting C Functions as Tcl Commands

When a C function is exported to a Tcl script, it becomes a command just like anything else. This is accomplished with the `Tcl_CreateObjCommand ()` function, which allows you to expose a host application function to the specified interpreter instance with the specified name.

Defining the Function

To start the example, you're going to define a C function called `RepeatString ()` that accepts a single string and an integer count parameter. The string will be printed to the console the specified number of times. Here's the function:

```
int RepeatString ( ClientData ClientData,
                   Tcl_Interp * pTclInterp,
                   int iParamCount,
                   Tcl_Obj * const pParamList [] )
{
    printf ( "\tRepeatString was called from Tcl:\n" );

    // Read in the string parameter
    char * pstrString;
    pstrString = Tcl_GetString ( pParamList [ 1 ] );

    // Read in the integer parameter
    int iRepCount;
    Tcl_GetIntFromObj ( pTclInterp, pParamList [ 2 ], & iRepCount );

    // Print out the string repetitions
    for ( int iCurrStringRep = 0; iCurrStringRep < iRepCount;
        ++ iCurrStringRep )
        printf ( "\t\t%d: %s\n", iCurrStringRep, pstrString );
    // Set the return value to an integer
    Tcl_SetObjResult ( pTclInterp, Tcl_NewIntObj ( iRepCount ) );

    // Return the success code to Tcl
    return TCL_OK;
}
```

Everything should look more or less understandable at first, but the function's signature certainly demands some explanation. Any function exported to a Tcl interpreter is required to match this prototype:

```
int RepeatString ( ClientData ClientData,
                   Tcl_Interp * pTclInterp,
                   int iParamCount,
                   Tcl_Obj * const pParamList [] );
```

`ClientData` can be ignored; it doesn't apply to these purposes. `pTclInterp` is a pointer to the interpreter whose script called the function. `iParamCount` is the number of parameters the script passed, and is analogous to the `argc` parameter often passed to a console application's `main ()` function. Lastly, `pParamList []` is an array of `Tcl_Obj` structures, each of which contains a parameter value. The size of this array is determined by `iParamCount`.

The prototype may seem a bit intimidating at first, but think about how much help it is—an exported function will automatically know which script called it, and have easy and structured access to the parameters.

Reading the Passed Parameters

Once inside the function's definition, the next order of business will usually be reading the parameters it was passed. This is done with two functions; `Tcl_GetString ()` and `Tcl_GetIntFromObj ()`, which read string and integer parameters, respectively.

NOTE

It's important to remember that the parameter array passed from Tcl should be read relative to the first index; in other words, the first parameter is found at index one, rather than zero, the second is at index two, rather than one, and so on.

You have the parameters, so you can put them to use by implementing this simple function's logic. Using `pstrString` and `iRepCount`, the string is printed the specified number of times, with each iteration on its own line and indented by a few tabs to help it stick out.

Returning Values

Lastly, values can be returned to the script using the Tcl_SetObjResult () function. This function requires as a pointer to the Tcl interpreter in which the function's caller is executing, and a pointer to a Tcl_Obj structure. You can create this structure on the fly to return an integer value with the Tcl_NewIntObj () function:

```
Tcl_Obj * Tcl_NewIntObj ( int intValue );
```

When passed an integer value, this function creates a Tcl object structure around it and returns the pointer. If you wanted to return a string, you could use the equally simple Tcl_NewStringObj () function:

```
Tcl_Obj * Tcl_NewStringObj ( char * bytes, int length );
```

This function is passed a pointer to a character string and an integer that specifies the string's length. Again, it returns a pointer to a Tcl object based on the string value.

This completes the function, so you return TCL_OK to let the Tcl interpreter know that everything went smoothly.

Exporting the Function

As stated, your now-finished function can be called using Tcl_CreateObjCommand (), which returns NULL in the event that the command couldn't be registered for some reason:

```
if ( ! Tcl_CreateObjCommand ( pTclInterp,
                              "RepeatString",
                              RepeatString,
                              ( ClientData ) NULL,
                              NULL ) )
{
    printf ( "Command could not be registered with Tcl interpreter." );
    return 0;
}
```

The first three parameters to this function are the only ones you need to be concerned with. The first is the Tcl interpreter to which the new command should be added, so you pass pTclInterp. The next is the name of the command, as you would like it to appear to scripts. I've chosen to leave the name the same, so the string "RepeatString" is passed. Lastly, RepeatString is passed as a function pointer. Once Tcl_CreateObjCommand () is successfully called, the function is available to any script in the specified interpreter as a command.

Calling the Exported Function from Tcl

The RepeatString function exported to Tcl can be called just like any other command. Let's modify the PrintStuff command a bit to call it:

```
proc PrintStuff {} {

    # Print some stuff to show we're alive
    puts "\tPrintStuff was called from the host."

    # Call the host API command RepeatString and print out its return value
    set RepCount [ RepeatString "String repetition." 4 ]
    puts "\tString was printed $RepCount times."
}
```

Upon executing this script from within your test program, the following results are printed to the console:

```
PrintStuff was called from the host.
RepeatString was called from Tcl:
        0: String repetition.
        1: String repetition.
        2: String repetition.
        3: String repetition.
String was printed 4 times.
```

Returning Values from Tcl Commands

You have already seen how to call Tcl commands from your program, but there may come a time when you want to call a custom Tcl command and receive a return value. As a demonstration, you can create a Tcl command in script_1.tcl called GetMax. When passed two integer values, this command will return the greater value:

```
proc GetMax { X Y } {

    # Print out the command name and parameters
    puts "\tGetMax was called from the host with $X, $Y."

    # Perform the maximum check
    if { $X > $Y } {
        return $X
    } else {
        return $Y
    }
}
```

This command is called like any other, using the techniques you've already seen. As a test, let's call it with the integer values 16 and 32:

```
Tcl_Eval ( pTclInterp, "GetMax 16 32" );
```

The command will of course return 32, but how exactly will it do so? At any time, the last command's return value can be extracted from the Tcl interpreter with the `Tcl_GetObjResult ()` function. Just pass it a pointer to the proper interpreter instance, and it will return a `Tcl_Obj` structure containing the value. You can then use the same helper functions used in the `RepeatString ()` example to extract the literal value from this structure. In this case, because you want an integer, you'll use `Tcl_GetIntFromObj ()`:

```
int iMax;
Tcl_Obj * pResultObj = Tcl_GetObjResult ( pTclInterp );
Tcl_GetIntFromObj ( pTclInterp, pResultObj, & iMax );

printf ( "\tResult from call to GetMax 16 32: %d\n\n", iMax );
```

With the value now in `iMax`, you can print it and produce the following result:

```
        GetMax was called from the host with 16, 32.
        Result from call to GetMax 16 32: 32
```

Manipulating Global Tcl Variables from C

The last feature worth mentioning in the interface between the host application and Tcl is the capability to modify a script's global variables. As an example, two global definitions will be added to `script_1.tcl`:

```
set GlobalInt 256
set GlobalString "Look maw..."
```

The first step is reading these values from the script into variables defined in your program. To do this, you need to create two `Tcl_Obj` structures, which is easily done with the `Tcl_NewObj ()` helper function:

```
Tcl_Obj * pGlobalIntObj = Tcl_NewObj ();
Tcl_Obj * pGlobalStringObj = Tcl_NewObj ();
```

pGlobalIntObj and pGlobalStringObj are pointers to integer and string Tcl objects, respectively. Reading values from a Tcl script's global variables into these structures is done with the Tcl_GetVar2Ex () function, like this:

```
pGlobalIntObj = Tcl_GetVar2Ex ( pTclInterp, "GlobalInt", NULL, NULL );
pGlobalStringObj = Tcl_GetVar2Ex ( pTclInterp, "GlobalString", NULL, NULL );
```

As has been the case a few times before, the last two parameters this function accepts don't concern you. All that matters are the first two—the pTclInterp, which is of course a pointer to the Tcl interpreter within which the appropriate script resides, and the name of the global you'd like to read. You pass "GlobalInt" and "GlobalString" and the function returns the proper Tcl object structures. You've already seen how values are read from Tcl objects a number of times, so the following should make sense:

```
int iGlobalInt;
Tcl_GetIntFromObj ( pTclInterp, pGlobalIntObj, & iGlobalInt );
char * pstrGlobalString = Tcl_GetString ( pGlobalStringObj );
```

You now have the values stored locally, so you can print them to test the process thus far:

```
printf ( "\tReading global varaibles...\n\n" );
printf ( "\t\tGlobalInt: %d\n", iGlobalInt );
printf ( "\t\tGlobalString: \"%s\"\n", pstrGlobalString );
```

Running the code as it currently stands produces the following:

```
        Reading global varaibles...

                GlobalInt: 256
                GlobalString: "Look maw..."
```

You can modify a global variable with a single function call, but to make the demo a bit more interesting, you'll also read the value immediately back out after making the change. Modifying Tcl globals is done with the Tcl_SetVar2Ex () function, an obvious compainion to the Tcl_GetVar2Ex () used earlier. Here's the code for modifying your global integer, GlobalInt:

```
Tcl_SetVar2Ex ( pTclInterp, "GlobalInt", NULL, Tcl_NewIntObj ( 512 ),
    NULL );
pGlobalIntObj = Tcl_GetVar2Ex ( pTclInterp, "GlobalInt", NULL, NULL );
Tcl_GetIntFromObj ( pTclInterp, pGlobalIntObj, & iGlobalInt );
```

Only the first, second, and fourth parameters matter in the context of this example. As always, start by passing the Tcl interpeter instance you'd like to use. This is followed by the name of the global you're interested in, a `NULL` parameter, and a Tcl object structure containing the value you'd like to update the global with. In this case, you use `Tcl_NewIntObj ()` to create an on-the-fly integer object with the value of `512`. Notice that immediately following the call to `Tcl_SetVar2Ex ()` is another call to `Tcl_GetVar2Ex ()`; this is done to re-read the updated global variable.

Modifying `GlobalString` isn't much harder, and is done with the `Tcl_SetVar2Ex ()` function as well. Let's start with the code:

```
char pstrNewString [] = "...I'm using TEH INTARWEB!";
Tcl_SetVar2Ex ( pTclInterp, "GlobalString", NULL,
    Tcl_NewStringObj ( pstrNewString, strlen ( pstrNewString ) ), NULL );
pGlobalStringObj = Tcl_GetVar2Ex ( pTclInterp, "GlobalString", NULL, NULL );
pstrGlobalString = Tcl_GetString ( pGlobalStringObj );
```

You can start by creating a local, statically allocated string with the new global value in it. `Tcl_SetVar2Ex ()` is then called with the same parameters as last time, except you're now passing a string value with the help of the `Tcl_NewStringObj ()` function. Because this function requires both a string pointer and an integer length value, it made things easier to define the string locally so you could use `strlen ()` to automatically pass the length. `Tcl_GetVar2Ex ()` is also called again to retrieve the updated global's value.

At this point you've updated both globals and re-read their values, so let's print them out and make sure everything worked:

```
        Writing and re-reading global variables...

                GlobalInt: 512
                GlobalString: "...I'm using TEH INTARWEB!"
```

The new values are reflected, so you're all set!

Recoding the Alien Head Demo

You've learned everything you need to know to smoothly interface with Tcl, so let's finish the job by committing your knowledge to a third and final version of the bouncing alien head demo.

Initial Evaluations

The approach to the demo isn't any different than it was when you were using Lua; you use the majority of the core logic (actually managing and updating the alien heads, as well as drawing each new frame) and rewrite it using Tcl. This will require a host API that wraps the core functionality of the host that the script will need access to, and the body of the C-version of the demo will be almost entirely gutted and replaced with calls to Tcl.

The Host API

The host API will be the same as it was in the Lua version, but here are the prototypes of the functions anyway, for reference. Remember, of course, the strict function signature that must be followed when creating a host API for a Tcl script. Remember also that these functions will be thought of within the script as commands.

```
int HAPI_GetRandomNumber ( ClientData ClientData, Tcl_Interp * pTclInterp,
    int iParamCount, Tcl_Obj * const pParamList [] );
int HAPI_BlitBG ( ClientData ClientData, Tcl_Interp * pTclInterp,
    int iParamCount, Tcl_Obj * const pParamList [] );
int HAPI_BlitSprite ( ClientData ClientData, Tcl_Interp * pTclInterp,
    int iParamCount, Tcl_Obj * const pParamList [] );
int HAPI_BlitFrame ( ClientData ClientData, Tcl_Interp * pTclInterp,
    int iParamCount, Tcl_Obj * const pParamList [] );
int HAPI_GetTimerState ( ClientData ClientData, Tcl_Interp * pTclInterp,
    int iParamCount, Tcl_Obj * const pParamList [] );
```

How these functions work hasn't changed either; aside from the fact that new helper functions are used to read parameters and return values, the logic that drives them remains unaltered.

The New Host Application

Because the intializazation of Tcl in the demo will actually entail both the creation of a Tcl interpreter instance, as well as the exporting of your host API, I've wrapped everything in the `InitTcl ()` and `ShutDownTcl ()` functions. Here's `InitTcl ()`:

```
void InitTcl ()
{
    // Create a Tcl interpreter
    g_pTclInterp = Tcl_CreateInterp ();

    // Register the host API
    Tcl_CreateObjCommand ( g_pTclInterp, "GetRandomNumber",
        HAPI_GetRandomNumber, ( ClientData ) NULL, NULL );
    Tcl_CreateObjCommand ( g_pTclInterp, "BlitBG", HAPI_BlitBG,
         ( ClientData ) NULL, NULL );
    Tcl_CreateObjCommand ( g_pTclInterp, "BlitSprite", HAPI_BlitSprite,
         ( ClientData ) NULL, NULL );
    Tcl_CreateObjCommand ( g_pTclInterp, "BlitFrame", HAPI_BlitFrame,
         ( ClientData ) NULL, NULL );
    Tcl_CreateObjCommand ( g_pTclInterp, "GetTimerState",
         HAPI_GetTimerState, ( ClientData ) NULL, NULL );
}
```

`g_pTclInterp` is a global pointer to the Tcl interpreter, and the multiple calls to `Tcl_CreateObjCommand ()` build up the host API your script will need. Notice that I omitted the `HAPI_` prefix when exporting the host API; this was just an arbitrary decision that could've gone either way.

As always, `ShutDownTcl ()` really just redundantly wraps `Tcl_DeleteInterp ()`, but I like having orthogonal functions. :)

```
void ShutDownTcl ()
{
    // Free the Tcl interpreter
    Tcl_DeleteInterp ( g_pTclInterp );
}
```

Now that Tcl itself is under control, you only need to call the proper script functions on a regular basis and your script will run. Of course, you haven't written the script yet, but it will follow the same format the Lua version did, which should help you follow along without immediately knowing the details.

The script, which I've named `script.tcl`, is loaded and initialized first, with the following code:

```
// Load your script
if ( Tcl_EvalFile ( g_pTclInterp, "script.tcl" ) == TCL_ERROR )
    W_ExitOnError ( "Could not load script." );

// Let the script initialize the rest
Tcl_Eval ( g_pTclInterp, "Init" );
```

You call `Tcl_EvalFile ()` to load the file into memory, and immediately follow up with a call to `Tcl_Eval ()` that runs the `Init` command. At this point, the script has been loaded into memory and is initialized, so the demo can begin. From here, it's just a matter of calling the `HandleFrame` command at each frame, again by using `Tcl_Eval ()`:

```
MainLoop
{
    // Start the current loop iteration
    HandleLoop
    {
        // Let Tcl handle the frame
        Tcl_Eval ( g_pTclInterp, "HandleFrame" );

        // Check for the Escape key and exit if it's down
        if ( W_GetKeyState ( W_KEY_ESC ) )
            W_Exit ();
    }
}
```

By running this command once per frame, the aliens will move around and be redrawn consistently. This wraps up the host application, so let's finish up by taking a look at the scripts that implement these two commands.

The Tcl Script

The structure of the Tcl script is purposely identical to that of the Lua version covered earlier in the chapter. I did this to help emphasize the natural similarities among scripting languages; often, a game scripted with at least the basic functionality of one language can be ported to another scripting language with minimal hassle.

As was the case in Lua, Tcl doesn't support constants. You can simulate them instead with global variables named using the traditional constant-naming convention:

```
set ALIEN_COUNT       12;          # Number of aliens onscreen

set MIN_VEL           2;           # Minimum velocity
set MAX_VEL           8;           # Maximum velocity

set ALIEN_WIDTH       128;         # Width of the alien sprite
set ALIEN_HEIGHT      128;         # Height of the alien sprite
set HALF_ALIEN_WIDTH  [ expr $ALIEN_WIDTH / 2 ];  # Half of the sprite
                                                  # width
set HALF_ALIEN_HEIGHT [ expr $ALIEN_HEIGHT / 2 ]; # Half of the sprite
                                                  # height

set ALIEN_FRAME_COUNT 32;          # Number of frames in the animation
set ALIEN_MAX_FRAME   [ expr $ALIEN_FRAME_COUNT - 1 ];  # Maximum valid
                                                        # frame
```

```
set ANIM_TIMER_INDEX  0;         # Animation timer index
set MOVE_TIMER_INDEX  1;         # Movement timer index
```

You also need two globals: an array to hold the alien heads, and a counter to track the current frame of the animation. Remember, Tcl's lack of multidimensionality can be easily sidestepped by cleverly naming indexes, so don't worry about the necessary dimensions in the declaration:

```
set Aliens() 0;                  # Sprites
set CurrAnimFrame 0;             # Current frame in the alien animation
```

Now onto the functions. As you saw in the Tcl version of the demo's host application, you need to define two new commands: `Init` and `HandleFrame`. Let's start with `Init`, which is called once when the demo starts up and is in charge of initializing the script.

```
    # Initializes the demo

    proc Init {} {

        # Import the constants we'll need
        global ALIEN_COUNT;
        global ALIEN_WIDTH;
        global ALIEN_HEIGHT;
        global MIN_VEL;
        global MAX_VEL;

        # Import the alien array
        global Aliens;

        # Initialize the alien sprites

        # Loop through each alien in the table and initialize it
        for { set CurrAlienIndex 0; } { $CurrAlienIndex < $ALIEN_COUNT }
            { incr CurrAlienIndex; } {

            # Set the X, Y location
            set Aliens($CurrAlienIndex,X)
                [ GetRandomNumber 0 [ expr 639 - $ALIEN_WIDTH ] ];
            set Aliens($CurrAlienIndex,Y)
                [ GetRandomNumber 0 [ expr 479 - $ALIEN_HEIGHT ] ];
            # Set the X, Y velocity
            set Aliens($CurrAlienIndex,XVel)
                [ GetRandomNumber $MIN_VEL $MAX_VEL ];
            set Aliens($CurrAlienIndex,YVel)
                [ GetRandomNumber $MIN_VEL $MAX_VEL ];

            # Set the spin direction
            set Aliens($CurrAlienIndex,SpinDir) [ GetRandomNumber 0 2 ];
        }
    }
```

Remember that your "constants" are actually just typical globals, which need to be imported into the command's local scope with the `global` command. You also need to import the `Aliens` array, a real global. The command then loops through each alien in the array and sets its fields. Notice, however, that the "fields" are actually just cleverly named indexes; what you're dealing with is a purely one-dimensional array that actually feels two-dimensional. Because you can use the comma in your index names, you can trick the syntax into appearing as if you're working with multiple dimensions. The host API command `GetRandomNumber` is used to fill all of the values—the X, Y location, X, Y velocity, and the spin direction.

The next and final command is `HandleFrame`, which is called once per frame and is responsible for moving the aliens around, handling their collisions with the side of the screen, and drawing and blitting the next frame:

```
    # Creates and blits the next frame of the demo
        proc HandleFrame {} {

        # Import the constants we'll need
        global ALIEN_COUNT;
        global ANIM_TIMER_INDEX;
        global MOVE_TIMER_INDEX;
        global ALIEN_FRAME_COUNT;
        global ALIEN_MAX_FRAME;
        global HALF_ALIEN_WIDTH;
        global HALF_ALIEN_HEIGHT

        # Import your globals
        global Aliens;
        global CurrAnimFrame;
```

```
# Blit the background image
BlitBG;

# Increment the current frame in the animation
if { [ GetTimerState $ANIM_TIMER_INDEX ] == 1 } {
    incr CurrAnimFrame;
    if { $CurrAnimFrame >= $ALIEN_FRAME_COUNT } {
        set CurrAnimFrame 0;
    }
}

# Blit each sprite
for { set CurrAlienIndex 0; } { $CurrAlienIndex < $ALIEN_COUNT }
    { incr CurrAlienIndex; } {

    # Get the X, Y location
    set X $Aliens($CurrAlienIndex,X);
    set Y $Aliens($CurrAlienIndex,Y);

    # Get the spin direction and determine the final frame for this
    # sprite based on it.
    set SpinDir $Aliens($CurrAlienIndex,SpinDir);
    if { $SpinDir == 1 } {
        set FinalAnimFrame
            [ expr $ALIEN_MAX_FRAME - $CurrAnimFrame ];
    } else {
        set FinalAnimFrame $CurrAnimFrame;
    }

    # Blit the sprite
    BlitSprite $FinalAnimFrame $X $Y;
}

# Blit the completed frame to the screen
BlitFrame;

# Move the sprites along their paths
if { [ GetTimerState $MOVE_TIMER_INDEX ] == 1 } {

    for { set CurrAlienIndex 0; } { $CurrAlienIndex < $ALIEN_COUNT }
        { incr CurrAlienIndex; } {
                # Get the X, Y location
                set X $Aliens($CurrAlienIndex,X);
                set Y $Aliens($CurrAlienIndex,Y);

                # Get the X, Y velocities
                set XVel $Aliens($CurrAlienIndex,XVel);
                set YVel $Aliens($CurrAlienIndex,YVel);

                # Increment the paths of the aliens
                incr X $XVel
                incr Y $YVel
                set Aliens($CurrAlienIndex,X) $X
                set Aliens($CurrAlienIndex,Y) $Y

                # Check for wall collisions
                if { $X > 640 - $HALF_ALIEN_WIDTH ||
                     $X < -$HALF_ALIEN_WIDTH } {
                    set XVel [ expr -$XVel ];
                }
                if { $Y > 480 - $HALF_ALIEN_HEIGHT ||
                     $Y < -$HALF_ALIEN_HEIGHT } {
                    set YVel [ expr -$YVel ];
                }
                set Aliens($CurrAlienIndex,XVel) $XVel
                set Aliens($CurrAlienIndex,YVel) $YVel
            }
        }
    }
```

This command does just what it did in the Lua and C versions of the demo. It increments the animation frame, draws each alien to the screen, moves each sprite and handles its collision with the wall, and blits the results to the screen. There's also nothing new here in terms of Tcl—everything this command does has been covered elsewhere in the chapter. Remember of course, the typical quirks— "constants" and globals must be imported into the command's scope before use with the `global` keyword, and array indexes that appear to be multidimensional are actually just single-dimensional keys that happen to contain a comma.

That's everything so check out the demo! You can find this and all other Chapter 6 programs in `Programs/Chapter 6/`.

Advanced Topics

As usual, I couldn't possibly fit a full description of the language here, so there's still plenty to learn if you're interested. Here are some of the semi-advanced to advanced topics to consider pursuing as you expand your knowledge of Tcl:

- **Tk.** Naturally, Tk is logical next step now that you've attained familiarity and comfort with the Tcl language. Tk may not be game-related enough to make it into the book, but most games need GUIs and some form of setup programs, and the Tk windowing toolkit is a great way to rapidly develop such interfaces. Tcl/Tk is also a great way to rapidly and easily develop fully graphical utilities like map editors and file-format converters.
- **Extensions.** Along with Tk, Tcl supports a wide range of useful extensions that provide countless new commands for everything from an HTTP interface to OggVorbis audio playback. As you can imagine, there's quite a bit of power to be drawn from these extensions, much of which you might find useful in the context of game development and scripting.
- **Lists.** I've covered Tcl's associative array, but the *list* is another aggregate data type supported by the language that is worth your time. Although it would've proved awkward to use in this demo and is often considered inefficient for large datasets, understanding Tcl lists is a valuable skill.
- **Exception Handling.** Tcl provides a robust error-handling system that resembles the exception mechanisms of languages such as C++ and Java. An understanding of how it works can lead to more stable and cleanly designed scripts.
- **String Pattern Matching with Regular Expressions.** Like other languages such as Perl, Tcl is equipped with a powerful set of string searching and pattern matching tools based on regular expressions. Anyone who's using Tcl for text-heavy applications should take the time to learn how these commands work.

Web Links

Tcl has been around for quite some time and has amassed a formidable following. Check out these Web links to continue your exploration of the Tcl system and community:

- **Tcl Developer Xchange:** `http://www.scriptics.com/`. A good place to get started with Tcl/Tk, and a frequently updated source of news and event information regarding the language and its community.

- **ActiveState:** `http://www.activestate.com/.` Makers of the ActiveStateTcl distribution used throughout this chapter.
- **The Tcl'ers Wiki:** `http://mini.net/tcl/.` A collaboratively edited Web site dedicated to Tcl and its user community. Good source of reference material, discussions, and projects.

WHICH SCRIPTING SYSTEM SHOULD YOU USE?

You've learned quite a bit about these three scripting systems in this chapter, but the real question is which one you should use, right? Well, as I'm sure you'd expect, there's no right or wrong answer to this question. The fact that I chose these particular languages to demonstrate in the first place should tell you that any of them would make a good choice, so you shouldn't have to worry too much about a bad decision. Furthermore, because you now understand both the details of each of the three systems' languages, as well as how to use their associated libraries and runtime environments, you'll be the best judge of what they can offer to your specific game project.

I explained three scripting systems in this chapter for a number of reasons. First of all, anyone who has intentions of designing his other own scripting system, as you certainly do, should obviously be as familiar as possible with what's out there. Chances are, Mercedes wouldn't make a particularly great car if they didn't spend a significant amount of time studying their competition. The more you know about how languages like Lua, Python, and Tcl are organized, the more insight and understanding you'll be able to leverage when designing one of your own.

Secondly, I wanted it to be as clear as possible to you that from one scripting system to the next, certain things change wildly (namely, language syntax and the general features that language supports), whereas others stay remarkably the same (such as the basic layout of a runtime environment or the utilities a distribution comes with). On the one hand, you'll need to know which parts of your scripting system should be designed with tradition and convention in mind, but it also helps to know where you're free to go nuts and do your own thing. You don't want to create a mangled train wreck of a scripting language that does everything in a wildly unorthodox way, but you certainly want to exercise your creativity as well.

Lastly, even though the point of this book is to build a scripting system of your own, there will always be reasons why using an existing solution is either as good a decision, or a smarter one. Here are a few:

- **Ease of development.** Building a scripting system is *hard* work, and lots of it. Creating a game is a lot of hard work as well. Put these two projects together and you have double the amount of long, difficult work ahead of you. Using an existing scripting package can make things quite a bit easier, and that means you'll have more energy to spend on making your game as good as it can be. Besides, that's what's really important anyway.

- **Speed of development.** Aside from difficulty, building a scripting system from scratch takes a long time. If you find yourself working on a commercial project for an established game company, or just don't want to spend two years from start to finish on a personal project, you may find that there simply aren't enough hours in the day to do both. Because game development is always the highest priority, the design and creation of a custom scripting language may have to be sacrificed in the name of actually getting something done.
- **Quality assurance.** Scripting systems are extremely complex pieces of software, and if there's one thing software engineers know, it's that bugs and complexity go hand in hand. The more code you have to deal with, the more potential there is for large and small bugs alike to run rampant. It's hard enough to get a 3D engine to work right; you shouldn't have to battle with your scripting system's stability issues at the same time.
- **Features.** Making your own scripting system is a lot of fun, and a great learning experience, but how long is it going to take to make something that can compete with what's already out there? How long will you spend adding object-orientation, garbage collection, and exceptions? Sometimes, one of the existing solutions might just be plain *better* than your own version.

Of course, I don't mean to sound too negative here. To be fair, I should mention that there are just as many reasons that you *should* design your own scripting system, or at least know how to do so. Here are a few:

- **Exiting solutions are overkill.** The last reason I mentioned to use someone else's scripting language is that it may simply boast more features than you're prepared to match. Of course, this can also be its downfall, because a bloated feature set may completely overshadow its utility value. You may not *need* objects, exceptions, and other high-level language features, and may just want a small, easy-to-use custom language. In these cases, creating an intentionally modest scripting system of your own design may be just what the project needes.
- **Existing languages are generic by design.** Tcl in particular, for example, was designed from the ground up to be as generic as possible, so it could be directly applied to a wide range of domains. Everyone from game programmers to robot designers to Web application developers can find a use for Tcl. But if you need a language designed entirely to control a specific aspect of your own game, you may have no choice but to do it yourself. For example, if you're writing a game that involves a huge amount of natural language processing, you may not really care much about mathematical functions and just want a string-heavy language with built-in parsing and analysis routines.

- **No one knows your game better than you.** Optimization and freedom of creativity are two things that are always on the minds of game developers. You may find that the only way to get a scripting language small enough, fast enough, or specific enough for your game is to build it yourself. To put it simply, scripting languages are sometimes better off when they're custom-tailored to one project or group of similar projects.

To sum things up, even an existing scripting system is not something to take lightly. Scripting has a huge impact on games and game engines, so make sure you weigh all of the pros and cons involved in the situation. It's difficult to make a decision when so many conflicting interests are involved, ranging from practicality and development time to creative freedom and feature sets, but it's a necessary evil. Good games and engines are characterized by the smart decisions made by their creators.

Scripting an Actual Game

Oh right… one last thing. Sure, you made the bouncing alien head demo work in four languages (C, Lua, Python, and Tcl), but you certainly couldn't call that a game. Game scripting is a complicated thing, and simply being able to load and run scripts isn't enough. A great deal of thought must go into the design and layout of your scripting strategy, in terms of how and where exactly scripting will be applied, what game entities need to be scripted and when, in addition to countless other issues.

On the other hand, you have learned quite a bit so far. You do know how to physically invoke and interface with a scripting system, you know how to load scripts for later use and assign them to specific events (in this case, assigning them to run at each frame of the main loop), and you have a good idea of what each system and language can do. You should probably be able to determine how this information is then applied to at least a small or mid-level game on your own.

Of course, this wouldn't be much of a book if that were my final word on the subject. You'll ultimately finish things up with a look at how scripting techniques are applied to a real game with real issues. The beauty is that when that time comes, you'll be able to use any language you want to do the job—including the one you'll develop—because the principals of game scripting are generally language-independent.

Summary

Well *that* was one heck of a chapter, huh? You came in naïve and headstrong, and you've come out one step closer to attaining scripting mastery. You now have the theoretical knowledge and practical experience necessary to do real game scripting in Lua, Python, and Tcl—not too shabby, huh? Along the way, you've learned a lot about how these three scripting systems work, which means you'll be much better prepared for the coming chapters, in which you design your own scripting language.

On the CD

We built three major projects throughout the course of this chapter by recoding the original bouncing alien head demo in three different scripting languages. All code relating to the chapter can be found in `Programs/Chapter 6/` on the accompanying CD.

- **Lua**/ Contains the demos for the Lua scripting language.
- **Python**/ Contains the demos for the Python scripting language.
- **Tcl**/ Contains the demos for the Tcl scripting language.

CHAPTER 7

Designing a Procedural Scripting Language

"It's a Cosby sweater. A COSBY SWEATAH!!!"
——Barry, High Fidelity

Now that you've learned how scripting systems are generally laid out, and even gained some hands-on experience with a few of the existing solutions, you're finally on the verge of getting started with the design and construction of your own scripting engine.

As you've learned, the high-level language is quite possibly the most important—or more specifically, the most *pivotal*—element in the entire system. The reason for this is simple; because it provides the human readable, high-level interface, it's the primary reason you're embarking on this project in the first place. Equally important is the fact that the underlying elements of the system, such as the low-level language and virtual machine, can be better designed in their own right when the high-level language they'll ultimately be accommodating is taken into account. This is analogous to the foundation for a building. The foundation under a house will support houses and other small, house-like buildings, but will hardly support skyscrapers or blimp hangars.

For these reasons and more, your first step is to design the language you're going to build the system around. As I've alluded to frequently in the chapters leading up to this point, the ultimate goal will be a high-level language that resembles commonly used existing languages like C, C++, Java, and so on. This is beneficial as it saves you the trouble of "switching gears" when you go from working on engine code written in C to script code, for example. More generally, though, C-style languages have been refined and tweaked for decades now, so they're definitely trusted syntaxes and layouts that you can safely capitalize on to help you design a good language that will be appropriate for game scripting. It's not always necessary to reinvent the wheel, and you should keep this in mind over the course of the chapter.

The point to all this is that you need to be sure about what you're doing here. A badly or hastily designed language will have negative and long-lasting repercussions, and will hamper your progress later. Like I said, you'll be much better prepared when designing other aspects of your scripting system when the language itself has been sorted out, so the information presented in this chapter is important.

In this chapter, we're going to:

- Learn about the different types of languages we can base our scripting system around.
- See how the necessity of a high-level language manifests itself, and watch its step-by-step evolution.
- Define the XtremeScript language and discuss its design goals.

NOTE

Sun's Java Virtual Machine (JVM) can technically support any number of languages, as long as they're compiled down to JVM bytecode. However, because the system was designed primarily for Java, that's the language that "fits" best with it and can best take advantage of its facilities. This should be your aim with XtremeScript as well; a language and runtime environment designed with each other in mind.

General Types of Languages

Programming languages, like people, for example, come in a wide variety of shapes and sizes. Also like people, certain languages are better at doing certain things than others. Some languages have broad and far-reaching applications, and seem to do pretty much everything well. Other languages are narrow and focused, being applicable to only a small handful of situations, but are totally unmatched in those particular fields. The area in which a given language is primarily intended for use is called its *domain*.

The beauty of a project like the scripting system you're about to begin building is that it gives you a chance to create your *own* language—something I'm sure every decent programmer has fantasized about once or twice. If you've ever found yourself wishing your language of choice could do this or that, your day has finally come! We're going to outline a language of our own design from the ground up, so it'll naturally be our job to decide exactly what its features are.

To start things off, you're going to have a look at a few basic models for scripting languages. As you move from one to the next, I'll note the increasing level of complexity that each one presents. Although none of the following language styles are "right" or "wrong" in general, it's obvious that certain games require more power and precision than others. Remember that the scripting requirements of a Pac-Man clone will probably differ considerably from that of a first person shooter.

Assembly-Style Languages

The first type of language we're going to cover is what I like to call "assembly-style" languages, so named because they're designed after native assembly languages, such as Intel 80X86. As was briefly covered in the first chapter, assembly languages work on the principal of *instructions* and *operands*. Instructions, just like the ones currently running on the computer I'm writing this book with, are executed sequentially (one at a time) by the virtual machine. Each instruction specifies a small, simple operation like moving the value of a variable around or performing arithmetic. Operands further describe instructions; like the parameters of a function, they tell the virtual machine exactly which data or values the instruction should operate on.

Let's start with an example. Say you're writing a script that maintains three variables: X, Y, and Z. Just to test this language, all you're going to do is move these variables' values around and perform some basic arithmetic. A script that does these things might look like this:

```
Move        X, 16
Move        Y, 32
Move        Z, 64
Add         Y, Z
Sub         Y, X
Move        X, Y
```

You can start off with a Move instruction, which "moves" the value of 16 into X. This is analogous to the assignment operator in most programming languages. In other words, the first line of code in the previous example is equivalent to this in C:

```
X = 16;
```

Get it? This first instruction in the script is followed by two more Moves; the first to assign 32 to Y, and the second to assign 64 to Z. Once the three variables are initialized, you can add Y and Z together with (surprise) the Add instruction, and then subtract (Sub) X from Y. The results of both of these instructions are placed into Y, so they're equivalent to the following lines in C:

```
Y += Z;
Y -= X;
```

Lastly, you can move the value of Y into X with a final Move instruction, which wraps everything up.

Assembly-style languages are good primarily because they're so easy to compile. Despite the obvious simplicity of the example you just looked at, assembly-style languages generally don't get much more complicated than that, and believe it or not, just about anything you can do in C can be done with a language like this. As you've already seen, assignment of values to variables, as well as arithmetic, is easy using the instruction/operand paradigm. To flesh out the language, you'd add some additional math instructions, for things like subtraction, multiplication, division, and so on. You might be wondering, however, how conditional logic and looping is handled. The answer to this is almost as simple as what you've seen so far. Both loops and branching are facilitated with line labels and *jump* instructions. Line labels, just like the ones you're allegedly not supposed to use in C, mark a specific instruction for later reference. Jump instructions are used to route the flow of the program, to change the otherwise purely sequential execution of instructions.

This makes endless loops very easy to code. Consider the following:

```
    Move       X, 0
Label:
    Add        X, 1
    Jump       Label
```

This simple code snippet will set a variable called `X` to zero, and then increment it infinitely. As soon as the virtual machine hits the `Jump` instruction, it will jump back to the instruction immediately following `Label`, which just happens to be `Add`. The jump will then be encountered again, and the process will repeat indefinitely. To help guide this otherwise mischievous block of code, you're going to need the ability to compare certain values to other values, and use the result of that comparison as the criteria for whether to make the jump. This is how the familiar `if` construct works in C, the only difference being that you're doing everything manually. A more refined attempt at the previous loop might look like this:

```
    Move       X, 0
Label:
    Add        X, 1
    JL         X, 10, Label
```

You'll notice that `Jump` has become `JL`. `JL` is an acronym for "**J**ump if **L**ess than." The instruction also works with three operands now, as opposed to the single one that `Jump` used. The first two are the operands for the comparison. Basically, you compare `X` to 10, and if it's *less than*, you jump back to `Label`, which is the start of the loop, and increment it again. As you can see, the loop will now politely stop when `X` reaches the desired value (10, in this case). This is just like the `while` loop in C, so the previous code could be rewritten in C like this:

```
X = 0;
while ( X < 10 )
{
    ++ X;
}
```

You should now begin to understand why it is that assembly-style languages, despite their apparent simplicity, can be used to do just about anything C can do. What you should also begin to notice, however, is that it takes quite a bit more work to do the simple things that C usually lets you take for granted. For this reason, assembly-style languages are simply too low-level for the sort of scripting system we want to create. Besides, as you learned in Chapter 5, the script compiler is going to convert a higher-level language down to an assembly language like this anyway. You have to build an assembly language no matter what, so you might as well focus your real efforts on the high-level language that will sit on top of it. As I mentioned previously, however, the one real advantage to a language like this is that it's really quite easy to compile. As you can probably imagine, code that looks like this:

```
X = Y * Z + ( Q / 10.5 ) + P - 2
```

Is considerably harder for a compiler to parse and understand than something simpler (albeit longer) like this:

```
Mov         X, Y
Mul         Y, Z
Div         Q, 10.5
Add         Y, Q
Sub         P, 2
Add         Y, P
```

If this sort of language still interests you, however, don't worry. Starting in the next chapter, you're going to design and implement an assembly language of your own, as well as its respective assembler, which will come in quite handy later on in the development of your scripting system. Until then, however, you can use it by itself to do the kind of low-level scripting seen here. So, you're going to learn exactly how this sort of language works either way.

In a nutshell, here are the pros and cons of building a scripting system around a language like this.

Pros:

- Very simple to compile.
- Relatively easy to use for basic stuff, due to its simplistic and fine-grained syntax.

Cons:

- Low-level syntax forces you to think in terms of small, single instructions. Complex expressions and conditional operations become tedious to code when you can't describe them with the high-level constructs of a language like C.

Upping the Ante

One of the biggest problems with the sort of language discussed previously is its lack of flexibility. The programmer is forced to reduce high-level things like complex arithmetic and Boolean expressions to a series of individual instructions, which is counter-intuitive and tedious at times. Most people don't mind having to do this when writing pure assembly language, as the speed-boost and reduced footprint certainly make it worthwhile. But having to do the same to script a game is just silly, at least from the perspective of the script coder. Scripts are usually slow compared to true, compiled machine code whether they're in the form of an assembly-style language or a higher level language, so you might as well make them easier to use.

NOTE

Technically, a script written purely in the virtual machine's assembly language would run *somewhat* faster than one compiled by a script compiler, but the speed difference would be negligible and pretty much cancel out the effort spent on it.

The first thing to add, then, is support for more complex expressions. This in itself is a rather large step. Code that can properly recognize and translate an expression like this:

```
Mov         X, Y * Q / ( Z + X ^ 2 ) + 3.14159 % 256
```

is definitely more complicated to write than code that can understand the same expression after the coder has gone to the trouble of reducing it to its constituent instructions.

You can't really add expressions alone, though; a few existing language constructs need to change along with their addition in order to truly exploit the power of this new feature. For example, conditional expressions are currently evaluated in a manner much like the way arithmetic is handled. Only two operands can be compared at once, causing a jump to a location elsewhere in the script if the comparison evaluates to `true`. This means that even with support for full expressions, you can still only compare two things at once. To change this, you could simply alter the jump instructions to accept two operands instead of four. In other words, instead of the *jump if less than or equal* instruction (for example) looking like this:

```
JLE         X, Y, Label
```

This code jumps to `Label` if `X` is less than or equal to `Y`. You could simply reduce all jump instructions to a single, all-purpose conditional jump that looks like this:

```
Jmp         Expression, Label
```

Now you can do things like this:

```
Jmp         X > Y && Y * 2 < Z, MyLabel
```

Which makes everything much more convenient. However, as long as you're going this far, you might as well cut to the chase and create the familiar `if` statement we're all used to. Take the following random block of code for instance:

```
    Jmp         X > Y && Z < Q, TrueBlock
FalseBlock:
    ; Handle false condition here
    Mov         Z, X
    Sub         Q, Y
    Jmp         SkipTrueBlock
```

```
TrueBlock:
    ; Handle true condition here
    Add        X, Y
    Mul        Z, 2
    Mov        Y, Z
SkipTrueBlock:
```

It works, and it works much better thanks to the ability to code Boolean expressions directly into scripts, but it's a bit backwards, and it's still too low level. First of all, you still have to use labels and jumps to route the flow of execution depending on the outcome of the comparison. In languages like C, you can use *code blocks* to group the `true` and `false` condition handling blocks, which are much cleaner. Second, the general layout of assembly-style languages forces you to put the `false` condition block above the `true` block, unless you want to invert all of your Boolean expressions. This is totally backwards from what you're probably used to, so it's yet another example of the counter-intuitive nature of this style of language. You can kill two birds with one language enhancement by adding support for the `if` construct. The block of code you saw previously can now be rewritten like this:

```
if ( X > Y && Z < Q )
{
    ; Handle true condition here
    Add        X, Y
    Mul        Z, 2
    Mov        Y, Z
}
else
{
    ; Handle false condition here
    Mov        Z, X
    Sub        Q, Y
}
```

Again, much nicer, eh? It's so much nicer, in fact, that you should probably do the same thing for loops. Currently, loops are handled with the same jump instruction set you were using to emulate the `if` construct before you added it. For example, consider this code block, which initializes a variable called `X` to zero, and then increments it as long as it's less than `Y`:

```
    Mov        X, 0
LoopStart:
    Inc        X
    Jmp        X < Y, LoopStart
```

Looks like the same problem, huh? You're being forced to emulate the nice, tidy organization of code blocks with labels and jumps, and the expression that you evaluate at each iteration of the loop to determine whether you should keep going is below the loop body, which is backwards from the `while` loop in C. Once again, these are things that the language should be doing for you. Adding a `while` construct of your own lets you rewrite the previous code in a much more elegant fashion:

```
Mov         X, 0
while ( X < Y )
{
      Inc         X
}
```

Now that you've got a language that supports `if` and `while`, along with the complex type of expressions that these constructs demand, you've taken some major steps towards designing a C-style language, and have seen its direct advantages over the more primitive, albeit easier to compile, assembly-style languages. In fact, you're actually almost there; one thing I haven't mentioned until now is that "instructions" as we know them are virtually useless at this point. There's no need for the `Mov` instruction, as well as its similar arithmetic instructions, now that you have expression support. I mean, why go to the trouble of writing this:

```
Mov         X, Y + Z * Q
```

When you can just write this:

```
X = Y + Z * Q;
```

The latter approach certainly looks more natural from the perspective of a C programmer. And because `if` and `while` have replaced the need for the `Jmp` instructions and the line labels it works with, you no longer need them either. So what are you left with? A language that looks like this:

```
X = Y;
if ( X < Z )
    X = Z;
else
    Z = X;
while ( Z < Q * 2 )
{
    Z = Z + X;
    X = X - 1;
}
```

Which is C, more or less. Granted, you still don't know how to actually code a compiler capable of handling this, but you've learned first-hand why these language constructs are necessary, working your way up from what is virtually the simplest type of language you could implement. Now that you know exactly why you should aim for a language like this, let's have a look at some of the more complex language features.

Functions

What if you wanted to add trigonometry to your expressions? In other words, what if you wanted to do something like this:

```
Theta = 180;
X = Cos ( Theta ) / Sin ( Theta );
```

NOTE

As you should certainly know, *functions* basically take simple code blocks to the next level by assigning them names and allowing them to be jumped to from anywhere in the program, as well as giving them the ability to receive parameters. The process of jumping to a function based on its name is called a *function call*, and is really the high-level evolution of the jump instructions and line labels from the early version of your language.

You could hardcode the trig functions directly into your compiler, so that it replaces `Cos ( X )` and `Sin ( X )` with a specialized instruction for evaluating cosines, but a better approach is to simply allow scripts to define their own functions.

Functions open up possibilities for a whole new style of programming by introducing the concept of *scope*. This language as it stands forces every line of code to reside in the same scope. In other words, every variable defined in the script is available everywhere else. When code is broken into functions, however, scripts take on a much more hierarchical form and allow data to be fenced off and exclusively accessible in its own particular area. Variables defined in a function are available only within that function, and therefore, the function's code *and* data is properly encapsulated. See Figure 7.1.

Recursion also becomes possible with functions. *Recursion* is a form of problem-solving that involves defining the problem in terms of itself. Recursive algorithms are usually implemented in C (or C-style languages, as your language is quickly becoming) by defining a function that calls itself. Take the following block of code for instance:

```
function Fibonacci ( X )
{
     if ( X <= 1 )
       return X;
     else
       return Fibonacci ( X - 1 ) + Fibonacci ( X - 2 );
}
```

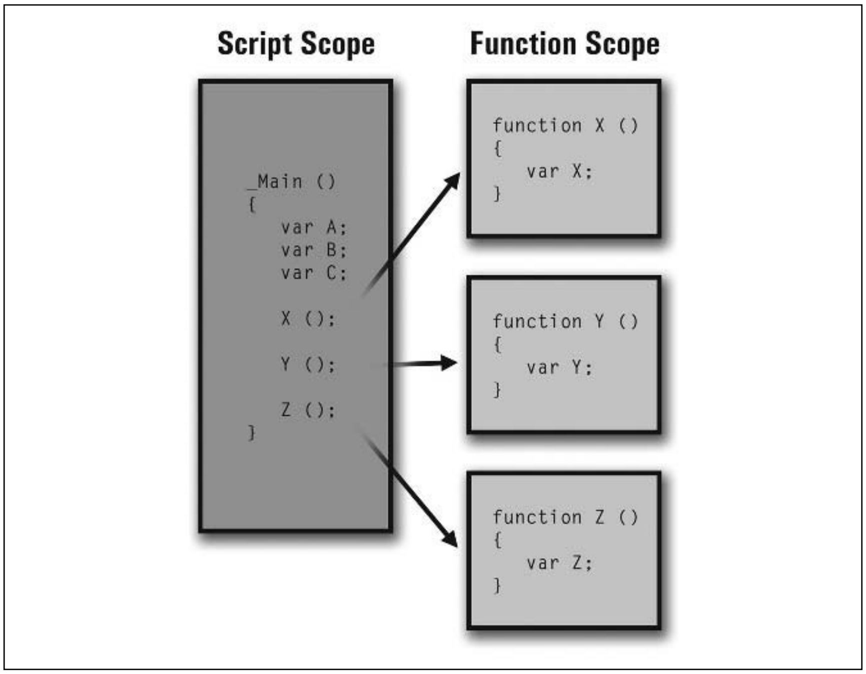


Figure 7.1

Functions create a two-level scope hierarchy: script scope and function scope.

This function of course computes the *Fibonacci Sequence*, a sequence defined such that each element *X* is defined as the sum of the previous two elements (in other words, *X - 1* and *X - 2*). The Fibonacci Sequence is a common example of basic recursive algorithms. For example, here are the first few terms from the sequence:

```
1,1,2,3,5,8,13,…
```

In general, functions change the way you code because they allow you to break scripts into specialized blocks of codes that work with one another via function calls. Functions promote code reuse, because you can write code once, assign it a logical name of some sort, and refer to it as many times as you want simply by using its name.

This also opens up the possibility of creating a *standard library* of functions that are commonly used among all scripts. For example, if you're scripting a game that employs a complex algorithm for leveling-up

TIP

While it's true that script-defined functions are vital, there are definite advantages to writing functions in C that the script can call by name. This allows functions to be written that run much faster than script-defined functions, and are capable of lower-level or more specialized tasks. Of course, a far more flexible method is simply defining C functions in the host API. We'll talk about this later on.

players, you may want to write that algorithm once in a function, and then call that function whenever you need to level-up a player from any subsequent scripts. C programmers are certainly familiar with the concept of a standard library, so you should be able to imagine the possibilities as they would relate to games, once a game project gets complicated enough. Figure 7.2 illustrates this concept.

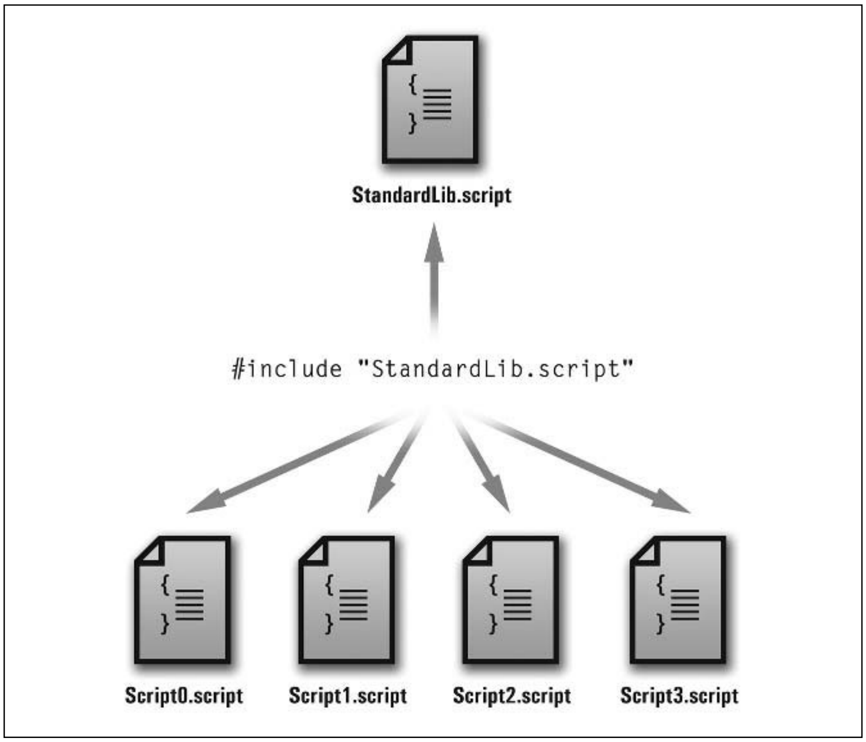


Figure 7.2

Using a standard library.

In general, functions (which are also known as *procedures*) turn your language into a *procedural language*, meaning a language whose programs are defined largely as collections of interrelated functions as opposed to a single, flat block of code. Languages like C and Pascal are procedural languages, so you should understand why you're aiming for something along those lines. They're easy to use and well accepted languages that are well suited to scripting a wide variety of games with plenty of flexibility and power.

Object-Oriented Programming

To round out this discussion, let's take a look at *object-oriented programming*, or *OOP*. As you may know, *objects* take the concept of functions a step further by merging code and data into a single structure. Generally speaking, objects manage both a set of data known as the *properties* that describe a given entity (such as an enemy in your game) as well as a group of functions known as *methods* that operate specifically on that data and implement the entity's behavior and functionality (see Figure 7.3).

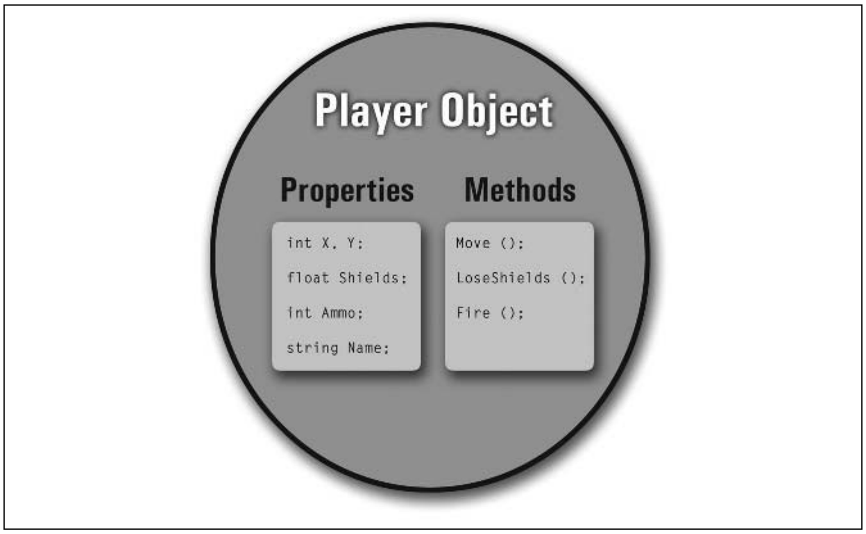


Figure 7.3

Objects combine data and code into single entities.

Object-oriented programs are very different from their procedural cousins. Rather than being a collection of functions that call and return values to each other, object-oriented programs are collections of objects, and can therefore be thought of as systems of interconnected entities that can communicate with each other by sending *messages*, as illustrated in Figure 7.4. Messaging in OOP-terminology really just refers to the process of one object calling the function of another object to get it to do something or return some value. In this regard, OOP programs are still somewhat procedural, but the real focus of these programs is that they're simply collections of nearly autonomous entities that fully define their own data and behavior.

An OOP program at runtime is very similar in a lot of ways to the real world, in the sense that it's composed of an underlying environment and a "population" of entities that live, function, and die within that environment. But I'm getting too philosophical; let's get back on track.

To bring this all back to the topic of game scripting, let's talk about how objects can be used to better control a game. If you think about it, objects are really a natural part of game programming as well as scripting. After all, games are also usually composed of an environment (the level, arena, game world, or whatever) that is inhabited by a number of autonomous entities that interact with each other (such as the player, enemies, power ups, weapons, and so on). With this in mind, an OOP-based scripting language seems almost ideal, because scripts can literally map the

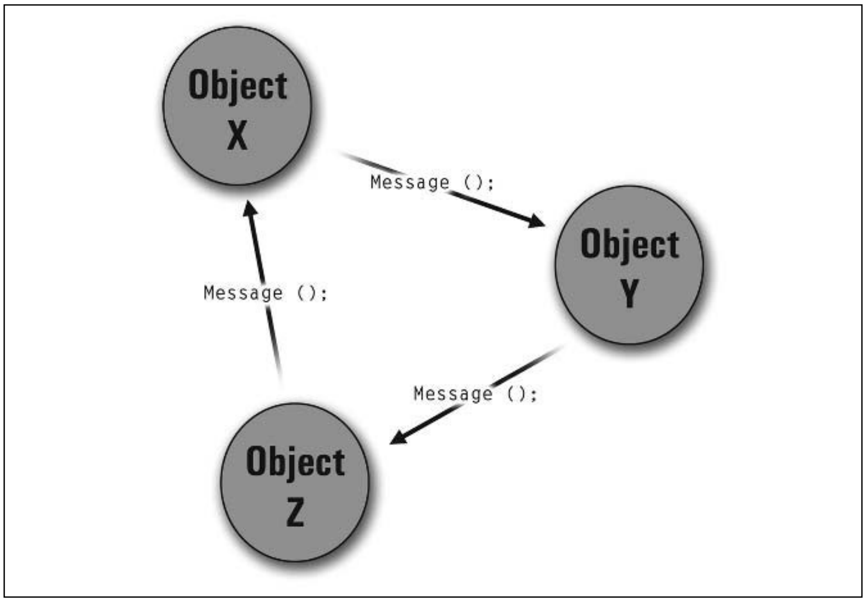


Figure 7.4

Objects communicate with each other via messages at runtime.

entities in a game to physical objects and therefore control them and their behavior in a very intuitive and lifelike manner. For example, UnrealScript, the scripting language used for the Unreal series of games, is based entirely around this concept.

However, OOP-based languages are not only far more complex to design than their procedural counterparts, they're also much more difficult to implement both in terms of compilers and run-time environments. I'll be passing on the OOP paradigm in this book, focusing instead on a purely procedural language. Don't worry about it, however; procedural code is still extremely powerful and can even emulate the functionality of OOP languages to varying degrees. As you'll

NOTE

It's a common misconception that object-oriented programming is just a matter of grouping code and functions together, when it is in fact *much* more. OOP brings with it not just the basic structure of objects, but also an endless collection of complex *design patterns*, which are basically ways to model common problems with objects in highly structured ways. If you were going to take the OOP route, it'd be best to go all-out and really do it right. Unfortunately, there'd hardly be room in a single book for both a full treatment of compiler design in *addition* to enough OOP info to make the design of an object-oriented language feasible.

see in the following sections, you'll have no shortage of flexibility when you actually start scripting. Regardless, OOP is still something important to keep in mind.

XtremeScript Language Overview

XtremeScript is the name of the scripting system you're going to build, but more importantly, it's the name of the language the system is based around. As a result, I'll usually be referring to the language specifically when I use the term, unless it's clearly in a different context. I just mention it because there's some potential for confusion. With that out of the way, let's see what's up with this language.

Design Goals

XtremeScript should be a C-like language for the reasons you've already seen. It helps you maintain the same state of mind you'll be in while working on the host application's game engine, because it will most likely be written in C or C++.

As you've also learned, the language must be truly procedural, which means that the structure of its programs (er, scripts) are based on functions rather than simply being one flat block of code. The procedural nature of this language will help you organize your thoughts better through encapsulation and the possibility of code reuse by grouping commonly used actions and algorithms into functions.

Going back to the C-style issue, the first order of business is syntax. For two important reasons, the syntax of the language should be a direct copy of C whenever possible. First of all, this is practically what will designate your language as "C-style" to begin with, because the look and feel of a language is almost as important as its feature set. Second, the layout of the C language has been in worldwide use for decades, which means you get a tried-and-tested syntax without having to spend months or years coming up with it yourself.

NOTE

C-syntax is extremely popular, and has been used as the basis for most new languages. C++, Java, JavaScript, C# and many other newer languages all use the familiar C-syntax as the basis for their structure. As you can see, all the cool kids are doing it.

C syntax brings up a few issues, however. First of all, how far is too far? You certainly want to emulate the look, feel, and even functionality of C, but a full implementation of the entire C language would not only take considerably longer to complete, it would be total overkill for the scripting needs of more than a few games anyway. As a result, you'll trim a few features here and there in the interest of getting this done some time before the next geological age.

You already know of some fairly serious differences from C, for example, like the fact that the language will be typeless and have built-in support for strings. These are more along the lines of additions to the language, however, as opposed to removals. The real differences will be in the form of features that will not be supported, such as pointers. Pointers not only add a whole new level of complexity to the compiler and runtime environment, but they also have little relevance in the scripting of many games. Any sort of aggregate data manipulation, for example, will be done with arrays, and there probably won't be much need for the dynamic allocation of memory. In general, pointers will not be necessary within XtremeScript and would bring about too many complexities to justify implementing.

NOTE

Pointers are usually only necessary when dealing with complex and dynamic data structures, which the majority of your game scripting needs won't involve. Scripting is by its very nature more simplistic than "true" programming most of the time, so this loss isn't too big a deal. Even Java doesn't support pointers (although it does mimic a lot of their functionality with *references*, but that's another matter).

Other, smaller differences will exist as well. For example, there will be no support for structures or unions. These again add a significant level of complexity to the compilation process, and the vast majority of their functionality can be emulated with arrays, which will be supported. In the following pages, you'll take a look at a complete language specification for XtremeScript; anything that isn't mentioned there will not be supported.

To continue in your efforts to simulate C, you'll even go so far as to add a basic preprocessor. The C preprocessor is so widely and heavily used that it's become a part of the language itself, and XtremeScript will reflect that. The basic preprocessor will duplicate the functionality of the C preprocessor's most popular and commonly used features.

Lastly, the language overall needs to feel as free-form and flexible as possible. This means that things like whitespace, capitalization, and coding style idiosyncrasies like indenting and placement of curly brackets should not factor into the compiler's understanding of source code. You, along with anyone else who ends up using your scripting system, should feel just as comfortable and at-home as you would in Microsoft Visual C++. A compiler intelligent enough to keep these things in check is definitely going to be more work, but it will be one of the most invaluable additions to the language. Trust me on that.

NOTE

Many people don't know this, but the same preprocessor used by C has been used in many other programs aside from C compilers themselves, and in this regard, is a separate entity of its own. Its file inclusion and macro expansion facilities have proven equally useful in text editors, for example.

Syntax and Features

Fortunately, I've done the (somewhat) hard part already and put together a full language specification for you to work from. As I said, it's a clear derivative of C, which gives it a familiar syntax and most of its popular features. There are a number of cutbacks here and there, in addition to a few small additions or modifications, but I think it's enough to make you feel comfortable using it. Without further ado, take your first look at your future language, XtremeScript.

Data Structures

Data structures in XtremeScript are simple, as only single variables and one-dimensional arrays are supported. There are no `classes`, `structs`, `unions` or other aggregate structures built-in, although many of them can be simulated due to the typeless nature of the language. Because XtremeScript is typeless, an array can be easily "transformed" into a general purpose structure similar to C's `struct` by treating each array index as a field, which works well because the XtremeScript array allows any data type to be stored at any index.

Variables

First up are of course variables. As I've mentioned a number of times, XtremeScript variables are typeless, which means they can hold any data type at any time. Because of this, however, there's no need for type-specific declaration statements, such as this:

```
int MyInt = 16;
float MyFloat = 3.14156;
string MyString = "Hello, world!";
```

Instead, variables of all types are declared with the `var` keyword, so the previous code would actually look like this:

```
var MyInt = 16;
var MyFloat = 3.14156;
var MyString = "Hello, world!";
```

Although variables can't be declared with any specific type, they can always use them. XtremeScript supports Boolean, integer, floating-point, and string values, so a variable called `X` can be assigned any of the following values at any time:

```
X = true;     // Boolean (true and false are built-in XtremeScript keywords)
X = 16;       // Integer
X = 3.14159;   // Floating-Point
X = "Hello!";  // String
```

Makes things easier, huh? The only restriction is that variables must be declared before using them, which contrasts with a number of other scripting languages that don't force you to do this. The reason I've chosen to enforce this policy is that positively *evil* logic errors can be the result of simple variable typos, such as the following:

NOTE

Internally, Boolean values will be represented as integers, wherein `true` is equal to one and `false` is equal to zero.

```
MyValue = 256;
if ( MyVolue )
    print ( "MyValue is nonzero." );
else
    print ( "MyValue is zero." );
```

As you can see, `MyValue` has accidentally been written as `MyVolue` in the `if` statement, which could go unnoticed for who knows how long, causing strange results (in this case, it will always be treated as zero, no matter what value you think it should have). Let me tell you from experience: identifying typo logic errors is like find your car keys—you'll end up derailing your entire schedule trying to find them, you'll tear everything apart in the process, and in the end you'll just end up feeling like an idiot when you find out that you left them in the ignition the whole time.

Lastly, even though it was briefly mentioned in a previous code example, the Boolean data type is directly supported with the `true` and `false` keywords, which can be used in expressions just like any other value. For example:

```
Flag = true;
if ( Flag )
{
    // Do something
    Flag = false;
}
```

Of course, I haven't mentioned `if` statements yet, but this code should be self-explanatory anyway.

Strings

First and foremost, strings should be considered just another data type in the context of this language, because there's no such thing as a "string variable"; rather, it's one of the many types that any given variable can hold if it wants to. However, strings have one important difference from the other types, which is that they can be accessed both as variables and arrays. For example, if you have two variables `X` and `Y`, you can manipulate them like this:

```
X = "Hello";     // Set X to a greeting
Y = "Goodbye";   // Set Y to the opposite
X = Y;           // Now X and Y both contain "Hello"
```

Which is the same way you'd deal with other data types, such as integers and Booleans. However, in the event that you need to access individual characters or substrings from variables, you can also use array notation:

```
X = "ABC";
Y = "DEF";
Y = X [ 1 ];      // Y now equals "B"
```

Which provides a more precise interface with string data. Remember that also like arrays, character data begins at index 0, so the `"A"` character in `X` from the previous example resides at index 0, whereas `"B"` and `"C"` can be found at 1 and 2.

Remember, pretty much any string-processing function can be derived from this simple ability to access characters based on an index.

Arrays

Arrays are the last member of the XtremeScript data structures family. They're declared in a manner very similar to C, which simply involves putting a bracket pair (`[]`) after an otherwise normal variable declaration to denote the array's dimensions. For example, a 16-element array called `MyArray` can be declared like this:

```
var MyArray [ 16 ];
```

Like I said, it's just like C. The only difference to keep in mind is that XtremeScript does not support the `{ … }` notation for initializing array elements at the time of the declaration. Also, unlike many other script languages, variables cannot be used as arrays unless they're specifically declared as such, and writing past the boundaries of an array is just as dangerous as it is in a language like C. For example, this is not allowed, as it is in many other scripting languages:

```
var X;
X [ 3 ] = "Hello!";   // Not allowed, X was never declared as an array
```

The following of course, is fine:

```
var X [ 16 ];
X [ 3 ] = "Hello!";   // No problem, X was declared as an array
```

Remember, even though more complex structures like C's `struct` aren't supported, you can simulate them with relative ease simply by using different elements of the array. For example, imagine that you wanted to port a structure like this from C++:

```
struct MyStruct
{
   bool X;
   int Y;
   float Z;
}

MyStruct Foo;

Foo.X = true;
Foo.Y = 32;
Foo.Z = 3.14159;     // I've really got a thing for pi, don't I?
```

It's simply a matter of declaring a three-element array and mapping each index to the appropriate field. Sure it's not quite as intuitive, but it works and the end result the same:

```
var Foo [ 3 ];

Foo [ 0 ] = true;
Foo [ 1 ] = 32;
Foo [ 2 ] = 3.14159;
```

NOTE

Yes, `structs` are extremely useful, and would definitely have their application in game scripting. However, a decent amount of complexity would accompany their inclusion in the compiler, so in the interest of keeping things as simple as possible, they've been left out. However, by the end of the book you should be capable of adding them yourself if you find their absence unacceptable.

Operators and Expressions

As you've learned in this chapter, expressions are an invaluable feature in any language, so you want to make sure XtremeScript doesn't fall short in this category. Let's just dive right in and look at the operators that the language provides.

Arithmetic

Arithmetic functions are the basis for most assignment expressions. XtremeScript supports the usual lineup, listed in Table 7.1:

Table 7.1 XtremeScript Arithmetic Operators

Operator	Description
+	Addition (Binary)
-	Subtraction (Binary)
$	String Concatenation (Binary)
*	Multiplication (Binary)
/	Division (Binary)
%	Modulus (Binary)
^	Exponent (Binary)
++	Increment (Unary)
--	Decrement (Unary)
+=	Addition assignment (Binary)
-=	Subtraction assignment (Binary)
*=	Multiplication assignment (Binary)
/=	Division assignment (Binary)
%=	Modulus assignment (Binary)
^=	Exponent assignment (Binary)

Notice that unlike C, this language provides a built-in exponent operator using the familiar caret (`^`). Also, as is the case with C, the increment (`++`) and decrement (`--`) operators come in both pre- and post- forms, so both of the following are legal:

```
X ++;
++ X;
```

NOTE

By the way, just in case you've forgotten, *binary* operators are those that take two operands, with one on each side of the operator. Examples are addition and subtraction, which are always in the form `X + Y` and `X - Y`. *Unary* operators accept only a single operand, which can be on either side depending on the definition of the operator. Increment, for example, which can take the form `++ X`, is a unary operator.

Bitwise

Bitwise operations are generally used for manipulating the individual bits of integer variables. XtremeScript's bitwise operators are listed in Table 7.2:

In another slight divergence from C, notice that the exclusive or operator is no longer the caret. I swapped that with the exponent operator. It is now the hash mark (#) instead.

Table 7.2 XtremeScript Bitwise Operators

Operator	Description
&	And (Binary)
|	Or (Binary)
#	XOr (Binary)
~	Not (Unary)
<<	Shift left (Binary)
>>	Shift right (Binary)
&=	And assignment (Binary)
|=	Or assignment (Binary)
#=	XOr assignment (Binary)
<<=	Shift left assignment (Binary)
>>=	Shift right assignment (Binary)

Logical and Relational

The last group of operators to mention are the logical and relational operators. Logical operators are used to implement Boolean logic in expressions, whereas relational operators define the *relationship* between entities (greater than, less than, etc.). XtremeScript's logical and relational operators are listed in Tables 7.3 and 7.4, respectively.

Table 7.3 XtremeScript Logical Operators

Operator	Description
`&&`	And (Binary)
`||`	Or (Binary)
`!`	Not (Unary)
`==`	Equal (Binary)
`!=`	Not Equal (Binary)

Table 7.4 XtremeScript Relational Operators

Operator	Description
`<`	Less Than (Binary)
`>`	Greater Than (Binary)
`<=`	Less Than or Equal (Binary)
`>=`	Less Than or Equal (Binary)

Precedence

Lastly, let's quickly touch on operator *precedence*. Precedence is a set of rules that determines the order in which operators are evaluated. For example, recall the *PEMDAS* mnemonic from school, which taught us that, for example, multiplication (M) is evaluated before subtraction (S). So, `8 - 4 * 2` is equal to zero,

NOTE

According to my editors, they've never heard of PEMDAS, so I'll explain it a bit in case you're confused too. My high school (in Northern California) math classes used the PEMDAS mnemonic to help us remember operator precedence. PEMDAS stood for "Please excuse my dear Aunt Sally", and, more specifically, "Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction". Popular derivatives involve Aunt Sally being executed and exfoliated. I leave it up to the reader to decide her fate.

because `4 * 2` is evaluated first, the result of which is then subtracted from `8`. If subtraction had higher precedence, the answer would be 8, because `8 - 4` would be multiplied by `2`.

XtremeScript operators follow pretty much the same precedence rules as other languages like C and Java, as illustrated in Table 7.5 (operators are listed in order of decreasing precedence, from left to right and top to bottom).

Table 7.5 XtremeScript Operator Precedence

Operator Type	Precedence
Arithmetic	`(* / + - ++ -- % ^ $)`
Bitwise	`(& | # ~ << >>)`
Assignment	`(= += -= *= /= &= |= #= ~= %= ^= <<= >>=)`
Logical/Relational	`(&& || == != < > <= >=)`
Unary Operators	`(- !)`

Code Blocks

Code blocks are a common part of C-style languages, as they group the code that's used by structures like `if`, `while`, and so on. Like C, code blocks don't need to be surrounded by curly brackets if they contain only one line of code (the exception to this rule is function notation; even single-line functions must be enclosed in brackets).

Control Structures

Control structures allow the flow of the program to be altered and controlled based on the evaluation of Boolean expressions. They include loops like `while` and `for` and conditional structures like `if` and `switch`. Let's look at the conditional/branching structures first.

Branching

First up is if, which works just like most other languages. It accepts a single Boolean expression and can route program flow to both a true or false block, with the help of the optional else keyword:

```
if ( Expression )
{
    // True
}
else
{
    // False
}
```

NOTE

It's worth noting that although many languages support a built-in elseif keyword, there's not really any need to do so. The if-else-else if structure can be assembled simply by placing an else and an if together on the same line without putting curly brackets around the else block.

Iteration

XtremeScript supports two simple methods for iteration. First up is the while loop, which looks like this:

```
while ( Expression )
{
    // Loop body
}
```

The while loop is often considered the most fundamental form of iteration in C-style languages, so it's technically all you'll need for most purposes. However, the for loop is equally popular, and often a more convenient way to think about looping, so let's include it as well:

```
for ( Initializer; Terminating-Condition; Iterator )
{
      // Loop body
}
```

The funny thing about the `for` loop is that it's really just another way to write a `while` loop. Consider the following code example:

```
for ( X = 0; X < 16; ++ X )
{
      Print ( X );
}
```

This code could be just as easily written as `while` loop, and behave in the exact same way:

```
X = 0;
while ( X < 16 )
{
     Print ( X );
     ++ X;
}
```

Nifty, huh? You might be able to capitalize on this fact later on when implementing the language. For now, though, just remember that the `while` loop is all you'd technically need, but that the `for` loop is more than convenient enough to justify its inclusion.

Lastly, you should include two other commonly used C keywords: `break` and `continue`. As you can see, `break` causes the current line of execution to exit the loop and "break" out of it, just like in a `case` block. `continue` causes the loop to unconditionally jump to the next iteration without finishing the current one.

NOTE

Technically, the `while` loop is limited by the fact that it will not always iterate at least once; something the `do...while` loop allows. The only difference with this new loop is that it starts with `do` instead of `while`, and the conditional expression is evaluated *after* the loop iterates, meaning it will always run at least once. The `do...while` loop is uncommon however, so I've chosen not to worry about it. Keep in mind, though, that it'd be an easy addition, so if you do really feel like you need it, you shouldn't have much trouble doing it yourself.

Functions

Functions are an important part of XtremeScript, and are the very reason why you call it a procedural language to begin with. You'll notice a small amount of deviation from C syntax, when dealing with XtremeScript functions, however, so take note of those details.

Functions are declared with the `func` keyword, unlike C functions, which are declared with the data type of their return value, or `void`. For example, a function that adds two integers and returns the result in C would look like this:

```
int Add ( int X, int Y )
{
    return X + Y;
}
```

In XtremeScript, it'd look like this:

```
func Add ( X, Y )
{
    return X + Y;
}
```

Because XtremeScript is typeless, there's no such thing as "return type". Rather, all functions can optionally return any value, so you simply declare them with `function`. Next, notice that the name of each parameter is simply an identifier. Again, because the language is typeless, there's no data type to declare them with. Usually you use the `var` keyword to declare variables, but there's no real need in the case of parameter lists because preceding each parameter with `var` in all cases would be redundant. Notice, though, that at least `return` works in XtremeScript just as it does in C.

The last issue to discuss with functions is how the compiler will gather function declaration information. In C, functions can be used only in the order they were declared. In other words, imagine the following:

```
void Func0 ()
{
    Func1 ();
}

void Func1 ()
{
    // Do something
}
```

This would cause a compile-time error because at the time `Func1 ()` is called in `Func0 ()`, `Func1 ()` hasn't been defined yet and the compiler has no evidence that it ever will be. C++ solves this problem with *function prototypes*, which are basically declarations of the function that precede its actual definition and look like this:

```
void Func0 ();
void Func1 ();

void Func0 ()
{
    Func1 ();
}

void Func1 ()
{
    // Do something
}
```

Function prototypes are basically a promise to the compiler that a definition exists somewhere, so it will allow calls to the function to be made at any time. I personally don't like this approach and think it's redundant, though. I don't like having to change my function prototype in two places whenever I modify its name or parameter list. So, the XtremeScript compiler will simply work in multiple passes; the first pass, for example, might simply scan through the file and build a list of functions. The second pass, which will actually perform the compilation, will refer to this table and therefore safely allow any function to be called from anywhere. I know this is getting a bit technical for a simple language overview, but it affects how code is written so I've included it. Naturally, we'll cover all of this in far greater detail later on, so just accept it for now.

TIP

I won't be covering it directly in this book, but a useful addition to your own implementation of the language would be an `inline` keyword for inlining functions. Inline functions work like macros defined with the preprocessor's `#define` keyword—their function calls are replaced with the function's code itself. This saves the overhead of physically calling the function (which we'll learn more about starting in the next chapter). Of course, in the context of scripting the affect of inlining may be completely unnoticeable, but it's always a nice option when writing performance-critical sections of code.

Escape Sequences

One important but often unnoticed addition to a language is the escape sequence. Escape sequences allow, most notably, double quotes to be used within string literal values without confusing the compiler. XtremeScript's escape sequence syntax is familiar, although we'll only be implementing two: `\"` for escaping double-quotes, and `\\`, for escaping the backslash itself (in other words, for using the backslash without invoking an escape sequence on the character that immediately follows it).

Comments

As you've probably noticed by now, XtremeScript will of course support the double-slash (`//`) comments that C++ popularized. However, C-style block comments will be included as well. All told, the two XtremeScript comment types will look like this:

```
//    This is a single line comment

/*
    This is a
    block comment.
*/
```

Single line comments simply cause every character after the double slashes to be treated as whitespace and thus ignored. Block comments work in a similar manner, but can of course span multiple lines. In addition, they're especially flexible in that they can be embedded in a line of code without affecting the code on either side. For example, the following line of code:

```
var MyVar /* Comment */ = 32;
```

Will appear to the compiler as though the comment were never there, like this:

```
var MyVar = 32;
```

The Preprocessor

As I mentioned, you'll even include a small preprocessor in the language to make things as easy as possible. Just as in C, the syntax for preprocessor directives will be the hash mark (#) followed by the directive itself.

The first and most obvious directive will be `#include`, which will allow external files to be dumped into the file containing the directive at compile-time, and looks like this:

```
#include "D:\Code\MyFile.xs"
```

Note the use of quotation marks. The XtremeScript compiler won't contain any default path information, so the greater-than/less-than symbol syntax used in C won't be included.

We'll also include a watered-down version of `#define`, which will be useful for declaring constants:

```
#define THIS_IS_A_CONSTANT 32
var X = THIS_IS_A_CONSTANT;
```

I say watered-down because this will be the only use of this directive. It will not support multi-line macros or parameters.

Reserved Word List

As a final note, let's just review everything by taking a look at the following simple list of each reserved word in the XtremeScript language as presented by Table 7.6

Table 7.6 XtremeScript Operator Precedence

Operator Type	Order Precedence
`var/var []`	Declares variables and arrays.
`true`	Built-in true constant.
`false`	Built-in false constant.
`if`	Used for conditional logic.
`else`	Used to specify else clauses.
`break`	Breaks the current loop.
`continue`	Forces the next iteration of the current loop to begin immediately.
`for`	Used for looping logic; another form of the `while` loop.
`while`	Used for looping logic.
`func`	Declares functions.
`return`	Immediately returns from the current function.

SUMMARY

This chapter has been a relatively easy one due to its largely theoretical nature, and I hope it's been fun (or at least interesting), because designing the language itself is usually the most enjoyable and creative part of creating a scripting system (in my opinion). More importantly, however, I hope that you've learned that creating a language even as simple as XtremeScript is not a trivial matter and should not be taken lightly. As you'll soon learn, the design of this language will have a pivotal effect on everything else you do in the process of building your scripting system, and you'll see first-hand how important the planning you've done in this chapter really is.

All stern warnings aside, however, creating languages can be a genuinely creative and even artistic process. Although the engineering aspect of a language's design, layout, and functionality is obviously important, its look and feel should not be understated. For matters of simplicity and accessibility, I've chosen to model XtremeScript mostly after a watered-down subset of C, but don't forget that when designing a scripting system of your own, you really do have the ability to create anything you want.

So with the language specification finished and in hand, let's finally get started on actually implementing this thing!

Part Four

Designing and Implementing a Low-Level Language

This page intentionally left blank

CHAPTER 8

Assembly Language Primer

"Are you insane in the membrane?"
——*Principal Blackman,* Strangers with Candy

In the last chapter, we finally sat down and designed the language you're ultimately going to implement later in the book. This was the first major step towards building your own scripting system, and it was a truly important one. Obviously, a scripting system hinges on the design of the language around which it's based; failing to take the design of this language into heavy consideration would be like designing and building a house without giving any thought to whom might end up living there, what they'll do with the place, and the things they'll need to do them.

As you've learned, however, high-level languages like the one you laid out aren't actually executed at runtime. Just like C or C++, they're compiled to an assembly language. This assembly version of the program can then be easily translated to executable bytecode, capable of running inside a virtual machine. In other words, assembly is like the middleman between your high-level script and the runtime environment with which it will be executed. This makes the design of the assembly language nearly as crucial as the design of the HLL (High Level Language).

In this chapter, you're going to

- Learn what exactly assembly language is, how it works, and why it's important.
- Learn how algorithms and techniques that normally apply to high-level languages can be replicated in assembly.
- Lay out the assembly language that the assembler you'll design and implement in the next chapter will understand.

What Is Assembly Language?

I've asked this question a number of times already, but here's the final answer: Assembly language is code that is directly understood by a hardware processor or virtual machine. It consists of small, fine-grained *instructions* that are almost analogous to the commands in a command-based language. Because of this, assembly is characterized by its rigid syntax and general inability to perform more than one major task per line of code.

Assembly language is necessary because processors, real and virtual alike, aren't designed to think on a large scale. When you play a video game, for example, the processor has no idea what's going on; it's simply shoveling instructions through its circuitry as fast as it possibly can. It'd be sorta like walking down the street, bent over in such a way that your face is only a foot or two off the ground. Your field of vision would be so narrow that you'd only be able to tell what was immediately around you, and would therefore have a hard time with large-scale strategies. If all you can see is the 2 foot x 2 foot surrounding area, it'd be hard to execute a plan like "walk to the center of the park." However, if someone broke it down into simple instructions, like "take four steps forward, and then take two steps right (to avoid the tree), and then take another 10 steps forward, turn 90 degrees, and stop" you'd find it to be just as easy as anything else. You wouldn't have much idea of where this plan would ultimately take you, but you'd have no trouble executing it.

This distinction is what separates machinery from intelligence. However, it's also what makes processors so fast. Because they have to focus only on one tiny operation at almost any given time, they're capable of running extremely quickly and with very low overhead. For this reason, assembly language programs are generally smaller and faster than their counterparts written in a HLL (although this is changing rapidly and is not nearly as true as it once was, thanks to advances made in optimizing compilers).

Assembly language is usually optional, however. Even when programming extremely compact systems like the Gameboy Advance, you still have the alternative of writing your code in C and having a compiler handle the messy business of assembly for you. Of course, no matter how abstracted and friendly the compiler is, there's always an assembly language under there somewhere. This is the burden of writing your own scripting system; you personally have to create and understand all of the mundane and technical low-level details you normally take for granted when coding.

Why Assembly Now?

You may be wondering why I'm covering assembly language at this point in the book, when I haven't really gone into much detail regarding the high-level language of the scripting system (aside from the last chapter). At first it seems like it'd be more intuitive to learn how to compile high-level code, and then learn how low-level code works after that, right? The problem is, doing so would be like building a house without a foundation. High-level code must be compiled *down to* assembly, which means without coverage of low-level languages now you'd be able to write only about 50% of your compiler.

Furthermore, it's quite possible to create a functional and useful scripting system that's based entirely on an assembly-style language, instead of a high-level one. These sort of scripting systems are easy and fast to create, are very powerful, and are fairly easy to use as well. By starting with low-level code now, you can have an initial version of your scripting system up and running within a few chapters. Once you have an assembly-based scripting language fully implemented, you'll either be able to get started with game scripting right away with it, or you can continue and add the high-level compiler. This order of events lets you move at your own pace and develop as much of the system as you want or need.

Besides, high-level code compilation is a large and complicated task and is orders of magnitude more difficult than the assembly of low-level code. It'll be nice to see a working version of your system early on to give you the motivation to push through such a difficult subject later.

How Assembly Works

Assembly language is often perceived by newcomers as awkward to use, esoteric, and generally difficult. Of course, most people say the same thing about computer programming in general, so it's probably not a good idea to believe the nay-sayers. Assembly is different than high-level coding to be sure; but it's just as easy as anything else if you learn it the right way. With that in mind, let's discuss each of the major facets of assembly-language programming.

Instructions

As stated previously, assembly languages are collections of *instructions*. An instruction is usually a short, single-word or abbreviation that corresponds to a simple action the CPU (or virtual machine) is capable of performing. For example, any CPU is going to be doing a lot of memory movement; taking values from one area of memory and putting them in another. This is done in Intel 80X86 assembly language by perhaps one of the most infamous instructions, `Mov` (short for **Mov**e). `Mov` can be thought of like a low-level version of C's assignment operator "`=`"; it'll transfer the contents of a source into a destination. For example, the following line in C:

```
MyVar0 = MyVar1;
```

Might be compiled down to this:

```
Mov    MyVar0, MyVar1
```

Essentially, this line of code is saying "move `MyVar1` into `MyVar0`" (this also brings up the issue of assembly language variables, but I'll get to that in a moment).

The collection of instructions a given assembly language offers is called its *instruction set*, and is responsible for providing its users with the capability to reproduce any high-level coding construct, from an `if` block to a function to a `while` loop, using only these lower-level instructions. Because of this, instructions can range from moving memory around, like the `Mov` instruction you've just seen, to performing simple arithmetic and bitwise operations, comparing values, or transferring the flow of execution to another instruction based on some conditional logic.

Operands

Instructions on their own aren't very useful, however. What gives them their true power are *operands*, which are passed to instructions, causing them to perform more specific actions. You saw operands in the `Mov` example. `Mov` is a general-purpose instruction for moving memory from one area to another. Without operands, you'd have no way to tell `Mov` what to move, or where to move it. Imagine a `Mov` instruction that simply looked like this:

```
Mov
```

Doesn't make much sense, does it? `Mov` does require operands, of course—two of them to be exact—the *destination* of the move, and the *source* of the data to put there. Operands are conceptually the same as the operands you passed to the commands in the command-based language developed in Chapters 3 and 4, as illustrated in Figure 8.1.

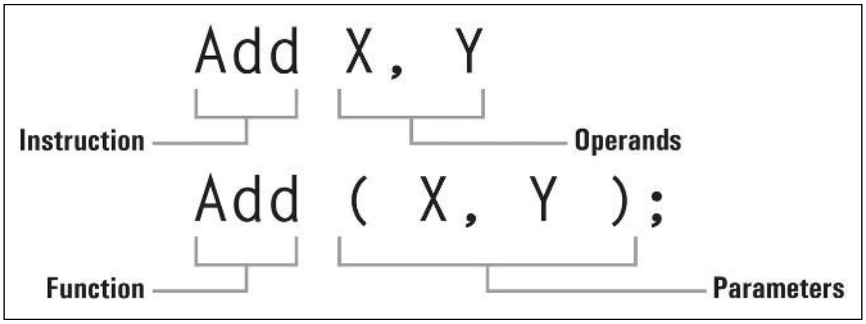


Figure 8.1

Operands are to instructions as parameters are to functions.

In fact, command-based languages and assembly languages are very similar in a lot of ways. Commands mirror instructions almost exactly, as do their operands. To use the analogy once again, instructions are like function calls. The instruction itself is like the function name, which specifies the action to be performed. The operands are like its parameters.

Expressions

To really get a feel for how instructions and operands relate to one another, let's look at how assembly languages manage expressions. Remember, this sort of thing isn't possible in assembly:

```
Mov    X, ( Y + Z ) * 2 / W
```

So what do you do if you need to represent an expression like this? You need to break it up into its constituent operations, using different assembly instructions to perform each one. For example, let's break down the expression `( Y + Z ) * 2 / W`:

- Because parentheses override the order of operations, `Y` and `Z` are added first.
- The sum of `Y` and `Z` is then multiplied by 2.
- The product of the multiplication is then divided by `W`.

So, this means you need to perform three arithmetic instructions: an addition, a multiplication, and a division. The result of these three operations will be the same as the single expression listed previously. You can then put this value in `X` and your task will be complete.

Here's one question though: step two says you have to multiply the sum of `Y` and `Z` by 2. How do you do this? Because assembly doesn't support *any* form of expression, you certainly can't do this:

```
Mul    Y + Z, 2
```

Besides, where is the sum going to go? "`Y + Z`" isn't a valid destination for the result. `Y + Z` is undoubtedly an expression (and by the way, `Mul`, short for Multiply, is an instruction that multiples the first operand by the second). Even though the sum isn't the final result of the expression, you still need to save it in some variable, at least temporarily. Consider the following:

```
Mov     Temp, Y
Add     Temp, Z
Mul     Temp, 2
```

`Temp` is used to store the sum of `Y` and `Z`, which is then multiplied separately by 2. This also introduced another new instruction: `Add` (which isn't short for anything! Ha!) is used to add the second operand to the first. In this case, `Z` was added to `Temp`, which already contained `Y`, to create the sum of the two. With temporary variables, the expression becomes trivial to implement. Here's the whole thing:

```
Mov    Temp, Y    ; Move Y into Temp
Add    Temp, Z    ; Add Z to Temp
Mul    Temp, 2    ; Multiply ( Y + Z ) times 2
Div    Temp, W    ; Divide the result by W, producing the final value
```

Two things first of all; yes, assembly languages generally use the semicolon to denote comments, which are single-line comments only. Second, the `Div` instruction, as you probably surmised, divides the first operand by the second (although in this case, as in the case of `Mul`, I haven't followed Intel 80X86 syntax exactly). To wrap things up, check out Figure 8.2. It illustrates the process of reducing a C-like expression to instructions.

NOTE

While it's true that a pure assembly language has no support for expressions, many modern assemblers, called *macro assemblers*, are capable of interpreting full expressions and automatically generating the proper instructions for them. While this definitely blurs the line between compilers and assemblers, it can really come in handy.

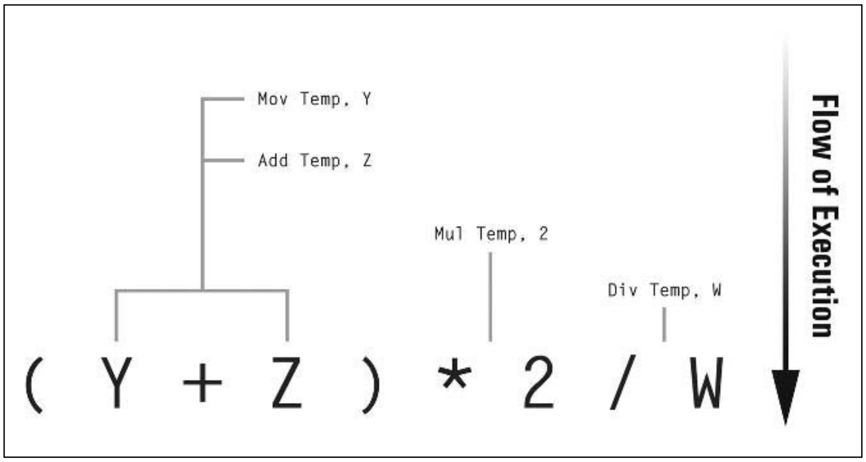


Figure 8.2

A C-style expression being reduced to instructions.

So, using only a handful of instructions (`Mov`, `Add`, `Mul`, and `Div`), you've managed to recreate the majority of the expression parsing abilities of C using assembly. Granted, it's a far less intuitive way to code, but once you get some practice and experience it becomes second nature.

Jump Instructions

Normally, assembly language executes in a sequential fashion from the first instruction to the last—just like a C program runs from the first statement to the last. However, the flow of execution in assembly can be controlled and re-routed by using instructions that strongly mimic C's `goto`. Although computer science teachers generally frown on `goto`'s use, it provides the very backbone of assembly language programming. These instructions are known as *jump* instructions, because they allow the flow of execution to "jump" from one instruction to another, thereby disrupting the otherwise sequential execution.

Jumps are key to understanding the concept of looping and iteration in assembly language. If a piece of code needs to be iterated more than once, you can use a jump instruction to move the flow of execution back to the start of the code that needs to be looped, thereby causing it to execute again. Imagine the following infinite loop in C:

```
while ( 1 )
{
    // ...
    // ...
    // ...
}
```

You can refer to the "top" of this block of code as the `while` line, whereas the "bottom" of the block is the closing bracket (`}`). Everything in between represents the actual loop itself. So, to rewrite this loop in assembly-like terms, consider the following:

```
LoopStart:
    ; ...
    ; ...
    ; ...
Jmp LoopStart
```

Just like in C, you can define line labels in assembly. The `Jmp` instruction seen in the last line (short for **Jump**) is known as an *unconditional jump*; or in other words, an instruction that *always* causes the flow of execution to move to the specified line label. Note that `while ( 1 )` is also "unconditional"; there is no condition under which that expression will ever fail (and if `1` ever *does* evaluate to false, we're all in a lot of trouble and will have much bigger problems to worry about anyway). In both cases, this is what makes the loops infinite. Check out Figure 8.3 to see this graphically.

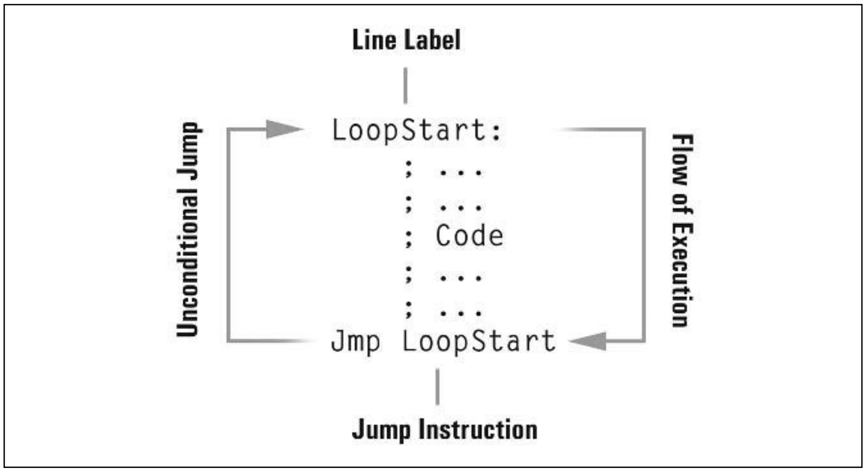


Figure 8.3

Using `Jmp` to form an infinite loop.

As a final note, consider rewriting this code in another form of C, but one that looks much more like the assembly version:

```
LoopStart:
    // ...
    // ...
    // ...
goto LoopStart;
```

Here, the code is almost identical, right? As you can see, assembly doesn't have to be *all* that different. In a lot of ways it strongly parallels C (which, in fact, was one of C's original design goals back in the ultra old-school K&R days).

NOTE

"K&R" is a term referring to the earliest versions of C, as initially created by Dennis Ritchie and Brian Kernighan. Many aspects of C have drastically changed from those days, hence the special term used to denote them.

Conditional Logic

Of course, unconditional jumps are about as useful as infinite loops are in C, so you need a more intelligent way to move the flow of code around. In C, you do this with the `if` construct; `if` allows you to branch to different parts of the program based on the outcome of a Boolean expression. This would be nice to do in assembly too, but expressions aren't an available luxury. Instead, you get the next best thing; comparison instructions and conditional jumping instructions. These two classes of instructions come together to simulate the full functionality of a C `if` statement, albeit in a significantly different way.

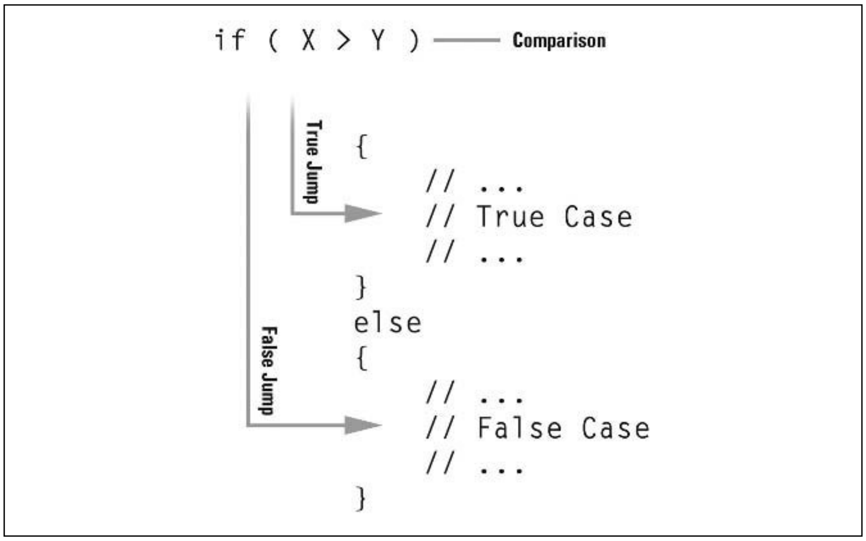
To understand how this works, first think about what an `if` statement really does. Consider the following code block:

```
if ( X > Y )
    // True case
else
    // False case
```

What this is basically saying is, "execute the true case if `X` is greater than `Y`, and execute the false case if the `X` is not greater than `Y`." This basically boils down to two fundamental operations; the *comparison* of `X` and `Y`, and the *jump* to the proper clause based on the result of that comparison. Figure 8.4 illustrates this process.

These two concepts are present in virtually all decision making. For example, imagine that you're standing in the lobby of an office building, and want to get into the elevator. Now imagine that there are two doors on the facing wall—one door that reads "Janitor Closet", and another that reads "To Elevators". Your brain will read the text written on both doors and *compare* it to what it's looking for. If one of the comparisons evaluates to truth, or equality, you'll *jump* (or walk, if you're a normal person), towards the proper door. In this case, "To Elevators" will result in equality when compared to what you're brain is looking for (a door that leads to an elevator).

Returning to the `if` example, the code will first *compare* `X` to `Y`, and then execute one of two supplied code blocks based on the outcome. This means that in order to simulate this functionality



```
if ( X > Y )
{
    // ...
    // True Case
    // ...
}
else
{
    // ...
    // False Case
    // ...
}
```

Figure 8.4

The `if` *block employs both a* comparison *and a* jump *to implement decision making.*

in assembly, you first need an instruction that facilitates comparisons. In the case of Intel 80X86 assembly, this instruction is called `Cmp` (short for Compare). Here's an example:

```
Cmp     X, Y
```

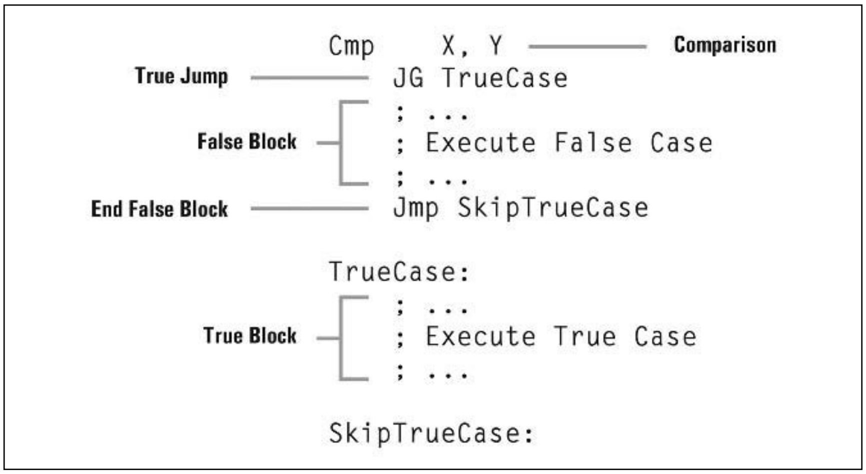
This instruction will compare the two values, just like you need. The question, though, is where does the result of the comparison go? For now, let's not worry about that. Instead, let's move on to the jump instructions you'll need to complete the assembly-version of the `if` construct. Because the original jump was unconditional, meaning it would cause the flow of instructions to change under all circumstances, it won't work here. What you need is a *conditional* jump; a type of jump instruction that will jump only in certain cases. In this case specifically, you should jump only if `X` is greater than `Y`. Here's an example:

```
Cmp     X, Y
JG      LineLabel
```

The new instruction here is called `JG`, which stands for Jump if Greater Than. `JG` will cause the flow of execution to jump to `LineLabel` *only* if the result of the *last* comparison was "greater than". `JG` doesn't actually care about the operands you compared themselves; it doesn't even know `X` and `Y` exist; all it cares about is that the first thing passed to `Cmp` was greater than the second thing, which `Cmp` has already determined. These two instructions, when coupled, provide the complete comparison/jump concept. Let's now take a look at how the code for each case (true and false) is actually executed.

When performing conditional logic in assembly, there are basically two ways to go about it. Both methods involve marking blocks of code with line labels, but the exact placement of the code blocks and labels differs. Here's the first approach (check out Figure 8.5 to see it graphically):

```
    Cmp    X, Y
    JG     TrueCase
    ; Execute false case
    Jmp    SkipTrueCase
TrueCase:
    ; Execute true case
SkipTrueCase:
    ; The "if construct" is complete,
    ; so the program continues.
```



```
Cmp    X, Y
    JG TrueCase
    ; ...
    ; Execute False Case
    ; ...
    Jmp SkipTrueCase

TrueCase:
    ; ...
    ; Execute True Case
    ; ...

SkipTrueCase:
```

Figure 8.5

The comparison and jump of an assembly language `if` *implementation.*

In this case, you first compare `X` to `Y` and perform the jump if greater than (`JG`) instruction. Naturally, you'll use this to make a jump to the true case (because you jump only if the condition was true, and in this case it was), which begins at the `TrueCase` line label. `TrueCase` continues onward until it reaches the `SkipTrueCase` line label. This label is simply there to mark the end of the true case block; it doesn't actually do anything, so execution of the program keeps moving, uninterrupted. If the comparison evaluates to false, however, you don't jump at all. This is because `JG` is only given one line label, and therefore can *only* change the flow of execution if the condition was true. If it's false, you keep on executing instructions beginning right after `JG`. Because of this, you need to put the false case directly under the conditional jump. However, because the false case is now above the true case, the sequential order of execution of assembly instructions will inadvertently cause the true case to be executed afterwards too, which isn't what you want. Because of this, you need to put an unconditional jump (`Jmp`) after the false case to skip past the true case. This ensures that no matter what, only one of the two cases will be executed based on the outcome of the comparison.

This approach works well, but there is one little gripe; the code blocks are upside down, at least compared to their usual configuration in C. C and C++ programmers are used to the idea of the true block coming before the false block, and you should do that in your assembly language coding as well. Here's an example of how to modify the previous code example to swap the blocks around:

```
    Cmp     X, Y
    JLE     FalseCase
    ; Execute true case
    Jmp     SkipFalseCase
FalseCase:
    ; Execute false case
SkipFalseCase:
    ; The "if construct" is complete,
    ; so the program continues.
```

As you can see, the true and false blocks are now in the proper order, but you're forced to make the opposite of the comparison you made earlier (note that `JLE` means Jump if Less than or Equal, which is the opposite of `JG`). Because you want the true case to come before the false case, you must rewrite the comparison so that it *doesn't* jump if true, instead of the other way around. In retrospect, I don't think the C-style placement of the true and false blocks is worth the reversed logic, however, and generally do my assembly coding in the style of the original example.

In either case, however, you should now understand how basic conditional logic works in assembly. Of course, there's a bit more to it than this; most notably, you need a lot more jump instructions in order to properly handle any situation. Examples of other jumps the Intel 80X86 is capable of making include `JE` (Jump if Equal), `JNE` (Jump if Not Equal), and `JGE` (Jump if Greater than or Equal).

Iteration

Conditional logic isn't all jump instructions are capable of. Looping is just as important in low-level languages as it is in high-level ones, and the jumps are an invaluable part of how iteration is implemented in assembly language programs (or scripts, as in this case).

Recall the infinite loop example, which showed you how jump instructions and line labels form the "top" and "bottom" of a loop's code block. Here it is again:

```
LoopStart:
    ; ...
    ; ...
    ; ...
Jmp LoopStart
```

Here, the loop executes exactly from the declaration of the `LoopStart` label, all the way down to the `Jmp`, before moving back to the label and reiterating. Once again, however, this loop would run indefinitely and therefore be of little use to you. Fortunately, however, you learned how conditional logic works in the last example. And, if you really analyze a `for` or `while` loop in C, you'll find that all finite loops involve conditional logic of some form (which is what makes them finite in the first place).

Take a `while` loop for example. A `while` loop has two major components—a Boolean expression and a code block. At each iteration of the loop, the expression is evaluated. If it evaluates to true, the code block is executed and the process repeats. Presumably, the code block (or some outside force) will eventually do something that causes the expression to evaluate to false, at which point the loop terminates and the program resumes its sequential execution. Take a look at the code:

```
while ( Expression )
{
    // ...;
    // ...;
    // ...;
}
```

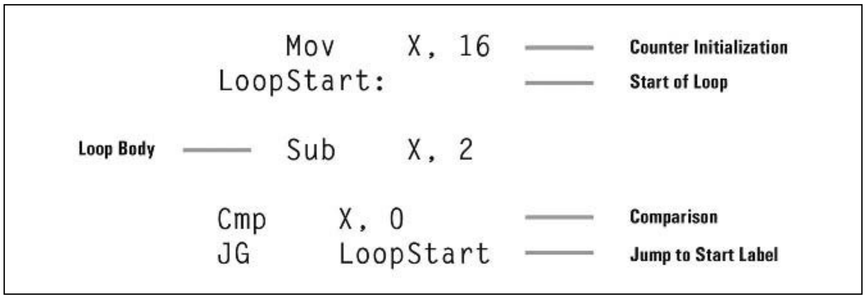
This means that in order to simulate this in assembly, you'll once again use the `Cmp` instruction, as well as a conditional jump instruction, to create the logic that will cause the loop to terminate at the proper time. As an example, let's attempt to reduce the following C loop to assembly:

```
int X = 16;           // Set X to 16
while ( X > 0 )       // Loop as long as X is greater than zero
    X -= 2;          // Decrement X by 2 at each iteration
```

Here, the "code block" is decidedly simple; a single line that decrements `X` by 2. The loop logic itself is designed to run as long as `X` is greater than zero, which will be around eight iterations because `X` starts out as 16. Look at the assembly equivalent:

```
    Mov    X, 16      // Set X to 16
LoopStart:            // Provide a label to jump back to
    Sub    X, 2       // Subtract 2 from X
Cmp    X, 0           // Compare X to zero
JG LoopStart          // If it's greater, reiterate the loop
```

Once again you're introduced to another instruction, `Sub`, which Subtracts the second operand from the first. As for the code itself, the example starts by `Mov`ing 16 into `X`, which implements the assignment statement in the C version. You then create a line label to denote the top of the loop block; this is what you'll jump back to at each iteration. Following the label is the loop body itself, which, as in the C version, is simply a matter of decrementing `X` by 2. Lastly, you implement the loop termination logic itself by comparing `X` to zero and only reiterating the loop if it's greater. Check out Figure 8.6, which illustrates the basic structure of an assembly loop.



```
            Mov     X, 16
        LoopStart:

            Sub     X, 2

        Cmp     X, 0
        JG      LoopStart
```

Figure 8.6

The structure of an assembly language loop.

The one difference between these two pieces of code, however, is that the loop behaves slightly differently in the assembly version. One of the major points of C's `while` loop is that it loops only if the expression is true; because of this, if the expression (for whatever reason) is false when the loop first begins, the loop will never execute. This is a stark contrast from your version, which will always execute at least once because the expression isn't checked until the loop body is finished. This is a problem that can be solved either by rethinking your loop logic to allow at least one iteration in all cases, or by rearranging the code block order like you did in the first conditional logic example in the last section.

As for `for` loops, remember that they're just another way of writing `while` loops. For example, consider the following:

```
for ( int X = 0; X < 16; ++ X )
{
    printf ( "Iteration %d", X );
}
```

This could just as well be written using `while`, like so:

```
int X = 0;
while ( X < 16 )
{
    printf ( "Iteration %d", X );
    ++ X;
}
```

And because you've already managed to translate a `while` loop to assembly (albeit a slightly reversed one), you can certainly manage `for` loops as well.

NOTE

Throughout this chapter, as well as any other time I mention assembly language, I'll use the terms "virtual machine", "runtime environment", "processor", and "CPU" interchangeably. Because a virtual machine is designed to literally mimic the layout and functionality of a real hardware CPU (hence the name), just about anything I say in regards to one applies to the other (unless otherwise stated).

You've made a lot of progress so far; understanding how expressions, conditional logic, and iteration work in assembly is a huge step forward. Now, let's dig a bit deeper and see how assembly will actually interact with the virtual machine.

Mnemonics versus Opcodes

In a nutshell, instructions represent the CPU's capabilities. Virtually anything the hardware is capable of doing is represented by an instruction. However, because it'd be silly to design a CPU that had to physically parse and interpret strings in order to read instructions, even short ones like "`Mov`" and "`Cmp`", the CPU won't literally see code like this:

```
Mov     X, Y
Add     X, Z
Div     Z, 2
```

Even though the previous example is written in assembly language, this still isn't the final step in creating an executable script. Remember, strings are handled by computers in a far less efficient manner than numeric data. The whole concept of digital computing is based on the processing of numbers, which is why binary data is, by nature, both faster and more compact than text-based/ASCII data.

I've mentioned before that assembly language is the lowest level language you can code in. This is true, but there is still another step that must be taken before your assembly code can be read by the VM. This step is performed by a program called an *assembler*, which, as you saw in Chapter 5, is to assembly what a compiler is to high-level code. An assembler takes human readable assembly source code and converts it directly into *machine code*. Machine code is a nearly exact, one-to-one conversion of assembly language. It describes programs in terms of the same instructions with the same operands in the same order. The only difference is that assembly is the text-based, human readable version, and machine code is expressed entirely with numbers.

To understand this concept of conversion better, think back to when you were a kid. If you were anything like me, you spent a lot of time sneaking around with your friends, on various deadly but noble missions to harass the girls of the neighborhood. Now neighborhood spying is risky business, and requires a secure method of communication in order to properly get orders to field agents without enemy forces intercepting the message. Because of this, we had to devise what is without a doubt the most foolproof, airtight method of encryption man has ever dared to dream of: *letter to number conversion.*

In a nutshell, this brilliant scheme (which I'll probably end up selling to the Department of Defense, so forget I mentioned this) involves assigning each letter of the alphabet a number. A becomes 0, B becomes 1, C becomes 2, and so on. A message like the following:

```
"Lisa is sitting on her steps with a book. This is clearly a vile attempt to thwart
our glorious mission. Mobilize all forces immediately. Use of deadly force (E.G.,
water balloons) is authorized. Godspeed."
```

could be encrypted by translating each letter to its numeric equivalent according to the code. The result is a string of numbers that expresses the *exact* same message while at the same time shedding its human readability (sort of). The code of course worked. Despite its simplicity, no one could crack it. However, it worked a bit too well, because not a lot of eight year olds have the patience to spend the 20 minutes it usually took to get through a few numerically encoded sentences, so we'd generally just get bored and go inside to play Nintendo. I think the nation truly owes a debt of gratitude to me and my friends for never pursuing careers with the CIA.

Getting back on track, my tale of nostalgia was intended to show you that the difference between assembly language and machine code is (usually) a purely cosmetic one. The data itself is the same in either case; the only difference is how it's expressed.

For example, take the following snippet of assembly:

```
Mov     X, Y
Add     X, Z
Div     Z, 2
```

If the goal is to reduce this code to a form that can be expressed entirely through numeric data, the first order of business should be assigning each instruction a unique integer code. Let's say `Mov` is assigned 0, `Add` is assigned 1, and `Div` is assigned 4 (assuming `Sub` and `Mul` take the 2 and 3 slots). The first attempt to reduce this to machine code will transform it into this:

```
0      X, Y
1      X, Z
4      Z, 2
```

Not too shabby. This is already a more efficient version because you've eliminated at least one third of the string processing required to read it. In fact, this is how things are really done—every assembler on earth really just boils down to a program that reads in instructions and maps them to numeric codes. Of course, these numeric codes have a name—they're called *opcodes*. "Opcode" is an abbreviation of Operation Code. This makes pretty good sense, because each numeric code corresponds to a specific operation, as you've seen. These are important terms, however, and a lot of people screw them up. Instructions can come in two forms; the numeric *opcode* that you've just seen, which is read by the VM, and the string-based *mnemonic*, which is the actual instruction name you've been using so far.

The remaining strings are mostly in the form of variable identifiers and literal values. Because the only literal value is 2 (the second operand of the `Div` instruction), which is already a number, you can leave it as-is. That means your next task is to reduce the variable names to numbers as well. Fortunately, this is easy too and follows a form very similar to the conversion of mnemonics to opcodes.

When virtually any language is compiled, whether it's assembly, C, or XtremeScript, the number of variables it contains is already known. New variables aren't created at runtime, which means that you have a fixed, known number of variables at compile-time. You can use this fact to help eliminate those names and replace them numerically. For example, the code snippet you've been working with in this example so far has three variables: `X`, `Y` and `Z`. Because the computer obviously doesn't care what the actual name of the variable is, as long as it can uniquely identify it, you can assign each variable a number, or index, as well. So, if `X` becomes 0, `Y` becomes 1, and `Z` becomes 2, you can further reduce the code to this:

```
0       0, 1
1       0, 2
4       2, 2
```

Cool, huh? You now have a version of your original code that, while retaining all of its original information, is now in an almost purely numeric code. There is one problem left, however, and that's all the spacing and commas. Because they, like instruction mnemonics and variable identifiers, exist only to enhance the script's readability, they too can be scrapped. Come to think of it, there's no need for line breaks either. In fact, this data shouldn't be expressed through text at all! All you really need is a stream of digits and you're done. Here's the previous code, condensed onto a single line with all extraneous spacing and commas removed:

```
001102422
```

As you can see, `001` represents the first instruction (`Mov X, Y`), `102` is the second instruction (`Add X, Z`), and `422` is the last (`Div Z, 2`). This final numeric string is the machine code, or *bytecode* as it's often called in the context of virtual machines. This isn't a perfect example of how an assembler works, but it's close enough and the concept should be clear. You'll put these techniques to real use in the next chapter, in which you construct an assembler for the assembly language you'll be designing shortly.

RISC versus CISC

So, now you understand how assembly language programming basically works and you have a good idea of the overall process of converting assembly to machine code. Throughout the last few pages you've had a lot of interaction with various instructions, from the arithmetic instructions (`Add` and `Mul`) to the conditional branching family (`Cmp`, `JG`, and so on). You now understand how these instructions work and how to reduce common C constructs to them, but where did they come from? Who decided what instructions would be available in the first place?

Because an instruction set is indicative of what a given CPU can do, deciding what instructions the set will offer is obviously an extremely important step in the design of such a machine. No matter what, there are always a number of basic instructions that virtually any processor, virtual machine, or runtime environment will offer. These are the basics: arithmetic, bit operations, comparisons, jumps, and so on and so forth. These are a lot like the basic elements of the programming languages you studied in the last chapter. Lua, Python, and Tcl may have strong differences between one another, but they all share a common "boiler plate" of syntax for describing conditional logic, iteration, and functions (among other things).

Beyond this basic set of bare-minimum functionality, however, is the possibility to add more features and instructions, in an attempt to make the instruction set easier to use, more powerful, or both. This is where the design of an instruction set splits into two starkly contrasting schools of thought—*RISC* and *CISC*.

Let's start with RISC first, which is an acronym for **R**educed **I**nstruction **S**et **C**omputing. RISC is a design methodology based on creating large instruction sets with many fine-grained instructions. Each instruction is assigned a small, simplistic task rather than a particularly complex one. Complex tasks are up to the programmer, as he or she must manually fashion more complicated algorithms and operations by combining many small instructions.

CISC, of course, is just the opposite. It stands for Complex Instruction Set Computing, and is based on the idea of a smaller instruction set, wherein each instruction does more. Programming tends to be easier for a CISC CPU, because more is done for you by each instruction and therefore, you have less to do yourself.

In the case of physical computing, the advantages of RISC over CISC are subtle but significant. First and foremost, the digital circuitry of a CPU must traverse a "list", so to speak, of hardwired instructions. These are the actual hardware implementations of instructions like `Mov` and `Add`. It doesn't take a PhD of computer science to know that a shorter list can be traversed faster than a longer one, so signals will be able to reach the proper instruction in a set of 100 faster than they can in a list of 2000 (see Figure 8.7). Furthermore, there is an overhead with executing an instruction just as there's an overhead involved in calling a function. If a CISC processor can perform a task in one instruction that a RISC would need to execute four instructions to match, the

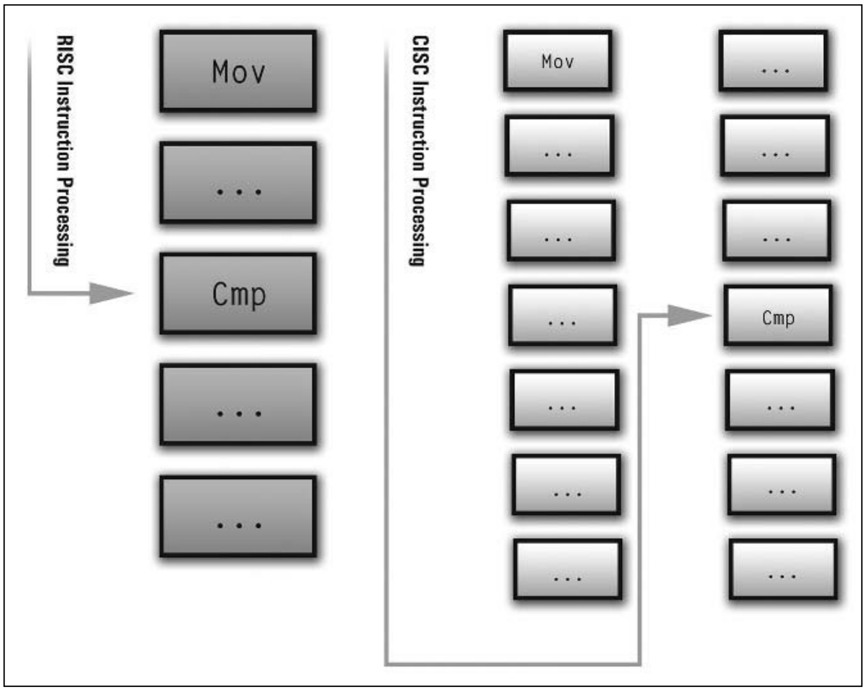


Figure 8.7

Short instruction lists can be traversed faster than long ones.

CISC system has reduced the overhead of instruction processing by a factor of four (despite the fact that the instruction itself will take longer to execute and be more complex on the CISC processor).

Electrical engineering is an interesting subject, but you're here to build a virtual machine for a scripting system, so let's shift the focus back to software. In a virtual context, CISC makes even more sense. This is true for a simple reason— scripting languages are *always* slower than natively compiled ones. Obviously, because even a compiled bytecode script has an entire layer of software abstraction between it and the physical CPU, everything it does will take longer than it would if it was written in C or C++. Because of this, a simple but vital rule to follow when designing the runtime environment for a scripting system is to *do as much as is humanly possible in C.* In other words, make sure to give your language all the luxuries and extra functions it needs. Anything you don't provide as a C implementation will have to be written manually in the other (slower) scripting language.

The moral of the story is that anything you can do in C *should* be done in C. The less the scripting language does, the better (or the faster, I should say). Even though conceptually speaking, scripting is a more intelligent way to code certain game functionality and logic due to its flexible nature, the reality is that performance is any game programmer's number one concern. The goal then, should be to strike a happy medium between a flexible language and as much hardcoded C

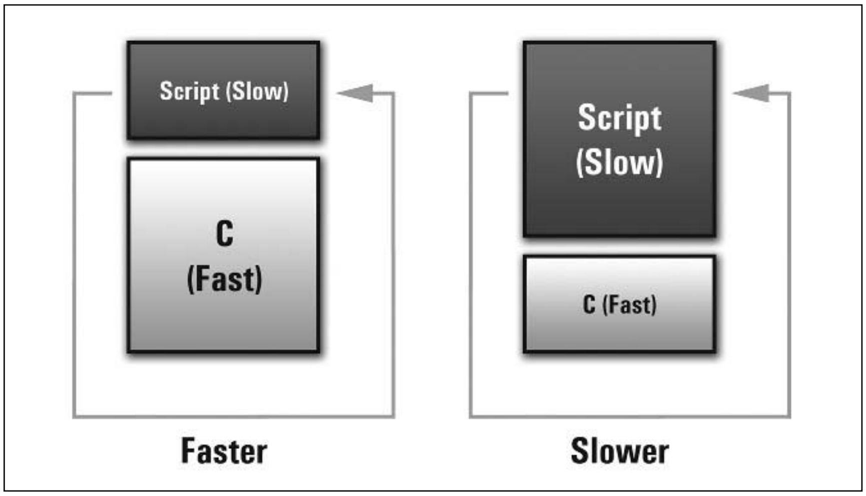


Figure 8.8

Two blocks of code. One spends more time in C, the other spends more time in the script. Obviously the first one will run faster.

as possible. You shouldn't do so much in C that you end up restricting the freedom of the scripts, because that'd defeat the whole purpose of this project, but you must remember that scripting involves significant overhead and should be minimized wherever possible.

Orthogonal Instruction Sets

In addition to the RISC versus CISC decision when designing an instruction set, another issue worth consideration is *orthogonality*. An instruction set is considered *orthogonal* when it's "evenly balanced", so to speak. What this means essentially is that, for example, an instruction for addition has a corresponding instruction for subtraction. Technically, subtraction can be defined as addition with negative numbers. You don't absolutely *need* a subtraction instruction to survive, but it makes things easier because you don't have to worry about constantly negating everything you want to subtract for use with an add instruction. In other words, it's the measure of how "complete" the instruction set is in terms of instructions that would logically seem to come in a group or pair, even if it's merely for convenience or completeness.

Orthnogonality can also extend to the functionality of certain instructions as opposed to the others they're logically grouped with. For example, the Intel 80X86 isn't totally orthogonal in its implementation of the basic arithmetic instructions, because of the difference in how the `Add` and `Sub` instructions work as opposed to `Mul` and `Div`. `Add` and `Sub` accept two operands, and add or subtract one from the other. `Mul` and `Div`, however, only accept a single operand and either multiply or divide its value by another value that's already been stored in a previously specified location (the `AX` register, to be technical, but I haven't discussed registers yet so don't worry if that doesn't make sense). This irregular design of such closely related instructions can be jarring to the programmer, so it's one of a few subtle details you'll be ironing out in the design of your own assembly language.

Registers

Before moving on, I'd like to address the issue of registers. Those of you who have some assembly experience might be wondering if the virtual machine of a scripting system has any sort of analog to a real CPU's register set. Before answering that, allow me to briefly explain what registers are to bring the unenlightened up to speed.

Simply put, *registers* are very fast, very compact storage locations that reside directly on the CPU. Unlike memory, which must travel across the data bus to reach the processor, and is also subject to the complexities and overhead of general memory access, registers are immediately available and provide a significant speed advantage. Assembly language programmers and compilers alike value registers quite highly; given their speed and limited numbers, they're a rare but precious commodity.

Without going into too much more detail, you can understand how important register usage is. As for their relevance to the XtremeScript Virtual Machine, however, registers are essentially useless. Remember, your entire virtual machine will exist in the same memory address space; no single part of it is any faster or more efficient than any other. As a result, the memory model within the XVM will be a simple, stack-based scheme with some additional random access capabilities. Defining a special group of "registers" would accomplish nothing, as they'd provide no practical advantage over anything else.

NOTE

Speed and simplicity aren't the only advantages of registers, however. Often, registers are utilized simply because they're accessible in the same way from all parts of an assembly language program, regardless of scope or function nesting. As a result, they're often a convenient way to pass simple data from one block of code to another that, for whatever reason, would be difficult with conventional memory. For this reason, you just may find a use for registers in the XVM yet.

The Stack

At this point you've learned how to do a lot with assembly, at least conceptually. In fact, you understand almost all of the major conversions between the structures and facilities of high-level languages to low-level ones, like expressions, branches, and loops. What I haven't discussed yet are functions, however. For this, you'll need to understand the concept of a *runtime stack*.

Most runtime environments, whether they're virtual or physical machines, provide some sort of a runtime stack (also known simply as a *stack*). The stack, due to its inherent ability to grow and shrink, as well as the rigid and predictable order in which data is pushed on and popped off, make it the ideal data structure for managing frequently changing data—namely, the turbulent behavior of function calls.

As your typical high-level program runs, it's constantly making function calls. These functions tend to call other functions. Recursive functions even call themselves. Altogether, functions and the calls to and between them "pile up" as their nesting grows deeper and deeper, and eventually unravel themselves. Luckily for you, this is exactly how a stack works.

To understand this better, first think about how a function is called in the first place. If you envision your compiled script as a simple array of instructions, with each instruction having a unique and sequential index, the actual location of a given instruction or block of instructions can be expressed as one of those indices. So, in order to call a function, you need to know the index of the function's first instruction in the array, known as the function's *entry point*. You then need to branch to this instruction, at which point the function will begin executing. This sounds like a typical jump instruction, right?

So far, so good. From here, the runtime environment will start executing the function's code just like it would anything else. But wait—how will the runtime environment know when the function is finished? Furthermore, even if it does know where the function ends, how will it know how to get back to the instruction that called it? After all, functions have to return the flow of execution to their callers. You can't just use a jump instruction to move back to the index of the instruction that called you, because you don't know where that is. Besides, functions can be called from anywhere in the code, which means you can't have a hardcoded jump back to a specific instruction. This would allow you to call the function from only that one place. See Figure 8.9.

Let's solve the second problem first. Once you know a function is over, how do you get back? Unfortunately, I'm asking this question at the wrong time. I should've planned for this before jumping to the function in the first place, which would've made things much easier. So, let's go back in time a few nanoseconds to the point at which you make the call and think for a moment. In order for the function you're about to invoke to know how to find its way back to you, you need to give it the index of the instruction that's actually making the call. Just as the function's entry point is defined as the index of its first instruction, the *return address* is defined as the index of the function that it needs to return when it's done. So, before you make the call to the function, you need to push the return address onto the stack. That way, the function just has to pop the top value off the stack to determine where it's going when it returns.

Before moving on, I'll quickly introduce the instructions most CPUs provide for accessing the stack. As you might guess, they're called `Push` and `Pop`. `Push` accepts a single value and pushes it onto the stack. `Pop` accepts a single memory reference and pops the top stack value into it. The

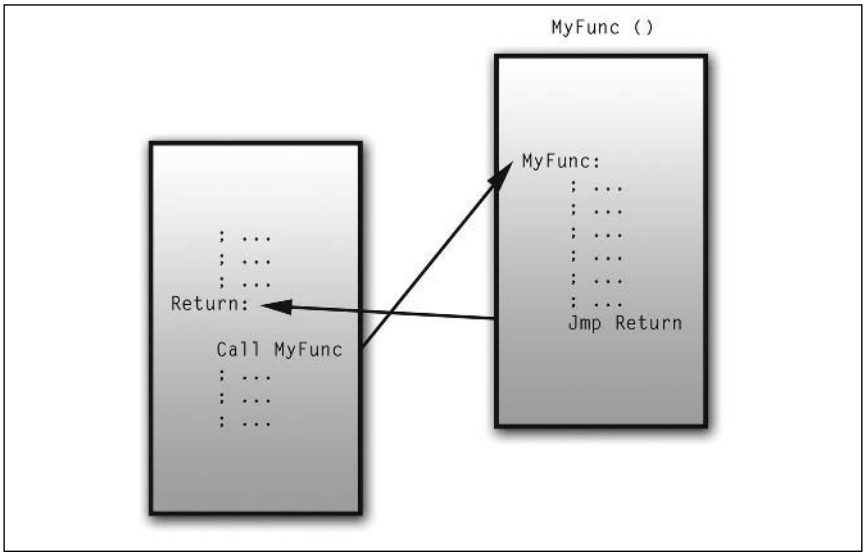


Figure 8.9

Functions can't simply jump back to a specific instruction, as this would bind their use to one place rather than making them available to the whole program.

stack itself is a global structure, meaning it's available to all parts of the program. That's why you can push something on before calling a function and still access it from within that function. Figure 8.10 shows general stack use in assembly.

Getting back on track, you don't need to "mark" the end of the function. Instead, you can just end it with another jump—one that jumps back to the return address. In fact, there are usually two instructions specifically designed just for this task: `Call` and `Ret`.

`Call` is a lot like `Jmp` in the sense that it causes the flow of execution to branch to another instruction. However, in addition to simply making an unconditional jump, it also pushes the current instruction index (which is its own index, as well as the return address) plus one onto the stack. It

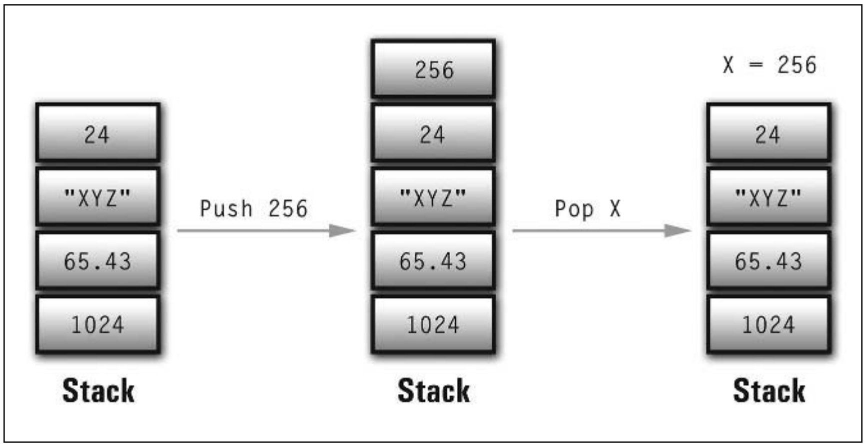


Figure 8.10

Pushing and popping values in assembly to the runtime stack.

adds one to its own address to make sure the function returns to the *following* instruction, not itself; otherwise you'd have an infinite loop on your hands. `Ret`, on the other hand, is a bit different. It also performs an unconditional jump, but you don't have to pass it a label. Instead, it jumps to whatever address it finds on the top of the stack. In other words, `Ret` pops the value off the top of the stack and uses it as the return address, expecting it to take it back to the caller. And if all goes well, it does. Together, `Call` and `Ret` expand on the simplistic jump instructions to provide a structured method for implementing functions.

And here's the best part to all of this—because you've used a stack to store return addresses, which grows and shrinks while automatically preserving the order of its elements, the function calls are inherently capable of nesting and recursion. If a new function is called from within a previously called function, the stack just grows higher. It grows and grows with each nested call, until finally the last call returns. Then, it slowly begins to shrink again, as each return address is subsequently popped back off. Because the functions were called in a sequential order, which was intrinsically preserved by the stack, they can return in the opposite of that order and be confident that the return addresses will always be the right ones. Figure 8.11 illustrates this concept.

CAUTION

There is one catch to this stack-based function call implementation. Because `Ret` assumes that the top value of the stack contains the return address, which it does at the time the function is invoked, the function itself *must* preserve the stack layout. This is done in two ways—either the function simply doesn't touch the stack at all, or it makes sure to pop all values it pushes before `Ret` executes to make sure that the original top of the stack, containing the return address, is once again on top, where it should be.

Stack Frames/Activation Records

Everything seems peachy so far, but there's one important issue I haven't yet discussed— parameters and return values. You've figured out how to use the stack to facilitate basic function calls, but functions usually want to pass data to and from one another. This will undoubtedly complicate things, but fortunately it's still a pretty straightforward process.

When a function passes parameters to another function, it's basically a way of sending information, which is something you've already done. Currently, your implementation of functions is capable of sending the return address to the function it calls, which is kind of like sending a single parameter at all times, right? So, as you might already be thinking, you can pass parameters in the exact same way—by pushing them onto the stack along with the return address.

When a function is called, its parameters are first pushed in a given order; either left-to-right or right-to-left. It doesn't matter which way you do it, as long as the function you're calling is expect-

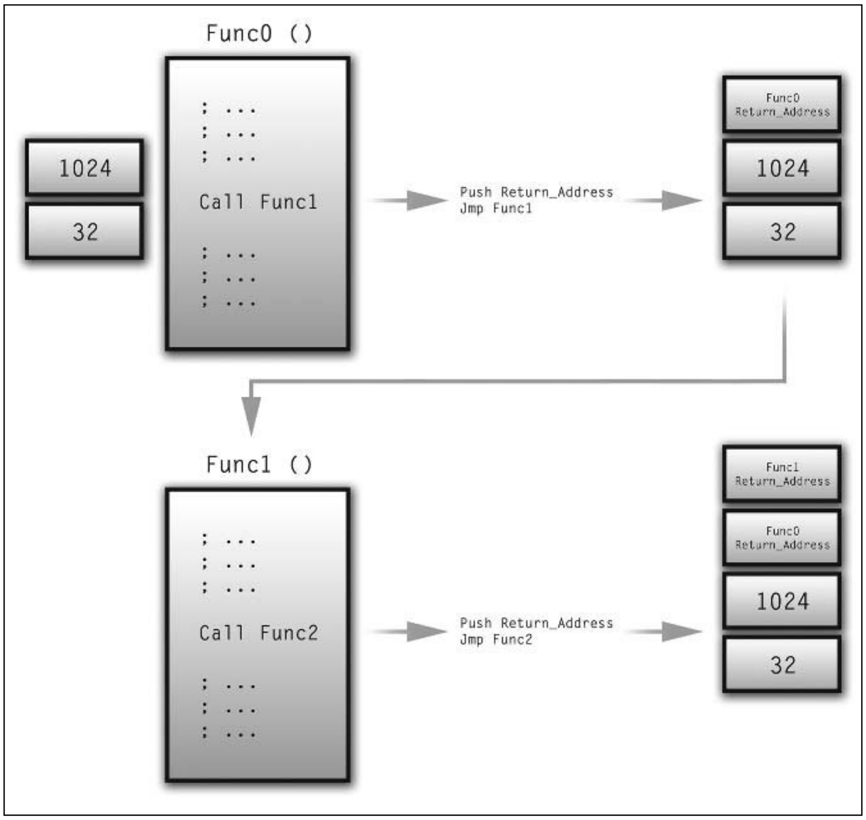


Figure 8.11

Using a stack to manage return addresses gives you automatic support for nested calls and recursion. It's stacktastic!

ing whichever method you choose. Following the parameters, the return address is pushed, as already discussed. The function is then invoked, and execution begins at its entry point. As the function executes, it will of course refer to these parameters you've sent it, which means it'll need to read the stack. Rather than pop the values off, however, it'll instead access the stack in a more arbitrary way; each parameter's identifier is actually just a symbol that represents an *offset* into the stack. So for example, if you have a function whose prototype looks like this:

```
Func MyFunc ( X, Y, Z );
```

it receives three parameters. If these parameters are pushed onto the stack, they can be accessed relative to the top of the stack. If the data you push onto the stack before the function is called is in this order:

```
Parameter X
Parameter Y
Parameter Z
Return Address
```

it'll be found in the reverse order if you move from the top of the stack down. The return address will be at the top, with everything else following it, so it'll look like this:

```
Return Address
Parameter Z
Parameter Y
Parameter X
```

This means that return address is at the top of the stack, `Z` is at the top of the stack minus 1, `Y` is at the top of the stack minus 2, and `X` is at the top of the stack minus 3. These are *relative stack indices*, and are used heavily within the code for a function. Remember, because of the way a stack works, the order in which you push parameters on means they'll be accessed in the *opposite* order. So, if the caller pushes them in `X`, `Y`, `Z` order, the function has to deal with them in `Z`, `Y`, `X` order. This is why I make a distinction between left-to-right and right-to-left parameter passing; you should decide whether you want the functions or the callers to be able to deal with parameters in their formally declared order.

NOTE

I recommend pushing function parameters onto the stack in the right-to-left order. Although this does mean the function itself will have to refer to its parameters in reverse order, it also means that every time you call the function, you can push the parameters in an order that makes intuitive sense. I always favor the caller over the function for a simple reason—you'll write code to call a given function countless times, but you'll write the function itself only once. Besides, as you'll see later, you'll design the assembly language syntax in a way that makes this easy.

Of course, when the function returns, there will be three stack elements that need to be popped back off (corresponding to the three variables you pushed on before the call). Normally, this would be the responsibility of the caller (because they put them there to begin with), but it's quite a hassle to have to follow every function call with a series of `Pop` instructions. As a result, the `Ret` instruction usually lets you pass a single parameter corresponding to how many stack elements you'd like it to automatically pop off. So, the three-parameter function would be with the following instruction:

```
Ret    3    ; Clean our 3 parameters off the stack
```

As you'll see, you will design your own assembly language to support this automatic stack cleanup, but in an even easier way.

We can pass parameters now, so what about return values? If parameters can be passed on the stack, return values can too, right? Well, it'd certainly be possible if your stack was laid out differently, but unfortunately the current implementation wouldn't support it. Why? Because the only way to a pass return value on the stack would involve the function pushing it with the intention of the caller popping it back off. Unfortunately, you'd push this value *after* the parameters and return address, meaning the return value would now be above everything else, on the top of the stack. The problem is that once the `Ret` instruction is executed, it'll attempt to restore the stack to the way it was before the function was called by popping the parameters and return address off. Inadvertently, this would end up prematurely popping the return value, and worse, only popping off parts of the parameter list and therefore leaving a corrupted stack for the caller to deal with.

So if the stack is out, what can you do? Aside from the stack, there aren't any storage locations that persist between function calls, which means there isn't really any common space the caller and function can share for such a purpose. To solve this problem let's look at what the 80X86 does.

The 80X86, unlike your culminating virtual machine, has a number of general-purpose registers. These registers provide storage locations that are always accessible from all parts of assembly language program, regardless of scope. Therefore, in order to return a value from a function to its caller, one merely has to put that value into a specific register, allowing the caller to read from it once the function returns. On the Intel platform, it's convention to use the accumulator `AX` (or `EAX on 32-bit platforms`) register for just this task (even compilers output code that follows this). So, a simple `Mov` instruction would be used to fill `AX` with the proper value, and the return value would be set. The caller then grabs the value of `AX`, and the process is complete. The only problem is that I've already stated that your VM will not include registers. This is true, at least in the case of general-purpose registers, but you will have to bend this rule just a bit in order to add a single register for this specific purpose of transporting return values.

The implementation of stacks is now somewhat more complex; rather than simply assigning a return address to each function as it's represented on the stack, you also have to make room for parameters. Things are no longer a matter of a simple push onto the stack; rather, they're beginning to take on the feel of a full data structure. You now have an implementation of function calls such that each time a call is made, a structure is set up to preserve the return address, passed parameters, and more information, as you'll see in the next section. This structure is known as a *stack frame*, or sometimes as an *activation record*. In essence, it's a structure designed to maintain the information for a given function. Figure 8.12 shows the concept of stack frames graphically.

Local Variables and Scope

So you can call functions and pass parameters via the stack, as well as return values with a specific register. What about the code of a function itself? Naturally, the code resides in a single place and is more or less unrelated to the stack. However, there is the matter of local variables to discuss.

Let's start by imagining a recursive function. Because this function will be calling itself over and over, you'll quickly reach a point where multiple instances of this function exist at once; for

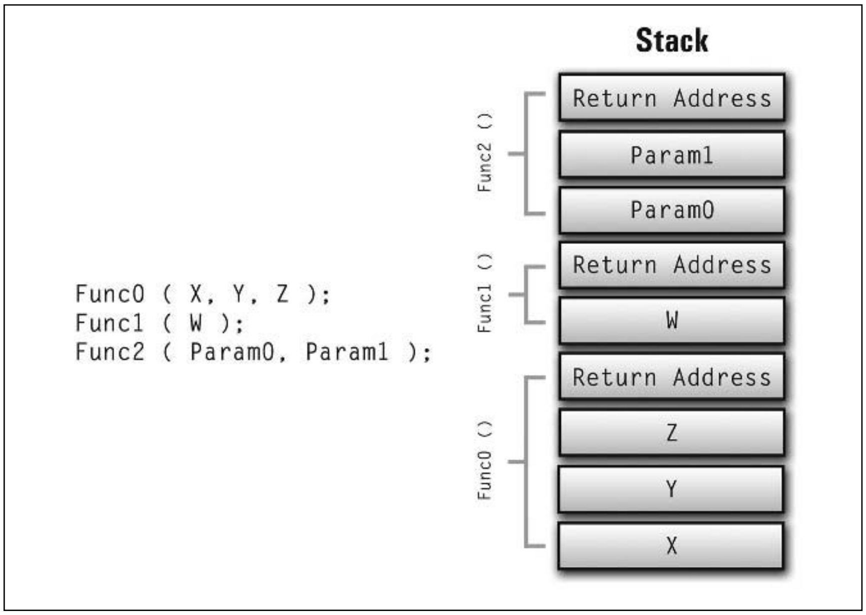


Figure 8.12

Stack frames (also known as activation records) now contain return addresses and parameter lists.

example, the function might be nested into itself six levels deep and thus have six stack frames on the stack. The code for the function is not repeated anywhere, because it doesn't change from one instance to the next. However, the data the function acts upon (namely, its locally defined variables) *does* change from one instance to another quite significantly. This is where a function's stack frame must expand considerably.

You're already storing a return address and the passed parameters, but it's time to make room for a whole lot more. Each instance of a function needs to have its own variables, and because you've already seen that the stack is the only intelligent way to manage the nested nature of function calls, it means that the reasonable place to store local variables themselves is on the stack as well. So now a stack frame is essentially the location at which *all* data for a given function resides. Check out Figure 8.13 for a more in-depth look at stack frames.

In fact, because the average program spends the vast majority of its time in functions (or even all of its time, in the case of languages like C which always start with `main ()`), this means you've decided on where to store virtually all of the script's data. All that's left are global variables and code that resides in the global scope (outside of functions). This, however, can be stack-based as well; data that resides in the global scope can be stored at the bottom of the stack. Therefore, only parameters and local variables are accessed relative to the top of the stack, with negative indices like `-1`, `-2`, `-3` and so on, globals are relative to the bottom of the stack, with indices like `0`, `1` and `2` (remember, negative stack indices are relative to the top of the stack, whereas positive are relative to the bottom).

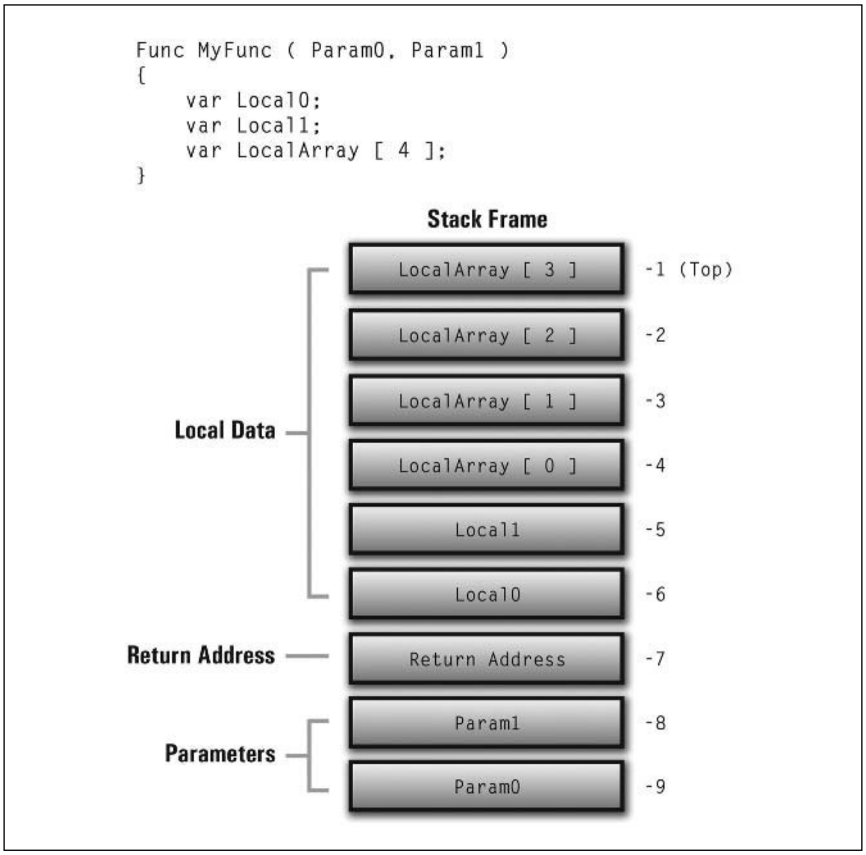


Figure 8.13

Stack frames represent an entire function in terms of its runtime data (local variables, parameters, and the return address).

All in all, this section is meant to show you how important the stack is when discussing runtime environments. Your language won't support dynamically allocated data, which means that the only structure you need to store an entire script's variables and arrays is a single runtime stack (in addition to a single register for returning values from functions to callers). In addition, it will manage the tracking and order of function calls, as well as provide a place for intermediate values during expression parsing. What this should tell you is that with few exceptions, the concept of "variables" in general is just a way of attaching symbolic names to what are really just stack indices relative to the current stack frame.

In a lot of ways, the runtime stack is the heart of it all.

INTRODUCING XVM ASSEMBLY

So where does this leave you? You're at a point now where you understand quite a bit about assembly language, so you might as well get started by laying out the low-level environment of our

XtremeScript system. You'll get started on that in this chapter by designing the assembly language of the XtremeScript virtual machine, which I like to call *XVM Assembly*.

XVM Assembly is what your scripts will ultimately be reduced to when you run them through the XtremeScript compiler that you'll develop later on in this book. For now, however, it'll be your first real scripting language, because within the next few chapters you'll actually reach a point where it becomes useable.

Because of this, you should design XVM Assembly to be useable by human coders. This will allow you to test the system in its early stages by writing pure-assembly scripts in place of higher-level ones. Of course, at the same time, the language must also be conducive to the compiler, so you'll need enough instructions to reduce a C-style language to it.

Initial Evaluations

Let's get started by analyzing exactly what the language needs to do. Fortunately, you spent the last chapter creating the high-level language that XVM Assembly will need to support, so you've got your requirements pretty well cut out for you.

First of all, XtremeScript is typeless, and has direct support for integers, floats, and strings (it also supports Booleans but let's treat `true` and `false` internally as the integer values 1 and 0, respectively). You could make the assembly language more strongly typed, letting it sort out the various storage requirements and casting necessary to manage each of these three data types in an efficient way, but that'd be an unnecessary hindrance to performance. There's no need to manually manage the different data types in terms of their actual binary representation in memory when you can just get C to do the majority of the work for you. So, you can make your assembly language typeless too. This means that even in the low-level code you can directly refer to integers, floats, and strings, without worrying about how it's all implemented. You can leave that all up to the runtime environment, which of course will be pure C and very fast. Code like the following will not be uncommon in XVM Assembly (although you certainly wouldn't find anything like this on a real CPU!):

```
Mov    MyInt, 16384
Mov    MyFloat, 123.456
Mov    MyString, "The terrible secret of space!"
```

As long as I'm on the subject of data, I should also cover XtremeScript arrays. This is another case where you could go one of two ways. On the one hand, you could provide assembly language scripts with the ability to request dynamically allocated memory from the runtime environment and use that to facilitate the translation of high-level arrays to low-level data structures, but as you'll see in the section on designing the XVM, you're better off not allowing dynamic allocation. Therefore, even the assembler must statically allocate arrays, and should therefore have array functionality built-in. So, in addition to variable references like this:

```
Mov    X, Y
```

XVM Assembly will also directly support array indexing like this:

```
Mov    X, MyArray [ Y ]
```

I'll talk about how to declare arrays a bit later.

The last real issue regarding data is how various instructions will interpret different data types. For example, `Div` is used to divide numeric values, so what happens if you try to divide 64 by a string? You have three basic choices in a situation like this:

- Halt the script and produce a runtime error.
- Convert data to and from data types intelligently. For example, dividing by the string value "`128`" would convert the string temporarily to the integer value 128.
- Silently nullify any bad data types. In other words, passing a numeric when a string was expected will convert the number temporarily to an empty string. Likewise, passing a string when a numeric was expected will temporarily replace the string with the integer value zero.

This is more an issue for the virtual machine design phase, but it will still have something of an effect on how you design the language itself. For now, let's defer the decision on exactly how data types will be managed until later, but definitely agree that you'll go with one of the second two choices. Rather than forcibly stop the coder from passing incorrect data types as operands to instructions with runtime errors, you'll allow it and choose a graceful method for handling it in a couple of chapters.

The XVM Instruction Set

The rest of the language description is primarily a run down of the instruction set, so what follows is such a reference, organized by instruction family. Also worth noting is that, just as you based the syntax for XtremeScript heavily on C, the XVM Assembly Language is strongly based on Intel's 80X86 syntax, although I will mention a few creative liberties I've taken to make various instructions more intuitive or convenient.

Memory

```
Mov           Destination, Source
```

The first and most obvious instruction, as always, is `Mov`. Every assembly language has some sort of general-purpose instruction for moving memory around, or a small group of slightly more specialized ones. One thing to note about `Mov`, however, is that its name is somewhat misleading. The instruction doesn't actually *move* anything, in the sense that the `Source` operand will no longer exist in its original location afterwards. A more logical name would be `Copy`, because the result of the instruction is two instances of `Source`. Expect `Mov` to be your most commonly used instruction, as it usually is in assembly programming.

As for restrictions on what sort of operands you can use for `Source` and `Destination`, `Source` can be anything—a literal integer, float, or string value, or a memory reference (which consists of variables and array indices). `Destination`, on the other hand, must be a memory reference of some sort, as it's illegal to "assign" a value to a literal. In other words, `Destination` really follows the same rules that describe an L-Value in C.

Arithmetic

```
Add          Destination, Source
Sub          Destination, Source
Mul          Destination, Source
Div          Destination, Source
Mod          Destination, Source
Exp          Destination, Power
Neg          Destination
Inc          Destination
Dec          Destination
```

The next most fundamental family of instructions is probably the arithmetic family. These functions, with the exception of `Neg`, follow the same operand rules as does `Mov`. In other words, `Source` can be any sort of value, whereas `Destination` must be a memory reference of some sort. These instructions work both on integer and floating-point data without trouble.

The three newcomers here are `Mod`, `Exp`, and `Neg`. `Mod` calculates the *modulus* of two numbers; that is, the remainder of `Destination / Source`, and places it in `Destination`. `Exp` handles exponents, by raising `Destination` to the power of `Power`. Lastly, `Neg` accepts a single parameter, `Destination`, which is a memory reference pointing to the value that should be negated.

This family of instructions is another example of the CISC approach you're taking with the instruction set; although there are actually more instructions here than are usually supplied for arithmetic on real CPUs, the VM will perform all of the operations that will be directly needed by the set of arithmetic operators XtremeScript supports. Imagine, for example, that you didn't provide an `Exp` instruction, but left the `^` (exponentiation) operator in XtremeScript anyway. When code that uses the operator is compiled down to assembly, you'll have no choice but to manually calculate the exponent using XVM assembly itself. This means you'd have to perform a loop of repetitive multiplication. This would be significantly slower than simply providing an Exp instruction that takes direct advantage of a far-faster C implementation. These extra instructions are good examples of how to offload more of the work to C, while preserving the flexibility of the scripting language.

NOTE

Users of Intel 80X86 assembly language will be happy to see the changes made to Mul and Div, which are now as easy to use and side-effect free as Add and Sub. Due to the language not being dependent on registers, you can be much more flexible in your definition of instructions, and therefore can avoid the small headaches sometimes associated with these two instructions on the 80X86. This is also an example of improving orthogonality.

Lastly, I've included the Inc and Dec instructions to round out the arithmetic family. These simple instructions increment and decrement the value contained in Destination, and are analogous to C's ++ and -- operators. Once again this a subtle example of the CISC approach; since a general purpose subtraction instruction is slightly more complicated than one that always subtracts one, we can (at least theoretically) improve performance by separating them.

Bitwise

```
And           Destination, Source
Or            Destination, Source
XOr           Destination, Source
Not           Destination
ShL           Destination, ShiftCount
ShR           Destination, ShiftCount
```

Up next is the XVM family of bitwise instructions. These instructions allow common bit manipulation functions to be carried out easily, and once again directly match the operator set of XtremeScript. These instructions are similar to the arithmetic family, and therefore also similar to Mov, in terms of their operand rules. All Destination operands must be memory references, whereas Source can be pretty much anything. Note that bitwise instructions will only have meaningful results when applied to integer data.

The rundown of the instructions is as follows. And, Or, XOr (eXclusive Or), and Not perform their respective bitwise operations between Source and Destination. ShL (Shift Left) and ShR (Shift Right) shift the bits of Destination to the right or left ShiftCount times.

String Processing

```
Concat         String0, String1
GetChar        Destination, Source, Index
SetChar        Index, Destination, Source
```

XtremeScript is a typeless language with built-in support for strings. In another example of a CISC-like design decision, I've chosen to provide a set of dedicated string-processing functions for easy manipulation of string data as opposed to simply providing a low-level interface to each character of the string. Especially in the case of string processing, allowing a C implementation to be directly leveraged in the form of the previous instructions is *far* more efficient (and convenient) than forcing the programmer to implement them in XVM Assembly.

The `Concat` instruction concatenates two strings by appending `String1` to `String0`. `GetChar` extracts the character at `Index` and places it in `Destination`. `SetChar` sets the character in `Destination` and `Index` to `Source`. All indices in XtremeScript are zero-based, which holds true for strings as well.

Conditional Branching

```
Jmp            Label
JE             Op0, Op1, Label
JNE            Op0, Op1, Label
JG             Op0, Op1, Label
JL             Op0, Op1, Label
JGE            Op0, Op1, Label
JLE            Op0, Op1, Label
```

The family of jump instructions provided by the XVM closely mimics the basic 80X86 jump instructions, with one major difference. Rather than provide a separate comparison instruction like the `Cmp` instruction I talked about earlier, all of the XVM's jumps have provisions for evaluating built-in comparisons. In other words, the operands you'd like to compare, the method of comparison, and the line label to jump to are all included in the same line. This approach to branching has a number of advantages, so I decided to change things around a bit.

NOTE
Line labels in XVM assembly are declared just as they are in 80X86 and C: with the label name itself and a colon, (like "`Label:`"). Labels can be declared on their own lines, or on the same line as the instruction they point to. You'll see more of this later.

`Jmp` performs an unconditional jump to `Label`, whereas the rest perform conditional jumps based on three criteria—`Op0`, `Op1`, and the type of comparison specified in the jump instruction itself, which are as follows: Jump if Equal (`JE`), Jump if Not Equal (`JNE`), Jump if Greater (`JG`), Jump if Less (`JL`), Jump if Greater or Equal (`JGE`), and Jump if Less or Equal (`JLE`). In all cases, `Label` must be a line label.

The Stack Interface

```
Push          Source
Pop           Destination
```

As you have learned, the runtime stack is vital to the execution of a program. In addition, this stack can be used to hold the temporary values discussed earlier when reducing a high-level expression like `X + Y * ( Z / Cos ( Theta ) ) ^ Pi` to assembly.

Fortunately, the stack interface is pretty simple, as it all just comes down to pushing and popping values. `Push` accepts a single operand, `Source`, which is pushed onto the stack. `Pop` accepts a single operand as well, `Destination`, which must be a memory reference to receive the value popped off the stack. Unlike on the 80X86, `Push` can be used with literal values, not just memory references.

The Function Interface

```
Call          FunctionName
Ret
CallHost      FunctionName
```

Functions are (almost) directly supported by XVM Assembly, which makes a number of things easier. First of all, it lets you write assembly code in a very natural way; you don't have to manually worry about relative stack indices and other such details. Furthermore, it makes the job of the compiler easier as well, because high-level functions defined in XtremeScript can be directly translated to XVM assembly.

A function can be called using the `Call` instruction, which pushes the return address onto the stack and makes a jump to the function's entry point. `FunctionName` must be a function name defined in the file, just as the parameter to a jump instruction must be a line label.

`Ret` does just the opposite. When called, it first grabs the return address from the current stack frame, and then clears it off entirely and jumps back to the caller. The cool thing about `Ret` is that it's usually optional, as you'll see when I discuss function declarations. Like `return` in C, you need to use `Ret` only if you're specifically returning from a specific area in the function. Most of the time, however, the function will simply end by "falling through" the bottom of its code block.

Lastly, there's `CallHost`. This instruction takes a function name as well, just like `Call`, except that the function's definition isn't expected in the script. Rather, it's assumed that the host API will provide a registered function of the same name. Without going into too much more detail, you can safely assume that this is how XtremeScript interacts with the host API. You'll find that this approach is rather similar to the scripting systems discussed in Chapter 6. I'll discuss the exact nature of the host interface in the coming chapters.

NOTE

As I mentioned, return values from function calls are facilitated via a register set aside from just this task. Actually using this register is very simple; it appears to be just another global variable called `_RetVal`. `_RetVal` can be used in all the same places normal variables can, and maintains its value from function call to function call.

Miscellaneous

```
Pause          Duration
Exit           Code
```

Lastly, there are a few extra instructions worth mentioning that didn't really have a home in any of the other categories.

The first is `Pause`, which can be used to pause the script's execution for a specified duration in milliseconds (provided by the `Duration` operand). The difference between the `Pause` instruction and a simple empty loop is that the host application, as well as any other, concurrently running scripts, will continue executing. This makes it useful for various issues of timing and latency wherein the script needs to idle for a given period without intruding on anything else. The `Duration` operand can be either a literal value or a memory reference, which means the `Pause` duration can be determined at runtime (which is useful).

The last instruction is `Exit`, which simply causes the script to unconditionally terminate. I also decided to add the `Code` operand on a whim, which will give you the ability to return a numeric code to the host application for whatever reason. I can't think of any real purposes for it just yet, but you never know— it just might save your life someday. :) Regardless, `Exit` is not required; scripts will automatically terminate on their own when their last instruction is reached.

XASM Directives

The XASM Assembler, of course, is primarily responsible for reducing a series of assembly language instructions to their purely numeric, machine code equivalent. However, in order to do its job in full, it needs a bit more information about the script it's compiling, as well as the executable script it will ultimately become. For example, how much stack space should be allocated for the script? What are the names of the script's variables and arrays, and how big should the arrays be? And perhaps most importantly, which code belongs to which functions?

All of these questions can be answered with *directives*. A directive is a special part of the script's source code that is not reduced to machine code and therefore is not part of the final executable. However, the information a directive provides helps the assembler shape the final version of the machine code output, and is therefore just as important as the source code itself in many ways. Directives will be used in the case of XVM Assembly to set the script's stack size, declare variables and arrays, and mark the beginning and ends of functions. Ultimately, directives help turn otherwise raw source code into a fully structured script.

Stack and Data

The first group of directives you'll explore relate to stack and data, which are closely linked (as you'll see soon). The first, `SetStackSize`, is the simplest and is solely responsible for telling the XVM how big a stack the script should be allocated. Here's an example:

```
SetStackSize    1024
```

When loaded and run, the executable version of the script will be given `1024` stack elements to work with. This is the same idea behind `lua_open ()` (see Chapter 6), which accepted a single stack size parameter for the script. This directive is optional, however. Omitting it will cause the script to ask for zero bytes, which is a code to the XVM to use whatever default value has been configured (it won't actually allocate it a zero-byte stack).

Next up is the data the script will operate on. As you learned in the last chapter, scripts operate on two major data structures: simple variables and one-dimensional arrays. First up are variables, which can be declared like this:

```
var MyVar0
var MyVar1
var MyVar2
```

For simplicity's sake, I decided against the capability to declare multiple variables on one line.

Of course, you'll often need large blocks of data to work with, rather than just single variables, so you can use the `[]` notation to create arrays of a given size:

```
var MyArray0 [ 16 ]
var MyArray1 [ 8192 ]
```

Variables and arrays can be declared both inside and outside of functions. Those declared outside are automatically considered global, and those declared elsewhere are considered local to wherever the place of that declaration may be.

Functions

The instruction set lets you write code, the `var` directives let you statically allocate data, so all that's really left is declaring functions. The `Func` directive can be used to "wrap" a block of code that collectively is considered a function with somewhat C-style notation. Here's an example:

```
Func Add
{
    Param   Y
    Param   X
    Var     Sum
    Mov     Sum, X
    Add     Sum, Y
    Mov     _RetVal, Sum
}
```

This code is of course an example of a simple `Add` function. Note that the `Func` directive doesn't allow the passing of formal parameters, but you can use the `Param` directive to make things easier (I'll get to `Param` in a moment). Notice that the return value is placed in `_RetVal`, which allows you to pass it back to the caller. Furthermore, note the lack of a `Ret` instruction, as I mentioned. `Ret` will be automatically appended to your function's code by the assembler, so you have to add it only when you want to exit the function based on some conditional logic.

The `Param` directive is required for accessing parameters on the stack. Each call to `Param` associates the specified identifier with its corresponding index within the parameter list section of the stack frame. So, if two parameters are pushed onto the stack before the call to `Add`, the following code:

```
Param    Y
Param    X
```

Would assign the second parameter to `Y` and the first parameter to `X` (remember the reversal of parameter order from within the function due to the LIFO nature of a stack). We'll see more about why this works the way it does in the next chapter, but for now, understand that without `Param`, parameters cannot be read from the stack.

Once a function has been declared with `Func`, its name can be used as the operand for a `Call` instruction.

Escape Sequences

Because game scripting often involves scripted dialogue sequences, it's not uncommon to find a heavy use of the double quote (") symbol for quotes. Unfortunately, because strings themselves are delimited with that same symbol, you need a way for the assembler to tell the difference between a quotation mark that's part of the string's content, and the one that marks the string's end. This is accomplished via *escape sequences*, also sometimes known as *backslash codes*.

Escape sequences are single- and sometimes multi-character codes preceded by a backslash (`\`). The backslash is a sign to the assembler that whatever character (or specially designated sequences of characters) immediately follows is a signal to do something or interpret something differently, rather than just another character in the string. Here's an example:

```
Mov    Quote, "General: \"Troops! Mobilize!\""
```

Here, the otherwise problematic quotation marks surrounding the `General`'s command are now safely interpreted by the assembler for what they really are. This is because any quotation mark preceded by a backslash is actually output to the final executable as quotation mark alone, so the final string will look like this:

```
General: "Troops! Mobilize!"
```

Just as intended. Of course, this brings up the issue of the backslash itself. If it's used to mark quotation marks, how do you simply use a backslash by itself if that's all you want? All you need to do is precede the backslash you want with another backslash, and that's that. For example:

```
Push "D:\\Gfx\\MySprite.bmp"
```

Of course, this ends up forcing you to use twice the amount of backslashes you need, but it's worth it to solve the quotation mark issue.

Comments

Lastly, I decided to throw comments into this section as well. Comments really aren't directives themselves, but I figured this was as good a place as any to mention them. Like most assemblers, XVM has a very simple commenting scheme that uses the semicolon to denote a single-line comment, like so:

```
; This is a comment.
Mov    Y, X    ; So is this.

; This is a
; multi-line
; comment.
```

Summary of XVM Assembly

You've covered a lot of ground here in a fairly short space, so here are a few important bullet points to remember just to make sure you stay sharp:

- *Assembly language* and machine code are basically the same thing; the only real difference is how they're expressed. Assembly is the human readable version that is fed to the assembler, and machine code is the purely numeric equivalent that the assembler produces. This is the version your virtual machine will actually read and execute.
- *Instructions* can be expressed in two ways: as a human readable *mnemonic*, such as "`Mov`" and "`Ret`", or as numeric *opcodes*, which are simply integer values.
- Instructions accept a variable-number of *operands*, which help direct the instruction to perform more specific actions.
- Conditional logic and iteration are handled exclusively with jump instructions and line labels.
- The RISC versus CISC debate centers upon how complex an instruction set is, in regards to the functionality of each instruction. CISC instruction sets can be faster in many applications, and was the chosen methodology for the design of the XVM instruction set.
- An instruction set's *orthogonality* is a measure of how complete the set is in terms of instructions that can be logically paired or grouped. XVM Assembly is designed to be reasonably orthogonal.
- The XVM Assembly instruction set is based on a somewhat reworked version of Intel 80X86 assembly, although it has almost no notion of registers because they wouldn't provide any of their physical advantages in the virtual context of the XVM. The `_RetVal` register is provided, however, for handling function return values.
- Expressions, which are ubiquitous and vital to high-level languages, don't exist in assembly and are instead reduced to a series of single instructions. Expressions often use the *stack* to store temporary values as these instructions are executed, which allows them to keep track of the overall result.
- The *stack* is vital to the execution of a program, because it provides a temporary storage location for the intermediate result values used when parsing expressions, and of course provides the foundation for function calls.
- A *stack frame* or *activation record* is a data structure pushed onto the stack for each function call that encapsulates that function's return value, parameter list, and all of its local variables and arrays.
- XVM stands for "XtremeScript Virtual Machine", but it's also the Roman numeral representation of 985. 985 kinda looks like "1985". I was born in 1981. 1985 - 1981 = 4, which is the exact number of letters in my first name! *COINCIDENCE!?!*

SUMMARY

Out of all the theoretical chapters in the book, this has hopefully been among the most informative. In only a few pages you've learned quite a lot about basic assembly language, different approaches to instruction set design, and even gotten your first taste of how an assembler works. I then moved on to cover the design of XVM Assembly, the low-level language for the XtremeScript system that will work hand-in-hand with the high-level language developed in the last chapter. You've got another major piece of the design puzzle out of the way, and you're about to put it to good use.

The next chapter will focus on the design and implementation of XASM (which I pronounce "Exasm", by the way), which is the XtremeScript Assembler. You'll be taking a big step, as this will mark your first actual work on the system you've spent so many pages planning. As you've also seen, the assembler will be more than just another part of a larger system. Once you also have a working VM (which will directly follow your work on the assembler), you'll have the first working version of your scripting system. The language itself may be less convenient than a high-level, C-style language, but will be capable of the same things. In other words, the following chapter will be your next step towards attaining scripting mastery (feel free to insert the Jedi-reference of your choice here).

This page intentionally left blank

CHAPTER 9

Building the XASM Assembler

"It's fair to say I'm stepping out on a limb,
but I am on the edge. And that's where it happens."
——*Max Cohen,* Pi

Over the course of the last eight chapters, you've been introduced to what scripting is and how it works, you've built a simple command-based language scripting system, you've learned how real procedural scripting is done on a conceptual level, you've learned how to use a number of existing scripting systems in real programs, and you've even designed both the high- and low-level languages the XtremeScript system will employ. At this point, you're poised and ready to begin your final mission—to take XtremeScript out of the blueprints and design docs in your head, and actually build it.

This chapter will mark the first major step in that process, as you design and implement XASM. XASM is short for *XtremeScript Assembler*, and, as the name implies, will be used to assemble scripts written in XVM Assembly down to executables capable of running on the XtremeScript virtual machine. This program will sit in between the high-level XtremeScript compiler (which outputs XVM assembly) and the XVM itself, and is therefore a vital part of the overall system. Figure 9.1 illustrates its relationship with its neighboring components.

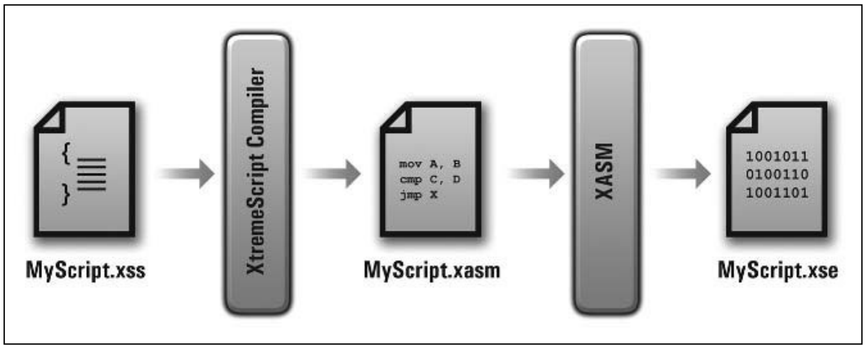


Figure 9.1

XASM sits in between the compiler and run-time environment as the final stage in the process of turning a script into an executable.

XASM is a good place to start because it's an inherently simple program, at least when compared to the complexities of a high-level language compiler. Despite the myriad of details you'll see in the following pages, its main job can still be described simply as the mapping of instruction mnemonics to their respective opcodes, as well as other text-to-numeric conversions. It's really just a "filter" of sorts; human-readable source code goes in one end, and executable machine code comes out the other.

With the pleasantries out of the way, it's time to roll up your sleeves and get started. This chapter will cover

- A much more in-depth exploration of how a generic assembler works.
- The exact details of how XASM works.
- An overall design plan for the construction of the assembler.
- A file format specification for the output of XASM, the XVM executable file.

I *strongly* encourage you to browse the code for the working XASM assembler as or after you read the chapter. It can be found on the accompanying CD and is heavily commented and organized. Regardless of how many times you read this chapter and how much you may think you "totally get it", the XASM source code itself is, for all intents and purposes, required reading. Once you understand the underlying concepts, you'll really stand to gain by seeing how it all fits together in a working program. In a lot of ways, this chapter is almost a commentary on the XASM source code specifically, so please don't underestimate the importance of taking the time to at least read through it when you're done here.

How a Simple Assembler Works

Before coding or designing anything, you need to understand how a simple assembler works on a conceptual level. You got a quick crash course in the process of reducing assembly to machine code in the last chapter, but you'll need a better understanding than that to get the job done here.

As you saw in Chapter 8, the basic job of an assembler is to translate human readable assembly source code to a purely numeric version known as machine code. Essentially, the process consists of the following major steps:

- Reducing each instruction mnemonic to its corresponding opcode based on a "master" instruction lookup table.
- Converting all variable and array references to relative stack indices, depending on the scope in which they reside.
- Taking note of each line label's index within the instruction stream and replacing all references to those instructions (in jump instructions and `Call`) with those indices.
- Discarding any extraneous or human-readable content like whitespace, as well as commas and other delimiting symbols. In other words, reducing everything to a binary form as opposed to ASCII.
- Writing the output to a binary file in a structured format recognized by the XVM as an executable.

NOTE

This is just me ranting about a huge pet peeve of mine, but have you ever thought about how stupid the term "lookup table" is? It's completely redundant. What other function does a table have other than lookups? Do tables exist that *don't* allow lookups? What purpose would such a table serve? It'd be like saying "read-from book" or "drive-around car" or "buy-from store". There's no point in prefixing the name of something with its sole purpose, because the name by itself already tells you what it does. Oh well, don't mind me, and feel free to disagree and send me flame e-mails calling me an idiot. :) I'll continue using the term just because everyone's already used to it, but know this—every time I say it, I die a little inside. In the meantime I'll just get back to writing this learn-from chapter using my type-on keyboard.

The next section discusses how the instructions of a script file are processed by a generic assembler, in reasonably complete detail. The output of this generic, theoretical assembler is known as an *instruction stream*, a term representing the resulting data when you combine all of the opcodes and operands and pack them together sequentially and contiguously. It represents everything the original source code did, but in a *much* faster and more efficient manner, designed to be blasted through the VM's virtual processor at high speeds.

Assembling Instructions

Primarily, an assembler is responsible for mapping instruction mnemonics to opcodes. This process involves a lookup table (ahem) containing strings that represent a given instruction, the opcode, and other such information. Whenever an instruction is read from the file, this table is searched to find the instruction's corresponding entry. If the entry is found, the associated opcode is used to replace the instruction string in the output file. If it's not found, you can assume the instruction is invalid (or just misspelled) and display an error. Check out Figure 9.2 to see this expressed visually.

The actual implementation of the table is up to the coder, but a hash table is generally the best approach because it allows strings to be used as indices in linear time. Of course, there's nothing particularly wrong with just using a pure C array and searching it manually by comparing each string. After all, although it is significantly slower than using a hash table or other, more sophisticated method of storage, you probably won't be writing scripts that are nearly big enough to cause noticeable slowdown. Besides, assembly isn't done at runtime, so the speed at which a script is assembled has no bearing on its ultimate runtime speed.

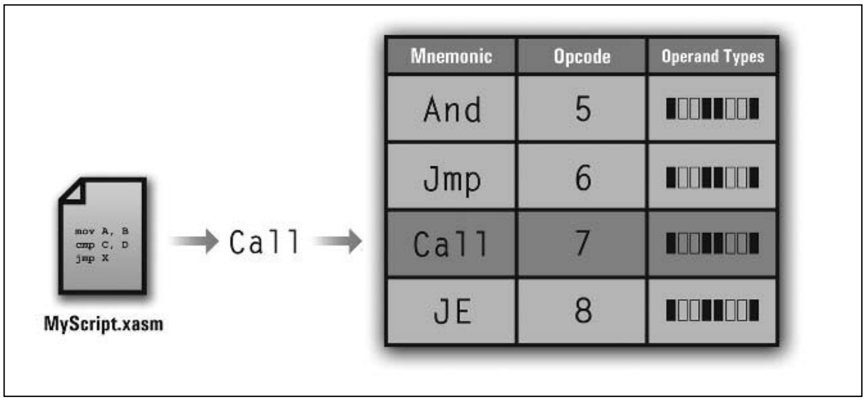


Figure 9.2

Looking up an instruction in the table to find its corresponding opcode.

NOTE

Hashtables are a great way to implement the instruction lookup table, so I highly recommend them in your own assemblers. C++ users can immediately leverage the existing STL hashtable, for example. I won't be using them in the source to XASM, however, because I find them to be somewhat obtrusive as far as teaching material goes; it's easier to understand the linear search of a C array than it is to understand even a totally black boxed hashtable. You'll find throughout the book that I usually chose simplicity over sophistication for this reason.

I also mentioned previously that in addition to the mnemonic string and the opcode, each entry in the table can contain additional information. Specifically, I like to store an instruction's opcode list here. The *opcode list* is just a series of flags of some sort (usually stored in a simple array of bit vectors) that the assembler uses to make sure the operands supplied for the given instruction are proper. For example, a `Mov` instruction accepts two parameters. The first parameter, `Destination`, must be a memory reference of some sort, because it's where the `Source` value will be stored. `Source`, on the other hand, can be anything—another memory location like `Destination`, or a literal value. So the first operand can be of one data type, while the second can be many. The lookup table would store an opcode list at the `Mov` instruction's index that specifies this.

The operand list can also be implemented any way you like, but as I said, I prefer using arrays of bit vectors. Each element in the array is a byte, integer, long integer, or whatever (depending on how many flags you need). Each element of the array corresponds to an operand, in the order they're expected. In the case of `Mov`, this would be a two-element array indexed from 0 to 1.

Element 0, corresponding to `Destination`, only allows memory references and would therefore have the `MEMORY_REF` flag set (for example), whereas the `LITERAL_VALUE` flag would be unset. Element 1, on the other hand, because it corresponds to `Source`, would have both the `MEMORY_REF` and `LITERAL_VALUE` flags set. Other operand types would exist as well, such as `LINE_LABEL` and `FUNCTION_REF` for jump instructions and `CALL` for example. This is explained in more detail in Figure 9.3.

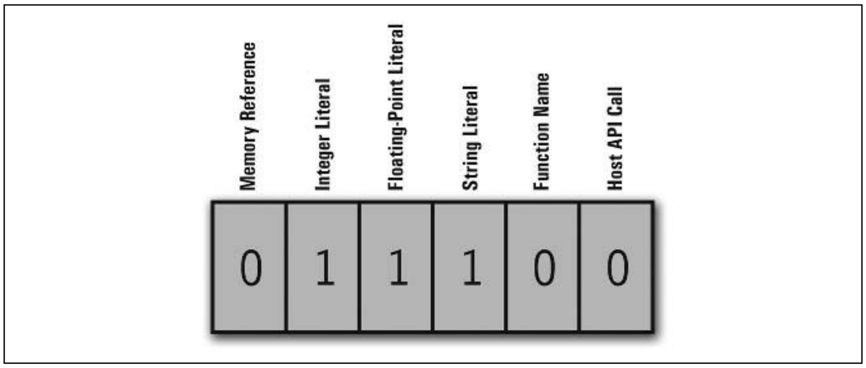


Figure 9.3

Bit vectors being used to store the description of an operand list.

This table, with its three major components, would be enough information to write a basic assembler capable of translating instructions with relative ease. As each instruction is read in, its name is validated to make sure it's in the table and is therefore a known mnemonic, the operands are checked against the operand list stored in the table, and finally, its opcode is written to the output.

The operands are written to the output as well, of course, but doing so is significantly more complex than assembling the instructions themselves. To understand how operand lists are assembled, you first have to know how each *type* of operand is assembled; only then can you process entire operand lists and write them to the output file. To get things started, let's learn how variable references are assembled, and then move on to operand assembly in general.

Assembling Variables

Variables are assembled in a reasonably straightforward way. As you learned in the last chapter, a variable or array index is really just a symbolic name that the programmer attaches to a relative stack index. The stack index is always relative to the top of the stack frame of the function in which it's declared. Even global variables can be placed on the stack (at the bottom, for example).

A function's stack frame generally consists of a number of parameters, the caller's return address, and local data. An active function's stack frame is always at the top of the stack. If that function makes another function call, the newly called function then takes over the top of the stack while it's running. Once the second function returns, its stack frame is popped off the stack, so that when the calling function continues executing, it's once again on top.

If you remember back to the discussion of Lua in Chapter 6, you may recall that the Lua stack can be accessed in two ways; with positive indices and with negative indices. Positive indices start from the bottom, so that the higher the index, the higher up you go into the stack. Negative indices, however, are used to read from the stack relative to the top. Therefore, -1 is the top of the stack, -2 is the second highest stack element, and so on. The lower the negative index, the lower into the stack you read. You use a similar technique when dealing with stack frames. Because a function's stack frame is always at the top (as long as it's the active function, which it obviously is if its code is executing), you can access elements relative to the top of the current stack frame by using negative indices. Check out Figure 9.4.

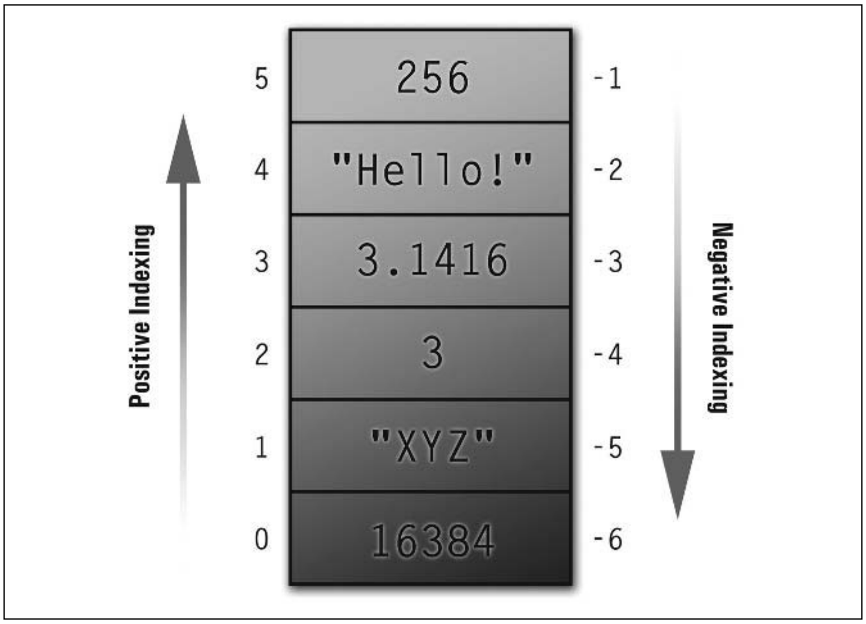


Figure 9.4

Stack indexing.

The stack frame consists of three major components. Starting from the top of the frame and working down, they are as follows: local data, the caller's return address, and the passed parameters (see Chapter 8 for more information on why it's laid out this way). So, if you have four local variables and two parameters, you know that the size of the stack frame is seven elements (4 + 2 + 1 = 7; always add 1 because the return address takes exactly one stack index in addition to everything else). Therefore, the stack frame takes up the *top* seven elements of the stack. The four local variables take indices -1 through -4, the return address is at -5, and the two parameters are at indices -6 and -7.

Figure 9.5 contains an example of a stack frame.

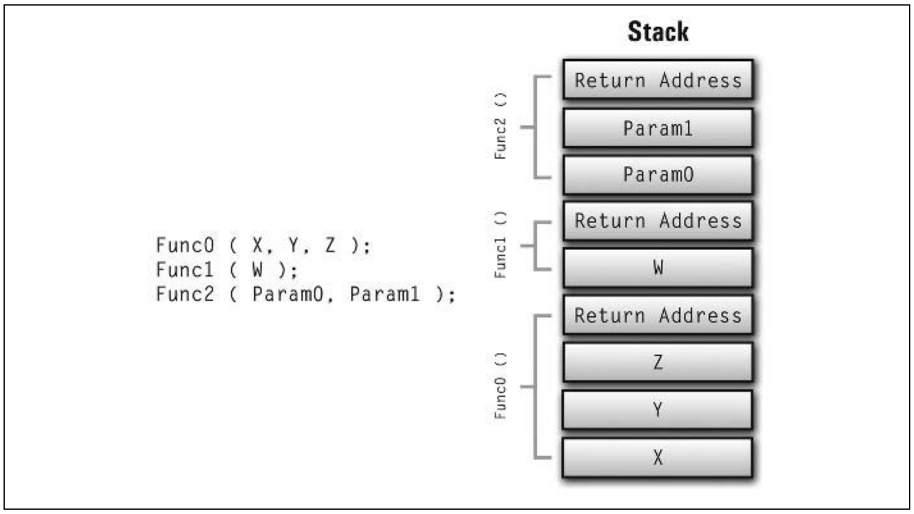


Figure 9.5

An example stack frame.

You can use this information to replace a variable name with a stack index. Let's assume the following code was used to declare the function's variables, and that variables are placed on the stack in the order they're declared (therefore, the first one declared is the lowest on the stack):

```
var X
var Y
var Z
var W
```

`X` would be placed on the stack first, followed by `Y`, `Z`, and `W`. Because `W` would be on the top of the stack, its relative index is -1. `Z` follows it at index -2, and `Y` and `X` complete the local data section of the frame with indices -3 and -4, respectively. You can then scan through the input file and, as you read each variable operand, replace it with the indices you've just calculated. Check out Figure 9.6.

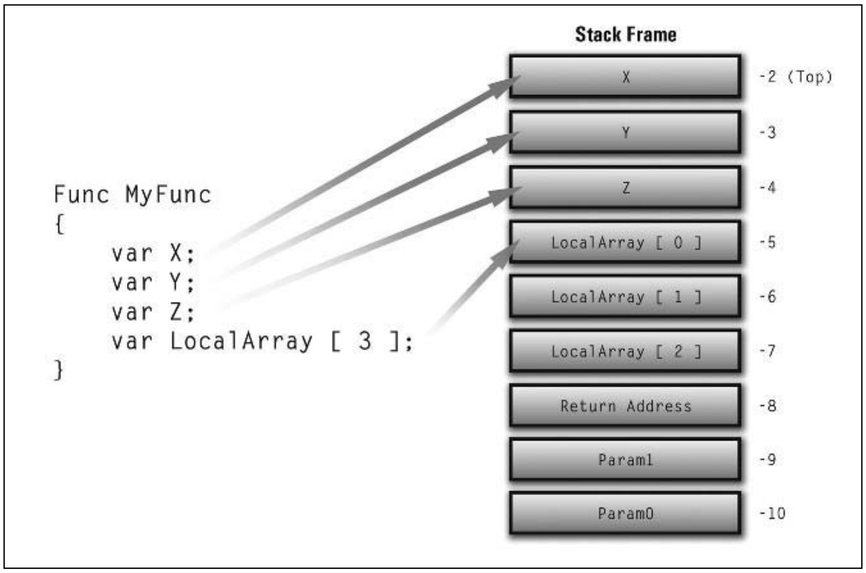
However, it isn't enough to simply replace a variable with a number. For example, there'd be no way to tell a stack index from a literal integer value. Imagine assembling the following instruction:

```
Mov    Z, 4
```

As previously determined, `Z` resides at index -2. Also assuming that the `Mov` instruction corresponds to opcode 0, your assembled output would look something like the following:

```
0 -2 4
```

The XVM, when it receives this data, is going to interpret it at as "`Move the value of 4 into -2.`", which doesn't make much sense. What you need to do is prefix the assembled operand with a flag of some sort so that it can tell the difference between an assembled variable (a relative stack



```
Func MyFunc
{
    var X;
    var Y;
    var Z;
    var LocalArray [ 3 ];
}
```

Figure 9.6

Variables and their association with stack indices relative to the current stack frame.

index) and an assembled integer variable. For example, let's say the code for a stack index is 0, and the code for an integer literal is 1. The new output of the assembler would look like this:

```
0 0 -2 1 4
```

As you can see, the new format for the `Mov` instruction is opcode, operand type, operand data, operand type, and operand data.

Lastly, there's the issue of referencing global variables. Because these reside at very different locations than local data, you need to make sure to be ready for them. I prefer storing globals at the bottom of the stack; this way, whether a given variable is local or global, you can always use stack indices to reference them. Because the bottom of the stack can be indexed using positive indices, you don't have to make any changes to the instruction stream.

NOTE

XASM and the XVM will actually work a bit differently than what I've described here. For reasons that will ultimately become clear in the next chapter, the stack indices generated for variables will begin at index -2, rather than -1. Since I don't want to bewilder you too much, the reason has to do with an extra value that the XVM pushes onto the stack *after* the stack frame, which causes everything to be pushed down by one index (thus, local data starts at -2 instead of -1). This extra value wasn't mentioned in chapter 8 because it's specific to the XVM--it needs it for some internal bookkeeping issues we'll get into in the next chapter. For now, just keep this detail in mind.

An assembled global variable reference is just like a local one; the only difference is the sign of the index.

Assembling Operands

You've already seen the first steps in assembling operands in the last section with the codes you used to distinguish variable stack indices from integer literals, but let's round the discussion out with coverage of other operand types. As you saw, operands are prefixed with an operand type code so that the runtime environment can determine what it should do with the operand data itself. In the case of a stack index operand type, the runtime environment expects a single integer value to follow (which is treated as the index itself). In the case of an integer literal operand type, a single integer value would again be expected. In this case, however, this is simply a literal value and is treated as such.

There are a number of operand types to consider, however. Table 9.1 lists them all.

Table 9.1 Operand Types

Type	Example	Description
Integer Literal	`256`	A literal integer value
Float Literal	`3.14159`	A literal float value
String Literal	`"L33T LIEK JEFF K.!!11"`	A literal string value
Variable	`MyVar`	A reference to a single variable
Array with Literal Index	`MyArray [ 15 ]`	An array indexed by an integer literal value
Array with Variable Index	`MyArray [ X ]`	An array indexed by a variable
Line Label	`MyLabel`	A line label, used in jump instructions
Function Name	`MyFunc`	The name of a function, used in the `Call` instruction
Host API Call	`MyHostAPIFunc`	The name of a host API function, used in the `CallHost` instruction

The list should be pretty straightforward, although you might be a bit confused by the idea of arrays indexed by literal values being considered different than arrays indexed by variables. The reason this is an issue is that the two operand types must be written to the output file with different pieces of information. For example, an array with an integer index must be written out with the base index of the array (where the array begins on the stack), as well as the array index itself (which will be added to the first value to find the absolute stack index, which is where the specific array element resides). In fact, you could even add the array index to the array's base at compile-time and write that single value out as a typical variable reference (which would be more efficient). An array indexed with a variable, on the other hand, cannot be resolved at assemble-time. There's no way to know what the indexing variable will contain, which means you can only write out the array's base index and the index of the variable index. These two methods of indexing arrays are different, and the runtime environment must be aware of this so it can process them properly. Check out Figure 9.7.

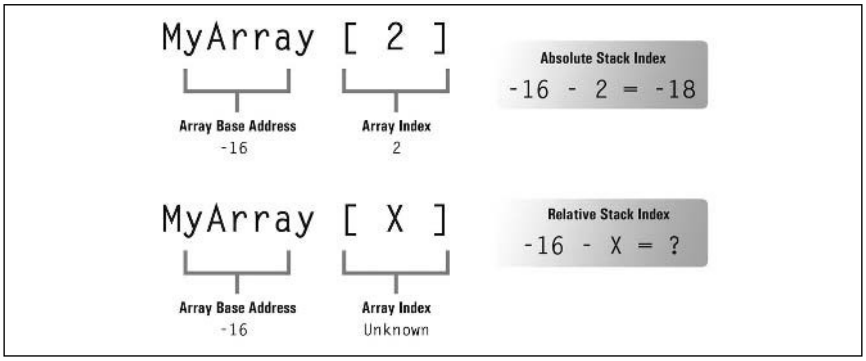


Figure 9.7

Arrays being indexed in different ways.

As for the operand type codes themselves, they're just simple integer values. An integer literal might be 0, floats and strings might be 1 and 2, variables and both array types might be 3, 4, and 5, and so on. As long as the assembler outputs codes that the VM recognizes, the actual values themselves don't matter.

Now that you can prefix each operand with a code that allows the VM to properly read and interpret its data based on its type, there's one last piece of information each instruction needs, and that's how many operands there are in total. This is another simple addition to the instruction stream output. In between the opcode and the first operand, you need only insert another integer value that holds the number of operands following. In the case of `Mov`, this would always be 2 (the `Source` and `Destination` operands). In the case of `Jmp` it'd always be 1 (the `Label` operand). So, if you have the following line of code:

```
Mov    MyVar, 16384
```

and MyVar is found at stack index -8, the machine-code equivalent would look like this:

```
0 2 3 -8 0 16384
```

Now, the order is basically this: first you output the opcode (0), and then you output the newly-added operand count (2, for two operands), and then the operand type of the first operand (a variable in this case, the code for which let's assume is 3), and then the variable's stack index (-8), and finally the second operand. The second operand is an integer, the code for which let's assume is 0, followed by the value itself (16384). Check out Figure 9.8 for a visual of this format.

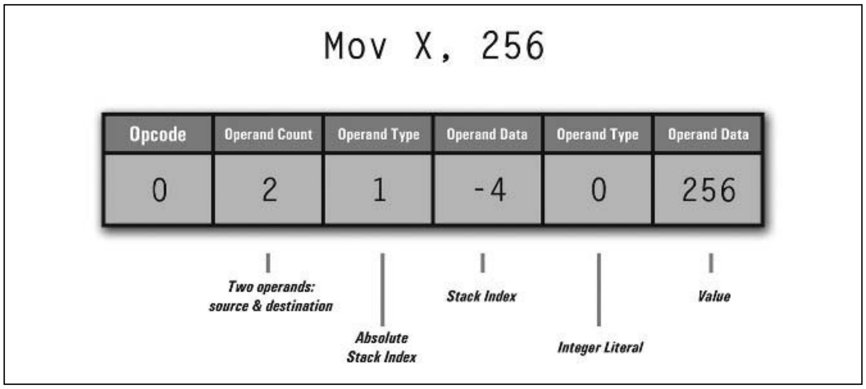


Figure 9.8

The new format of an assembled instruction consists of the opcode, followed by N *operands, each of which consist of an operand type code and operand data.*

You might be wondering why you need to include the operand count at all. As you've seen, these instructions have a fixed number of operands. For example, Mov always has two operands, Jmp always has 1, and so on. There doesn't seem to be much need to include this data when you can just derive it from the opcode itself. The reason I like to include it, however, is that it may become advantageous at some point to give instructions the ability to accept a variable number of operands. For example, you might want to alter the Exit instruction so that it can be called without the return code, thereby making it optional (so Exit might be interpreted the same as Exit 0, for example). If you decide to do this, however, you'll need some way for the VM to know that sometimes, Exit is called with a return code, and sometimes it isn't. Adding a simple variable count to the instruction stream allows you to do this easily.

Assembling String Literals

Strings are simple to assemble, but it may not be done in the way you'd imagine. Simple literal values like integers can be embedded directly into the instruction, immediately following the operand type code. You could do the same thing with strings, but that means clogging up your otherwise simplistic instruction stream with chunks of string data. Consider the following two lines of code:

```
Mov    X, "This is a string literal."
Mov    Y, 16384
```

The instruction stream would look something like this:

```
0 2 3 8 This is a string literal 0 2 3 9 0 16384
```

I personally happen to find this implementation a bit messy; loading the instruction stream from the disk when the script is loaded into the runtime environment will become a more complicated affair, because you'll have to manage the reading of strings in addition to simply reading in simple integer values (and floats, in the case of float literals).

Instead of clogging up the instruction stream, I suggest strings be grouped at assemble-time and loaded into a separate structure called the *string table*. The string table contains all of a script's string literals, and assigns each a sequential index (which means it's just a simple array). Then, instead of placing a string literal itself in the instruction stream, you substitute it with its corresponding index into the string table. The string table itself is then written out in full to another part of the output file.

In the case of the previous example, because the two-line script has only one string, it'd be loaded into the string table at index 0. Therefore, the instruction stream itself would now take on a much cleaner, more compact form:

0 2 3 8 **0** 0 2 3 9 0 16384

Ahhh, much better. Figure 9.9 illustrates the separation between the instruction stream and the string table.

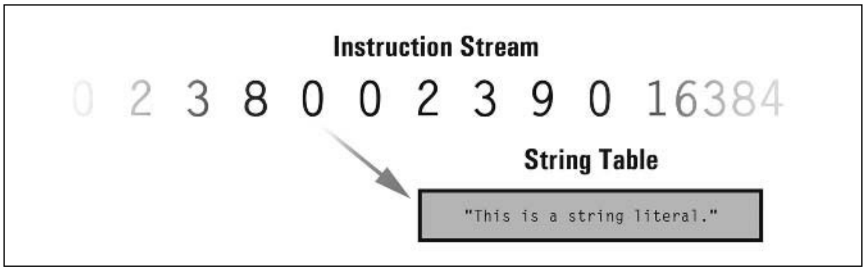


Figure 9.9

The string table separates strings from the instruction stream, allowing for cleaner encapsulation and logical grouping of data.

Assembling Jumps and Function Calls

The last real aspect of the instruction stream to discuss in this initial overview of the assembly process deals with line labels and functions. Line labels are used to mark the location of a given instruction with a symbolic name that can be used to reach it with a jump. Function names are similar; rather than marking a single instruction, however, they mark a block of them and give the code within that block its own scope and stack frame.

Line labels and jumps are often approached with one of two popular methods when assembling code for a real hardware system. The first method is called the *two-pass* method, because the calculation of line labels is handled in one complete pass over the source file, whereas the second pass assigns the results of the first pass (the index of each line label) to those line label's respective references in jump instructions.

You have a number of options when approaching this issue in your own assembler. Regardless of how you do it, though, the underlying goal of this phase is twofold; to determine which instruction each line label corresponds to, and to use the index of those instructions to replace the label's references in jump instructions. The following code provides an example:

```
Label0:
        Mov    X, Y
        And    Z, Q
        Jmp    Label0
        Pop    W
        Pause  U
        JLE    X, Y, Label1
        Push   256
Label1: Jmp    Label0
        Exit   0
```

Here you have two labels and three jump instructions (forget about the actual code itself, it's just there to fill space). The first label points to the first instruction (`Mov X, Y`), whereas the second (and last) label points to the eighth instruction (`Jmp Label0`). Notice here that the actual instruction pointed to by a given label is *always* the one that immediately follows it. The label and the instruction can be separated by any amount of whitespace, including line breaks, which is why the two don't have to appear on the same physical line to be linked. Here's the same code again with line numbers to help explain how this all works:

```
   Label0:
0:         Mov    X, Y
1:         And    Z, Q
2:         Jmp    Label0
3:         Pop    W
4:         Pause  U
5:         JLE    X, Y, Label1
6:         Push   256
7: Label1: Jmp    Label0
8:         Exit   0
```

According to the diagram, these nine instructions are indexed from 0-8, and any lines that do not contain instructions (even if they contain a label declaration) don't count. Also, notice that line labels can be declared after references to them are made, as in the case of `Label1`. Here, notice that `Label1` is referenced in the `JLE` instruction on line 5 before being declared on line 7. This is called a *forward reference*, and is vital to assembly programming for obvious reasons (refer to Chapter 8's intro to assembly language coding for examples). However, this ability for label references to precede their declarations is what makes line label assembly somewhat tricky. Before I get into that, however, let's take a look at the previous code after its line labels have been assembled:

```
Mov    X, Y
And    Z, Q
Jmp    0
Pop    W
Pause  U
JLE    X, Y, 7
Push   256
Jmp    0
Exit   0
```

Check out Figure 9.10 for a graphical representation of this process.

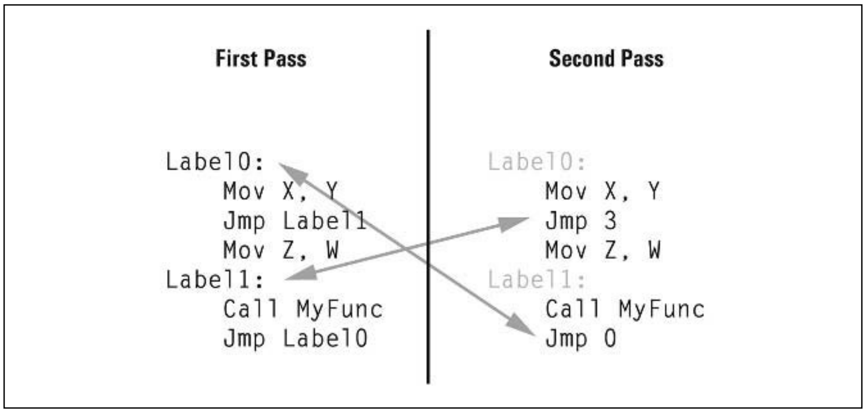


Figure 9.10

Line labels and jumps are matched up over the course of two passes

As you can see, the label declarations are gone. In place of label references are simple integer values that correspond to the index of a target instruction. The runtime environment should route the flow of execution to this target instruction when the jump is executed. What you're more interested in, though, is the actual process of calculating these instruction indices. I think the two-pass approach is simpler and more straightforward, so let's take a look at how that works.

- The first pass begins with the assembler scanning through the entire source code file and assigning a sequential index to each instruction. It's important to note that the amount of lines in the file is *not* necessarily equal to the amount of instructions it contains (in fact, this is rarely the case and will ultimately be impossible when the final XVM assembly syntax is taken into account). Lines that *only* contain directives, labels, whitespace, and comments don't count.
- The first pass utilizes an array of line labels that is similar in structure to the master instruction lookup table discussed earlier. Each element of this array contains the line label string itself, as well as the index of the instruction it points to. With these two fields, you have enough data to match up jumps with their target instructions in the resulting machine code.
- Whenever a line label declaration is detected, a new element in the array is created, with its name field set to the name of the label string. So, if you encounter `MyLabel:` in the source code, a new element is created in the line label array containing the string "`MyLabel`" (note the removal of the colon). Care must also be taken to ensure that the same label is not declared twice; this is a simple matter of checking the label string against all array elements to make sure it doesn't already exist.
- Remember, a line label always points to the instruction immediately *following* it. So, whenever a label is detected, you copy the instruction counter to a temporary variable and use that value as the label's target index. This value, along with the label's name, is placed into the array and the label is recorded. The process of determining a line label's target instruction is called *resolving* the label.
- This process continues until the entire source file has been scanned. The end result is an array containing each line label and its corresponding instruction index. The instruction stream has not been generated in any form yet, however; this pass is not meant to produce any code.

This completes the first pass, so let's take a look at the steps involved in the second pass. The second pass is where you actually assemble the entire source file and dump out its corresponding machine code. All you're worried about in this section, however, is the processing of line labels, so let's just focus on that and ignore the rest.

- The second pass scans through each instruction of the source file, just as the first did. As each instruction is found, it's converted to its machine-code equivalent using the techniques discussed previously. In the case of jump instructions, however, you need to output a line label operand type. What this actually consists of isn't the line label string, but rather the target instruction's index.
- Whenever a jump instruction is found, its line label is read and used as a key to search the line label array constructed in the last pass. When the corresponding element is found, you grab the instruction index field and use that value to replace the label in the

machine code you output. Just like with labels, this is called *resolving* the jump. Note also that if the label cannot be found in the label array, you know it's an invalid label (or again, just a misspelling) and must alert the users with an error.

That, in a nutshell, is how line labels are processed in a two-pass fashion. The end results are jump instructions that vector directly to their target instructions, as if you never used the labels to begin with. Slick, huh?

Functions and `Call` instructions are processed in virtually the same way. In the same first pass you use to gather and resolve line labels, you can also detect instances of the `Func` directive, which, to refresh your memory, looks like this:

```
Func Add
{
    Param  X                ; Assign names to the two parameters
    Param  Y
    Var    Sum              ; Create a local variable for the sum
    Mov    Sum, X           ; Perform the addition
    Add    Sum, Y
    Mov    _RetVal, Sum     ; Store the sum in the _RetVal register
}
```

This is a simple addition function, defined using the `Func` directive. In a lot of ways, `Func` is just a glorified line label; its major purpose (aside from establishing the function's scope, which is why you need the curly braces as well) is simply to help you find the entry point of the function.

Because `Call` is basically just an unconditional jump that also saves the return address on the stack, you can approach the resolution of function names, as well as the assembling of `Call` instructions, in roughly the same way you approached line labels and jumps. In the first pass (the same "first pass" discussed previously), you gather and resolve each function name, associating it with the index of the first instruction within its scope, or its *entry point*. In the case of the `Add` function, the entry point is `Mov Sum, X` (remember, directives like `Param` and `Var` don't count as instructions), and therefore, the index of that instruction will be stored along with the "`Add`" name string within an array of functions. This array will be structured just as the label array is; each element contains a function name and an index.

The second pass will then replace the function name parameter in each `Call` instruction with the index of the function's entry point. So, if `Add`'s entry point is the 204th instruction in the script, any `Call` instruction that calls the function would go from this:

```
Call    Add
```

to this:

```
Call    204
```

Simple, right? Of course, functions are more than just labels, and calling a function is more than just jumping to it—otherwise, you'd just use the jump instructions and typical line labels instead. A function also brings with it a concept of scope and builds itself an appropriate stack frame upon its invocation—containing the parameters passed, return address and local data.

Because of this extra baggage, you won't actually replace the function name in a `Call` instruction with the function's entry point, but rather, an index into a *function table*. The function table is a structure that will be created during the assembly of the script and persist all the way up to the script's runtime. Whenever a function is called, this index is used to extract information about the requested function from the table. This information will primarily pertain to the construction of function's stack frame, but will also include the basics like, of course, the entry point itself.

The issue of functions and their stack frames is highly specific from machine to machine, from language to language, and from compiler to compiler. As a result, I won't be covering it just yet (although I will later in this chapter). This section is just meant to be a conceptual overview of a generic assembler, and discussing the details of the stack frames and the function invocation and return sequence would go too far beyond that. I'll return to this subject later.

XASM Overview

You now should understand the majority of how a generic assembler does its job in theory, so I'll now expand that into a description of how XASM will work in practice. XASM is, more or less, a typical assembler; the only major difference is that it's designed to produce code for a typeless virtual machine, which makes things a lot easier on you.

In addition to the basic assembler functionality, it brings with it a number of added features like the directives discussed in the last chapter for declaring variables, arrays, functions and so on. Overall, the assembler will be responsible for the following major steps:

- A first pass over the source code that processes directives, including the processing of line label indices and function entry points.
- A second pass over the source that assembles each instruction into its machine code equivalent, also resolving jump labels and function calls as well.
- Writing the completed data out to a binary file using a structured format that the XVM can recognize.

This is a very broad roadmap, but it's more or less the task you're responsible for. I'm now going to discuss a variety of topics that relate to the construction of this assembler, getting closer and closer to a full, specific game plan with each one. Eventually you'll reach a point where you understand enough individual topics to put everything together into a fully functional program.

Memory Management

First and foremost, it's important to be aware of the different ways in which both the script source data, as well as the final executable data, can be stored. Early compilers and assemblers ran on machines with claustrophobically small amounts of memory, and as a result, kept as much information on the hard drive as possible at all times. Source files were read from the disk in very small chunks, processed individually, and immediately written to either temporary files or the final executable to clear room for the next chunk. This process was repeated incrementally until the entire file had been scanned and processed.

Today, however, you find yourself in a much different situation. Memory is much cheaper and far more ubiquitous, giving compiler writers a lot more room to stretch out. As a result, you're usually free to load an entire source file into memory, perform as many passes and analysis phases as you want, and write the results to disk at your leisure. Of course, no matter how much memory you've got at your fingertips, it's never a good idea to be wasteful or irresponsible.

Because of this, you've got a decision to make early on. You already know that you'll be making repeated passes over the source file—at least two—and might want to load everything into memory for that reason alone. Furthermore, loading the file into memory allows you to easily make on-the-fly changes to it; various preprocessing tasks could be performed, for example, that translate the file into slightly different or more convenient forms for further processing.

In a nutshell, having the entire file loaded into memory makes things a lot easier; data access is faster and flexibility is dramatically increased. Furthermore, memory requirements in general will rarely be an issue in this case. Unlike the average assembler or compiler, which may be responsible for the translation of five or ten million lines of code, an assembler for a scripting language is unlikely to ever be in such a position. Scripts, almost by their very nature, tend to be dramatically smaller than programs.

Of course, it's not necessarily an open and shut case. There are definitely reasons to consider leaving the source file (among other things) on the disk and only using a small amount of actual memory for its processing. For example, you might want to distribute your assembler and compiler along with your game, or with a special version of your game that's designed to be expanded by mod authors or other game hackers. In this case, when the program will be run on tens, hundreds, or even thousands of different end users' machines, available memory will fluctuate wildly and occasionally demand a small footprint. Furthermore, despite these comments, it's always possible that your game project, for whatever reason, will demand massive scripts that occupy huge amounts of memory. Although I personally find this scenario rather unlikely, you can never rule something out entirely. See Figure 9.11 for a visual representation of this concept.

In the end, it's all up to you. There's a strong case to be made for both methods. As as long as there aren't any blatantly obvious reasons to go one way over the other, you really can't go wrong.

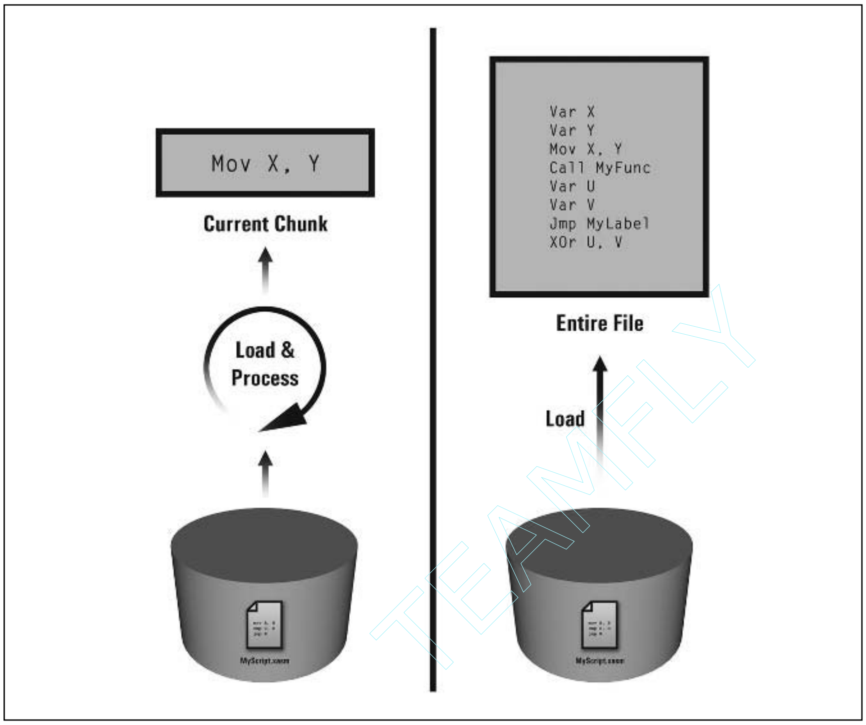


Figure 9.11

The source file can be loaded into memory once at the start of the assembly process, or left on the disk and read from incrementally.

Either method will serve you well if it's implemented correctly. However, for the purpose of this book, you'll load the entire script into memory, rather than constantly making references to an external file, for a number of reasons:

- It's a lot easier to learn the concepts involved when everything is loaded into a structured memory location rather than the disk, so learning the overall process of assembly will be simpler.
- You're free to do more with the file once you have it loaded; you can move blocks of code around, make small changes, perform various preprocessing tasks, and the like.
- Overall, the assembler will run faster. Because it's making multiple passes over the source file, you avoid repetitious disk access.

Input: Structure of an XVM Assembly Script

Whenever approaching a difficult situation, the most important thing is to know your enemy. In this case, the enemy is clear—the source code of an XVM Assembly script. These scripts, although more than readable for pansy humans, are overflowing with fluff and other extraneous data that software simply chokes on. Whitespace? Hmph! Line breaks? Hmph! An assembler craves not these things. It's your job to filter them out.

Parsing and understanding human-readable data of any sort is always a tricky affair. Style and technique differ wildly from human to human, which means you have to make all sorts of generalizations and minimize your assumptions in order to properly support everyone. Whitespace and line breaks abound, huge strings of case-sensitive characters are often required for a human to express what software could express in a single byte, and above all else, errors and typos can potentially be anywhere. Indeed, above all else, compiler theory will teach you to appreciate the cold, calculated order and structure of software.

The point, however, is that the input you'll be dealing with is complex, and the best way to ensure things go smoothly is to understand and be prepared for anything the enemy can throw at you. To this end, this section is concerned with everything a given XVM Assembly script can contain, as well as the different orders and styles these things can be presented in.

Remember, even though the XtremeScript compiler will ultimately replace humans as the source of input for XASM, there's always the possibility of writing assembly-level scripts by hand, or editing the assembly output of the compiler. This will be particularly useful before the compiler is finished, because you'll be forced to use XASM directly. Because of this, you should write the program to be equally accommodating to both the clean, predictable style of a compiler's output, and the haphazard mess of a human.

The following subsections deal with each major component of a script. I initially listed these in the last chapter, but I'll delve into more detail here and provide examples of how they may be encountered.

Directives

Before the instructions themselves, most scripts will present a number of directives to help guide the assembler and VM in their handling of the script's code. Remember, unlike instructions, directives are not reduced to machine code but are rather treated as directions for the assembler to follow. Directives allow the script writer to exert more specific control over the assembler's output.

SetStackSize

The first directive is called `SetStackSize` and allows the stack size for the script to be set by the script itself. It's a rather simple directive that accepts a single numeric parameter, which is of course the stack size. For example,

```
SetStackSize 1024
```

will set the size of the script's stack to 1024 elements. Here are some notes to keep in mind:

- 0 can be passed as a stack size as well, which is a special flag for the VM to allocate the default stack size to the script.
- The directive does not have to appear in the script at all; just like requesting a stack size of zero elements, this is another way to tell the VM to simply use the default size.
- The stack size parameter itself must be an integer literal value and cannot be negative.
- The directive cannot appear in a single script file more than once. Multiple occurrences of the directive should produce an error.

Func

Perhaps the most important directive is Func, because it's the primary method of organization within a script. All code in a script *must* reside in a function; any code found in the global scope will cause an error. Remember, of course, that the term *code* only refers to instructions. Certain directives, like Var for instance (which I'll cover next), can be found both inside and outside of functions.

However, a script that consists solely of user-defined functions won't do anything when executed; just like a C program with no main (), none of a script's functions will execute if they aren't explicitly called. Usually this is desirable, because most of the time you simply want to load a script into memory and call specific functions from it when you feel necessary, rather than immediately executing it (which you learned about first-hand in Chapter 6). However, it's often important for certain scripts to have the ability to execute automatically on their own, without the host having to call a specific function.

In this case, XVM scripts mirror C somewhat in the sense that they can optionally define a function called _Main () that is considered the script's entry point. Just as a function's entry point is the first instruction that's executed upon its invocation, a script's entry point can be thought of as the first function that should be called when it begins running. The XVM will recognize _Main () and know to run it automatically. Here' an example:

NOTE

Why is _Main () preceded with an underscore? As the book progresses, a naming convention will become more and more clear wherein any default, special, or compiler-generated identifiers are preceded with an underscore. As long as users of the assembler and compiler are discouraged from using leading underscores in their own identifiers, this is a good way to prevent name clashing. If, for whatever reason, the user wanted to create a function called Main () that didn't have the property of being automatically executed, he or she could do so. Always keep these possibilities in mind-- name clashing of any sort can result in irritating limitations for the script writer.

```
; This function will run automatically when a script is executed
Func _Main
{
    ; Script execution begins here
}
```

XASM will need to take note of whether a `_Main ()` function was found, and set the proper flags in the output file accordingly so as to pass the information on to the XVM. Because identifiers, including function names, are not preserved after the assembly phase, the XVM will have no way to tell on its own whether a given function was initially called `_Main ()` and therefore relies on the assembler to properly flag it.

Getting back to the `Func` directive in general, let's have a look at its general structure:

```
Func FuncName
{
    ; Code
}
```

Functions can be passed parameters, but this is not reflected in the syntax of the function declaration itself and can therefore be ignored for now. All you really need to do to ensure the validity of a given function is make sure the general directive syntax is followed and that the function's name is not already being used by another function. Also, for reasons you'll see later, the assembler will automatically support alternate coding styles, such as:

```
Func FuncName {
    ; Code
}
Func FuncName { ; Code }
Func FuncName
    {
         ; Code
    }
```

People tend to get pretty defensive about their personal choice of placement for curly braces and that sort of thing—and I'm no exception—so it's always nice to respect that (even if my style *is* right and you're all doing it wrong).

Unlike languages like Pascal, functions cannot be nested. Therefore, the following will cause an error:

```
Func Super
{
    ; Code
    Func Sub
    {
        ; Code
    }
    ; Code
}
```

The last issue in regards to `Func` is that `Ret` is not explicitly required at the end of a function. A `Ret` instruction will always be appended to the end of a function (even if you put one there yourself, not that it'd make a difference), to save the user having to add it to each function manually. Generally speaking, if you can find something that the users will have to type themselves in *all* cases, you might as well let them intentionally omit it so the assembler or compiler can add it automatically.

Var/Var []

The `Var` directive is used to declare variables. The directive itself is independent of scope, which means it can be placed both inside and outside of functions. Any instance of `Var` found inside a function (even the special `_Main ()` function) will be local to that function only. `Var` declarations outside of functions, however, are used to declare globals that can be referenced automatically inside any function.

The syntax of the simple `Var` directive is as follows:

```
Var VarName
```

Unlike a lot of languages, I've chosen to keep things simple, so `Var` cannot be used to declare a comma-delimited series of varaibles, like this:

```
Var X, Y, Z
```

Instead, they must be declared one at a time, like this:

```
Var X
Var Y
Var Z
```

The naming rules of variables are the same as functions; no two variables, regardless of scope, can share the same identifier. Notice that last comment I made; unlike languages like C, which let you "shadow" global variables by declaring locals with the same name, XVM Assembly prevents this. This is just another way to keep things simple. Of course, this doesn't mean that two variables in two different functions can't use the same identifier; that'd be silly. Perhaps I should phrase it this way: no two variables within the same or *overlapping* scope can share a name.

`Var` also has a modified form that can be used to declare arrays, which has the following syntax:

```
Var ArrayName [ ArraySize ]
```

All variable and array declarations in XtremeScript are static, however, which means that only a constant can be used in place of `ArraySize`. Attempting to use a variable as the size of the array should cause an error. Because arrays are always referenced with `[]` notation, it would be possible to allow variables and arrays to share certain names. For example, it's easy to tell the following apart:

```
Var    X
Var    X [ 16 ]
Mov    X, "Hello!"
Mov    X [ 2 ], X
```

The `X` array is always followed by an open-bracket, whereas the `X` variable is not. However, it's yet another complexity you don't really need, so you will treat all variables and arrays the same way when validating their names.

When a `Func` block is being assembled, the number of `Var` directives found within its braces is used to determine the total size of the function's local data. Take the following function for example:

```
Func MyFunc
{
    Var    X
    Var    Y
    Var    MyArray [ 16 ]
}
```

The two `Var` instances mean you have two local variables, and the single `Var []` instance declares a single local array of 16 elements. "Local data" is defined as the total sum of variables and arrays a given function declares, and therefore, this function's local data size is 18 stack elements. Just to recap what you learned earlier, this means that `X` will refer to index -2, `Y` will be -3, and `MyArray [ 0 ]` through `MyArray [ 15 ]` will represent indices -4 through -19. (Remember, XASM and XVM expect all local data to start at index -2, rather than -1).

Variable declarations, like most directives, will be assessed during the first pass over the source, which means that forward references will be possible. In other words, the following code fragment is acceptable:

```
Mov    X, 128.256
Var    X
```

I strongly advise against this for two reasons, however:

- The code is far less readable, especially if there's a considerable amount of code between the variable's reference and its declaration. Although forward referencing is a must for line labels, it's in no way required with variables.
- It's generally good practice to declare all variables before code anyway, or at least declare variables before the block of code in which they'll be used.

Given a choice between the two, I'd personally rather the language not support forward variable references at all, but as we'll soon see, it's actually easier to allow them—you'd have to go out of your way to stop them, and because the goal here is to keep things nearly as simple as possible, let's leave it alone for now.

Param

The `Param` directive is similar to `Var` in that it assigns a symbolic name to a relative stack index. Unlike `Var`, however, `Param` doesn't create any new space; rather, it simply references a stack element already pushed on by the caller of a function and is therefore used to assign names to parameters. Because of this, `Param` can only appear inside functions; there's no such thing as a "global parameter" and as such, any instance of `Param` found in the global scope will cause an error. Lastly, `Param` cannot be used to declare arrays, so `Param []`, regardless of the scope it's found in, will cause an error as well.

Just for completeness, `Param` has the following syntax:

```
Param ParamName
```

`Param` also plays a pivotal role when processing a `Func` block. Just as the number of `Var` instances could be summed to determine the total size of the function's local data, the number of `Params` can be added to this number, along with an additional element to hold the return address, to determine the complete size of the function's stack frame. As an example, let's expand the function from the last section to accept three parameters:

```
Func MyFunc
{
    ; Declare parameters
    Param  U
    Param  V
    Param  W
    ; Declare local variables
    Var    X
    Var    Y
    Var MyArray [ 16 ]
    ; Begin function code
    Mov    MyArray [ 0 ], U
    Mov    MyArray [ 1 ], V
    Mov    MyArray [ 2 ], W
}
```

This function is now designed to accept three parameters. This means that, in addition to the single stack element reserved for the return address, as well as the 18 stack elements worth of local data, the total size of this function's stack frame at runtime will be 3 + 1 + 18 = 22 elements.

Use of the `Param` directive is *required* for any function that accepts parameters. Due to the syntax of XVM Assembly, there's no other way to perform random access of the stack, which means parameters will be inaccessible unless the function assigned names to the parameter's indices within the stack using `Param`.

Also worth noting is the relationship between the number of `Param` directives found in a function, and the number of parameters `Push`ed onto the stack by the caller. Unlike higher level languages like C and even XtremeScript, there's no way to enforce a specific function prototype on callers; the callers simply push whatever they want onto the stack and use `Call` to invoke the function. If the caller pushes too many parameters onto the stack, meaning, the number of elements pushed on is greater than the number of `Param` directives, nothing serious should occur; the function simply won't reference them, and the stack space will be wasted. However, if too few values are pushed onto the stack, references to certain parameters will return garbage values (because they'll be reading from below the stack frame, and therefore reading from the caller's local data). This in itself is not a huge problem, but serious consequences may follow when the function returns. Because functions automatically purge the stack of their stack frame, the function will inadvertently pop off part of the caller's local data as well, because the supplied stack frame was smaller than expected. In short, always make sure to call functions with enough parameters to match the number expected.

Lastly, the order of `Param` directives is important. For example, imagine you'd like to use the following XtremeScript-style prototype in XVM Assembly:

```
Func MyFunc ( U, V, W );
```

The assembly version of the function *must* declare its parameters in either the same order or the exact reverse order:

```
Func MyFunc
{
    Param U
    Param V
    Param W
}
```

The stack indices will be assigned to the parameter names in the order they're encountered, which explains why it's so important. Note, however, that I implied you might want to list the parameters in reverse order, like this:

```
Func MyFunc
{
    Param W
    Param V
    Param U
}
```

This is actually preferable to the first method, because it allows the caller to push the parameters onto the stack in `U, V, W` order rather than forcing the `W, V, U` order. Check out Figure 9.12 to see this difference depicted graphically.

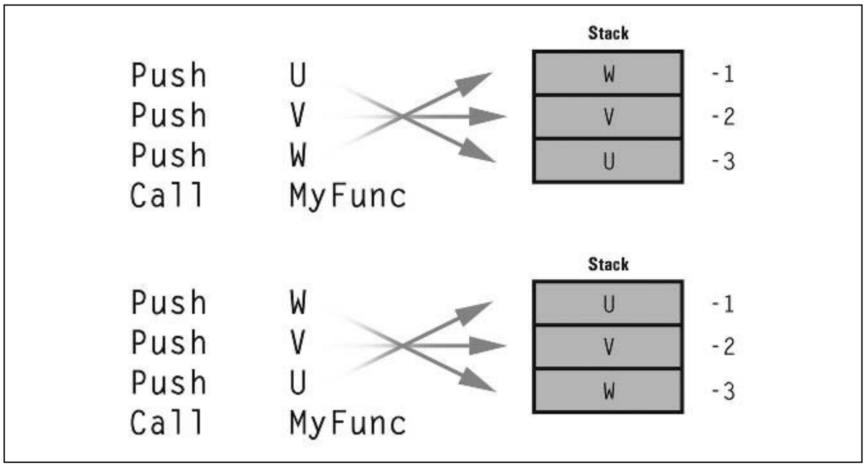


Figure 9.12

Calling a function with two different parameter-passing orders.

Identifiers

With all this talk of functions, variables, and parameters, you should make sure to define a given standard by which all identifiers should be named. Like most languages, let's make it simple and say that all identifiers must consist of letters, numbers, and underscores, and can't begin with a number.

Also, *unlike* most languages, everything in XVM Assembly, namely identifiers, is case-insensitive. I personally don't like the idea of case sensitivity; the only real advantage I can see is being able to explicitly differentiate between two variables named like this:

```
Mov MyVar, myVar
```

And this is just bad practice. The two names are so close that you're only going to end up confusing yourself, so I've taken it out of the realm of possibilities altogether.

Instructions

Despite the obvious importance of directives, instructions are what you're really interested in. Because they ultimately drive the output of machine code, instructions are the "meat" of the script and are also the most complex aspects of translating a source script to an executable.

The XVM instruction set is of a decent size, but despite its reasonable diversity, each instruction still follows the same basic form:

```
Mnemonic    Operand, Operand, Operand
```

Within this form there's a lot of leeway, however. First of all, an instruction can have anywhere from 0-*N* operands, which means the mnemonic alone is enough in the case of zero-parameter instructions. Also, you'll notice that I generally put more space between the mnemonic and the first operand than I do between each individual operand. It's customary to put one or two tab stops between the mnemonic and its operand list so that operands always line up on the same column. Operands are also subject to convention; like in C, I always put a single space between the trailing comma of an operand and the following operand. However, none of these is directly enforced, so the following instruction:

```
Mov    X, Y
```

Can also be written in any of the following ways:

```
Mov X, Y
Mov                X,Y
Mov X     ,Y
```

and so forth.

However, unlike C, you'll notice a lack of a semicolon after each line. This means that instructions must stay within the confines of a physical line; no multi-line instructions are allowed. Also, there must exist at least one space or tab between the instruction mnemonic and the operand list, but operands themselves can be entirely devoid of whitespace because ultimately it's only the commas that separate them.

Instructions and the general format of their operands is the easy part. The real complexity involved in parsing an instruction is handling the operands properly. As you learned, there are a number of strongly differing operand types that all must be supported by the parser. Depending on which operand types are supported, at least, the instruction parser needs to be ready for any of the following:

- **Integer and floating-point literals.** Integer literals are defined as strings of digits, optionally preceded by a negative sign. Floats are similar, although they can additionally contain one (and only one) radix point. Exponential notation and other permutations of floating-point form are not supported, but can be added rather easily.
- **String literals.** These are defined simply as any sequence of characters between two double quotes, like most languages. The string literal also supports two escape sequences; `\"`, which embeds a double quote into the string without terminating it, as well as `\\`, which embeds a *single* backslash into the string. Remember that single backslashes cannot be directly used because they'll inadvertently register an escape sequence, which will most likely be incorrect. The general rule is to always use twice as many backslashes as you actually need to ensure that escape sequences aren't accidentally triggered.
- **Variables.** These can be found in two places—either as the entire operand, or as the index in an array operand.
- **Array Indices.** Arrays can be found as operands in two forms: those that are indexed with integer literals, and those that are indexed with variables. It should be noted that arrays *cannot* appear without an index. For example, an array called `MyArray` can only appear as an operand as `MyArray [ Index ]`, never as simply `MyArray`.
- **Line Labels, Functions, and Host API Calls.** These operands are pretty much as simple as variables; only the identifier needs to be read. A common newbie mistake, however, is to add the colon to the line label reference like you would in the declaration. `Jmp MyLabel:`, however, will cause an error because the `:` is not a valid identifier character and is only used in the declaration.

Any operand list that does not contain as many operands as the instruction requires will cause an error.

Line Labels

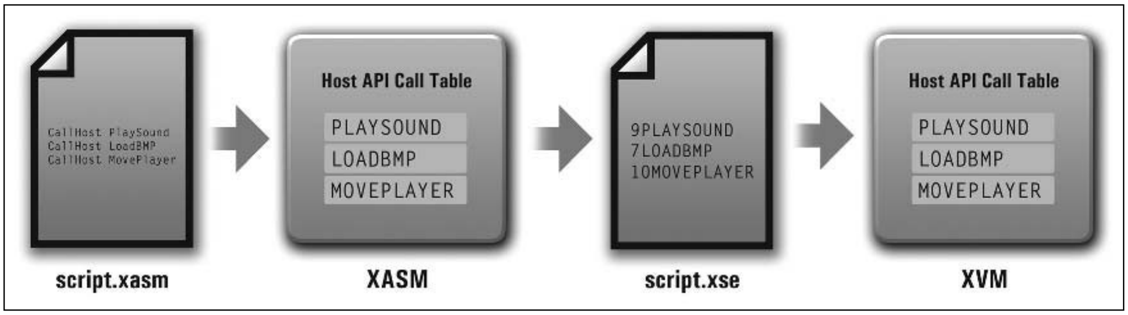
Line labels can be defined anywhere, but are subject to the same scope rules as variables and arrays. Also, like the `Param` directive, they cannot appear outside functions. Line labels are always declared with the following syntax:

```
Label:
```

Host API Function Calls

In addition to functions defined within the script and invoked with `Call`, host API functions can be called with the `CallHost` instruction. `CallHost` works just like `Call` does; the only difference is that the function it refers to is defined by the host application and exposed to the scripting system through its inclusion in the host API.

Everything about calling a host API function is syntactically identical to calling a script function. You pass parameters by pushing them onto the stack, you receive return values via _RetVal, and so on. The only major difference lies within the assembler, because you can't just check the specified function name against an array of function information. In fact, you have to save the entire function name string, as-is, in the executable file because you'll need it at runtime (because the host API's functions will not be known at assemble-time). Figure 9.13 illustrates this.



```
CallHost PlaySound
CallHost LoadBMP
CallHost MovePlayer
```

```
9PLAYSOUND
7LOADBMP
10MOVEPLAYER
```

Figure 9.13

Host API function calls being preserved until runtime.

The only real check you can do at assemble-time is make sure the function name string is a valid identifier—in other words, that it consists solely of letters, numbers, and digits, and does not begin with a number.

The _Main () Function

As mentioned, scripts can optionally define a _Main () function that contains code that is automatically executed when the script is run. Scripts that do not include this function are also valid, as they're usually just designed to provide a group of functions to the host application, but neither type of script may include code in the global scope.

Aside from its ability to run automatically and that Param directives are not allowed, the _Main () function does not have any other special properties. Also, for reasons that you'll learn of soon, the _Main function *must* be appended with an Exit instruction (as opposed to Ret, like other functions). This ensures that the script will end properly when _Main () returns.

The _RetVal Register

_RetVal is a special type of operand that can be used in all the same places as variables, arrays, or parameters can be used. You can store any type of variable in it at any time, and use it in any instruction where such an operand would be valid. However, because _RetVal exists permanently in the global scope, its value isn't changed or erased as functions are called and returned; this is what makes it so useful for returning values.

Comments

Lastly, let's talk about comments. Comments are somewhat flexible in XVM Assembly, in the sense that they can easily appear both on their own lines, or can follow the instruction on a line of code. For example:

```
; This is a comment.
Mov    X, Y        ; So is this.
```

Comments are approached in a simple manner; as the assembler scans through the source file, each line is initially preprocessed to strip any comments it contains. This means the code that actually analyzes and processes the source code line doesn't even have to know comments exist, making the code cleaner and easier to write. Because of this, comments have very little impact on the code overall. Because they're immediately stripped away before you have much of a chance to look at them, you can almost pretend they don't exist.

One drawback to comments, however, is that multi-line comments are not supported. Only the single-line `; comment` is recognized by XASM.

A Complete Example Script

That's pretty much all you'll need to know to prepare for the rest of the chapter. Now that I've discussed every major aspect of a script file, you're ready to move on. Before you do, however, it's a good idea to solidify your knowledge by applying everything to a simple example script that demonstrates how things will appear in relation to one another:

```
; Example script

; Demonstrates the basic layout of an XVM
; assembly-language script.

; ---- Globals ----------------------------------------------

    Var    GlobalVar
    Var    GlobalArray [ 256 ]
```

```
; ---- Functions ----------------------------------------------

    ; A simple addition function
    Func MyAdd
    {
        ; Import our parameters
        Param   Y
        Param   X
        ; Declare local data
        Var     Sum
        Mov     Sum, X
        Add     Sum, Y
        ; Put the result in the _RetVal register
        Mov     _RetVal, Sum
        ; Remember, Ret will be automatically added
    }

    ; Just a bizarre function that does nothing in particular
    Func MyFunc
    {
        ; This function doesn't accept parameters
        ; But it does have local data
        Var     MySum
        ; We're going to test the Add function, so we'll
        ; start by pushing two integer parameters.
        Push    16
        Push    32
        ; Next we make the function call itself
        Call    MyAdd
        ; And finally, we grab the return value from _RetVal
        Mov     MySum, _RetVal
        ; Multiply MySum by 2 and store it in GlobalVar
        Mul     MySum, 2
        Mov     GlobalVar, MySum
        ; Set some array values
        Mov     GlobalArray [ 0 ], "This"
        Mov     GlobalArray [ 1 ], "is"
        Mov     GlobalArray [ 2 ], "an"
        Mov     GlobalArray [ 3 ], "array."
    }
```

```
; The special _Main () function, which will be automatically executed
Func _Main
{
    ; Call the MyFunc test function
    Call MyFunc
}
```

Whew! Think you're clever enough to write an assembler that can understand everything here, and more? There's only one way to find out, so let's keep moving.

Output: Structure of an XVM Executable

So you know what sort of input to expect, and you'll learn about the actual processing and assembly of that input in the next section. What that leaves you with now, however, are the details of the output.

As I've mentioned before, XASM will directly output XVM executable files, which have the .XSE (**X**tremeScript **S**cript **E**xecutable) extension. These files are read by the XVM and loaded into memory for execution by the host application. As such, you must make sure you output files that follow the structure the XVM expects exactly.

I'm covering this section here because in the next section, when you actually get to work on implementing XASM itself, it'll be nice to have an idea of what you're outputting so I can refer to the various structures of the executable file without having to introduce them as well. Let's get started.

Overview

.XSE files are tightly-packed binary files that encapsulate assembled scripts. This means there's no extraneous spacing or buffering in between various data elements; each element of the file directly precedes the last.

For the most part, data is written in the form of bytes, words and double words (1-byte, 2-byte and 4-byte structures, respectively). However, floating-point data is written directly to the file as-is using C's standard I/O functions, and as a result, is subject to whatever floating-point format the C compiler for the platform it's compiled on uses. String data is stored as an uncompressed, byte-for-byte copy, but is preceded by a four-byte length indicator, rather than being null-terminated. Check out figure 9.14.

The .XSE format is designed for speed and simplicity, providing a fast, structured method for storing assembled script data in a way that can be loaded quickly and without a lot of drama.

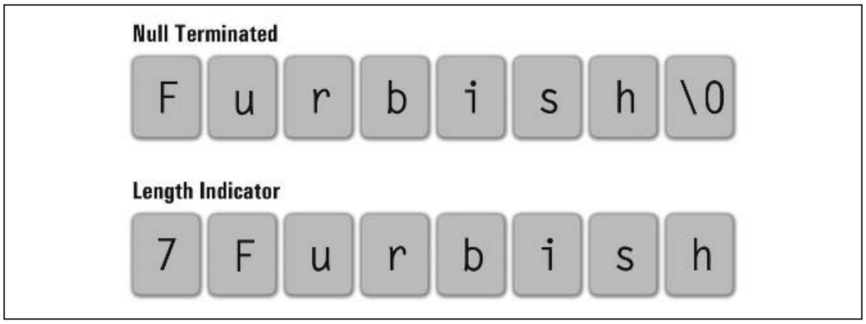


Figure 9.14

Using a string-length indicator instead of a null terminator.

Each field of the file is prefixed by a size field, rather than followed by a terminating flag of some sort. This, for example, allows entire blocks of the file to be loaded into memory very quickly by C's buffered input routines in a single call. In addition to the speed and simplicity by which a file can be loaded, the .XSE format is of course far from human-readable and thus means scripts can be distributed with your games without fear of players being able to hack and exploit your scripts. This can be especially beneficial in the case of multiplayer games where cheating actually has an effect on other human players.

The following subsections each explain a separate component of the file, and are listed in order. Figure 9.15 displays the format graphically, but do read the following subsections to understand the details in full.

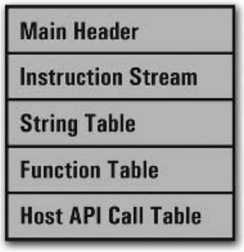


Figure 9.15

An overview of the .XSE executable format.

The Main Header

The first part of the file is the main header, where general information about the script is stored. The main header is the only fixed-size structure in the file, and is described in Table 9.2 and Figure 9.16.

In a nutshell, this header structure contains all of the basic information the XVM will need to handle the script once it's loaded. The ID string is a common feature among file formats; it's the quickest and easiest way to identify the incoming file type without having to perform complex checks on the rest of the structure. This is always set to `"XSE0"`. The version field allows you to specify up to two digits worth of version information, in Major.Minor format. The nice thing about this is that your VM can maintain backwards compatibility with old scripts, even if you make radical changes to the file format, because it'll be able to recognize "legacy" executables. For now you're going to set this for version 0.4. The stack size field, of course, is directly copied from the

Table 9.2 XSE Main Header

Name	Size (in Bytes)	Description
ID String	4	Four-character string containing the .XSE ID, "XSE0"
Version	2	Version number (first byte is major, second byte is minor)
Stack Size	4	Requested stack size (set by SetStackSize directive; 0 means use default)
Global Data Size	4	The total size of all global data
Is _Main () Present?	1	Set to 1 if the script implemented a _Main () function, 0 otherwise.
_Main () Index	4	Index into the function table at which _Main () resides.

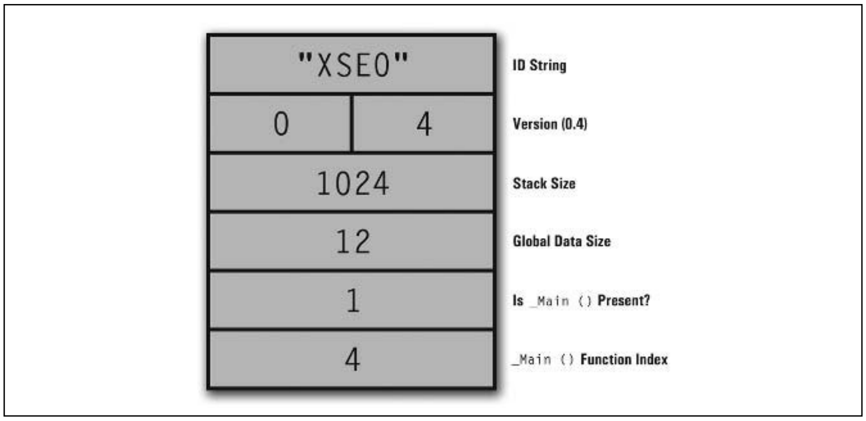


Figure 9.16

The main header.

SetStackSize directive, and defaults to zero if the directive was not present in the script. Following this field is the size of all global data in the program, which is collected incrementally during the assembly phase. Lastly, we store information regarding the _Main () function– the first is a 1-byte flag that just lets us know if it was present at all. If it was, the following field is its 4-byte index into the function table.

The Instruction Stream

The instruction stream itself is the heart of the executable; it of course represents the logic of the script in the form of assembled bytecode. The instruction stream itself is a very simple structure; it consists of a four-byte header that specifies how many instructions are found in the stream (which means you can assemble up to 2^32 instructions total, or well over 4 billion), followed by the actual stream data.

The real complexity lies in the instructions and their representation within the stream. As you learned, encoding an instruction involves a number of fields that help delimit and describe its various components. The instruction stream overall can be thought of as a hierarchical structure consisting of a simple sequence of instructions at its highest level. Within each instruction you find an opcode and an operand stream. Within the operand stream is the operand count followed by the operands themselves. Within each operand you find the operand type, followed by the operand data. Phew! Tables 9.3-9.6 summarize the instruction stream and its various levels of detail.

Overall this might come across as a complex structure, but it's honestly quite simple; just work your way through it slowly and it should all make sense. Check out Figure 9.17 for a visual representation of a sample instruction stream.

Table 9.3 The Instruction Stream Structure

Name	Size (in Bytes)	Description
Size	4	The number of instructions in the stream (not the stream size in bytes)
Stream	N	A variable-length stream of instruction structures

Table 9.4 The Instruction Structure

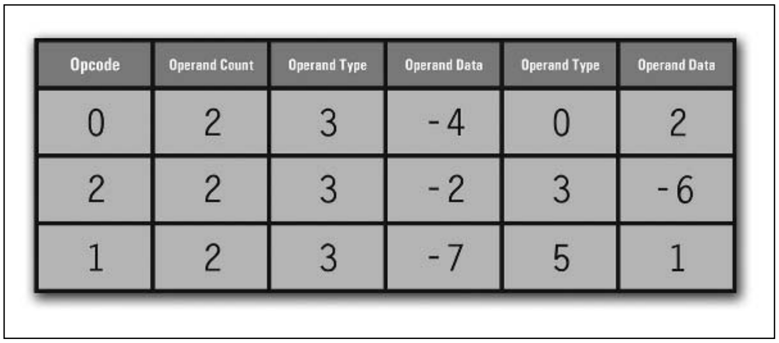
Name	Size (in Bytes)	Description
Opcode	2	The instruction's opcode, corresponding to a specific VM action
Operand Stream	N	Contains the instruction's operand data

Table 9.5 The Operand Stream Structure

Name	Size (in Bytes)	Description
Size	1	The number of operands in the stream (the operand count)
Stream	N	A variable-length stream of operand structures

Table 9.6 The Operand Structure

Name	Size (in Bytes)	Description
Type	1	The type of operand (integer literal, variable, and so on)
Data	N	The operand data itself, which can be any size



Opcode	Operand Count	Operand Type	Operand Data	Operand Type	Operand Data
0	2	3	-4	0	2
2	2	3	-2	3	-6
1	2	3	-7	5	1

Figure 9.17

A sample instruction stream. Note the hierarchical nature of the structure; an instruction stream contains instructions, which (in addition to the opcode) contain operands, which in turn contain operand types and operand data fields.

Operand Types

The last issue regarding the instruction stream is one of the various operand types the operands can assume. In addition to the code for each type, you also need to know what form the operand data itself will be found in. Let's first take a look at the operand type codes themselves, found in Table 9.7.

You'll notice this list differs slightly from the more theoretical version discussed earlier. This one, however, is more suited towards the specific assembly language and virtual machine. Each value in the *Code* column of the table refers to the actual value you'll find in the operand type field.

Some of these fields may be a bit confusing, so let's run through them real quick. First up are the literal values; `integer`, `float`, and `string`. Integers and floats will be written directly into the instruction stream, so they're nothing to worry about. String literals, however, as you learned earlier, are only indirectly represented within the stream. Instead of stuffing the string itself in the operand data field, you use a single integer index that corresponds to a string within the string table (which I'll discuss in more detail later).

Beyond literal values are stack indices, which are used to represent variables in the assembled script. Stack indices come in two forms; one is an *absolute stack index*, which is a single signed integer value that should be used to read from the stack. As usual, negative values mean the index is relative to the top of the stack (local), whereas positives mean the index is relative to the bottom (global). An absolute stack index is used for representing single variables mostly, but is also used for arrays when the index of the array is an integer literal. As you know, if an array called `MyArray []` begins at stack index -8 (known as the array's *base address*), `MyArray [ 4 ]` is simply the base address plus 4. -8 + 4 = -4, so `MyArray [ 4 ]` can be written to the instruction stream simply as -4. The VM doesn't need to know an array was ever even involved; all it cares about is that absolute stack index. From the VM's perspective, creating `MyArray [ 4 ]` is no different than manually creating `MyArray0`, `MyArray1`, `MyArray2` and `MyArray3` as separate, single variables.

Relative stack indices are slightly more complex, and are only used when an array is indexed with a variable. If the assembler encounters `MyArray [ X ]`, it can't tell what the final stack index will be

Table 9.7 Operand Type Codes

Code	Name	Description
0	Integer Literal	An integer literal like `256` or `-1024`.
1	Floating-Point Literal	A floating-point value like `3.14159` or -`987.654`.
2	String Literal Index	An index into the string table representing a string literal value.
3	Absolute Stack Index	A direct index into the stack, like -6 (relative to the top) or 8 (relative to the bottom). Direct stack indices are used for both variables and arrays indexed with a literal value.
4	Relative Stack Index	A base index into the stack that is offset by the contents of a variable's value at runtime. Used for arrays indexed with variables.
5	Instruction Index	An index into the instruction stream, used as jump targets.
6	Function Index	An index into the function table, used for function calls via `Call`.
7	Host API Call Index	An index into the host API call table, used for host API calls via `CallHost`.
8	Register	Code specifying a specific register (currently used only for `_RetVal`).

because the value of `X` won't be known until runtime. So, you instead write the base address of `MyArray []` to the file, followed by the stack index at which `X` resides, so that the VM can add the value of `X` to `MyArray []`'s base address at runtime and find the absolute index. I know this can all come across as complicated, but remember—it's just one level of indirection, which is easy to follow as long as you go slowly. Check out Figure 9.18 for a visual.

You're out of the woods with stack indices, which brings you to the next two codes. The Instruction Index code means the operand contains a single integer value that should be treated as an index into the instruction stream. So, if a line label resolves to instruction `512`, and you make a jump to that label, the operand of that jump instruction will be the integer value `512`.

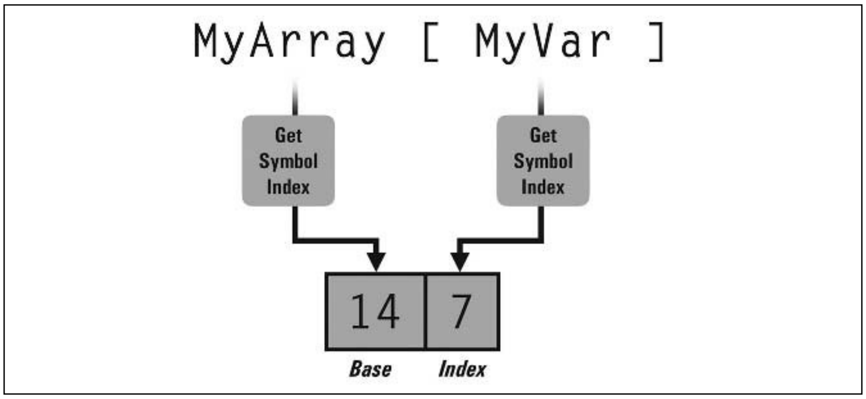


Figure 9.18

Arrays indexed with variables must be stored as relative stack indices*—that is, the array's base index followed by the stack index of the variable whose runtime value will be used to index it.*

The Function Index code is similar, and is used as the operand for the `Call` instruction. Rather than provide a direct instruction index to jump to, however, a function index refers to an element within the function table, which I'll discuss in detail later.

Similar to the Function Call Index is the Host API Call Index. Because the names of the host API's functions aren't known until runtime, you need to store the name string itself in the executable file for use by the VM. The host API call table collects the function name operands accepted by the `CallHost` instruction and saves them to be dumped into the executable file. Much like string literals, these function name strings are then replaced in the instruction stream with an index into the table.

The last operand type is `Register`. The `Register` type uses a single integer code to specify a certain register as an operand, usually as the source or destination in a `Mov` instruction. You'll remember from the last chapter that your VM won't need any registers, with the exception of `_RetVal`. `_RetVal`, used for returning values from functions, is the only register the XVM needs or offers and is therefore specified with code 0. I have, however, allowed for the possibility of future expansion by implementing it this way; if you ever find a need for a new register, you can simply add another code to this operand type, rather than hard-coding new registers in separate operand types.

The String Table

The string table is a simple structure that immediately follows the instruction stream and contains all of a script's string literal values. The indices of this table are *implicit*; in other words, the strings are purposely written out to the table in their proper order, so the string corresponding to index 4 will be the fourth string found in the table, the string corresponding to index 12 will be the twelfth, and so on.

The string table is one of the simpler parts of an .XSE file. It consists of a four-byte header containing the number of strings in the table. The string data itself immediately follows; each string in the table is preceded by its own individual four-byte header specifying the string length. The string length is then followed by the string's characters. Note that the strings are not padded or aligned in any way; if a string's header contains the value 37, the string is exactly 37 characters (not including a null-terminator, because it's not needed here), which in turn means that the next string begins immediately after the 37th character is read. Tables 9.8 and 9.9 outline the string table in its entirety.

Check out Figure 9.19 for a visual layout of the table.

Table 9.8 The String Table Structure

Name	Size (in Bytes)	Description
Size	4	The number of strings in the table (not the total table size in bytes)
Strings	N	String data

Table 9.9 The String Structure

Name	Size (in Bytes)	Description
Size	4	The number of characters in the string
Characters	N	Raw string data itself (*not* null terminated)

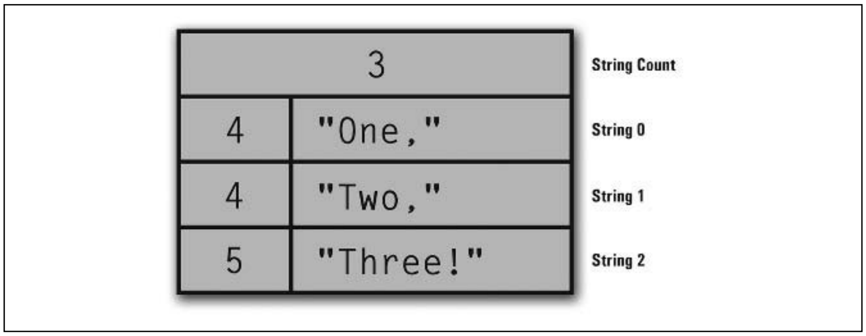


Figure 9.19

A sample string table.

The Function Table

The function table is the .XSE format's next structure and maintains a profile of each function in the script. Each element of the table contains the function's entry point (the index of its first instruction), the number of parameters it takes, and the total size of its local data. This information is used at runtime to prepare stack frames, for example.

As you can see, the total size of the function's stack frame can be derived from this table, by adding the Parameter Count field to the Local Data Size and adding one to make room for the return address. The XVM will use this calculated size to physically create the stack frame as the function is called. This is partially why you can't simply use an instruction index as the operand for a `Call` instruction—the VM needs this additional information to properly facilitate the function call. Lastly, of course, the Entry Point field is used to make the final jump to the function once the stack frame has been prepared.

Table 9.10 The Function Table Structure

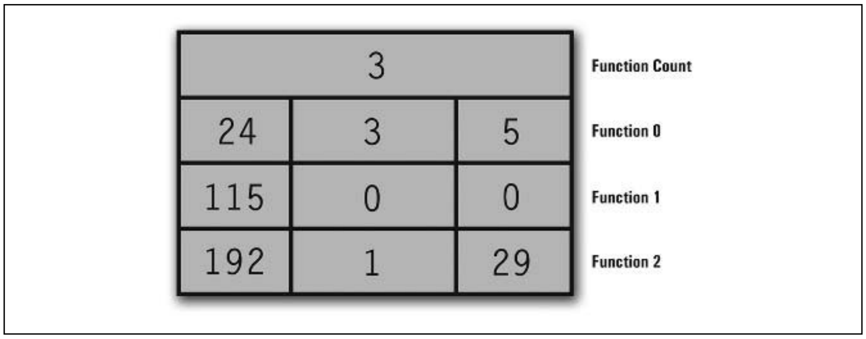
Name	Size (in Bytes)	Description
Size	4	The number of functions in the table.
Functions	N	Function data.

Table 9.11 The Function Structure

Name	Size (in Bytes)	Description
Entry Point	4	The index of the first instruction of the function.
Parameter Count	1	The number of parameters the function accepts.
Local Data Size	4	The total size of the function's local data (the sum of all local variables and arrays).

The `_Main ()` function is also contained in this table, and is always stored at index zero (unless the script doesn't implement `_Main ()`, in which case index zero can be used for something else). The main header of the .XSE file contains a field that lets the VM know whether the `_Main ()` method is present. Note also that the `_Main ()` method will always set the Parameter Count field to zero, because it cannot accept parameters.

Take a look at Figure 9.20, which illustrates the function table.



3			Function Count
24	3	5	Function 0
115	0	0	Function 1
192	1	29	Function 2

Figure 9.20

A sample function table.

The Host API Call Table

As was mentioned, the names of host API functions are not known at runtime. Therefore, you must collect and save the strings that compose the function name operand accepted by the `CallHost` instruction, because the XVM will need them in order to bind host API function calls with the actual host API functions. This is a process called *late binding*.

The Host API Call Table is much like the string literal table; it's simply an array of strings with implicit indices that the instruction stream makes references to. Tables 9.12 and 9.13 list the table and its elements in detail:

Table 9.12 The Host API Call Table Structure

Name	Size (in Bytes)	Description
Size	4	The number of host API calls in the table (not the total table size in bytes)
Host API Calls	N	Host API calls

Table 9.13 The Host API Call Structure

Name	Size (in Bytes)	Description
Size	1	The number of characters in host API function name
Characters	N	The host API function name string (*not* null terminated)

That's basically it. Aside from maybe the instruction stream, which gets a bit tricky, the .XSE format overall is a simple and straightforward structure for storing executable scripts. It's an easy and clean format to both read and write, so you shouldn't have much trouble working with it. Despite its simplicity, however, it's still quite powerful and complete, and will serve you well. Regardless, it's also designed to be expanded, as the built-in version field will allow any changes you make to seamlessly merge with your existing code base. Multiple script versions can certainly co-exist peacefully as long as they can identify themselves properly to the XVM at load-time.

Once again, to help solidify your understanding of the format, is a graphical representation of a basic .XSE file in Figure 9.21.

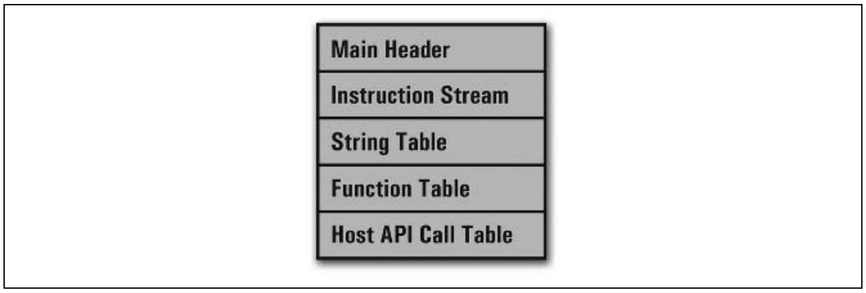


Figure 9.21

Another graphical view of the .XSE file, now that you understand all of its fields and components.

Implementing the Assembler

You now understand the type of input you can expect, and you've got a very detailed idea of what your output will be like. Between these two concepts lies the assembler itself, of course, which translates the input to the output in the first place. At this point you have enough background knowledge on the task at hand to get started.

Before moving on, I'd like to say that what you're about to work on is going to be your first real taste of compiler theory. I discussed some of these principals in a much more simplistic manner back in the command-based language chapters, but what you're about to build is far more complex and a great deal more powerful. The scripts you'll be able to write with this program can do almost anything a C-style language can do (just without the C-style syntax), but that kind of flexibility brings with it a level of internal complexity you're just beginning to understand. I'm going to explain things reasonably slowly, however, so you should be fine as long as you stay sharp and don't rush it.

In a nutshell, the assembler's job is to open an input file, convert its contents, and write the results to an output file. Obviously, the majority of this process is spent in the middle phase; converting the contents. This will be a two-pass process, wherein the first pass scans through the file and collects general information about the script based on directives and other things, and the second pass uses that information to facilitate the assembly of the code itself. To actually explain this process I'm going to switch back and forth between top-down and bottom-up approaches, because it helps to first introduce the basic theory in a bottom-up fashion, and then cover the program itself from a top-down perspective.

Basic Lexing/Parsing Theory

Technically, the principals behind building this assembler will correspond strongly with the underlying field of study known as *compiler theory*. Compiler theory, as the name suggests, concerns itself with the design and implementation of language processors of all sorts, but namely the high-level compilers used to process languages like C, C++, and Java. These general concepts can be applied to any sort of language interpretation and translation, which means it wouldn't be a bad idea to just teach you the stuff now.

However, as you'd suspect, compiler theory is a rough subject that can really chew you up and spit you out if you don't approach it with the right preparation and frame of mind. Furthermore, despite its relative difficulty, it just flat-out takes a long time to cover. This is the only chapter you're going to spend on the construction of XASM, so there's just no room for a decent compiler theory primer either way.

Fortunately, you can get by without it. The type of translation you'll be doing as you write XASM is so minimal by comparison to the translation of a language like C, that you can easily make do with an ad-hoc, bare minimum understanding. Don't worry though, because you're only a few chapters away from the section of the book that deals with the construction of the XtremeScript compiler. *That's* where I'll wheel out the big guns, and you'll learn intermediate compiler theory the right way (you'll need it, too). Until then, I'll keep it simple.

This section, then, will proceed with highly simplified discussions of the two major techniques you'll be employing in the implementation of XASM—lexing and parsing. Together, these two concepts form the basis for a language processor capable of understanding, validating, and translating XVM Assembly Language.

Lexing

To get things started, let's once again consider the `Add` function, a common example throughout the last two chapters:

```
Func MyAdd
{
    Param  X                 ; Assign names to the two parameters
    Param  Y
    Var    Sum               ; Create a local variable for the sum
    Mov    Sum, X            ; Perform the addition
    Add    Sum, Y
    Mov    _RetVal, Sum      ; Store the sum in the _RetVal register
}
```

To humans it's simple, but it seems like it'd be pretty complicated for a piece of software to somehow understand it, right? This is true; being able to scan through this block of code, character by character, and manage to do everything an assembler does *is* complicated. But like most complicated things, it all starts with the basics.

The first thing to understand is that not everything the assembler is going to do overall will be done at once. Language processors almost invariably work in incremental phases, wherein each phase focuses on a small, reasonably simple job, thus making the job of the following phase even simpler. Together these phases form a pipeline, at each stage of which the source is in a progressively more developed, validated, or translated form.

Generally speaking, the first phase when translating any language is *lexical analysis.* Lexical analysis, or *lexing* for short, is the process of breaking up the source file into its constituent "words". These "words", in the context of lexing, are known as *lexemes.* For example, consider the following line of code:

```
Mov    Sum, X            ; Perform the addition
```

This line contains four separate lexemes; `Mov`, `Sum`, `,` (the comma), and `X` (note that the whitespace and comments are automatically stripped away and do not count). Already you should see how much easier this makes your task. Right off the bat, the lexer allows the users to fill their code with as much whitespace and commenting as they want, and you never have to know about it. As long as the lexer can filter this content out and simply provide the lexemes, you get each isolated piece of code presented in a clean, clutter-free manner. But the lexer does a lot more than just this.

The unfiltered source code, as it enters your assembler's processing pipeline, is called a *character stream*, because it's a stream of raw source code expressed as a sequence of characters. Once it passes through the first phase of the lexer, it becomes a *lexeme stream*, because each element in the stream is now a separate lexeme. Figure 9.22 helps visualize this.

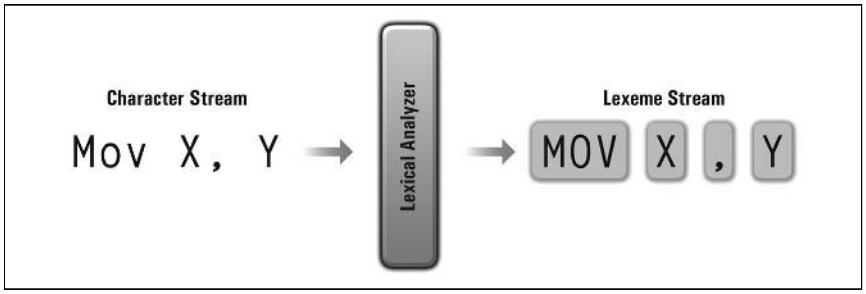


Figure 9.22

A character stream becoming a lexeme stream.

In addition to isolating and extracting lexemes, the real job of the lexer is to convert the lexeme stream to a *token stream*. *Tokens*, unlike lexemes, are not strings at all; rather, they're simple codes (usually implemented as integers) that tell you what exactly the lexeme *is*. For example, the line of code used in the last example, after being converted to a lexeme stream, looks like this (note that for simplicity, everything is converted to uppercase by the lexer):

```
MOV SUM , X
```

The new stream of lexemes is indeed easier to process, but take a look at the token stream (each element in the following stream is actually a numeric constant):

```
TOKEN_TYPE_INSTR TOKEN_TYPE_IDENT TOKEN_TYPE_COMMA TOKEN_TYPE_IDENT
```

Just for reference, it might be easier to mentally process the token stream when it's listed vertically:

```
TOKEN_TYPE_INSTR
TOKEN_TYPE_IDENT
TOKEN_TYPE_COMMA
TOKEN_TYPE_IDENT
```

NOTE

Technically, lexers and tokenizers are two different objects, but they work so closely together and are so similar that they're usually referred to and even implemented as a singular object.

Do you understand what's happened here? Instead of physically dealing with the lexeme strings themselves, which is often only of limited use, you can instead just worry about the token type. As you can see by looking at the original line of code, the token stream tells you that it consists of an instruction (`TOKEN_TYPE_INSTR`), an identifier, (`TOKEN_TYPE_IDENT`), a comma, (TOKEN_TYPE_COMMA), and finally another identifier. These tokens of course directly correspond to Mov, Sum, ,, and X, respectively. This process of turning the lexeme stream into a token stream is known as *tokenization*, and because of this, lexers are often referred to as *tokenizers*.

Without getting into the nitty gritties, I can tell you that the lexer is one of the easier parts of a compiler (or assembler) to build. Yet, as you can see, its contribution to the overall language-processing pipeline is considerable. After only the first major stage of translation, you can already tell, on a basic level, what the script is trying to say. Of course, simply converting the character stream to a token stream isn't enough to understand everything that's going on. To do this, you must advance to the next stage of the pipeline.

Parsing

The parser immediately follows the lexer and tokenizer in the pipeline, and has a very important job. Given a stream of tokens, the parser is in charge of piecing together its overall meaning when taken as a collective unit. So, although the tokenizer is in charge of breaking down the source file from a giant, unruly string of characters to a collection of easy-to-use tokens, the parser takes those tokens and builds them back up again, but into a far more structured view of the overall source code. See Figure 9.23.

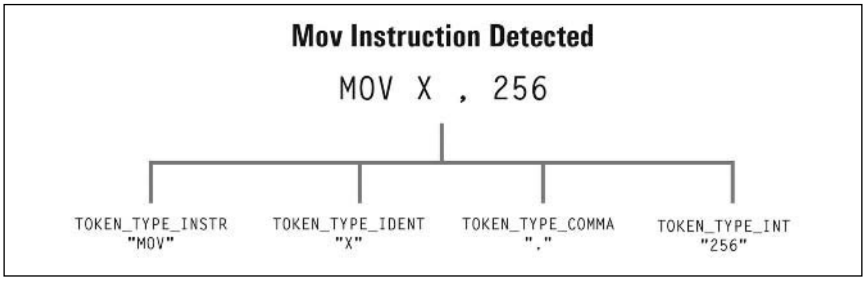


Figure 9.23

The parser uses tokens and lexemes to determine what the source code is trying to say.

There are many approaches to parsing, and building a parser is easily one of the most complex aspects of building a compiler. Fortunately, certain methods of parsing are easier than others, and the easy ones can be applied quite effectively to XASM.

In this chapter, you won't have to worry about the fine points of parsing theory and all the various terms and concepts that are associated with it. Rather, you're going to take a somewhat brute-force approach that, although not necessarily as clever as some of the methods you'll find in a compiler theory textbook, definitely get the job done in a clean, highly structured, and, dare I say, somewhat elegant manner.

In a nutshell, the parser will read groups of tokens until it finds a pattern between them that indicates the overall purpose of that particular token group. This process starts by reading in a single token. Based on this initial token's type, you can predict what tokens should theoretically come next, and compare that to the actual token stream. If the tokens match up the way you think they do, you can group them as a logical unit and consider them valid and ready to assemble. Figure 9.24 illustrates this.

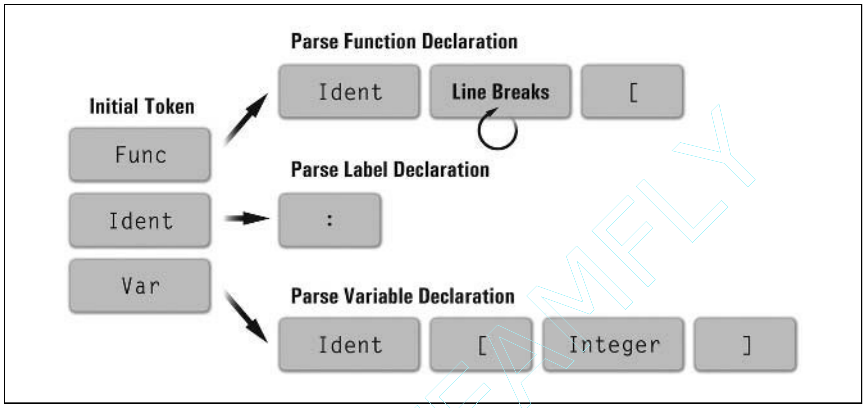


Figure 9.24

Each initial token invokes a different parsing sequence.

I think an example is in order. Imagine a new fragment of example code:

```
Func MyFunc {
```

As you can see, this is the beginning of a function declaration. It's cut off just before the function's code begins, because all you're worried about right now is the declaration itself. After the lexer performs its initial breakdown of the character stream, the tokenizer will go to work examining the incoming lexemes and convert them to a token stream. The token stream for the previous line of code will consist of:

```
TOKEN_TYPE_FUNC
TOKEN_TYPE_IDENT
TOKEN_TYPE_OPEN_BRACKET
```

Notice that you can reserve an entire token simply for the `Func` directive. This is common among reserved words; for example, a C tokenizer would consider the `if`, `while`, and `for` keywords to each be separate tokens. Anyway, with the tokens identified, the parser will be invoked and the second step towards assembly will begin.

The parser begins by requesting the first token in the stream from the tokenizer, which will return `TOKEN_TYPE_FUNC`. Based on this, the parser will immediately realize that a function declaration must be starting. This is how you predict which tokens must follow based on the first one read. Armed with the knowledge of XVM Assembly, you know that a function declaration consists of the `Func` keyword, an identifier that represents the function's name, and the open bracket symbol. So, the following two tokens *must* be TOKEN_TYPE_IDENT and TOKEN_TYPE_OPEN_BRACKET. If either of these tokens is incorrect, or if they appear in the wrong order, you've detected a syntax error and can halt the assembly process to alert the users. If these two tokens are successfully read, on the other hand, you know the function declaration is valid and can record the function in some form before moving on to parse the next series of tokens.

Check out the following pseudo-code, which illustrates the basic parsing process for a function declaration:

```
Token CurrToken = GetNextToken ();   // Read the next token from the stream
if ( CurrToken == TOKEN_TYPE_FUNC )  // Is a function declaration starting?
{
    if ( GetNextToken () == TOKEN_TYPE_IDENT ) // Look for a valid identifier
    {
        string FuncName = GetCurrLexeme (); // The current lexeme is the
                                            // function name, so save it
        if ( GetNextToken () != TOKEN_TYPE_OPEN_BRACKET ) // Make sure the open
                                                          // bracket is present
        {
            Error ( "'{' expected." );
        }
        Error ( "Identifier expected." );
    }
}
// Check for remaining token types...
```

The code starts by reading a single token from the stream using GetNextToken (). It then determines whether the token's type is TOKEN_TYPE_FUNC. If so, it begins the code that parses a function declaration, which consists of reading and validating the identifier (function name) and then ensuring the presence of the open bracket. If a valid identifier is found, it's saved to the string variable FuncName.

Remember, the token itself is *not* the function name; the token is simply a code representing the type of the current lexeme (in this case, an identifier). The lexeme itself is what you want to copy, because it's the actual string containing the function's name. Therefore, you use the function GetCurrLexeme () to get the lexeme associated with the current token (which you got with GetNextToken ()). If the token associated with the function name lexeme is not of type TOKEN_TYPE_IDENT, it means a non-identifier lexeme was read, such as a number or symbol (or some other invalid function name). In this case, you use the Error () function to report the error that an identifier was expected. If an identifier was found, you proceed to verify the presence of the open bracket token, and use Error () again to alert the users that the open bracket was expected if it's not found.

Hopefully this has helped you understand the general process of parsing. Along with lexing and tokenization, you should at least have a conceptual idea of how this process works. Once you've properly parsed a given group of tokens, you're all set to translate it. After parsing an instruction, for example, you use the instruction lookup table to verify its operands and convert it to machine code. In the case of directives like `Func`, you add a new entry to the function table (which, if you recall, stores information on the script's functions, like their entry points, parameter counts, and local data sizes).

With the basic idea behind lexing, parsing, and ultimately translation under your belt, let's move forward and start to learn how these various concepts are actually implemented.

Basic String Processing

As you should already be able to tell simply by looking at the last few examples, the process of translating assembly source to machine code will involve *massive* amounts of string processing. Especially in the case of the lexer and tokenizer, almost everything you do will involve the analysis, manipulation, or conversion of string data. So, before you take another step forward, you need to make a quick detour into the world of string processing and put together a small but vital library of functions for managing the formidable load of string processing that awaits you.

Vocabulary

You have to talk the talk in order to understand what's going on. In the case of string processing, there's a small vocabulary of terms you'll need to have under your belt in order to follow the discussion. Most of this stuff should be second nature to you, as high-level programming tends to involve a certain amount of string processing by nature, but I'll go over them anyway just to be sure you're on the same page.

The Basics

On the most basic level, as you obviously know, a *string* is simply a sequence of *characters*. Each character represents one of the symbols provided by the ASCII character set, or whichever character set you happen to be using. Other examples include Unicode, which uses 16-bits to represent a character rather than the 8-bits ASCII uses, which gives it the ability to reference up to 65,536 unique characters as opposed to only 255. You're of course concerning yourself only with ASCII for now.

Substrings

A *substring* is defined as a smaller, contiguous chunk of a larger string. In the string `"ABCDEF"`, `"ABC"`, `"DEF"`, and `"BCD"` are all examples of substrings. A substring is defined by two indices: the *starting index* and the *ending index*. The substring data itself is defined as all characters between and including the indices.

Whitespace

Whitespace can exist in any string, and is usually defined simply as non-visible characters such as spaces, tabs, and line breaks. However, it is often important to distinguish between whitespace that includes line breaks, and whitespace that doesn't. For example, in the case of C, where statements can span multiple lines, whitespace can include line breaks because the line break character itself doesn't have meaning. However, in the case of most assembly languages, including yours, whitespace cannot include line breaks because the line break character is used to represent the end of instructions.

A common whitespace operation is *trimming*, also known as *clamping* or *chomping*, wherein the whitespace on either or both sides of a string is removed. Take the following string for example:

```
"   This is a padded string.    "
```

A *left trim* would remove all whitespace on the string's left side, transforming it into:

```
"This is a padded string.    "
```

A *right trim* would remove all whitespace on the string's right side, like this:

```
"   This is a padded string."
```

Lastly, a full trim would produce:

```
"This is a padded string."
```

Trimming is often done by or before the lexing phase to make sure extraneous whitespace is removed early in the pipeline.

Classification

Strings and characters can be grouped and categorized in a number of ways. For example, if a character is within the range of `0..9`, you can say that string is a numeric digit. If it's within the range of `a..z` or `A..Z`, you can say it's an alphabetic character. Additionally, if it's within the range of `0..9`, `a..z` or `A..Z`, you can call it an *alphanumeric*, which is the union of both numeric digits and alphabetic characters.

This sort of classification can be extended to strings as well. For example, a string consisting entirely of characters that each individually satisfies the requirements of being considered numeric digits can be considered a numeric string. Examples include "`111`", "`123456`", "`0034870523850235`" and "`6`". By prefixing a numeric string with an optional negation sign (`-`), you can easily extend the class of numeric strings to *signed* numeric strings. By further adding the allowance of one radix point (`.`) somewhere within the string (but not before the sign, if present, and not after the last digit), you can create another class called *signed floating-point* numeric strings. See figure 9.25 for a visual.

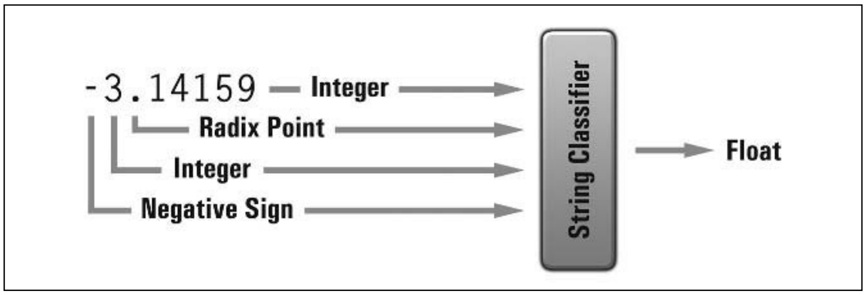


Figure 9.25

String classification.

As you can see, this sort of classification is a useful and frequent operation when developing an assembler or compiler. You'll often have to validate various string types, ranging from identifiers to floating point numbers to single characters like open brackets and double quotes. This is also a common function when determining a lexeme's corresponding token. Your string-processing library will include an extensive collection of string-classification functions.

A String-Processing Library

As the assembler is written, you'll find that what you need most frequently are string classification functions. Substring extraction and other such operations are performed much less frequently, so you'll usually just hardcode them where you need them.

Let's start small by writing up a collection of functions you can use to classify single characters. Generally, as you work your way through the source code, you'll need to know if a given character is any of the following things:

- A numeric digit (`0-9`).
- A character from a valid identifier (`0-9`, `a-z`, `A-Z`, or _ [underscore]).
- A whitespace character (space or tab).
- A delimiter character (something that separates elements; braces, commas, and so on).

Generally, these characters are easy to detect. I'll just show you the source to each function (in actual C, because this is a much lower-level operation), because they should be pretty self-explanatory:

```
// Determines if a character is a numeric digit
int IsCharNumeric ( char cChar )
{
    // Return true if the character is between 0 and 9 inclusive.
    if ( cChar >= '0' && cChar <= '9' )
        return TRUE;
    else
        return FALSE;
}

// Determines if a character is whitespace
int IsCharWhitespace ( char cChar )
{
    // Return true if the character is a space or tab.
    if ( cChar == ' ' || cChar == '\t' )
        return TRUE;
    else
        return FALSE;
}

// Determines if a character could be part of a valid identifier
int IsCharIdent ( char cChar )
{
    // Return true if the character is between 0 or 9 inclusive or is
    // an uppercase or lowercase letter
    if ( ( cChar >= '0' && cChar <= '9' ) ||
         ( cChar >= 'A' && cChar <= 'Z' ) ||
         ( cChar >= 'a' && cChar <= 'z' ) ||
           cChar >= '_' )
        return TRUE;
    else
        return FALSE;
}

// Determines if a character is part of a delimiter
int IsCharDelimiter ( char cChar )
{
    // Return true if the character is a delimiter
    if ( cChar == ':' || cChar == ',' || cChar == '"' ||
         cChar == '[' || cChar == ']' ||
         cChar == '{' || cChar == '}' ||
         IsCharWhitespace ( cChar ) )
        return TRUE;
    else
        return FALSE;
}
```

Simple enough, right? Each function basically works by comparing the character in question to either a set of specific characters or a range of characters and returning `TRUE` or `FALSE` based on the results.

Now that you can classify individual characters, let's expand the library to include functions for doing the same with strings. Because these functions are a bit more complex than their single-character counterparts, I'll introduce and explain them individually.

Let's first write some numerical classification functions. One immediate difference between characters and strings is that there's no differentiation between an "integer character" and a "float character", because a numeric character is simply defined as being within the range of `0..9`. With strings however, there's the possibility of the radix point being involved, which allows you to differentiate between integers and floats. Let's first see some code for classifying a string as an integer:

```
int IsStringInt ( char * pstrString )
{
    if ( ! pstrString )
        return FALSE;

    if ( strlen ( pstrString ) == 0 )
        return FALSE;

    unsigned int iCurrCharIndex;

    for ( iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( ! IsCharNumeric ( pstrString [ iCurrCharIndex ] ) &&
             ! ( pstrString [ iCurrCharIndex ] == '-' ) )
            return FALSE;

    for ( iCurrCharIndex = 1;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( pstrString [ iCurrCharIndex ] == '-' )
            return FALSE;

    return TRUE;
}
```

Essentially what you're doing here is simple. First, you do some initial checks to make sure the string pointer is valid and not empty. You then make an initial scan through the string to make sure that all characters are either numeric digits or the negation sign. Of course, at this stage, a number like `-867-5309` would be considered valid. So, to complete the process, you make one more scan through to make sure that the negation sign, if present at all, is only the first character.

NOTE

You'll notice that my implementation of these string-classification functions isn't necessarily the most efficient or most clever. Often, state machines are used for string validation and classification, and provide an elegant and generic mechanism for such operations. However, because the theme of this chapter has consistently and intentionally been "watered down compiler theory that just gets the job done", my focus is more on readable, intuitive solutions.

So you can classify integer strings, but what about floats? Well, it's more or less the same principal, the only difference being the radix point you now have to watch for as well.

```
int IsStringFloat ( char * pstrString )
{
    if ( ! pstrString )
         return FALSE;

    if ( strlen ( pstrString ) == 0 )
         return FALSE;

    unsigned int iCurrCharIndex;

    for ( iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( ! IsCharNumeric ( pstrString [ iCurrCharIndex ] ) &&
             ! ( pstrString [ iCurrCharIndex ] == '.' ) &&
             ! ( pstrString [ iCurrCharIndex ] == '-' ) )
            return FALSE;

    int iRadixPointFound = FALSE;

    for ( iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( pstrString [ iCurrCharIndex ] == '.' )
            if ( iRadixPointFound )
                return FALSE;
            else
                iRadixPointFound = TRUE;

    for ( iCurrCharIndex = 1;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( pstrString [ iCurrCharIndex ] == '-' )
            return FALSE;

    if ( iRadixPointFound )
        return TRUE;
    else
        return FALSE;
}
```

Once again, you start off with the typical checks for bad strings. You then move on to make sure the number consists solely of numbers, radix points, and negation signs. Once you know the characters themselves are all valid, you make sure the semantics of the number are correct as well, insomuch as there's only one radix point and negation operator.

With the numeric classification functions out of the way, let's move on to something a bit more abstract—determining whether a string is whitespace. Here's the code:

```
int IsStringWhitespace ( char * pstrString )
{
    if ( ! pstrString )
        return FALSE;

    if ( strlen ( pstrString ) == 0 )
        return TRUE;

    for ( unsigned int iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( ! IsCharWhitespace ( pstrString [ iCurrCharIndex ] ) )
             return FALSE;

    return TRUE;
}
```

This is a very simple function; all that's necessary is to pass each character in the string to our previously defined `IsCharWhitespace ()` function and exit if non-whitespace is found. One extra note, however—note that unlike the last two functions you've written, this function returns `TRUE` in the event of an empty string. You do this because a lack of characters can usually be considered whitespace as well.

Let's write one more, shall we? To make sure each of your character classifying functions has a corresponding string version, you need to make a function for determining whether a string is a valid identifier. Let's take a look:

```
int IsStringIdent ( char * pstrString )
{
    if ( ! pstrString )
        return FALSE;

    if ( strlen ( pstrString ) == 0 )
        return FALSE;

    if ( pstrString [ 0 ] >= '0' && pstrString [ 0 ] <= '9' )
        return FALSE;

    for ( unsigned int iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrString );
          ++ iCurrCharIndex )
        if ( ! IsCharIdent ( pstrString [ iCurrCharIndex ] ) )
        return FALSE;

    return TRUE;
}
```

This one's pretty easy too—all it does is make sure the first character is not a digit (which isn't allowed in an identifier), and then uses `IsCharIdent ()` to make sure that each subsequent character is a valid identifier character.

The Assembler's Framework

To begin implementing the assembler itself, you must first establish the major structures and helper functions that the lexer and parser will need as they traverse and assemble the source file. There's quite a bit of data to be managed as this process progresses, much of which won't make it to the executable file but rather will help shape that executable's final form.

The General Interface

Just to get it out of the way, let's start with a description of how the assembler will be implemented specifically. XASM will be a simple console application, which makes the code portable and the interface easy to design. The user will specify the input and output files using command-line parameters, and all messages to be displayed (error messages, a summary of script statistics gathered during the assembly process, and so on) will be written directly to the console as well (as opposed to a log file or something along those lines). Here's a simple usage example:

```
XASM MyScript.XASM
```

This will compile `MyScript.xasm` into `MyScript.xse`, producing the executable in the same directory. If, for whatever reason, the user wants the executable to have a different name, this can be specified as a second, optional parameter:

```
XASM MyScript.XASM MyExec.XSE
```

Note also that the assembler will automatically detect and append missing file extensions, so `MyScript` and `MyExec` will be considered just as valid as `MyScript.XASM` and `MyExec`.XSE.

A Structural Overview

With the general interface out of the way, let's check out a bird's eye view of the assembler and its major internal structures. One thing I'd like to mention up front is that the assembler is primarily composed of tables for managing various script-defined elements, like variables, functions, and labels. Because the quantity of these elements will vary significantly from script to script, linked lists will be employed for the majority of these tables to allow them to incrementally grow only as large as is necessary.

The actual implementation of the lists can vary from project to project and from coder to coder. I personally think a simple C++ class that provides basic linked list functionality (or possibly one provided by the STL) is the cleanest way to go. Others may write a generic, pure-C implementation that can be used in the same way. Still others may simply hard code the list over and over again for each table so that a generic structure isn't involved in any way. When writing your own assembler (or compiler, or any of these programs), just go with whatever you're most comfortable with. For absolute simplicity's sake, I'll be using a very basic C implementation.

Source Code Representation

As I mentioned previously, I decided to buffer everything in memory rather than incrementally read from the source file on the hard drive. Overall, this makes the process faster, and it's just easier to work with the data when it's immediately available in arrays.

At load time, the number of lines in the source file will be counted, and a suitably sized array of static strings called `g_ppstrSourceCode` will be allocated. These static strings will be large enough to hold what you predefine as the largest possible line the assembler supports. I usually use `4096` for this value. Chances are this is *much* bigger than anything you will ever need, but you never know. Besides, it's easy to change if you feel the need to do so. Here's the declaration and allocation of the structure:

```
#define MAX_SOURCE_LINE_SIZE        4096

char ** g_ppstrSourceCode = NULL;
int g_iSourceCodeSize;
```

When the loading of the source script is complete, you'll have a separate string representing each line of code easily and sequentially accessible in memory. Figure 9.26 illustrates the source code array in relation to the source on disk.

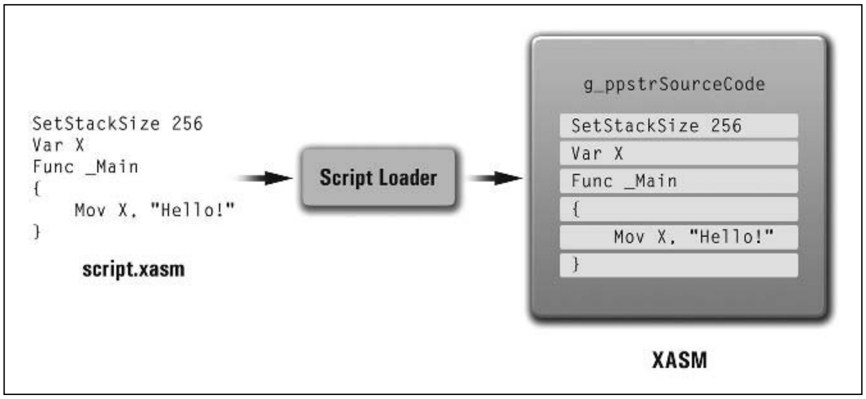


Figure 9.26

The source code array in relation to the original source file.

The Assembled Instruction Stream

In addition to buffering the incoming source code, you'll also buffer the outgoing assembled instruction stream. Just as the source file is loaded once and then forgotten about, the output file (the executable) will be entirely out of the picture during the assembly process; only when the process has completed in full will the output file be opened, written to in one quick phase, and closed.

The storage of the assembled instruction stream in memory will almost directly mimic the structure it will be stored with in the executable file, which I discussed earlier. This means it follows the same hierarchical form, and therefore must exist as a number of nested structures that are ultimately put to use in a large array.

Instructions

The instruction structure will need to contain the instruction's opcode, the number of operands it accepts, and a pointer to the operand data itself:

```
typedef struct _Instr          // An instruction
{
    int iOpcode;               // Opcode
    int iOpCount;              // Number of operands
    Op * pOpList;              // Pointer to operand list
}
    Instr;
```

Operands

The `Op` (operand) pointer points to a dynamically allocated `Op` array. The `Op` structure itself looks like this:

```
typedef struct _Op             // An assembled operand
{
    int iType;                 // Type
    union                      // The value
    {
        int iIntLiteral;       // Integer literal
        float fFloatLiteral;   // Float literal
        int iStringTableIndex; // String table index
        int iStackIndex;       // Stack index
        int iInstrIndex;       // Instruction index
        int iFuncIndex;        // Function index
        int iHostAPICallIndex; // Host API Call index
        int iReg;              // Register code
    };
    int iOffsetIndex;          // Index of the offset
}
```

It primarily consists of a `union` nested inside a larger `struct`. The `union` structure was chosen because in a typeless language, any operand can contain any data type at any time; an efficient way to support each of these types simultaneously is to let them share the same memory locations. Of course, certain fields needed to be kept separate and were thus declared outside of the `union`. These were `iType`, which is used to determine which data type is currently being used, and `iOffsetIndex`. Figure 9.27 depicts the instruction stream in action:

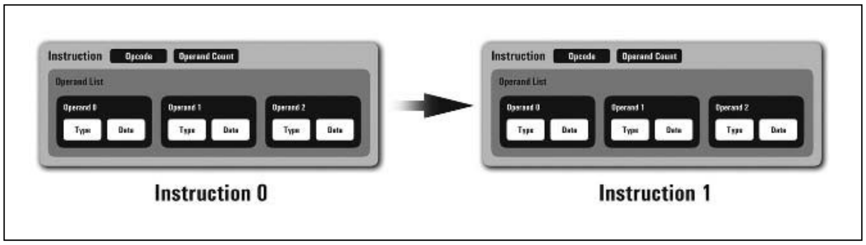


Figure 9.27

The assembled instruction stream structure.

`iOffsetIndex` is only used when the active data type within the `union` is `iStackIndex`. In the cases where an operand is defined as a *relative* stack index, we need to store the base index and the offset. Since we can't have two members of the `union` active at the same time without overwriting each other, the offset field is kept separate.

During the first pass, the number of instructions will be counted, and `g_pInstrStream []` will be allocated with this number before the start of the second pass.

Here's the declaration

```
Instr * g_pInstrStream = NULL;
int g_iInstrStreamSize;
```

The Script Header

The .XSE file provides a main header data area that provides general information in regards to the script as a whole, and you'll store in an internal structure for gathering and maintaining some of the data. You'll call it `g_ScriptHeader`, and it'll simply be an instance of the `ScriptHeader` structure:

```
typedef struct _ScriptHeader    // Script header data
{
    int iStackSize;             // Requested stack size
    int iGlobalDataSize;        // The size of the script's global data
    int iIsMainFuncPresent;     // Is _Main () present?
    int iMainFuncIndex;         // _Main ()'s function index
}
    ScriptHeader;
```

As you can see, you don't need to represent the entire header in this structure. The ID string and version numbers can simply be kept in `#define`s and written at the last moment to the output file because they won't change on a per-script basis.

A Simple Linked List Implementation

All of the remaining structures in XASM are built on linked lists to allow them to grow dynamically as the source file is assembled. Before we go any further, I'm going to cover a simple C linked list implementation that will be the basis for the remaining tables.

Linked lists consist of two structures: the list itself, and the node. Here they are:

```
typedef struct _LinkedListNode  // A linked list node
{
    void * pData;               // Pointer to the node's data
    _LinkedListNode * pNext;    // Pointer to the next node in the list
}
    LinkedListNode;

typedef struct _LinkedList      // A linked list
{
    LinkedListNode * pHead,    // Pointer to head node
                   * pTail;    // Pointer to tail node
    int iNodeCount;            // The number of nodes in the list
}
    LinkedList;
```

The list structure itself is very generic, but the key is the `pData` pointer in the node structure. Since this is a `void` pointer, it can be used to store anything, which makes the list flexible enough to handle all of XASM's tables.

Lists can be declared easily using these structures like so:

```
LinkedList MyList;
```

This structure is illustrated in Figure 9.28.

Once you've created a list, it needs to be initialized. This is performed with a call to `InitLinkedList ()`:

```
void InitLinkedList ( LinkedList * pList )
{
    // Set both the head and tail pointers to null
    pList->pHead = NULL;
    pList->pTail = NULL;

    // Set the node count to zero, since the list is currently empty
    pList->iNodeCount = 0;
}
```

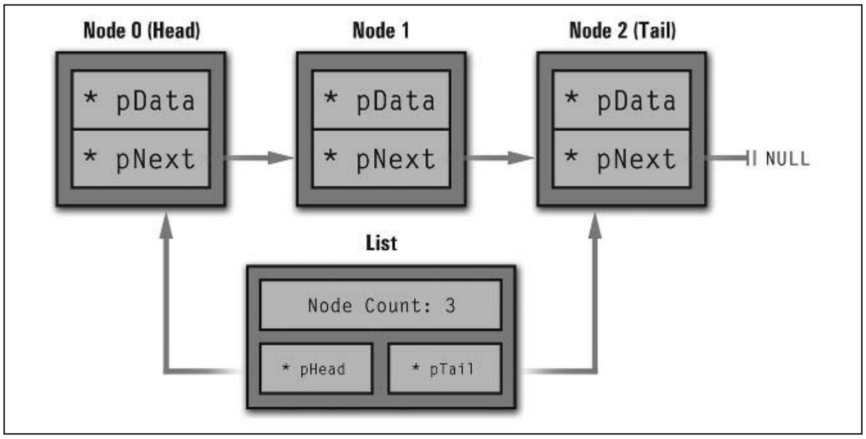


Figure 9.28

A basic linked list structure.

All this function does is set the head and tail pointers to `NULL`, and set the node count to zero. Once the list is initialized, you can start adding nodes to it with `AddNode ()`:

```
int AddNode ( LinkedList * pList, void * pData )
{
    // Create a new node
    LinkedListNode * pNewNode = ( LinkedListNode * )
       malloc ( sizeof ( LinkedListNode ) );

    // Set the node's data to the specified pointer
    pNewNode->pData = pData;

    // Set the next pointer to NULL, since nothing will lie beyond it
    pNewNode->pNext = NULL;

    // If the list is currently empty, set both the head and tail pointers
    // to the new node
    if ( ! pList->iNodeCount )
    {
        // Point the head and tail of the list at the node
        pList->pHead = pNewNode;
        pList->pTail = pNewNode;
    }

    // Otherwise append it to the list and update the tail pointer
    else
    {
        // Alter the tail's next pointer to point to the new node
        pList->pTail->pNext = pNewNode;
        // Update the list's tail pointer
        pList->pTail = pNewNode;
    }

    // Increment the node count
    ++ pList->iNodeCount;

    // Return the new size of the linked list - 1, which is the node's index
    return pList->iNodeCount - 1;
}
```

The function begins by allocating space for the node and initializing its pointers. The node count of the list is then checked– if the list is empty, this node will become both the head and tail, and the `pHead` and pTail pointers should be updated accordingly. If not, the node becomes the new tail, which requires the list's `pTail` to be updated, as well as the `pNext` pointer of the old tail node. Lastly, the node count is incremented and the list's new size is returned to the caller (which is actually treated as the new node's index).

When you're done with the list, the memory used for both its data and the nodes themselves must be freed. This is handled with `FreeLinkedList ()`:

```
void FreeLinkedList ( LinkedList * pList )
{
    // If the list is empty, exit
    if ( ! pList )
        return;

    // If the list is not empty, free each node
    if ( pList->iNodeCount )
    {
        // Create a pointer to hold each current node and the next node
        LinkedListNode * pCurrNode,
                       * pNextNode;

        // Set the current node to the head of the list
        pCurrNode = pList->pHead;

        // Traverse the list
        while ( TRUE )
```

```
            {
                // Save the pointer to the next node before freeing the current one
                pNextNode = pCurrNode->pNext;

                // Clear the current node's data
                if ( pCurrNode->pData )
                    free ( pCurrNode->pData );

                // Clear the node itself
                if ( pCurrNode )
                    free ( pCurrNode );

                // Move to the next node if it exists; otherwise, exit the loop
                if ( pNextNode )
                    pCurrNode = pNextNode;
                else
                    break;
            }
        }
    }
```

The function boils down to a loop that iterates through each node and frees both it and its data.

We now have a linked list capable of implementing each of the tables XASM will need to maintain. Let's have a look at the tables themselves.

The String Table

As the script's instructions are processed, string literal values will most likely pop up here and there. Because you want to remove these from the outgoing instruction stream and instead replace them with references to a separate table, this table will need to be constructed, as well an appropriate set of functions for interfacing with it.

The table is built on the linked list covered in the previous section, which means there's not a whole lot left to implement. The table's declaration is also quite simple:

```
LinkedList g_StringTable;
```

The `pData` member in each node will simply point to a typical C-style null-terminated string, which means all that's necessary is creating a simple wrapper based around `AddNode ()` that will make it easy to add strings directly to the table from anywhere in the program. This function will appropriately be named `AddString`.

```
int AddString ( LinkedList * pList, char * pstrString )
{
    // ---- First check to see if the string is already in the list

    // Create a node to traverse the list
    LinkedListNode * pNode = pList->pHead;

    // Loop through each node in the list
    for ( int iCurrNode = 0; iCurrNode < pList->iNodeCount; ++ iCurrNode )
    {
        // If the current node's string equals the specified string, return
        // its index
        if ( strcmp ( ( char * ) pNode->pData, pstrString ) == 0 )
             return iCurrNode;
        // Otherwise move along to the next node
        pNode = pNode->pNext;
    }

    // ---- Add the new string, since it wasn't added

    // Create space on the heap for the specified string
    char * pstrStringNode = ( char * ) malloc ( strlen ( pstrString ) + 1 );
    strcpy ( pstrStringNode, pstrString );

    // Add the string to the list and return its index
    return AddNode ( pList, pstrStringNode );
}
```

With this function you can add a string to the table from anywhere in your code and immediately get the index into the table at which it was added. This will come in very handy when parsing instructions later. Notice also that the function first checks to make sure the specified string isn't already in the table. This is really just a small space optimization; there's no need to store the same string literal value in the executable more than once.

NOTE

Remember, `FreeLinkedList ()` automatically frees the `pData` pointer as it frees each node, so we don't have to write an extra function for freeing the string table.

Lastly, you may be wondering why `AddString ()` also asks for a linked list pointer. The string will always be added to `g_StringTable`, won't it? Not necessarily. As we'll see later on, the host API call table is almost identical to the string table; in fact, it pretty much *is* identical. Since we can really just think of it as another string table, there's no point in writing the same function twice just so it can have a different name. Because of this, I used `AddString ()` in both places, and thus, the caller has to specify which list to add to.

The Function Table

The next table of interest is the *function table*, which collects information on each function the script defines. This table is required to maintain information regarding scope, stack frame details, and so on. Once again we'll be leveraging our previously defined linked list structure.

What sort of information is important when keeping track of functions? Right off the bat you need to record its name, because that's how it'll be referenced in the code. You also need to keep track of everything that falls within the function's scope. This primarily means variables and line labels. And lastly, you need to describe a function's stack frame as well; the XVM will need this information at runtime to prepare the stack when function calls are made. The stack frame primarily consists of local data. In addition, however, it also contains the function's parameters, so you'll need to track those too. Lastly, we'll need to record the function's entry point. Together, these fields will provide enough information to definitively describe a function. Here's the structure:

```
typedef struct _FuncNode        // A function table node
{
    int iIndex;                 // Index
    char pstrName [ MAX_IDENT_SIZE ];    // Name
    int iEntryPoint;            // Entry point
    int iParamCount;            // Param count
    int iLocalDataSize;         // Local data size
}
    FuncNode;
```

And here's the table itself:

```
LinkedList g_FuncTable;
```

Now, the structure has provisions for tracking the *number* of parameters and variables a function has, but what about the parameters and variables themselves? These are stored separately in another table called the *symbol table*. This goes for labels as well, which are stored in a label table. These two structures will be described in a moment.

You can now represent functions, so the next step is the ability to add them, right? Right. Let's have a look at a function you can use to easily add functions to the table.

```
int AddFunc ( char * pstrName, int iEntryPoint )
{
    // If a function already exists with the specified name, exit and return
    // an invalid index
    if ( GetFuncByName ( pstrName ) )
        return -1;

    // Create a new function node
    FuncNode * pNewFunc = ( FuncNode * ) malloc ( sizeof ( FuncNode ) );

    // Initialize the new function
    strcpy ( pNewFunc->pstrName, pstrName );
    pNewFunc->iEntryPoint = iEntryPoint;

    // Add the function to the list and get its index
    int iIndex = AddNode ( & g_FuncTable, pNewFunc );

    // Set the function node's index
    pNewFunc->iIndex = iIndex;

    // Return the new function's index
    return iIndex;
}
```

The function begins by determining whether or not the specified function already exists in the table, using `GetFuncByName ()`. As you can probably guess, this function returns a pointer to the matching node, which is how we can determine if the function has already been added. Of course, I haven't covered this function yet, so just take it on faith for now. We'll get to it in a moment. If the function already exists, `-1` is returned as an error code to the caller. Otherwise, we create a new function node, initialize it, and add it to the table. The index returned by `AddNode ()` is saved in the function's `iIndex` field, which lets each node in the table keep a local copy of its position in the table. This index is also returned to the caller.

Note that the newly added function has only set a few of its fields. The function never initialized its parameter count, local data size, or stack frame size. The reason for this, which you'll discover later as you write the parser, is that as you scan through the file, you need to first save the function's name and retrieve a unique function table index. From that point forward, you gradually collect the function's data and eventually complete the structure by sending the remaining info. Of course, in order to send that info anywhere, you need a function index, which you'll have because the function has already been created.

The function you'll use to add this remaining data looks like this:

```
void SetFuncInfo ( char * pstrName, int iParamCount, int iLocalDataSize )
{
    // Based on the function's name, find its node in the list
    FuncNode * pFunc = GetFuncByName ( pstrName );

    // Set the remaining fields
    pFunc->iParamCount = iParamCount;
    pFunc->iLocalDataSize = iLocalDataSize;
}
```

Again the function begins with a call to `GetFuncByName ()`, but beyond that it's just a matter of setting some fields.

Unlike the string table, the function table is not just written to. For the most part, you can pack your strings into the table and forget about them; the only time they'll be read is when they're ultimately dumped out to the executable file. It's important to interact with functions in the function table on a regular basis, however; as you parse the file in the second pass, you'll need to refer to the function table frequently to verify scope and other such matters. Because of this, you also need the ability to quickly and easily get a function's node based on its name. For this you'll create a function called `GetFuncByName ()`:

```
FuncNode * GetFuncByName ( char * pstrName )
{
    // If the table is empty, return a NULL pointer
    if ( ! g_FuncTable.iNodeCount )
        return NULL;

    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = g_FuncTable.pHead;

    // Traverse the list until the matching structure is found
    for ( int iCurrNode = 0; iCurrNode < g_FuncTable.iNodeCount; ++ iCurrNode )
    {
        // Create a pointer to the current function structure
        FuncNode * pCurrFunc = ( FuncNode * ) pCurrNode->pData;

        // If the names match, return the current pointer
        if ( strcmp ( pCurrFunc->pstrName, pstrName ) == 0 )
            return pCurrFunc;
        // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The structure was not found, so return a NULL pointer
    return NULL;
}
```

With this function, you can immediately retrieve any function's node at any time, based solely on its name. For example, when parsing a `Call` instruction, you simply need to grab the function name string from the source code, pass it to this function, and use the `Index` member of the structure it returns to fill in the assembled `Call`'s operand data.

The Symbol Table

The *symbol table* was mentioned in the last section, and is where you're going to store the script's variables and arrays. Like functions, variable and array information is initially collected in the first pass and then used heavily during the assembly process of the second pass. It's yet another application of our linked list; here's the declaration:

```
LinkedList g_SymbolTable;
```

To adequately represent a variable within the symbol table, you need the variable's identifier, its size (which is always 1 for elements, but can vary for arrays), and of course, its stack index. In addition, however, you'll naturally need some way to record the variable's scope as well. You'll do this by storing the index into the function table of the function in which the variable is declared. Then, whenever you need to retrieve a variable based on its identifier, you'll also pass the function index so that it'll know exactly which identifier to match it with (otherwise, you wouldn't be able to reuse the same identifiers in different functions). Here's the structure:

```
typedef struct _SymbolNode      // A symbol table node
{
    int iIndex;                 // Index
    char pstrIdent [ MAX_IDENT_SIZE ];    // Identifier
    int iSize;                  // Size (1 for variables, N for arrays)
    int iStackIndex;            // The stack index to which the symbol points
    int iFuncIndex;             // Function in which the symbol resides
}
    SymbolNode;
```

Like always, let's create a function that can add a variable or array to the symbol table easily:

```
int AddSymbol ( char * pstrIdent, int iSize, int iStackIndex, int iFuncIndex )
{
    // If a label already exists
    if ( GetSymbolByIdent ( pstrIdent, iFuncIndex ) )
        return -1;

    // Create a new symbol node
    SymbolNode * pNewSymbol = ( SymbolNode * )
        malloc ( sizeof ( SymbolNode ) );

    // Initialize the new label
    strcpy ( pNewSymbol->pstrIdent, pstrIdent );
    pNewSymbol->iSize = iSize;
    pNewSymbol->iStackIndex = iStackIndex;
    pNewSymbol->iFuncIndex = iFuncIndex;

    // Add the symbol to the list and get its index
    int iIndex = AddNode ( & g_SymbolTable, pNewSymbol );

    // Set the symbol node's index
    pNewSymbol->iIndex = iIndex;

    // Return the new symbol's index
    return iIndex;
}
```

With the new symbol added, you'll need the ability to retrieve it based both on its identifier and its function index. This function will be called `GetSymbolByIdent ()`:

```
SymbolNode * GetSymbolByIdent ( string Ident, int FuncIndex )
{
    // Traverse the linked list until a symbol with the proper
    // identifier and scope is found.
    // First latch onto the initial node
    SymbolNode * CurrSymbol = SymbolTable.Head;
    // Loop through each node in the list
    for ( CurrIndex = 0; CurrIndex < SymbolTable.SymbolCount; ++ CurrIndex )
    {
        // Check to see if the current node matches the specified identifier
        if ( CurrNode.Ident == Ident )
            // Now see if their scopes are the same or overlap (global/local)
            if ( CurrNode.FuncIndex == FuncIndex || CurrNode.StackIndex >= 0 )
                return CurrNode;
        // Otherwise move on to the next in the list
        CurrNode = CurrNode.Next;
    }
    // The specified symbol was not found, so return NULL
    return NULL;
}
```

Just pass it the symbol's identifier and function index, and this function will return the full node, allowing you access to anything you need. Variables declared in functions are also prohibited from sharing identifiers with globals. This is what the line in the previous code is all about:

```
if ( CurrNode.FuncIndex == FuncIndex || CurrNode.StackIndex >= 0 )
```

If the two identifiers don't share the same function, they might still conflict if the node already in the table is global. To determine whether this is the case, you simply compare the stack index to zero. If it's greater, it means you aren't using negative stack indices, which is an invariable characteristic of globals. Clever, huh? Remember, stack indices that are relative to the bottom are positive, which is where globals are stored. Variables, because they're always relative to the top of the stack inside their respective stack frames, are referenced with negative indices.

Before moving on, there are two other helper functions that will come in handy when we get to the parser. In addition to retrieving the pointer to a whole symbol node structure, there will also be times when it's nice to be able to extract specific fields based on a variable's identifier. Here's a function that allows you to get a symbol's stack index:

```
int GetStackIndexByIdent ( char * pstrIdent, int iFuncIndex )
{
    // Get the symbol's information
    SymbolNode * pSymbol = GetSymbolByIdent ( pstrIdent, iFuncIndex );

    // Return its stack index
    return pSymbol->iStackIndex;
}
```

It's naturally simple since it's just based on the existing `GetSymbolByIdent ()` function we already covered. The other function returns a symbol's size:

```
int GetSizeByIdent ( char * pstrIdent, int iFuncIndex )
{
    // Get the symbol's information
    SymbolNode * pSymbol = GetSymbolByIdent ( pstrIdent, iFuncIndex );
    // Return its size
    return pSymbol->iSize;
}
```

NOTE

Technically, the term *symbol table* is usually applied to a much broader range of information and stores information for *all* of the program's symbols (the term *symbol* just being a synonym for *identifier*). This means that symbol tables usually store information regarding functions, line labels, etc. However, I think it's easier and cleaner to work with multiple, specialized tables rather than one big collection of everything. I just retain the term "symbol table" for posterity's sake.

The Label Table

Completing the set of function- and scope-related tables is the *label table*. This table maintains a list of all of the script's line labels, which is useful because all references to these labels must eventually be replaced with indices corresponding to the label's target instruction. Of course, it's another linked list, so it has a rather predictable declaration:

```
LinkedList g_LabelTable;
```

Unlike functions and symbols, line labels don't need to be stored with much. All a label really needs is its name (the label itself), the index of its target instruction, and the index of the function in which it's declared. This should translate into a pretty self-explanatory set of structures, especially after seeing so many already, so I'll just list them both:

```
typedef struct _LabelNode       // A label table node
{
    int iIndex;                 // Index
    char pstrIdent [ MAX_IDENT_SIZE ];    // Identifier
    int iTargetIndex;           // Index of the target instruction
    int iFuncIndex;             // Function in which the label resides
}
```

And, as you'd expect, you need functions both for adding labels and retrieving them based on their identifier and scope. Here they are (there's nothing new, so the comments should be explanation enough):

```
int AddLabel ( char * pstrIdent, int iTargetIndex, int iFuncIndex )
{
    // If a label already exists, return -1
    if ( GetLabelByIdent ( pstrIdent, iFuncIndex ) )
        return -1;

    // Create a new label node
    LabelNode * pNewLabel = ( LabelNode * ) malloc ( sizeof ( LabelNode ) );

    // Initialize the new label
    strcpy ( pNewLabel->pstrIdent, pstrIdent );
    pNewLabel->iTargetIndex = iTargetIndex;
    pNewLabel->iFuncIndex = iFuncIndex;

    // Add the label to the list and get its index
    int iIndex = AddNode ( & g_LabelTable, pNewLabel );

    // Set the index of the label node
    pNewLabel->iIndex = iIndex;

    // Return the new label's index
    return iIndex;
}
```

Once we've got the label in the table, we can read it back out with `GetLabelByIdent ()`:

```
LabelNode * GetLabelByIdent ( char * pstrIdent, int iFuncIndex )
{
    // If the table is empty, return a NULL pointer
    if ( ! g_LabelTable.iNodeCount )
        return NULL;

    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = g_LabelTable.pHead;

    // Traverse the list until the matching structure is found
    for ( int iCurrNode = 0; iCurrNode < g_LabelTable.iNodeCount;
          ++ iCurrNode )
    {
        // Create a pointer to the current label structure
        LabelNode * pCurrLabel = ( LabelNode * ) pCurrNode->pData;
        // If the names and scopes match, return the current pointer
        if ( strcmp ( pCurrLabel->pstrIdent, pstrIdent ) == 0 &&
             pCurrLabel->iFuncIndex == iFuncIndex )
            return pCurrLabel;

        // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The structure was not found, so return a NULL pointer
    return NULL;
}
```

As you'd imagine, it traverses the list until a suitable match is found, at which point it returns the index. Otherwise it returns `NULL`.

The Host API Call Table

The *host API call table* stores the actual function name strings that are found as operands to the `CallHost` instruction. These are saved in the executable and loaded by the VM to perform late binding in which the strings supplied by the script are matched up to the names of functions provided by the host. This is our last linked list example, so here's the declaration:

```
LinkedList g_HostAPICallTable;
```

The actual implementation of the host API call table is almost identical to that of the string table, because it really just *is* a string table underneath. The only real technical difference is its name, and the fact that it's written to a different part of the executable. This is why `AddString ()` was designed to support different lists; just pass it a pointer to `g_HostAPICallTable` instead of `g_StringTable`, and you're good to go. Check out Figure 9.29 for a visual.

The Instruction Lookup Table

The last major structure to discuss here is the instruction lookup table, which contains a description of the entire XVM instruction set. This table is used to ensure that each instruction read from the input file is a valid instruction and is being used properly.

Defining Instructions

Since the instruction set won't change often, and certainly won't change during the assembly process itself, there's no need to wheel out yet another linked list. Instead, it's just a statically

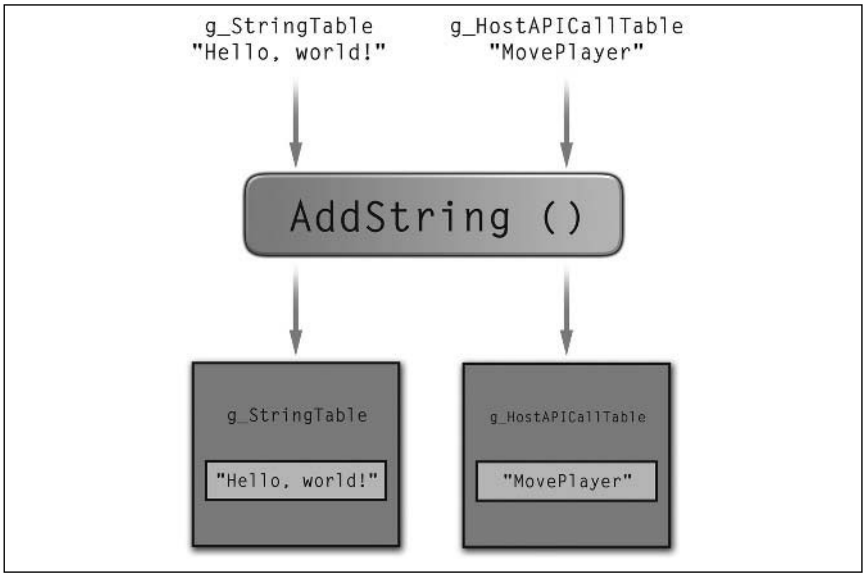


Figure 9.29

AddString () can add a string node to any linked list when provided with the proper pointer.

allocated array of InstrLookup structures. The InstrLookup structure encapsulates a single instruction, and looks like this:

```
typedef struct _InstrLookup      // An instruction lookup
{
    char pstrMnemonic [ MAX_INSTR_MNEMONIC_SIZE ];  // Mnemonic string
    int iOpcode;                 // Opcode
    int iOpCount;                // Number of operands
    OpTypes * OpList;            // Pointer to operand list
}
    InstrLookup;
```

As you can see, the structure maintains the instruction's mnemonic, its opcode, the number of operands it accepts, and a pointer to the operand list. As I mentioned earlier in the chapter, each operand type that a given operand can accept is represented in a bitfield. OpTypes is just an alias type that wraps int, since int gives us a simple 4-byte bitfield to work with:

```
typedef int OpTypes;
```

These structures, as mentioned above, are stored in a statically allocated global array. Here's the declaration:

```
#define MAX_INSTR_LOOKUP_COUNT      256      // The maximum number of
                                             // instructions the lookup table
                                             // will hold
```

```
#define MAX_INSTR_MNEMONIC_SIZE     16       // Maximum size of an instruction
                                             // mnemonic's string
InstrLookup g_InstrTable [ MAX_INSTR_LOOKUP_COUNT ];
```

Adding Instructions

Two functions will be necessary to populate the table– one to add new instructions, and one to define the individual operands. Let's look at the function for adding instructions first, which is of course called `AddInstrLookup ()`:

```
int AddInstrLookup ( char * pstrMnemonic, int iOpcode, int iOpCount )
{
    // Just use a simple static int to keep track of the next instruction
    // index in the table.
    static int iInstrIndex = 0;

    // Make sure we haven't run out of instruction indices
    if ( iInstrIndex >= MAX_INSTR_LOOKUP_COUNT )
        return -1;

    // Set the mnemonic, opcode and operand count fields
    strcpy ( g_InstrTable [ iInstrIndex ].pstrMnemonic, pstrMnemonic );
    strupr ( g_InstrTable [ iInstrIndex ].pstrMnemonic );
    g_InstrTable [ iInstrIndex ].iOpcode = iOpcode;
    g_InstrTable [ iInstrIndex ].iOpCount = iOpCount;

    // Allocate space for the operand list
    g_InstrTable [ iInstrIndex ].OpList = ( OpTypes * )
        malloc ( iOpCount * sizeof ( OpTypes ) );

    // Copy the instruction index into another variable so it can be returned
    // to the caller
    int iReturnInstrIndex = iInstrIndex;

    // Increment the index for the next instruction
    ++ iInstrIndex;

    // Return the used index to the caller
    return iReturnInstrIndex;
}
```

Given a mnemonic, opcode, and operand count, `AddInstrLookup ()` will create the specified instruction at the next free index within the table (maintained via the `static int`) and return the index to the caller. It also allocates a dynamic array of `OpTypes`, giving the instruction room to define each of its operands. That process is facilitated with a function called `SetOpType ()`:

```
void SetOpType ( int iInstrIndex, int iOpIndex, OpTypes iOpType )
{
    g_InstrTable [ iInstrIndex ].OpList [ iOpIndex ] = iOpType;
}
```

Pretty simple, huh? Given an instruction index, the `iOpType` bitfield will be assigned to the specified operand. The bitfield itself is constructed on the caller's end, by combining a number of operand type masks with a bitwise `or`. Each of these masks represents a specific operand data type and is assigned a power of two that allows it to flip its respective bit in the field. Table 9.14 lists them.

You'll notice that these operand types don't line up exactly with a lot of the other operand type tables you've seen. This is because you can be a lot more general when describing what type of operand a given instruction *can* accept than you can when describing what type of operand that

Table 9.14 Operand Type Bitfield Masks

Constant	Value	Description
`OP_FLAG_TYPE_INT`	1	Integer literal value
`OP_FLAG_TYPE_FLOAT`	2	Floating-point literal value
`OP_FLAG_TYPE_STRING`	4	String literal value
`OP_FLAG_TYPE_MEM_REF`	8	Memory reference (variable or array index)
`OP_FLAG_TYPE_LINE_LABEL`	16	Line label (used in jump instructions)
`OP_FLAG_TYPE_FUNC_NAME`	32	Function name (used in the `Call` instruction)
`OP_FLAG_TYPE_HOST_API_CALL`	64	Host API call (used in the `CallHost` instruction)
`OP_FLAG_TYPE_REG`	128	A register, which is always the `_RetVal` register in our case

instruction *did* accept. For example, the Mov instruction's destination operand can be a variable or array index. The parser doesn't care which it is; it only wants to make sure it's one of them.

So we've got the two functions we need, as well as our bitfield flags. Let's look at an example of how a few instructions in the set are defined. Here's Mov:

```
iInstrIndex = AddInstrLookup ( "Mov", 0, 2 );
SetOpType ( iInstrIndex, 0, OP_FLAG_TYPE_MEM_REF |
                            OP_FLAG_TYPE_REG );
SetOpType ( iInstrIndex, 1, OP_FLAG_TYPE_INT |
                            OP_FLAG_TYPE_FLOAT |
                            OP_FLAG_TYPE_STRING |
                            OP_FLAG_TYPE_MEM_REF |
                            OP_FLAG_TYPE_REG );
```

Here, the instruction is added first with a call to AddInstrLookup. Along with the mnemonic, we pass an opcode of zero and an operand count of two. The two operands are then defined with two calls to SetOpType (). Notice how whatever data types the operand may need are simply combined with a bitwise or; it makes for very easy operand description. Here's the definition of JGE:

```
iInstrIndex = AddInstrLookup ( "JGE", 24, 3 );
SetOpType ( iInstrIndex, 0, OP_FLAG_TYPE_INT |
                            OP_FLAG_TYPE_FLOAT |
                            OP_FLAG_TYPE_STRING |
                            OP_FLAG_TYPE_MEM_REF |
                            OP_FLAG_TYPE_REG );
SetOpType ( iInstrIndex, 1, OP_FLAG_TYPE_INT |
                            OP_FLAG_TYPE_FLOAT |
                            OP_FLAG_TYPE_STRING |
                            OP_FLAG_TYPE_MEM_REF |
                            OP_FLAG_TYPE_REG );
SetOpType ( iInstrIndex, 2, OP_FLAG_TYPE_LINE_LABEL );
```

This instruction represents opcode 24, and accepts three operands. The first two can be virtually anything, but notice that the last parameter must be a line label. Let's wrap things up with a look at a really simple one, Call:

```
iInstrIndex = AddInstrLookup ( "Call", 28, 1 );
SetOpType ( iInstrIndex, 0, OP_FLAG_TYPE_FUNC_NAME );
```

Call is added to the list as opcode 28 with one operand, which must be a function name.

NOTE

Check out the XASM source to see the rest of the instructions' definitions. The instruction set is initialized in a single function called InitInstrTable ().

Of course, if you *really* want to go all out, you could store your language description in an external file that is read in by the assembler when it initializes. This would literally allow a single assembler to implement multiple instruction sets, which may be advantageous if you have a number of different virtual machines that you use in various game projects.

When dealing with real hardware, it'd take a lot more than a simple description of instructions and operands to define an entire assembly language, but in the case of a virtual machine like ours, you may very well decide that you want to change the instruction set for your next game. If you continue work on the first game, or revise it with a new version or sequel, you may find yourself working with two different instruction sets at once, for two different virtual machines. Designing your assembler with swappable language definitions in mind will allow you to easily handle this situation.

For example, you may want to simply define your languages with a basic ASCII file so you can quickly make changes in a text editor. This would most easily be done in a tab-delimited *flatfile*. Flatfiles are easy to parse because each element of the file is separated by the same, single-character `\t` code. Here's an example of what it might look like:

```
Mov   0   2
MemRef
Int   Float   String   MemRef
Jmp   19      1
Label
```

In this particular example, the first line defined the `Mov` instruction. Following the mnemonic string, was a `0` and a `2`, signifying the opcode (zero) and the instruction's two operands. The parser would then know that the following two lines are the operand definitions. Each of these lines consist of tab-delimited strings. The strings are identified by the parser as different operand types, like `MemRef` and `String` in this case. Following the two operand lines is another instruction definition, this time for `Jmp`, as well as its single operand definition. The parser would continue reading these instruction definitions until the end of the file was reached, at which point it would consider the language complete. The end result is a simple and flexible solution to multiple game projects that allows you to leverage your existing assembler without even having to recompile. In fact, to make it easier, a new directive could be added to the assembler's overall vocabulary that specified which instruction set to use; this way scripts can define their own "dialect" without the user needing to manually handle the language swapping (which would otherwise have to be done with a command-line parameter, configuration file, or other such interface mechanism). Check out Figure 9.30 for a graphical take on this concept.

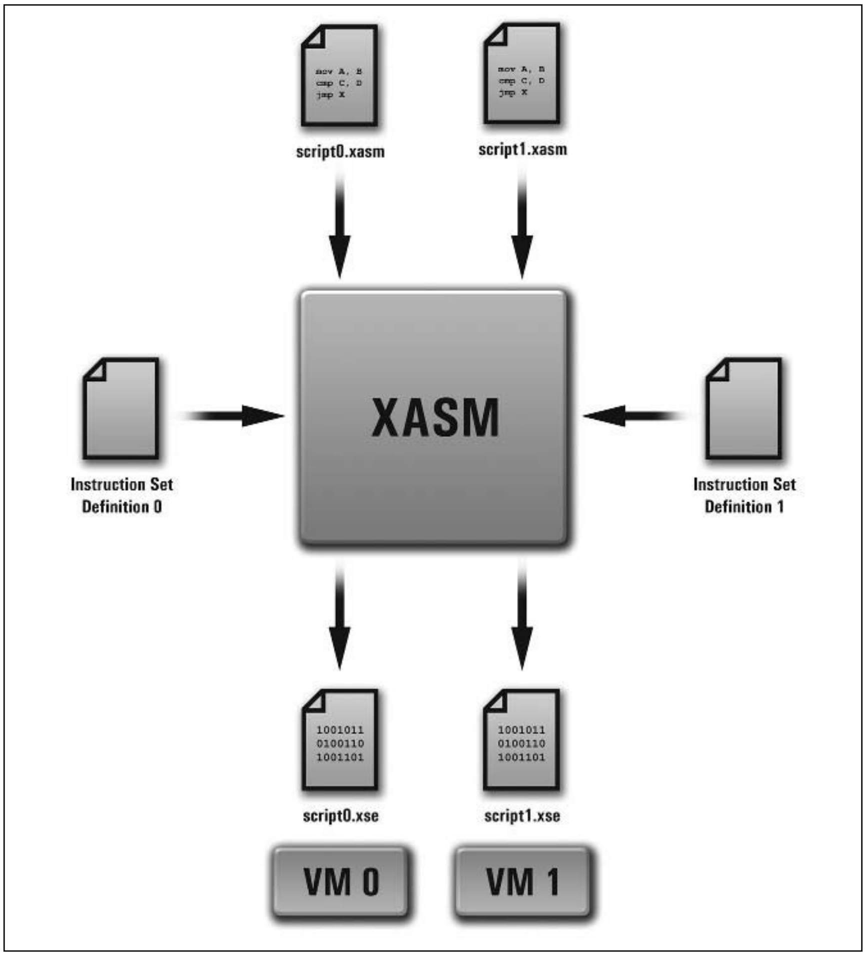


Figure 9.30

Building the assembler to support "swappable" instruction sets.

Accessing Instruction Definitions

Once the table is populated, the parser (and even the lexer) will need to be able to easily retrieve the instruction lookup structure based on a supplied mnemonic. This will be enabled with a function called `GetInstrByMnemonic ()`. Here's the code:

```
int GetInstrByMnemonic ( char * pstrMnemonic, InstrLookup * pInstr )
{
    // Loop through each instruction in the lookup table
    for ( int iCurrInstrIndex = 0;
          iCurrInstrIndex < MAX_INSTR_LOOKUP_COUNT; ++ iCurrInstrIndex )
    {
        // Compare the instruction's mnemonic to the specified one
        if ( strcmp ( g_InstrTable [ iCurrInstrIndex ].pstrMnemonic,
             pstrMnemonic ) == 0 )
        {
            // Set the instruction definition to the user-specified pointer
            * pInstr = g_InstrTable [ iCurrInstrIndex ];
            // Return TRUE to signify success
            return TRUE;
        }
    }

    // A match was not found, so return FALSE
    return FALSE;
}
```

Structural Overview Summary

So you've got a number of global structures, which, altogether, form the assembler's internal representation of the script as the assembly process progresses. Here's a summary in the form of these structures' global declarations:

```
// Source code representation
char ** g_ppstrSourceCode = NULL;
int g_iSourceCodeSize;

// The instruction lookup table
InstrLookup g_InstrTable [ MAX_INSTR_LOOKUP_COUNT ];

// The assembled instruction stream
Instr * g_pInstrStream = NULL;
int g_iInstrStreamSize;

// The script header
ScriptHeader g_ScriptHeader;


// The main tables
LinkedList g_StringTable;
LinkedList g_FuncTable;
LinkedList g_SymbolTable;
LinkedList g_LabelTable;
LinkedList g_HostAPICallTable;
```

Each (or most) of these global structures also has a small interface of functions used to manipulate the data it contains. Let's run through them one more time to make sure you're clear with everything.

Starting with the string table:

```
int AddString ( LinkedList * pList, char * pstrString );
```

Next up is the function table:

```
int AddFunc ( char * pstrName, int iEntryPoint );
FuncNode * GetFuncByName ( char * pstrName );
void SetFuncInfo ( char * pstrName, int iParamCount, int iLocalDataSize );
```

Followed by the symbol and label tables:

```
int AddSymbol ( char * pstrIdent, int iSize, int iStackIndex, int iFuncIndex );
SymbolNode * GetSymbolByIdent ( char * pstrIdent, int iFuncIndex );
int GetStackIndexByIdent ( char * pstrIdent, int iFuncIndex );
int GetSizeByIdent ( char * pstrIdent, int iFuncIndex );

int AddLabel ( char * pstrIdent, int iTargetIndex, int iFuncIndex );
LabelNode * GetLabelByIdent ( char * pstrIdent, int iFuncIndex );
```

Lastly, there's the instruction lookup table:

```
int AddInstrLookup ( char * pstrMnemonic, int iOpcode, int iOpCount );
void SetOpType ( int iInstrIndex, int iOpIndex, OpTypes iOpType );
int GetInstrByMnemonic ( char * pstrMnemonic, InstrLookup * pInstr );
```

Lastly, check out Figure 9.31 for a graphical overview of XASM's major structures.

Lexical Analysis/Tokenization

From here on out, I will refer to the lexical analysis phase as the combination of both the lexer and the tokenizer. Therefore, according to the new definition, the lexer's input is the character stream, and its output is the token stream. The lexeme stream will really only exist abstractly.

Therefore, the task in this section is to write a software layer that sits between the raw source code and the parser, intercepting the incoming character stream and outputting a token stream that the parser can immediately attempt to identify and translate. This will be our lexer.

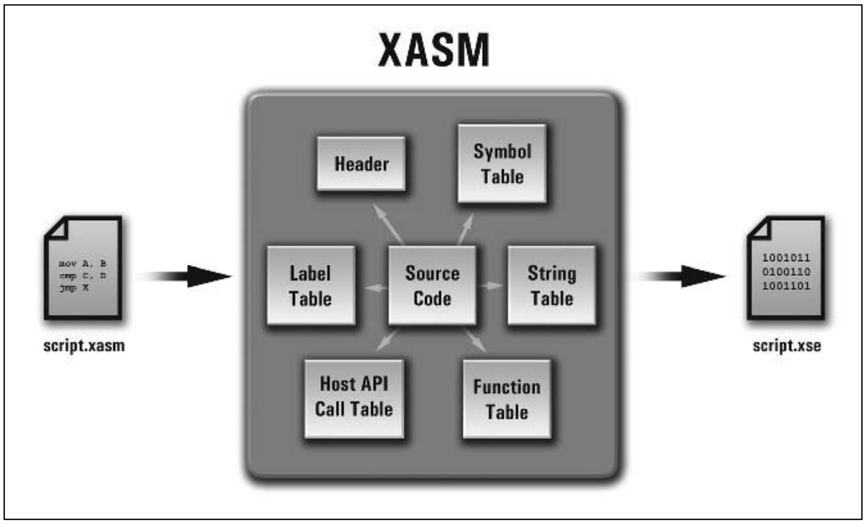


Figure 9.31

A structural overview of XASM.

The Lexer's Interface and Implementation

The implementation of the lexical analyzer is embodied by a small group of functions and structures. The primary interface will come down to a few main functions: `GetNextToken ()`, `GetCurrLexeme ()`, `GetLookAheadChar ()`,`SkipToNextLine ()`, and `ResetLexer ()`.

GetNextToken ()

`GetNextToken ()` returns the current token and advances the token stream by one. Its prototype looks like this:

```
int GetNextToken ();
```

As you can see, it doesn't require any parameters but returns an `int`. This integer value is the token, which can be any of the number of token types I'll define later in this section. Aside from returning the token, however, `GetNextToken ()` does quite a bit of behind-the-stage processing. Namely, the token stream will advance by one, which means that repetitive calls to `GetTokenStream ()` will continually produce new results automatically and eventually cycle through every token in the source file. In other words, the parser and other areas of the assembler won't have to manage their own token stream pointers; it's all handled internally.

In addition to returning the current token and advancing the stream, `GetNextToken ()` also fills the `g_Lexer` structure to reflect all of the current token's information, which I'll get to momentarily.

GetCurrLexeme ()

GetCurrLexeme () returns a character pointer to the string containing the current lexeme. For example, if GetNextToken () returns TOKEN_TYPE_IDENT, GetCurrLexeme () will return the actual identifier itself. Its prototype looks like this:

```
char * GetCurrLexeme ();
```

The string pointed to by GetNextLexeme () belongs to the g_Tokenizer structure, however, which means you shouldn't alter it unless you make a local copy of it. Once you've used GetNextToken () to bring the next token in the stream into focus and determine its type, you can follow up with a call to GetCurrLexeme () to take further action based on the content of the lexeme itself.

GetLookAheadChar ()

Thus far I haven't discussed look-aheads, so I'll introduce them here. You'll learn about this concept in much fuller detail later, but for now, all you really need to know is that a *look-ahead* is the process of the parser looking past the current token to characters that lie beyond it. However, although it does read the character, it doesn't advance the stream in any way, so the next call to GetNextToken () will still behave just as it would have before the look-ahead.

Look-aheads are often necessary because some aspect of the language is not *deterministic*. To explain what this means in a simple and appropriate context, consider the following example. Imagine the parser encountering the following variable declaration:

```
Var MyVar
```

The tokenizer will reduce this to the following tokens: TOKEN_TYPE_VAR and TOKEN_TYPE_IDENT. When the identifier token is parsed, the parser will be at a "crossroads", so to speak. On the one hand, this may be a complete variable declaration, and if so, you can move on to the next line. On the other hand, you may only be partially through with an *array* declaration, which involves extra tokens (the brackets and the array size). Remember, the parser can't look at the line of code as a whole like humans can. When it reaches the identifier token, it can literally only see up to that point. That means that if, in reality, the previous line was actually this:

```
Var MyVar [ 256 ]
```

The parser would have no idea whatsoever. So, you use a look-ahead in these cases, where the currently read set of parsed tokens isn't enough for you to *determine* exactly what the remaining tokens (if any) should be (hence the term "deterministic"). Rather than read the next token, however, you simply want to "peek" and find out what lies ahead without the stream being advanced, because advancing the stream would throw every subsequent call to GetNextToken () out of sync. By reading even the first character of the next token, you can determine what you're dealing with. In this particular case, that single character would actually be the entire token—the open bracket. This character alone would be enough to let you know that the variable declaration is in fact an array declaration and that the line isn't finished. Of course, if an open bracket isn't found, it means that the current line is indeed finished, and you can move on to the next token without fear of the stream being out of sync.

As you'll see throughout the development of the parser, you'll only need a one-character look-ahead. In other words, at worst you'll only need to see the first character of the next token in order to resolve an ambiguity. In most cases, however, your language is deterministic enough to parse without help from the look-ahead at all.

NOTE

Look-aheads don't always have to be a single character. Certain languages, depending on their complexity and general layout, may need multiple-character look-aheads to fully resolve a non-deterministic situation. Certain languages can even become so ambiguous that entire tokens must be looked ahead to.

The combination of these three functions should be enough for the parser to do its job, so let's look at how they're actually implemented.

`SkipToNextLine ()`

You might run into situations in which you simply want to ignore an entire line of tokens. Because the source code is internally stored as a series of separate lines, all this function really has to do is increment the current line counter and reset the tokenizer position within it. `SkipToNextLine ()` has an understandably simple prototype:

```
void SkipToNextLine ();
```

`ResetLexer ()`

`ResetLexer ()` is the last function involved in the lexer's interface, and performs the simple task of resetting everything. This function will only be used twice, as the lexer will need to be reset before each of the two passes over the source is performed.

The Lexer Implementation

The lexer, despite its vital role in the assembly process, is not a particularly complex piece of software. Its work is done in two phases—lexing, wherein the next lexeme is extracted from the character stream, and tokenization, which identifies the lexeme as belonging to one of a number of token type classes.

Token Types

To get things started, Table 9.15 lists the different types of tokens the lexer will output. Remember, a token is determined by examination of its corresponding lexeme.

Table 9.15 Token Type Constants

Constant	Description
`TOKEN_TYPE_INT`	An integer literal
`TOKEN_TYPE_FLOAT`	A floating-point literal
`TOKEN_TYPE_STRING`	A string literal value, *not* including the surrounding quotes. Quotes are considered separate tokens.
`TOKEN_TYPE_QUOTE`	A double quote "
`TOKEN_TYPE_IDENT`	An identifier
`TOKEN_TYPE_COLON`	A colon :
`TOKEN_TYPE_OPEN_BRACKET`	An opening bracket `[`
`TOKEN_TYPE_CLOSE_BRACKET`	A closing bracket `]`
`TOKEN_TYPE_COMMA`	A comma ,
`TOKEN_TYPE_OPEN_BRACE`	An opening curly brace `{`
`TOKEN_TYPE_CLOSE_BRACE`	A closing curly brace `}`
`TOKEN_TYPE_NEWLINE`	A line break
`TOKEN_TYPE_INSTR`	An instruction
`TOKEN_TYPE_SETSTACKSIZE`	The `SetStackSize` directive
`TOKEN_TYPE_VAR`	A `Var` directive
`TOKEN_TYPE_FUNC`	A `Func` directive
`TOKEN_TYPE_PARAM`	A `Param` directive
`TOKEN_TYPE_REG_RETVAL`	The `_RetVal` register
`TOKEN_TYPE_INVALID`	Error code for invalid tokens
`END_OF_TOKEN_STREAM`	The end of the stream has been reached

Note the `END_OF_TOKEN_STREAM` constant, which actually isn't a token in itself but rather a sign that the token stream has ended.

Even though the token type is just a simple integer value, it's often convenient to wrap primitive data types in more descriptive names using `typedef` (plus it looks cool!). In the case of your tokenizer, you can create a `Token` type based on `int`:

```
typedef int Token;
```

Now, for example, the prototype for `GetNextToken ()` can look like this:

```
Token GetNextToken ();
```

This also lets you change the underlying implementation of the tokenizer without breaking code that would otherwise be dependant on the `int` type. You never know when something like that might come in handy. I'll make use of the `Token` type throughout the remainder of this chapter, and in the XASM source.

Initial Source Line Prepping

Before the lexer goes to work, I like to prep the source line as much as possible to make its job easier. This involves stripping any comments that may be found on the line, and then trimming whitespace on both sides. After this process, you might even find that the line was pure whitespace to begin with, or consisted solely of a comment. In these cases, the line can be skipped altogether and you can move on to the next.

Comments are stripped first, which is a simple process, although there is one gotcha to be aware of. XVM Assembly defines comments as anything behind the semicolon character, including the semicolon itself. Imagine the following line of code:

```
Mov     X, Y                ; Move Y into X
```

The comments can be stripped from this line very easily by scanning through the string until the semicolon is found. If you place a null-terminator at the index of the semicolon, the semicolon and everything behind it will no longer be a part of the string, and we'll have the following:

```
Mov     X, Y
```

Sounds pretty easy, right? The one caveat to this approach, however, is strings. Imagine the following line:

```
Mov     X, "This curse; it is your birthright."    ; Creepy line of dialogue
```

The currently unintelligent scanner would, in its well-meaning attempts to rid you of the comments, reduce the line of code to this:

```
Mov     X, "This curse
```

This is not only a different string than was intended, but it won't even assemble. You therefore need a way to make sure that the scanner knows when it's inside a string, so it can ignore any semicolons until the string ends. Fortunately, this is easily solved: as the scanner moves through the string, it also needs to keep watch for double-quote characters. When it finds one, it sets a flag stating that a string is currently being scanned. When it finds the next double-quote, the flag is turned back off (because presumably, these two quotes were delimiting a string). This process repeats throughout the entire line of code, so strings won't trip it up. Let's look at some code:

```
void StripComments ( char * pstrSourceLine )
{
    unsigned int iCurrCharIndex;
    int iInString;

    // Scan through the source line and terminate the string at
    // the first semicolon
    iInString = 0;
    for ( iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrSourceLine ) - 1;
          ++ iCurrCharIndex )
    {
        // Look out for strings; they can contain semicolons too
        if ( pstrSourceLine [ iCurrCharIndex ] == '"' )
            if ( iInString )
                iInString = 0;
            else
                iInString = 1;

        // If a non-string semicolon is found, terminate the string
        // at its position
        if ( pstrSourceLine [ iCurrCharIndex ] == ';' )
        {
            if ( ! iInString )
            {
                pstrSourceLine [ iCurrCharIndex ] = '\n';
                pstrSourceLine [ iCurrCharIndex + 1 ] = '\0';
                break;
            }
        }
    }
}
```

Running the initial line of code through this function will yield the correct output:

```
Mov    X, "This curse; it is your birthright."
```

See a visual of this process in figure 9.32.

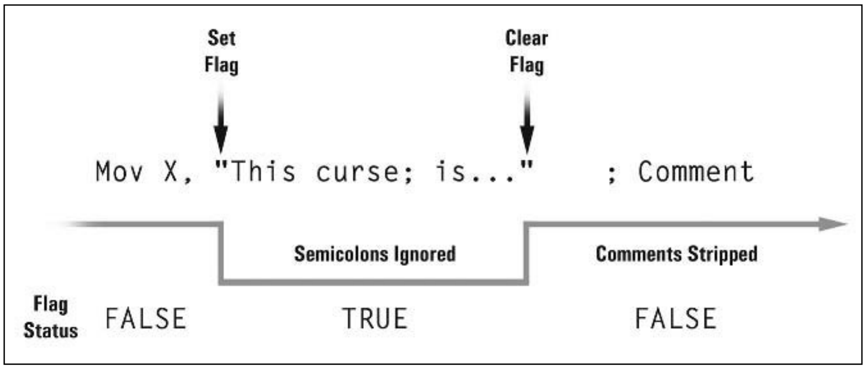


Figure 9.32

`StripComments ()` maintains a flag that is set and cleared as semicolons are read, since they presumably denote the beginnings and endings of string literals.

Trimming the whitespace from the stripped source line comes next. Trimming is usually pretty straightforward, but in C it's a bit trickier than some higher level languages due to its low-level approach to strings. Here's a function for trimming the whitespace off both ends of a string:

```
void TrimWhitespace ( char * pstrString )
{
    unsigned int iStringLength = strlen ( pstrString );
    unsigned int iPadLength;
    unsigned int iCurrCharIndex;

    if ( iStringLength > 1 )
    {
        // First determine whitespace quantity on the left
        for ( iCurrCharIndex = 0;
              iCurrCharIndex < iStringLength;
              ++ iCurrCharIndex )
            if ( ! IsCharWhitespace ( pstrString [ iCurrCharIndex ] ) )
                break;

        // Slide string to the left to overwrite whitespace
        iPadLength = iCurrCharIndex;
        if ( iPadLength )
        {
            for ( iCurrCharIndex = iPadLength;
                  iCurrCharIndex < iStringLength;
                  ++ iCurrCharIndex )
                pstrString [ iCurrCharIndex - iPadLength ] =
                pstrString [ iCurrCharIndex ];

            for ( iCurrCharIndex = iStringLength - iPadLength;
                  iCurrCharIndex < iStringLength;
                  ++ iCurrCharIndex )
                pstrString [ iCurrCharIndex ] = ' ';
        }

        // Terminate string at the start of right hand whitespace
        for ( iCurrCharIndex = iStringLength - 1;
              iCurrCharIndex > 0;
              -- iCurrCharIndex )
        {
            if ( ! IsCharWhitespace ( pstrString [ iCurrCharIndex ] ) )
            {
                pstrString [ iCurrCharIndex + 1 ] = '\0';
                break;
            }
        }
    }
}
```

This function begins by scanning through the string from left to right, counting the number of whitespace characters it finds using `IsCharWhitespace ()`. It then performs a manual string copy to physically slide each character over by the number of whitespace characters it found, effectively overwriting it. For example, if the original string looked like this:

```
"    This is a string.  "
```

It would look like this after the first step was complete:

```
"This is a string.  g.  "
```

The right-hand whitespace is easily cleared by setting the null terminator right after the last non-whitespace character in the string. Thus, the end result is:

```
"This is a string."
```

Figure 9.33 illustrates how `TrimWhitespace ()` works:

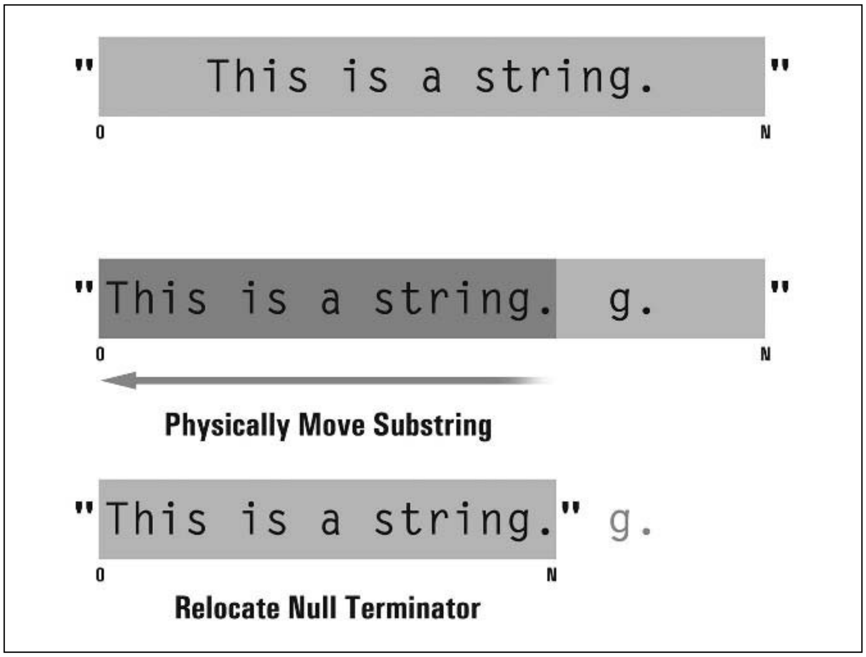


Figure 9.33

`TrimWhitespace ()` in action.

Lexing and Tokenizing

Here's where the real work begins. At this point you have a list of token type constants to produce, your line of source code has been prepped and is ready to go, so all that's left to do is isolate the next lexeme and identify its token type. This, of course, is the most complicated part.

The first thing to understand is where the lexer gets its data. Recall that the source code of the entire script is stored in a global array of strings, so if you had a small script that looked like this:

```
Func MyFunc                ; Just a meaningless function
{
    Param  X               ; Declare some parameters
    Param  Y
    Var    Product         ; Declare a local
    Mov    Product, X      ; Multiply X by Y
    Mul    Product, Y
}
```

It'd be stored in your source code array like this:

```
0: Func MyFunc                ; Just a meaningless function
1: {
2:     Param  X               ; Declare some parameters
3:     Param  Y
4:     Var    Product        ; Declare a local
5:     Mov    Product, X     ; Multiply X by Y
6:     Mul    Product, Y
7: }
```

And would look like this after each line was prepped:

```
0: Func MyFunc
1: {
2: Param  X
3: Param  Y
4: Var    Product
5: Mov    Product, X
6: Mul    Product, Y
7: }
```

The assembly process moves from line to line, which, in this case, would take you from string 0 to string 7. What's important is that at any given time, the current line (and the rest of the script, for that matter) is conveniently available in this array. The lexer, however, is specifically designed to ignore this fact that makes it appear as if everything is a continual token stream. Line breaks are ultimately reduced to `TOKEN_TYPE_NEWLINE`, and in that regard, are treated like just another token.

Because this array allows you such convenient and structured access to the script, there's no point in making another copy of the current line just for the lexer to work with. Instead, you'll just work directly with the source code array. This will make everything a lot easier because there won't be any extraneous string allocation and copying to worry about.

Let's now reiterate exactly what the lexer needs to do for you. As an example, assume the source code line in question is line 5, which looks like this:

```
Mov    Product, X
```

You can tell with your eyes that five lexemes compose this line:

```
Mov
Product
,
X
(Newline)
```

The question is, how do you get the lexer to do the same thing? Unfortunately, there aren't any hard-and-fast rules, at least not at first glance. Ideally, it'd be nice if lexemes were defined by a simple premise: for example, that all lexemes are separated by whitespace. This would make your job very simple, and perhaps even let you use the standard C library tokenizing function, `strtok ()`. Unfortunately, one of the four lexemes found previously was not separated from the lexeme before it by a space. Look at the `Product` and comma lexemes:

```
Mov    Product, X
```

There's no whitespace between them, so that throws the simple rule out the window. There are a number of ways to approach this problem, some of which are more structured and flexible than others, but I've got a rather simple solution that will fit the needs here well.

The actual rule you can apply to your lexer isn't much more complicated than the original whitespace rule. In fact, it's the same rule—just with a broader definition. All lexemes *are* separated by the same thing— *delimiter characters*. A delimiter character, as defined in the string-processing function `IsCharDelimiter ()`, are any of the characters used to separate or group common elements. In XVM Assembly, these are colons, commas, double quotes, curly braces, brackets, and yes, whitespace. So, if you scan through the source line and consider lexemes to be defined as the strings in between each delimiting character, you'll have a much more robust lexer.

There is one extra problem defined with this approach, however, because with the exception of whitespace, delimiting characters are *themselves* lexemes as well. The comma can be used to separate the `Product` lexeme from the `X` lexeme, but it's still a lexeme of its own, and one that you'll definitely need the lexer to return. So the final rule is that lexemes are separated by delimiting characters, and with the exception of whitespace, include the delimiters themselves as well. This rule will return the proper lexemes:

```
Mov
Product
,
X
(Newline)
```

Or at least, it almost will. The one other aspect of the lexer you have to be aware of is its ability to skip past arbitrary amounts of whitespace. For example, there's more than a single space between the `Mov` and `Product` lexemes. Because of this, the lexer must be smart enough to know that a lexeme doesn't start until the first non-whitespace character is found. It will therefore scan through all whitespace and ignore it until the lexeme begins. It then scans from that point forward until the first delimiter is found. The string between these two indices contains the lexeme.

You'll therefore need to manage two pointers as you traverse the string and attempt to identify the next lexeme. Both of these pointers will begin just *after* the last character of the last lexeme. When the tokenizer is first initialized, this means they'll both point to index zero. The first pointer will then move forward until it finds the first non-whitespace character, which represents the beginning of the next lexeme. The second pointer is then repositioned to equal the first. Both pointers are now positioned on the first character of the lexeme. The second pointer then scans forward until the first delimiter character is found, and stops just before that character is read. At this point, the two pointers will exactly surround the lexeme. Check out Figure 9.34 for a visual representation of this process.

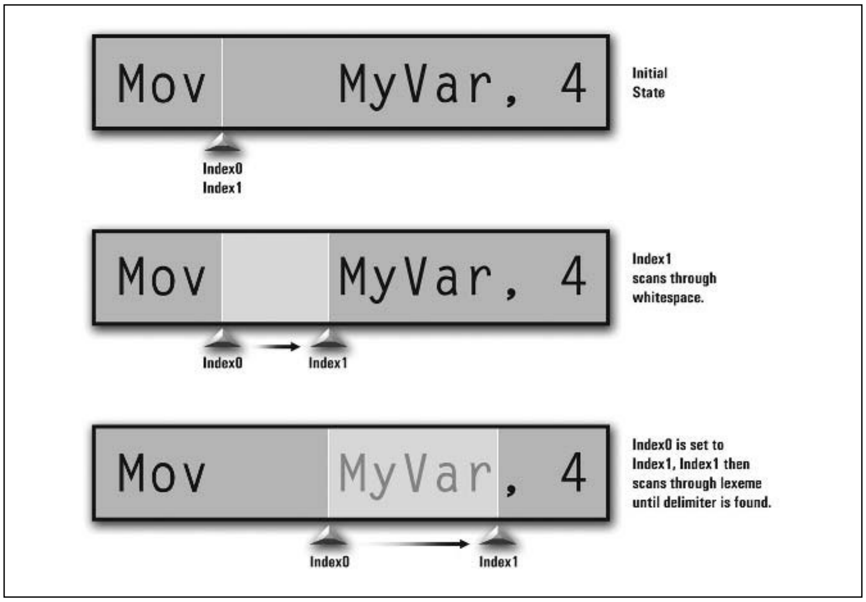

Mov MyVar, 4
Initial State
Index0
Index1
Mov MyVar, 4
Index1 scans through whitespace.
Index0
Index1
Mov MyVar, 4
Index0 is set to Index1, Index1 then scans through lexeme until delimiter is found.
Index0
Index1

Figure 9.34

Two indices traverse the source line to isolate the next lexeme amidst arbitrary whitespace and delimiters.

This substring is then copied into a global string. This global string is the current lexeme, a pointer to which is returned by `GetCurrLexeme ()`. At this point, the lexer has done its job and the tokenizer can begin. Fortunately, this is the easy part, and it's made even easier by the string processing functions covered earlier.

The first thing to check for are single-character tokens, which mostly include delimiters. You can use a `switch` block to compare this single character to each possible delimiter: the comma, the colon, the double-quote, the opening and closing brackets, newlines, and the opening and closing curly braces. If any of these matches are made, you return the corresponding `TOKEN_TYPE_*` constant.

Single-character tokens are listed in Table 9.16.

If the lexeme is longer than a single character, you know it's not a delimiter of any sort and can move on to checking for the multi-character tokens. These consist of integer and float literals, identifiers, the `_RetVal` register, and all of the XASM directives. Check out Table 9.17 for a list of them.

Table 9.16 Single-Character Tokens

Token	Description
TOKEN_TYPE_QUOTE	A quotation mark "
TOKEN_TYPE_COMMA	A comma ,
TOKEN_TYPE_COLON	A colon :
TOKEN_TYPE_OPEN_BRACKET	An opening bracket [
TOKEN_TYPE_CLOSE_BRACKET	A closing bracket]
TOKEN_TYPE_NEWLINE	A line break
TOKEN_TYPE_OPEN_BRACE	An opening curly brace {
TOKEN_TYPE_CLOSE_BRACE	A closing curly brace }

Table 9.17 Multi-Character Tokens

Token	Description
TOKEN_TYPE_INT	An integer literal
TOKEN_TYPE_FLOAT	A floating-point literal
TOKEN_TYPE_IDENT	An identifier
TOKEN_TYPE_INSTR	An instruction
TOKEN_TYPE_SETSTACKSIZE	The SetStackSize directive
TOKEN_TYPE_VAR	A Var directive
TOKEN_TYPE_FUNC	A Func directive
TOKEN_TYPE_PARAM	A Param directive
TOKEN_TYPE_REG_RETVAL	The _RetVal register

To check for integers, floats, and identifiers, you can use the functions covered earlier: `IsStringInt ()`, `IsStringFloat ()`, and `IsStringIdent ()`. Every other token is a specific string like `"VAR"` or `"_RETVAL"` and can be tested with a simple string comparison.

What I've described so far is a lexer capable of isolating and identifying all of the language's tokens, regardless of whitespace. This is quite an accomplishment! There is one little detail I've left out so far, however, and that's the issue of string literal tokens. This may not seem like much of an issue, but it's actually quite a bit trickier than anything else we've lexed so far. The problem with string literals is that they don't follow the rules laid down for every other token type. For example, consider the following:

```
Mov    StringVal, "This is a string."
```

The lexer will do fine until it runs into the first space in the string. This will be interpreted as a delimiter, and ultimately the lexer will produce the following series of lexemes and tokens:

```
MOV            TOKEN_TYPE_INSTR
STRINGVAL      TOKEN_TYPE_IDENT
,              TOKEN_TYPE_COMMA
"              TOKEN_TYPE_QUOTE
THIS           TOKEN_TYPE_IDENT
IS             TOKEN_TYPE_IDENT
A              TOKEN_TYPE_IDENT
STRING.        TOKEN_TYPE_IDENT
"              TOKEN_TYPE_QUOTE
```

This certainly isn't what you want. The value of a string literal should be returned just like the value of integers and floats are returned. What you're really looking for from the lexer is the following:

```
MOV                TOKEN_TYPE_NSTR
STRINGVAL          TOKEN_TYPE_IDENT
,                  TOKEN_TYPE_COMMA
"                  TOKEN_TYPE_QUOTE
This is a string.  TOKEN_TYPE_STRING
"                  TOKEN_TYPE_QUOTE
```

This means that the lexer must somehow know when it's extracting a string literal value, because it:

- Cannot be disrupted by the delimiting symbols that usually mark the end of a lexeme, because strings can and often do contain these same symbols.
- Should not convert the resulting lexeme to uppercase, because this would alter the string's content.

- Should replace the \" and \\ escape sequences with their respective single-character values.
- Should only stop scanning when it hits a non-escape sequence double-quote.

As you can see, strings add quite a bit of complexity to the otherwise simplistic lexer, so let's discuss the solutions to each of these problems. First of all, you need the ability to tell whether you're processing a string lexeme. This is done rather easily; whenever a double quote lexeme is detected, the flag is set, unless it's already set, in which case it's unset. This works in the same way your comment stripper function did; it simply treats double quotes as toggle switches for the string lexeme state.

As a typical lexeme is scanned, you must continually check to see if it's ended due to the presence of a delimiter character. If the lexer is to support strings, however, you must now first determine whether the string lexeme state is active; if it is, you only check for the presence of a double quote; if not, you check for any delimiter as usual.

This isn't enough, however. A single flag will only give us some of the information we need to properly maintain the state of the lexer, which will result in all tokens after the first string being interpreted as strings as well. Why? Because the toggling of the string lexeme flag when a double-quote is read isn't intelligent enough to differentiate between an opening quote and a closing quote. When a double-quote is first read, we'll go from the non-string state to the string state. We'll then read the string, and with the string state active, the lexer will know to treat the string differently, by ignoring delimiters, including whitespace, not converting the final lexeme to whitespace, etc. So far, so good, right?

The problem occurs when the string ends. A double quote will be read, which is the only character that can terminate a string lexeme. So the lexer will switch back to its non-string state, and return the string lexeme. The lexer will then be called again, at which point it will read the closing double quote (because, if you remember, delimiters are considered separate tokens). When this token is read, it will once again switch to the string lexing state, just as it did with the first quote. The lexer will continue to haphazardly alternate between strings and non-strings, greatly confusing the token stream. Check out figure 9.35 to see what I mean.

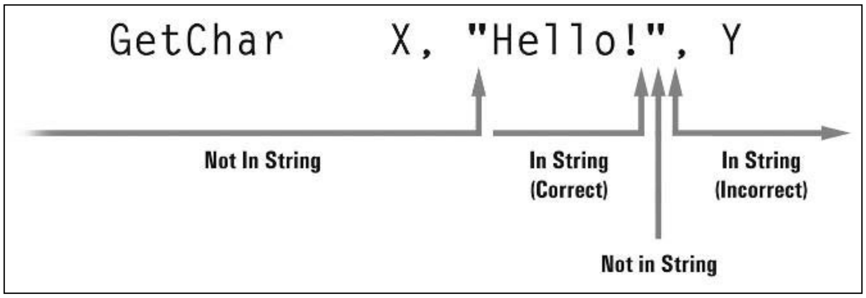


Figure 9.35

The problem with using two states to manage strings in the lexer.

The solution is to design the lexer with *three* states in mind, rather than two. The first state, LEX_STATE_NO_STRING, is active by default and is used for all non-string lexemes. When a double-quote is read, this state switches to LEX_STATE_IN_STRING, which allows it to properly handle string lexemes. When the next double quote is read, it will know that LEX_STATE_IN_STRING must transition into LEX_STATE_END_STRING. This state only exists briefly to keep the lexer from confusing opening and closing quotes. LEX_STATE_END_STRING transitions to LEX_STATE_NO_STRING, and the cycle continues.

Lastly, you may be wondering why we didn't take a simpler route by not even trying to separate double quotes from their respective strings. When a double quote character is read, the lexer could just read until the closing quote is found, and consider that whole thing one big lexeme. This would eliminate the need for lexer states and other such complexities. However, it'd make things harder on the parser, which would end up having to worry about the separation of the string from its surrounding quotes. Since I prefer to keep *all* string processing tasks within the lexer's implementation, I decided against this. As we'll see later on, it'll make the parser's job simpler. Figure 9.36 illustrates how that method would work.

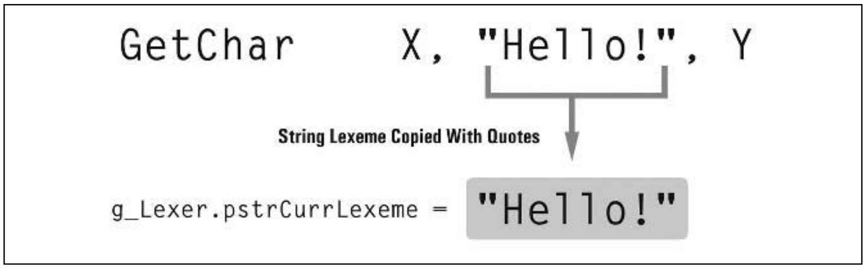

GetChar X, "Hello!", Y
String Lexeme Copied With Quotes
g_Lexer.pstrCurrLexeme = "Hello!"

Figure 9.36

A simpler way to lex strings.

The last issue is that of escape sequences. In order to support this, your scanner must also continually check for the backslash character. When one is found, you react by simply jumping ahead two characters. You do this because at this stage, you only want to ignore the sequence. You'll perform the actual processing of the sequence later.

With these changes implemented, the lexer will be capable of handling strings. Just as before, once the lexeme has been isolated, it's copied into a local lexeme string and made available to the rest of the program. To properly handle escape sequences, however, this copying process must be altered a little. As the lexeme is being copied, character-by-character, you must again keep watch for backslashes. When one is found, the backslash itself is not written to the lexeme string, but the character immediately following it instead. The process then picks up again after that character.

That's basically the story behind XASM's simple but functional lexer. Let's have a look at the final implementation:

```
Token GetNextToken ()
{
    // ---- Lexeme Extraction

    // Move the first index (Index0) past the end of the last token,
    // which is marked by the second index (Index1).

    g_Lexer.iIndex0 = g_Lexer.iIndex1;

    // Make sure we aren't past the end of the current line. If a string is
    // 8 characters long, it's indexed from 0 to 7; therefore, indices 8
    // and beyond lie outside of the string and require us to move to the
    // next line. This is why I use >= for the comparison rather than >.
    // The value returned by strlen () is always one greater than the last
    // valid character index.

    if ( g_Lexer.iIndex0 >= strlen
        ( g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ] ) )
    {
        // If so, skip to the next line but make sure we don't go past the
        // end of the file. SkipToNextLine () will return FALSE if we hit
        // the end of the file, which is the end of the token stream.

        if ( ! SkipToNextLine () )
            return END_OF_TOKEN_STREAM;
    }

    // If we just ended a string, tell the lexer to stop lexing
    // strings and return to the normal state

    if ( g_Lexer.iCurrLexState == LEX_STATE_END_STRING )
        g_Lexer.iCurrLexState = LEX_STATE_NO_STRING;

    // Scan through the potential whitespace preceding the next lexeme, but
    // ONLY if we're not currently parsing a string lexeme (since strings
    // can contain arbitrary whitespace which must be preserved).

    if ( g_Lexer.iCurrLexState != LEX_STATE_IN_STRING )
    {
        // Scan through the whitespace and check for the end of the line
        while ( TRUE )
        {
            // If the current character is not whitespace, exit the loop
            // because the lexeme is starting.

            if ( ! IsCharWhitespace ( g_ppstrSourceCode
                [ g_Lexer.iCurrSourceLine ][ g_Lexer.iIndex0 ] ) )
                break;

            // It is whitespace, however, so move to the next character and
            // continue scanning

            ++ g_Lexer.iIndex0;
        }
    }

    // Bring the second index (Index1) to the lexeme's starting character,
    // which is marked by the first index (Index0)

    g_Lexer.iIndex1 = g_Lexer.iIndex0;

    // Scan through the lexeme until a delimiter is hit, incrementing
    // Index1 each time

    while ( TRUE )
    {
        // Are we currently scanning through a string?

        if ( g_Lexer.iCurrLexState == LEX_STATE_IN_STRING )
        {
            // If we're at the end of the line, return an invalid token
            // since the string has no ending double-quote on the line

            if ( g_Lexer.iIndex1 >= strlen ( g_ppstrSourceCode
                [ g_Lexer.iCurrSourceLine ] ) )
            {
                g_Lexer.CurrToken = TOKEN_TYPE_INVALID;
                return g_Lexer.CurrToken;
            }
            // If the current character is a backslash, move ahead two
            // characters to skip the escape sequence and jump to the next
            // iteration of the loop

            if ( g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ]
                [ g_Lexer.iIndex1 ] == '\\' )
            {
                g_Lexer.iIndex1 += 2;
                continue;
            }

            // If the current character isn't a double-quote, move to the
            // next, otherwise exit the loop, because the string has ended.

            if ( g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ]
                [ g_Lexer.iIndex1 ] == '"' )
                break;

            ++ g_Lexer.iIndex1;
        }

        // We are not currently scanning through a string

        else
        {
            // If we're at the end of the line, the lexeme has ended so
            // exit the loop

            if ( g_Lexer.iIndex1 >= strlen (
                g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ] ) )
                break;

            // If the current character isn't a delimiter, move to the
            // next, otherwise exit the loop

            if ( IsCharDelimiter ( g_ppstrSourceCode
                [ g_Lexer.iCurrSourceLine ][ g_Lexer.iIndex1 ] ) )
                break;

            ++ g_Lexer.iIndex1;
        }
    }
```

```
// Single-character lexemes will appear to be zero characters at this
// point (since Index1 will equal Index0), so move Index1 over by one
// to give it some noticeable width

if ( g_Lexer.iIndex1 - g_Lexer.iIndex0 == 0 )
    ++ g_Lexer.iIndex1;

// The lexeme has been isolated and lies between Index0 and Index1
// (inclusive), so make a local copy for the lexer

unsigned int iCurrDestIndex = 0;
for ( unsigned int iCurrSourceIndex = g_Lexer.iIndex0;
    iCurrSourceIndex < g_Lexer.iIndex1; ++ iCurrSourceIndex )
{
    // If we're parsing a string, check for escape sequences and just
    // copy the character after the backslash

    if ( g_Lexer.iCurrLexState == LEX_STATE_IN_STRING )
        if ( g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ]
            [ iCurrSourceIndex ] == '\\' )
            ++ iCurrSourceIndex;

    // Copy the character from the source line to the lexeme

    g_Lexer.pstrCurrLexeme [ iCurrDestIndex ] = g_ppstrSourceCode
       [ g_Lexer.iCurrSourceLine ][ iCurrSourceIndex ];

    // Advance the destination index

    ++ iCurrDestIndex;
}

// Set the null terminator

g_Lexer.pstrCurrLexeme [ iCurrDestIndex ] = '\0';

// Convert it to uppercase if it's not a string

if ( g_Lexer.iCurrLexState != LEX_STATE_IN_STRING )
    strupr ( g_Lexer.pstrCurrLexeme );

// ---- Token Identification

// Let's find out what sort of token our new lexeme is
// We'll set the type to invalid now just in case the lexer doesn't
// match any token types

g_Lexer.CurrToken = TOKEN_TYPE_INVALID;

// The first case is the easiest-- if the string lexeme state is
// active, we know we're dealing with a string token. However, if the
// string is the double-quote sign, it means we've read an empty string
// and should return a double-quote instead

if ( strlen ( g_Lexer.pstrCurrLexeme ) > 1 ||
     g_Lexer.pstrCurrLexeme [ 0 ] != '"' )
{
    if ( g_Lexer.iCurrLexState == LEX_STATE_IN_STRING )
    {
        g_Lexer.CurrToken = TOKEN_TYPE_STRING;
        return TOKEN_TYPE_STRING;
    }
}

// Now let's check for the single-character tokens

if ( strlen ( g_Lexer.pstrCurrLexeme ) == 1 )
{
    switch ( g_Lexer.pstrCurrLexeme [ 0 ] )
    {
        // Double-Quote

        case '"':
            // If a quote is read, advance the lexing state so that
            // strings are lexed properly

            switch ( g_Lexer.iCurrLexState )
            {
                // If we're not lexing strings, tell the lexer we're
                // now in a string

                case LEX_STATE_NO_STRING:
                    g_Lexer.iCurrLexState = LEX_STATE_IN_STRING;
                    break;
                // If we're in a string, tell the lexer we just ended a
                // string

                case LEX_STATE_IN_STRING:
                    g_Lexer.iCurrLexState = LEX_STATE_END_STRING;
                    break;
            }

            g_Lexer.CurrToken = TOKEN_TYPE_QUOTE;
            break;

        // Comma

        case ',':
            g_Lexer.CurrToken = TOKEN_TYPE_COMMA;
            break;

        // Colon

        case ':':
            g_Lexer.CurrToken = TOKEN_TYPE_COLON;
            break;

        // Opening Bracket

        case '[':
            g_Lexer.CurrToken = TOKEN_TYPE_OPEN_BRACKET;
            break;

        // Closing Bracket

        case ']':
            g_Lexer.CurrToken = TOKEN_TYPE_CLOSE_BRACKET;
            break;

        // Opening Brace

        case '{':
            g_Lexer.CurrToken = TOKEN_TYPE_OPEN_BRACE;
            break;
```

```
            // Closing Brace

            case '}':
                g_Lexer.CurrToken = TOKEN_TYPE_CLOSE_BRACE;
                break;

            // Newline

            case '\n':
                g_Lexer.CurrToken = TOKEN_TYPE_NEWLINE;
                break;
        }
    }

    // Now let's check for the multi-character tokens

    // Is it an integer?

    if ( IsStringInteger ( g_Lexer.pstrCurrLexeme ) )
        g_Lexer.CurrToken = TOKEN_TYPE_INT;

    // Is it a float?

    if ( IsStringFloat ( g_Lexer.pstrCurrLexeme ) )
        g_Lexer.CurrToken = TOKEN_TYPE_FLOAT;

    // Is it an identifier (which may also be a line label or instruction)?

    if ( IsStringIdent ( g_Lexer.pstrCurrLexeme ) )
        g_Lexer.CurrToken = TOKEN_TYPE_IDENT;

    // Check for directives or _RetVal

    // Is it SetStackSize?

    if ( strcmp ( g_Lexer.pstrCurrLexeme, "SETSTACKSIZE" ) == 0 )
        g_Lexer.CurrToken = TOKEN_TYPE_SETSTACKSIZE;

    // Is it Var/Var []?

    if ( strcmp ( g_Lexer.pstrCurrLexeme, "VAR" ) == 0 )
        g_Lexer.CurrToken = TOKEN_TYPE_VAR;
    // Is it Func?

    if ( strcmp ( g_Lexer.pstrCurrLexeme, "FUNC" ) == 0 )
        g_Lexer.CurrToken = TOKEN_TYPE_FUNC;

    // Is it Param?

    if ( strcmp ( g_Lexer.pstrCurrLexeme, "PARAM" ) == 0 )
        g_Lexer.CurrToken =TOKEN_TYPE_PARAM;

    // Is it RetVal?

    if ( strcmp ( g_Lexer.pstrCurrLexeme, "_RETVAL" ) == 0 )
         g_Lexer.CurrToken = TOKEN_TYPE_REG_RETVAL;

    // Is it an instruction?

    InstrLookup Instr;
    if ( GetInstrByMnemonic ( g_Lexer.pstrCurrLexeme, & Instr ) )
        g_Lexer.CurrToken = TOKEN_TYPE_INSTR;

    return g_Lexer.CurrToken;
}
```

Our lexer is finished, and it definitely gets the job done. I should mention, however, that our take on the lexing process has been something of a "brute force" approach. It's not the most elegant or flexible method, and while it serves our purposes nicely, it's not the way we'll implement the lexer for the XtremeScript compiler. We'll of course get into the details of the textbook method later on, but since I'm sure I've already piqued your interest, I'll give you the gist here.

Lexical analysis is most commonly implemented with a *state machine*, which is a simple loop that uses each incoming character to form a progressively more accurate idea of what the string is. The term "state machine" refers to the fact that the entire lexer is composed of a single loop (remember that our lexer entered and exited a number of separate loops). At each iteration of this loop, the function is in one of a finite number of states (which is why, more specifically, it's a ***finite** state machine*) that determine how it will react to the next character. The final state when the loop ends corresponds directly to the token type.

Let's take a look at a simple example to understand this better. Imagine that this particular lexer is very simple and can only distinguish between different types of numbers. When the loop starts, it will be in an initial state, which we can call `STATE_INIT`. The loop iterates once, reading in one character. The character is analyzed, and it's identified as whitespace. The lexer now knows that it has an arbitrary amount of leading whitespace to deal with, so it switches into `STATE_WHITESPACE`, which will consume whitespace until a non-whitespace is found. Finally a non-whitespace character is found. If this is a number, the state will switch into `STATE_INT`. It turns out to be a minus sign, however, which causes it to switch into `STATE_NEG_INT` instead. The machine is now expecting to read a negative integer. If it were to read more whitespace, for example, it would return an error. It reads the next few characters, all of which are numbers, and thus in accordance with what that particular state expects. If the token were to end here, the `STATE_NEG_INT` would reflect a negative integer, which is exactly what the token would be. However, a period character is read, which means we're dealing with a float. The machine switches into `STATE_NEG_FLOAT`, and the remaining numbers are read. At any time, the current state alone is enough to handle erroneous input and ultimately reflect the token type. When the loop ends, the final state is `STATE_NEG_FLOAT`, which we can directly map to a token type. As you can see, the states changed in a way that brought us closer and closer to a conclusion. This means that the real guts of a state machine loop is a potentially large `switch` block that defines the rules by which the current state can switch to the next. These are called *state transition rules*, or *edges*.

To further drive the point home, check out Figure 9.37.

The state machine approach is definitely the most elegant way to go, so you might be wondering why I didn't just use it here. The reason is primarily that despite its benefits, a state machine isn't really the most intuitive way to parse strings–at least not at first. I personally came up with the

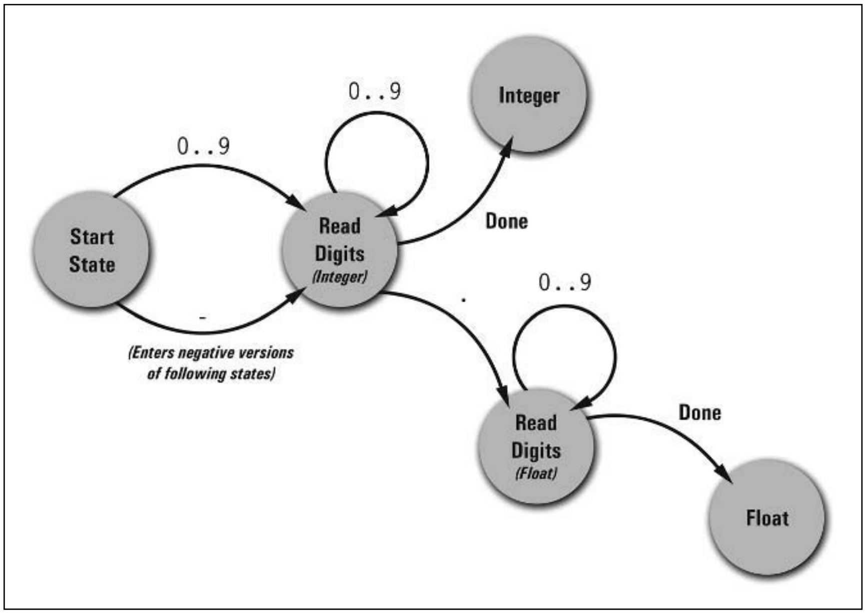


Figure 9.37

A state machine for a simple number lexer.

brute force method on my own, long before learning about state machines, and I think that's indicative of a lot of aspiring compiler/assembler writers.

> **NOTE**
>
> **Aside from general complexity, one downside to the state-machine approach is that often, to properly lex an entire language, literally *hundreds* of state transition rules must be written. To alleviate this, lexers are often actually generated by separate programs that work with an input file specifying the language's lexing rules. Of course, we're getting way ahead of ourselves-- we'll learn all about the details of this starting in Chapter 12.**

These ad-hoc methods just come more naturally, so I like the idea of covering them instead of pretending they don't exist like a lot of textbooks tend to do. In a lot of ways, the XASM assembler implementation was designed to deliberately incorporate these more primitive approaches to lexing and parsing, because they're very easy to understand and have ultimately provided you with a much stronger footing for understanding the more esoteric approaches we'll be learning about when we build the actual XtremeScript compiler. Note that the state machine approach can even be applied to our string processing library functions (and often is).

Final Details

`GetNextToken ()` was by far the biggest hurdle in completing the lexer's interface, but let's wrap things up by taking a quick look at the other functions. Up next is `SkipToNextLine ()`, which is a rather simple one:

```
int SkipToNextLine ()
{
    // Increment the current line

    ++ g_Lexer.iCurrSourceLine;

    // Return FALSE if we've gone past the end of the source code

    if ( g_Lexer.iCurrSourceLine >= g_iSourceCodeSize )
        return FALSE;

    // Set both indices to point to the start of the string

    g_Lexer.iIndex0 = 0;
    g_Lexer.iIndex1 = 0;
```

```
        // Turn off string lexeme mode, since strings can't span multiple lines

        g_Lexer.iCurrLexState = LEX_STATE_NO_STRING;

        // Return TRUE to indicate success

        return TRUE;
    }
```

It starts by incrementing the pointer to the current line, which moves us to the next line. It then makes sure we haven't moved beyond the last line in the file by comparing the new position to `g_iSourceCodeSize`. If this test passes, it sets both lexer indices to zero and resets the lexer state to `LEX_STATE_NO_STRING`. It returns `TRUE` to let the caller know that the next line was reached successfully.

I'll cover `ResetLexer ()` next because it's very similar to `SkipToNextLine ()` and is even simpler. Here's the code:

```
    void ResetLexer ()
    {
        // Set the current line to the start of the file

        g_Lexer.iCurrSourceLine = 0;

        // Set both indices to point to the start of the string

        g_Lexer.iIndex0 = 0;
        g_Lexer.iIndex1 = 0;

        // Set the token type to invalid, since a token hasn't been read yet

        g_Lexer.CurrToken = TOKEN_TYPE_INVALID;

        // Set the lexing state to no strings

        g_Lexer.iCurrLexState = LEX_STATE_NO_STRING;
    }
```

As you can see, it does many of the things `SkipToNextLine ()` does. The only major difference is that it sets the source line to zero rather than incrementing it, which lets us start fresh at the beginning of the file. It sets the initial token type to `TOKEN_TYPE_INVALID`, just to ensure a clean slate, and resets the lexer state as well.

The last function in our lexer interface is `GetLookAheadChar ()`, which scans through the source code from the current position until it finds the first character of the next token. Let's have a look at its implementation:

```
char GetLookAheadChar ()
{
    // We don't actually want to move the lexer's indices, so we'll
    // make a copy of them

    int iCurrSourceLine = g_Lexer.iCurrSourceLine;
    unsigned int iIndex = g_Lexer.iIndex1;

    // If the next lexeme is not a string, scan past any potential
   // leading whitespace

    if ( g_Lexer.iCurrLexState != LEX_STATE_IN_STRING )
    {
        // Scan through the whitespace and check for the end of the line

        while ( TRUE )
        {
            // If we've passed the end of the line, skip to the next
            // line and reset the index to zero

            if ( iIndex >= strlen ( g_ppstrSourceCode
                [ iCurrSourceLine ] ) )
            {
                // Increment the source code index

                iCurrSourceLine += 1;

                // If we've passed the end of the source file, just
                // return a null character

                if ( iCurrSourceLine >= g_iSourceCodeSize )
                    return 0;

                // Otherwise, reset the index to the first character on
                // the new line

                iIndex = 0;
            }
```

```
                // If the current character is not whitespace, return it, since
                // it's the first character of the next lexeme and is thus the
                // look-ahead

                if ( ! IsCharWhitespace ( g_ppstrSourceCode
                    [ iCurrSourceLine ][ iIndex ] ) )
                    break;

                // It is whitespace, however, so move to the next character
                // and continue scanning

                ++ iIndex;
            }
        }

        // Return whatever character the loop left iIndex at

        return g_ppstrSourceCode [ iCurrSourceLine ][ iIndex ];
    }
```

The function starts by making a copy of the lexer's internal indices into the current source line. Remember, since `GetLookAheadChar ()` is specifically designed to "peek" into the next token without actually advancing the stream, we can't make any permanent changes to the lexer's current state. Figure 9.38 illustrates the look-ahead.

As long as the current lexeme isn't a string, the function scans through any whitespace to find its way to the first non-whitespace character. If a whitespace character is found, the scanning loop breaks and the function returns whatever character it stopped on. Line breaks are also handled transparently, but without the aid of `SkipToNextLine ()` of course, since that would alter the lexer state.

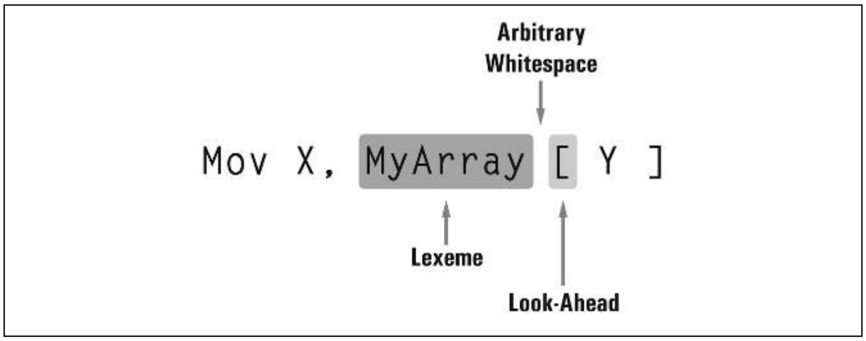


Figure 9.38

A look-ahead character being read.

Error Handling

We're just about ready to dive into parsing, but before we do, there's one important issue to address– how will we handle errors? There are three major aspects of error handling: detection, resynchronization, and message output. Detection is all about determining when an error has occurred in the first place, as well as what type of error it was. Resynchronization is the process of getting the parser back on track so that it can resume its processing, allowing the program to flag multiple errors (this is how most modern compilers, like Visual C++, produce "cascading" error messages). Lastly, and most importantly, the error message must be output to the screen or a log file of some sort in order to alert the user.

XASM is designed to be a simple and to-the-point middleman between the XtremeScript compiler we'll develop later and the XVM. As such, error handling will be clean but minimal. Because of this, we'll skip the resynchronization phase and design the program to halt the assembly process entirely at the first sign of an error.

Errors will be handled with three basic functions. Let's look at the first one, `ExitOnError ()`. This function causes the program to display an error message and terminate:

```
void ExitOnError ( char * pstrErrorMssg )
{
    // Print the message
    printf ( "Fatal Error: %s.\n", pstrErrorMssg );

    // Exit the program
    Exit ();
}
```

As you can see, it's all rather simple. The function spits out the error message (with an automatically appended period, which is nice), and terminates. One thing to note about this function, however, is that it's not meant to be used for code errors. It's only for general program errors, like problems with File I/O and the like. The next two functions will deal specifically with code errors.

NOTE
You may notice the call to `Exit ()` at the end of each error function. This is just a simple function that wraps the shutdown procedure of the assembler. It's really quite inconsequential, but if you're curious, there's only one way to find out the details-- ***look at the source!***

Next up, let's look at XASM's most versatile code-error handling function, `ExitOnCodeError ()`:

```
void ExitOnCodeError ( char * pstrErrorMssg )
{
    // Print the message
    printf ( "Error: %s.\n\n", pstrErrorMssg );
    printf ( "Line %d\n", g_Lexer.iCurrSourceLine );

    // Reduce all of the source line's spaces to tabs so it takes less
    // space and so the caret lines up with the current token properly
    char pstrSourceLine [ MAX_SOURCE_LINE_SIZE ];
    strcpy ( pstrSourceLine, g_ppstrSourceCode [ g_Lexer.iCurrSourceLine ] );

    // Loop through each character and replace tabs with spaces
    for ( unsigned int iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrSourceLine ); ++ iCurrCharIndex )
        if ( pstrSourceLine [ iCurrCharIndex ] == '\t' )
            pstrSourceLine [ iCurrCharIndex ] = ' ';

    // Print the offending source line
    printf ( "%s", pstrSourceLine );

    // Print a caret at the start of the (presumably) offending lexeme
    for ( unsigned int iCurrSpace = 0; iCurrSpace < g_Lexer.iIndex0;
          ++ iCurrSpace )
        printf ( " " );
    printf ( "^\n" );

    // Print message indicating that the script could not be assembled
    printf ( "Could not assemble %s.\n", g_pstrExecFilename );

    // Exit the program
    Exit ();
}
```

The output of this function is very cool. First, the current source line is printed to the screen so the user can actually see the offending code. The lexer's internal indices are then used to place a caret symbol directly under the character or token that caused the problem; since most code errors will involve a specific token, this produces accurate results virtually every time. Also, to save space and to make the process of aligning the caret easier, tabs are filtered out of a local copy of the source line.

There are times, however, when all that's necessary is to let the user know that a specific character was expected but not found. For this, there's `ExitOnCharExpectedError ()`:

```
void ExitOnCharExpectedError ( char cChar )
{
    // Create an error message based on the character
    char * pstrErrorMssg = ( char * ) malloc ( strlen ( "' ' expected" ) );
    sprintf ( pstrErrorMssg, "'%c' expected", cChar );

    // Exit on the code error
    ExitOnCodeError ( pstrErrorMssg );
}
```

As you can see, the function is built on top of `ExitOnCodeError ()`, so we get the extra formatting for free.

Parsing

With the lexical analyzer up and running, you're ready to build the parser around it. The nice thing about parsing is that you no longer have to worry about messy string manipulation and processing. Instead, you just deal with "building blocks" so to speak; the much higher level tokens and lexemes that your lexer provides you with. At this point, parsing becomes a rather easy job (at least, given the method of parsing you're going to use).

At this point in the pipeline, you're dealing with a very clean dataset, which is illustrated in Figure 9.39. Whitespace and comments don't exist, and your only input comes in the form of tokens (and optional lexemes or look-ahead characters when you request them). From the perspective of the parser, the human element of source code is almost entirely eliminated. You still have large-scale evidence of a human presence, such as syntax errors (and in fact, this is the phase in

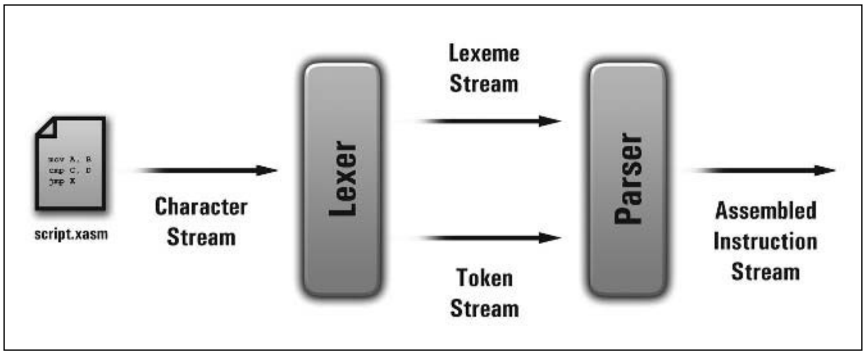


Figure 9.39

How the parser fits into the assembly pipeline.

which you'll detect them), but you don't have to worry about mixed caps, spacing, or anything along those lines.

The actual process of parsing the token stream is relatively simple. As mentioned in the parsing introduction, the main principal is identifying the initial token and predicting what should follow based on how that initial token fits into the rules of the language. Based on these initial tokens, you can determine what sort of line you're dealing with—whether it's a directive, instruction, line label, whatever—and easily parse the remaining tokens. Once the first line is finished, you read the next token in the stream (which will correspond with the first token of the next line), and start the process over, treating the newly read token as the initial token.

NOTE

The XASM parser is a somewhat ad-hoc implementation that most closely resembles a parsing method known as *recursive descent*, without the recursive element. Most generally, this represents an approach to parsing called *top-down* parsing, because we start with a general idea of what the source (the "initial token") is saying and work our way down the details. All you need to know at this point is that it's an easy-to-implement approach that gets the job done without a lot of fuss. Writing a top-down parser won't exactly put you in line for the Nobel Prize, but it's a good way to implement simple translation programs like this assembler that only need to handle a small, narrowly-defined language. The real goal is to ultimately compile a high-level language, so there's no point in spending months developing the perfect assembler. This is really just a means to an end.

Initializing the Parser

Before either pass can begin, the parser must be initialized. During the parsing process, a number of global variables are maintained that track the status of the script. For example, since the `SetStackSize` directive can only appear once, a flag that monitors its presence is checked when the directive is encountered and subsequently set. I'll list the code first, then we'll look at each line:

```
// ---- Initialize the script header

g_ScriptHeader.iStackSize = 0;
g_ScriptHeader.iIsMainFuncPresent = FALSE;
```

```
// ---- Set some initial variables

g_iInstrStreamSize = 0;
g_iIsSetStackSizeFound = FALSE;
g_ScriptHeader.iGlobalDataSize = 0;

// Set the current function's flags and variables

int iIsFuncActive = FALSE;
FuncNode * pCurrFunc;
int iCurrFuncIndex;
char pstrCurrFuncName [ MAX_IDENT_SIZE ];
int iCurrFuncParamCount = 0;
int iCurrFuncLocalDataSize = 0;

// Create an instruction definition structure to hold instruction information
// when dealing with instructions.
InstrLookup CurrInstr;

// Reset the lexer
ResetLexer ();
```

First the script header is initialized by setting the stack size to zero and clearing the flag that monitors the presence of `_Main ()`. The instruction stream size is then set to zero, the `SetStackSize` flag I mentioned above is cleared, and the global data size is set to zero.

A number of local flags are then declared that the parser will use to keep track of where it is in the script. `iIsFuncActive` is a flag that tells us whether or not the current line of code is within a function. Of course, this is cleared by default. The remaining variables in this section keep track of the current function's information; a pointer to its node in the function table, its index, its name, and so on.

An empty instruction lookup structure is then created, which is passed to `GetInstrByMnemonic ()` whenever an instruction's definition is needed. Lastly, the lexer is reset with a call to `ResetLexer ()`, and the show is ready to start. With this basic initialization stuff out of the way, we're going to knock down each parsing topic one by one, starting with directives.

Directives

I'm now going to cover each of the directives the assembler supports and discuss how each can be parsed. Remember, directives don't translate into actual machine code, so the translation stage that follows the parsing of a directive really just means storing its information in the appropriate tables and moving on.

At each iteration of the first pass, an initial token is read with a call to GetNextToken (), like this:

```
if ( GetNextToken () == END_OF_TOKEN_STREAM )
    break;
```

Note that before doing anything, we make sure we haven't passed the end of the token stream. If we have, the loop ends and the second pass begins. Otherwise, a switch is entered, wherein each case handles a different initial token. Each of the following subsections will represent one of the cases of this switch:

```
switch ( g_Lexer.CurrToken )
{
```

After the following section, you'll understand enough to write an assembler that can parse and translate directives (remember, *check out the included XASM source!*).

SetStackSize

Let's start with an easy one—SetStackSize. Here's an example of its usage:

```
SetStackSize 1024
```

This simple directive is reduced to only two tokens: TOKEN_TYPE_SETSTACKSIZE and TOKEN_TYPE_INT. Remember, the stack size must be set by an integer literal. Anything else will result in an error. Here's the case that handles SetStackSize:

```
case TOKEN_TYPE_SETSTACKSIZE:
    // SetStackSize can only be found in the global scope, so make sure we
    // aren't in a function.
    if ( iIsFuncActive )
        ExitOnCodeError ( ERROR_MSSG_LOCAL_SETSTACKSIZE );

    // It can only be found once, so make sure we haven't already found it
    if ( g_iIsSetStackSizeFound )
        ExitOnCodeError ( ERROR_MSSG_MULTIPLE_SETSTACKSIZES );

    // Read the next lexeme, which should contain the stack size
    if ( GetNextToken () != TOKEN_TYPE_INT )
        ExitOnCodeError ( ERROR_MSSG_INVALID_STACK_SIZE );
    // Convert the lexeme to an integer value from its string
    // representation and store it in the script header
    g_ScriptHeader.iStackSize = atoi ( GetCurrLexeme () );

    // Mark the presence of SetStackSize for future encounters
    g_iIsSetStackSizeFound = TRUE;

    break;
```

That wasn't so bad, huh? That's how parsing works. This pattern, as simple as it seems, can be applied to the entire language and yield just the results you're after. See how easy this otherwise intimidating assembly process becomes with the help of structured phases like lexical analysis and tokenization?

The basic process is as follows. The code first checks `iIsFuncActive` to make sure the directive wasn't found inside a function. If it was, an error occurs. Another test is performed, this time to make sure another instance of `SetStackSize` hasn't already been found. If it has, another error occurs. Otherwise, the next lexeme is read, which should be the stack size. If this isn't an integer token, it's an invalid stack size and a third error occurs. Otherwise, the lexeme is converted to an integer value with `atoi ()` and the stack size is set in the script header, along with the `g_iIsSetStackSizeFound` flag.

Func

Functions are declared with the `Func` directive, and consist of three tokens. For example:

```
Func MovePlayer
{
```

This code consists primarily of three tokens: `TOKEN_TYPE_FUNC` for the `Func` directive, `TOKEN_TYPE_IDENT` for the `MovePlayer` function name, and `TOKEN_TYPE_OPEN_BRACE`. There is one issue, however, because there's a line break between the function name and the brace. In this particular case, the lexer would return:

```
TOKEN_TYPE_FUNC
TOKEN_TYPE_IDENT
TOKEN_TYPE_NEWLINE
TOKEN_TYPE_OPEN_BRACE
```

However, as I mentioned earlier in the chapter, the syntax of our language is designed to gracefully support any particular curly-brace style, which may mean that the function name and curly brace won't be separated by any line breaks at all. To cover all bases, the parser is going to have to check for any number of line breaks, from 0 to *N*, between the name of the function and the opening brace. This will allow the users to use whatever style they're used to. Let's look at some code to parse it (also check out Figure 9.40):

```
case TOKEN_TYPE_FUNC:
{
    // First make sure we aren't in a function already, since nested functions
    // are illegal
    if ( iIsFuncActive )
        ExitOnCodeError ( ERROR_MSSG_NESTED_FUNC );

    // Read the next lexeme, which is the function name
    if ( GetNextToken () != TOKEN_TYPE_IDENT )
        ExitOnCodeError ( ERROR_MSSG_IDENT_EXPECTED );
    char * pstrFuncName = GetCurrLexeme ();

    // Calculate the function's entry point, which is the instruction
    // immediately following the current one, which is in turn equal to the
    // instruction stream size
    int iEntryPoint = g_iInstrStreamSize;

    // Try adding it to the function table, and print an error if it's already
    // been declared
    int iFuncIndex = AddFunc ( pstrFuncName, iEntryPoint );
    if ( iFuncIndex == -1 )
        ExitOnCodeError ( ERROR_MSSG_FUNC_REDEFINITION );

    // Is this the _Main () function?
    if ( strcmp ( pstrFuncName, MAIN_FUNC_NAME ) == 0 )
    {
        g_ScriptHeader.iIsMainFuncPresent = TRUE;
        g_ScriptHeader.iMainFuncIndex = iFuncIndex;
    }

    // Set the function flag to true for any future encounters and
    // reinitialize function tracking variables
    iIsFuncActive = TRUE;
    strcpy ( pstrCurrFuncName, pstrFuncName );
    iCurrFuncIndex = iFuncIndex;
    iCurrFuncParamCount = 0;
    iCurrFuncLocalDataSize = 0;
```

```
        // Read any number of line breaks until the opening brace is found
        while ( GetNextToken () == TOKEN_TYPE_NEWLINE );

        // Make sure the lexeme was an opening brace
        if ( g_Lexer.CurrToken != TOKEN_TYPE_OPEN_BRACE )
            ExitOnCharExpectedError ( '{' );

        // All functions are automatically appended with Ret, so increment the
        // required size of the instruction stream
        ++ g_iInstrStreamSize;

        break;
    }
```

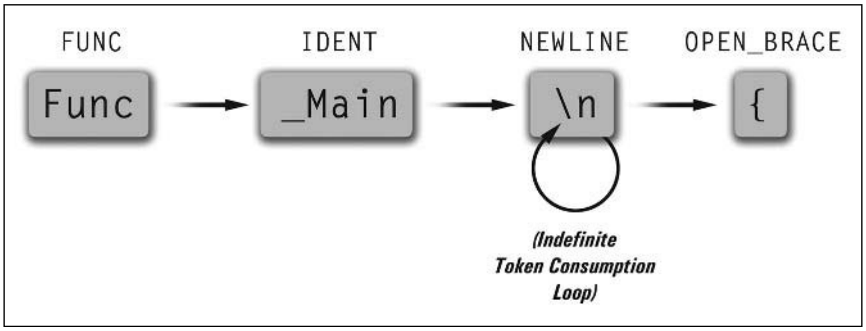


Figure 9.40

Parsing a function declaration with a flexible handling of line breaks.

We begin by first making sure a function isn't already being parsed. If it is, the current `Func` directive is illegal and an error is reported. Otherwise, the next lexeme is read, which should be the function name. If it's not a valid identifier, an error is reported. The function's entry point is then calculated, which is always equal to the current number of instructions in the stream. This initial function information (the name and entry point) is added to the function table with a call to `AddFunc ()`.

The function name is then analyzed to find out if it's `_Main ()`. If it is, the `_Main ()` flag is set in the script header and the function's index is recorded. We then set the function tracking variables so that subsequent iterations of the parser know:

- We're currently inside a function.
- The current function's name.
- The current function's index.

During the parsing of the function's body, you need to count the number of parameters and local variables as the function is parsed, which is why we initialize `iCurrFuncParamCount` and `iCurrFuncLocalDataSize` to zero. When the end of the function is reached, you can send this information to `SetFuncInfo ()` to finalize the function's entry in the table. Speaking of the end of a function, you need to parse that too, of course. You haven't learned how the instructions between the curly braces are parsed yet, so you're basically making a jump from the start of the function to the end, but I'll fill in the guts soon.

The end of a function is probably the easiest thing to parse in the whole language, because you just have to read a `TOKEN_TYPE_CLOSE_BRACE` token. Once the function is read, you need to check the global flags to make sure that a function is active (otherwise you have a dangling closing curly brace out in the middle of nowhere). If it is, you can fill in the function's data with the completed totals set in `iCurrFuncParamCount` and `iCurrFuncLocalDataSize`.

Lastly, there's one other thing the parser needs to do to translate the end of a function. Remember that XASM will automatically append the necessary `Ret` instruction to the end of each function. Remember also that the first pass of the assembler counts each instruction in order to allocate an instruction stream of the proper size before the second pass begins. Because of this, you need to remember to increment the instruction count by one each time a function ends to make room for the extra `Ret`.

```
Here's the code:
case TOKEN_TYPE_CLOSE_BRACE:
    // This should be closing a function, so make sure we're in one
    if ( ! iIsFuncActive )
        ExitOnCharExpectedError ( '}' );

    // Set the fields we've collected
    SetFuncInfo ( pstrCurrFuncName, iCurrFuncParamCount,
        iCurrFuncLocalDataSize );

    // Close the function
    iIsFuncActive = FALSE;

    break;
```

All we need to do is make sure we're in a function (reporting an error otherwise), save the information about the function we collected with `SetFuncInfo ()`, and clear the active function flag. Now that you can parse functions, let's look at how you parse the directives you'll find inside them; namely, `Var` and `Param`.

Var/Var []

The `Var` and `Var []` directives can occur both inside and outside of functions. As you've learned, those found outside declare variables and arrays within the global scope, and those found inside declare them in a scope local to that function.

Like I mentioned earlier when discussing the lexer, you'll need to utilize a one-character look-ahead when parsing the `Var` directive due to its optional `[]` notation for declaring arrays. Because the identifier following the `Var` lexeme might not be the end of the line, you'll find yourself in a non-deterministic situation that can only be resolved by examining the first character ahead of the current position. Check out Figure 9.41.

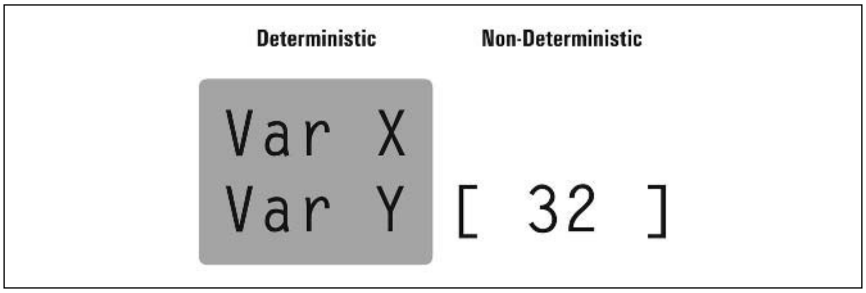


Figure 9.41

The non-deterministic nature of variable/ array declaration.

Let's start small and just handle single variables. Variables are declared in the form of the following example:

```
Var X
```

Which, fortunately, only translates to two tokens: `TOKEN_TYPE_VAR` and `TOKEN_TYPE_IDENT`. When a variable is encountered, you of course add it immediately to the symbol table. However, in order to properly determine its stack index, you need to know whether you're in a global or local scope. To do this, you check the value of `g_IsFuncActive`.

If you're in a function, you subtract the value of `iCurrFuncLocalDataSize` plus two from zero to obtain the relative stack index. Why do you do this? Think of it like this—although positive stack indices start from zero, negatives always start from -1 (because negative and positive indices can't "share" the zero index). When you encounter your first local variable, whose stack index should be -1, `iCurrFuncLocalDataSize` will be set to zero. However, for reasons we'll see in the next chapter, the top element of the stack (residing at index -1) has to be reserved for some of the VM's internal bookkeeping, so our variables get pushed down to index -2. Adding two to `iCurrFuncLocalDataSize` will result in a sum of two, which, when subtracted from zero, yields -2–the correct stack index. When the second variable is read, `iCurrFuncIndex` will equal 1. You increment this by two, subtract it from zero, resulting in -3, and you have the next stack index. This continues onward as more and more variables are read. See Figure 9.42 if you're as confused as I am.

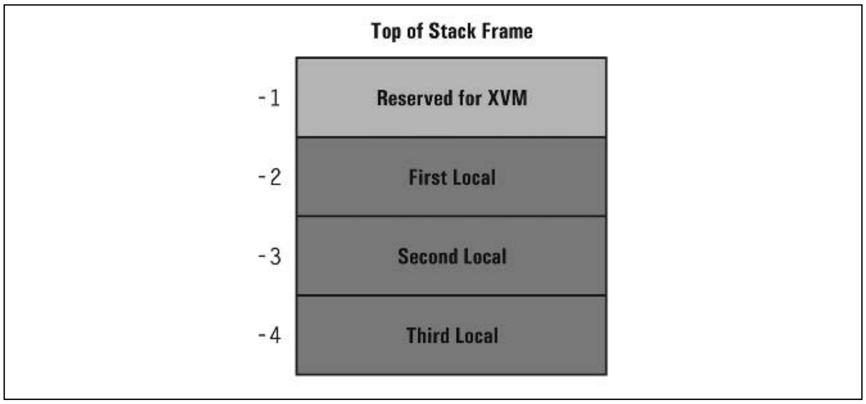


Figure 9.42

Determining a local variable's stack index.

Things are different in the case of globals. Global variables should get their own counter, because they're separate from locals and because, technically, global declarations can appear in between function declarations. This is why we initialized `g_ScriptHeader.iGlobalDataSize` to zero earlier. Every time a global variable is encountered, the current global data size is used as its index. Because this size starts out at zero, and the first global's stack index is zero, you can see how this relationship works. Check out Figure 9.43 for a better view of how locals and globals co-exist on the stack.

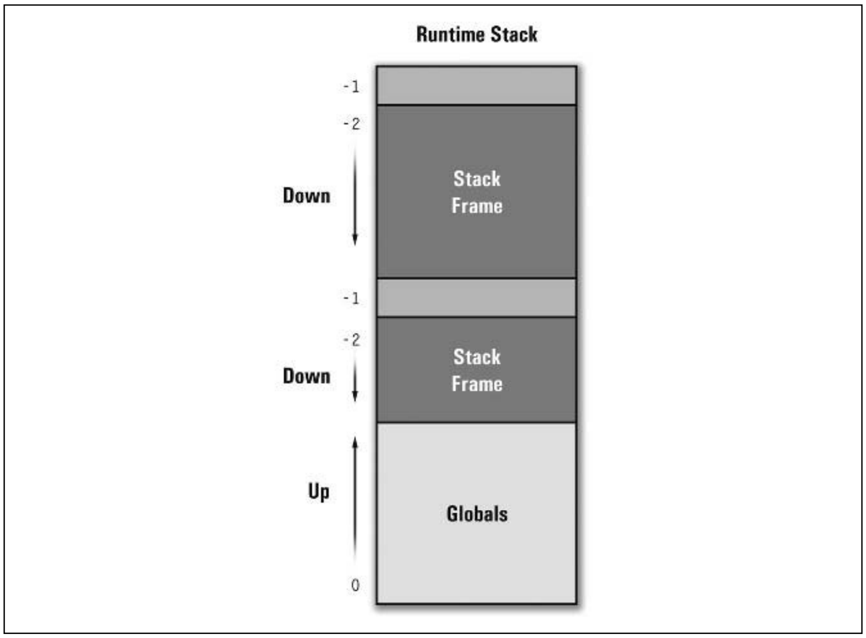


Figure 9.43

Local and global variables are stored on the same stack, with globals starting at zero and locals starting at –2 (relative to their particular stack frame).

With all that sorted out, let's take a look at the code for parsing single variables, in both the local and global scope:

```
case TOKEN_TYPE_VAR:
{
    // Get the variable's identifier
    if ( GetNextToken () != TOKEN_TYPE_IDENT )
        ExitOnCodeError ( ERROR_MSSG_IDENT_EXPECTED );
    char pstrIdent [ MAX_IDENT_SIZE ];
    strcpy ( pstrIdent, GetCurrLexeme () );

    // This version of the code only handles single variables
    int iSize = 1;

    // Determine the variable's index into the stack

    // If the variable is local, then its stack index is always the local data
    // size + 2 subtracted from zero
    int iStackIndex;
    if ( iIsFuncActive )
        iStackIndex = -( iCurrFuncLocalDataSize + 2 );
    // Otherwise it's global, so it's equal to the current global data size
    else
        iStackIndex = g_ScriptHeader.iGlobalDataSize;

    // Attempt to add the symbol to the table
    if ( AddSymbol ( pstrIdent, iSize, iStackIndex, iCurrFuncIndex ) == -1 )
        ExitOnCodeError ( ERROR_MSSG_IDENT_REDEFINITION );

    // Depending on the scope, increment either the local or global data size
    // by the size of the variable
    if ( iIsFuncActive )
        iCurrFuncLocalDataSize += iSize;
    else
        g_ScriptHeader.iGlobalDataSize += iSize;

    break;
}
```

Once `Var` is read, the first thing to do is make sure the following token is an identifier. If not, the declaration is invalid and an error is reported, otherwise, a local copy is made. For now, we set the variable size (stored in `iSize`) to 1 by default since this initial code won't handle arrays. The variable's stack index is then calculated using the same algorithm described earlier, and saved in `iStackIndex`. Using this information, a new symbol is added using `AddSymbol ()`, which reports an error in the event of a variable redefinition. Lastly, the current function's local data size is incremented by the size of the variable if the scope is local. Otherwise, the global data size is incremented.

This works, but of course, only handles single variables. To support arrays, you need to start by adding extra parsing code to interpret the extra tokens an array declaration brings with it. Here's an example:

```
Var MyArray [ 16384 ]
```

This code creates an array of 16,384 elements and is reduced by the lexer to the following tokens: `TOKEN_TYPE_VAR`, `TOKEN_TYPE_IDENT`, `TOKEN_TYPE_OPEN_BRACE`, `TOKEN_TYPE_INT`, and `TOKEN_TYPE_CLOSE_BRACE`.

Of course, in order to interpret these extra tokens, you need to use a look-ahead. Once you've parsed the array, the actual addition to the symbol table isn't much different. The only real change is taking the larger size into account in a few places, because the only difference between a variable and an array in XVM Assembly is how many stack elements it occupies.

Here's a new version of the code, now augmented to parse and translate array declarations as well:

```
case TOKEN_TYPE_VAR:
{
    // Get the variable's identifier
    if ( GetNextToken () != TOKEN_TYPE_IDENT )
        ExitOnCodeError ( ERROR_MSSG_IDENT_EXPECTED );

    char pstrIdent [ MAX_IDENT_SIZE ];
    strcpy ( pstrIdent, GetCurrLexeme () );

    // Now determine its size by finding out if it's an array or not, otherwise
    // default to 1.
    int iSize = 1;

    // Find out if an opening bracket lies ahead
    if ( GetLookAheadChar () == '[' )
    {
        // Validate and consume the opening bracket
        if ( GetNextToken () != TOKEN_TYPE_OPEN_BRACKET )
            ExitOnCharExpectedError ( '[' );
        // We're parsing an array, so the next lexeme should be an integer
        // describing the array's size
        if ( GetNextToken () != TOKEN_TYPE_INT )
            ExitOnCodeError ( ERROR_MSSG_INVALID_ARRAY_SIZE );

        // Convert the size lexeme to an integer value
        iSize = atoi ( GetCurrLexeme () );

        // Make sure the size is valid, in that it's greater than zero
        if ( iSize <= 0 )
            ExitOnCodeError ( ERROR_MSSG_INVALID_ARRAY_SIZE );

        // Make sure the closing bracket is present as well
        if ( GetNextToken () != TOKEN_TYPE_CLOSE_BRACKET )
            ExitOnCharExpectedError ( ']' );
    }

    // Determine the variable's index into the stack

    // If the variable is local, then its stack index is always the local data
    // size + 2 subtracted from zero
    int iStackIndex;
    if ( iIsFuncActive )
        iStackIndex = -( iCurrFuncLocalDataSize + 2 );
    // Otherwise it's global, so it's equal to the current global data size
    else
        iStackIndex = g_ScriptHeader.iGlobalDataSize;

    // Attempt to add the symbol to the table
    if ( AddSymbol ( pstrIdent, iSize, iStackIndex, iCurrFuncIndex ) == -1 )
        ExitOnCodeError ( ERROR_MSSG_IDENT_REDEFINITION );

    // Depending on the scope, increment either the local or global data size
    // by the size of the variable
    if ( iIsFuncActive )
        iCurrFuncLocalDataSize += iSize;
    else
        g_ScriptHeader.iGlobalDataSize += iSize;

    break;
}
```

Pretty simple addition, huh? It was just a matter of taking the new variable size into account. If the look-ahead reveals an open bracket, two tokens are read. The first should be the bracket itself, and the second should be an integer token correlating to the size of the array. The lexeme is translated into a real integer with `atoi ()`, and the value is saved in `iSize`. Finally, the closing bracket is verified and the process continues as normal.

Param

Although you wouldn't initially assume it, `Param` is an exception to the usual convention of parsing all directives in the first pass. The reason you have to save this until the second pass is because a parameter's location on the stack is entirely relative to the final size of the function's local data. For example, if a function declares four variables, the last local variable will reside on the stack at index -5 (remember, local variables start at index -2), the return address will be at -6, and the first parameter will be at -7. If the function declares only two local variables, the first parameter will be found at -5. If the function declares eight variables and an array of 12 elements, the parameters won't start until index -23. The total size of the function's local data isn't known until the function has been fully scanned, which means you'll have already missed the `Param` directives, and thus, have to wait until the second pass. The parser does make a note of `Param` instances in the first pass simply to count them and increment `g_FuncParamCount`, but the parameters themselves are not recorded to the symbol table until the second. Figure 9.44 should help the brain swelling go down.

What this also means is that unlike variables, parameters cannot be forward referenced. Of course, you shouldn't be using forward parameter references to begin with, so this won't be a problem. :)

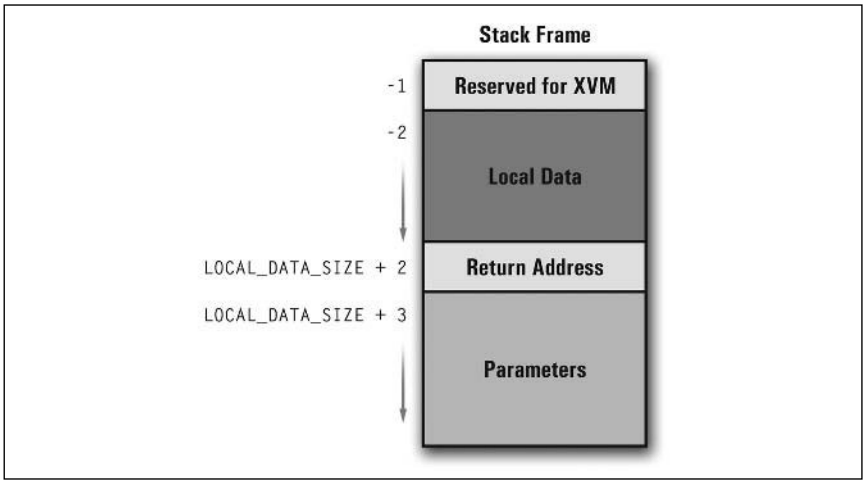


Figure 9.44

How parameters fit into the stack frame.

Just as in the first pass, the second pass will keep track of which function it's in, which is helpful so you can assign it to the parameter with the proper scope. You'll also need to once again keep track of `iCurrFuncParamCount` for each function, because the current parameter count will help you determine the stack index. The stack index for a parameter is always relative to the function's local data size (as usual, the extra 1 is for the return address). Therefore, if the local data size is 6, the parameter's stack address is (-7 - 2), or -9. The `Param` directive has the same form of a single variable declaration, so here's an example:

```
Param Y
```

The lexer will reduce this line of code to `TOKEN_TYPE_PARAM`, and `TOKEN_TYPE_IDENT`. Here's some code for parsing parameter declarations:

```
case TOKEN_TYPE_PARAM:
{
    // Read the next token to get the identifier
    if ( GetNextToken () != TOKEN_TYPE_IDENT )
        ExitOnCodeError ( ERROR_MSSG_IDENT_EXPECTED );

    // Read the identifier, which is the current lexeme
    char * pstrIdent = GetCurrLexeme ();

    // Calculate the parameter's stack index
    int iStackIndex = -( pCurrFunc->iLocalDataSize + 2 +
        ( iCurrFuncParamCount + 1 ) );

    // Add the parameter to the symbol table
    if ( AddSymbol ( pstrIdent, 1, iStackIndex, iCurrFuncIndex ) == -1 )
        ExitOnCodeError ( ERROR_MSSG_IDENT_REDEFINITION );

    // Increment the current parameter count
    ++ iCurrFuncParamCount;

    break;
}
```

This simple parser first makes sure that the current token is an identifer, much like `Var` did. Once the identifier has been validated, the parameter's stack index is calculated by adding two to the local data size, plus the current number of parameters, plus one (to make room for the return address).

And there you have it—parsing code for handling each directive.

Line Labels

Line labels will first appear to the parser in the form of an identifier token, since that's what a label is. This means that any time your initial token is TOKEN_TYPE_IDENT, the look-ahead character can be used to find out if the following token is a colon. If so, it's definitely a line label declaration.

Here's an example of a line label:

```
MyLabel:
```

It's yet another simple structure to parse. The lexer will spit this out as TOKEN_TYPE_IDENT and TOKEN_TYPE_COLON, which makes your job pretty easy. Here's the code:

```
case TOKEN_TYPE_IDENT:
{
    // Make sure it's a line label
    if ( GetLookAheadChar () != ':' )
        ExitOnCodeError ( ERROR_MSSG_INVALID_INSTR );

    // Make sure we're in a function, since labels can only appear there
    if ( ! iIsFuncActive )
        ExitOnCodeError ( ERROR_MSSG_GLOBAL_LINE_LABEL );

    // The current lexeme is the label's identifier
    char * pstrIdent = GetCurrLexeme ();

    // The target instruction is always the value of the current
    // instruction count, which is the current size - 1
    int iTargetIndex = g_iInstrStreamSize - 1;

    // Save the label's function index as well
    int iFuncIndex = iCurrFuncIndex;

    // Try adding the label to the label table, and print an error if it
    // already exists
    if ( AddLabel ( pstrIdent, iTargetIndex, iFuncIndex ) == -1 )
        ExitOnCodeError ( ERROR_MSSG_LINE_LABEL_REDEFINITION );

    break;
}
```

The code begins by making sure a colon follows the identifier. If not, we can assume that it actually wasn't a label, but rather an invalid instruction. The label's scope is then checked to make sure it's not being declared globally, which is illegal. Both of these cases result in errors. The current lexeme contains the label itself, and the current instruction (which is always equal to the current size of the instruction stream minus one) is locally saved as the label's target instruction index. The function in which the label resides is also recorded, and all of this information is saved in a new entry in the label table using `AddLabel ()`. If the label already exists, a label redefinition error is reported.

Done and done. At this point, the only thing your theoretical parser can't do is handle instructions. Of course, I've saved the biggest job for last.

Instructions

Like the parsing of `Param` directives, instruction parsing takes place in the second pass. During this pass, with the exception of parameter information, you know everything you need to know about the script. You know all about its functions, what instructions each line label targets, and have information on all of the script's local and global variables. In other words, you're capable of resolving any operand you come across and reducing instructions to machine code.

Generally speaking, there are two basic ways to approach the interpretation of an instruction set. Rather than introduce them here, I'll let them speak for themselves in the following subsections.

The Brute Force Approach

The first and most obvious approach is just to use brute force. Whenever an instruction needs to be parsed, you enter a giant `if-else if-else` block that compares the lexeme to each instruction mnemonic in the language. Once the mnemonic has been matched, it's just a simple matter of parsing the instruction's operands like you've parsed everything else.

Here's a pseudo-code example of parsing a `Mov` instruction:

```
// Save the instruction's mnemonic
string InstrMnemonic = GetCurrLexeme ();
// Are we dealing with a Mov instruction?
if ( InstrMnemonic == "MOV" )
{
    // Parse first operand
    // Parse comma
    if ( GetNextToken () != TOKEN_TYPE_COMMA )
       ExitOnCharExpectedError ( ',' );
    // Parse second operand
    // etc.
}
```

Notice that I've pretty much glossed over the process of parsing the operands. This is because operand parsing is a rather huge job and would only end up cluttering this example. In fact, it's easily the most complex part of parsing an instruction. In fact, therein lies the problem.

Think about it—any given operand can be one of any number of types. Some of these involve single tokens; others involve many. There are some simple ones, like integer and float literals, the `_RetVal` register, and line labels and function calls, all of which are deterministic and simple to parse. Then there are deterministic operands that take up multiple tokens; for example, strings that always start with a double quote, followed by a string literal value, followed by a closing double quote. And lastly, there are multiple-token operands that are non-deterministic; namely, variable references (which are themselves single-token) and array references. And within array references you've got two further "subtypes", because you have to differentiate between integer literal array indices and variable indices! In a word, it's complicated.

However, parsing line labels and every supported directive was complicated too, and you solved it relatively easily with a simple, methodical parsing approach. You can do the same here. The problem, however, is that you have a lot of instructions, and if each is represented individually by its own `else if` block, you're going to have to physically duplicate the potentially huge operand-parsing logic countless times, which is unacceptable.

This is why it's generally a bad idea to manually write parsing code for each instruction. Furthermore, it's a rigid approach as well. If you want to add, remove, or worst of all, change a given instruction, you have to mess with this huge, unruly block of code. This in itself is an error-prone and laborious process that I think we'd all like to avoid if possible.

Fortunately, there's a solution that's not only elegant and easy to implement, but infinitely more robust, flexible, and compact.

A Generic Instruction Parser

If for no other reason, you probably knew from the start that the brute force approach outlined previously wasn't going to be the final word on instruction parsing because it ignores one of the first things you learned about how assemblers work—the instruction lookup table. There's no need for such a table if each instruction is represented with its own block of code, but I probably wouldn't have wasted everyone's time mentioning it in the first place if you weren't going to use it, right?

Your intuition has served you well, because this is exactly right. Rather than directly code an individual parser for each instruction, you'll instead write a single generic one. However, the "single instruction" that this parser understands can be changed based on a number of input values, which it'll read from the master instruction lookup table. These values will tell it which instruction to parse and what sort of operands to anticipate. Check out Figure 9.45.

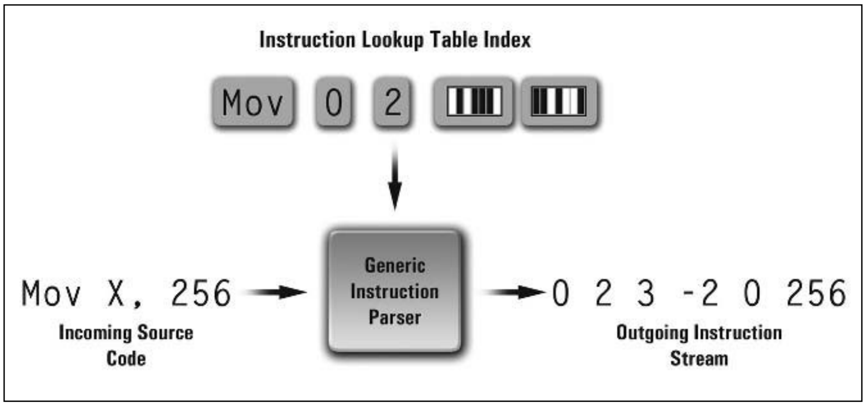

Instruction Lookup Table Index
Mov
0
2
Generic Instruction Parser
Mov X, 256
Incoming Source Code
0 2 3 -2 0 256
Outgoing Instruction Stream

Figure 9.45

A generic instruction parser.

Since we've already designed and implemented the instruction lookup table, we have everything we need to get started. Just as a refresher, the each entry in the instruction lookup table contains:

- The instruction's mnemonic, which is used to map instructions in the source file to their entries in the table.
- The opcode.
- The number of operands the instruction accepts.
- A dynamic array of 4-byte bitfields, each of which contains a series of 1-bit flags that determine which data types the corresponding operand can accept.

Let's see how this data can be applied to a generic instruction parser.

Assembling the Opcode

The first and most obvious step in assembling an instruction is translating the mnemonic to an opcode. This is accomplished with a simple call to `GetInstrByMnemonic ()`, which fills an `InstrLookup` structure with information regarding the instruction. Here's the initial code for the instruction parser:

```
case TOKEN_TYPE_INSTR:
{
    // Get the instruction's info using the current lexeme (the mnemonic )
    GetInstrByMnemonic ( GetCurrLexeme (), & CurrInstr );

    // Write the opcode to the stream
    g_pInstrStream [ g_iCurrInstrIndex ].iOpcode = CurrInstr.iOpcode;
```

This code is invoked when the lexer returns an instruction token, and begins by using the current lexeme (which contains the instruction mnemonic) to retrieve the instruction's lookup structure. This is why we declared the `CurrInstr` structure when the parser was initialized. This structure is initially used to write the opcode to the instruction stream at the index specified by `g_iCurrInstrIndex`.

The parser thus far will produce an assembled instruction stream that represents each source code instruction as an opcode. There aren't any operands yet, but it's definitely a start and provides a true, assembled "skeleton" of the final script.

Assembling the Operand Count

The next logical step in your instruction parser is the ability to add the operand count to the assembled instruction stream. If you recall earlier discussions, each instruction in the stream is composed of the following components: the opcode, the operand count, and the operand data itself. Because the operands are easily the most complex aspect of assembling instructions, you can work your way up by first adding the operand count field.

```
// Write the operand count to the stream
g_pInstrStream [ g_iCurrInstrIndex ].iOpCount = CurrInstr.iOpCount;

// Allocate space to hold the operand list
Op * pOpList = ( Op * ) malloc ( CurrInstr.iOpCount * sizeof ( Op ) );
```

This next block of code in the instruction parser reads the `iOpCount` field from the `CurrInstr` structure and writes it to the corresponding field in the current instruction in the assembled stream. In addition, it also goes ahead and allocates the space for the assembled operands; once we have the operand count, we have enough information to do this. This new array will be used by the remainder of the instruction parser to hold the assembled operands' types and data.

At this point, two thirds of the instruction has been assembled, so let's check out the final step.

Assembling the Operands

Handling the operands of an instruction is a two-fold process. First, and most obviously, there's the issue of parsing and assembling them. However, before you do this, you need to know exactly which operands you're looking for in the first place. For example, you'll parse a line label differently than you will a string or array index, so if you're parsing a jump instruction's line label operand, there's no need to waste time looking for other operand types.

Since each operand in the instruction lookup table is defined with a bitfield, we created a number of masks that could be used to read and write individual bits. Table 9.18 reiterates these masks to refresh your memory.

Table 9.18 Operand Type Bitfield Masks

Constant	Value	Description
`OP_FLAG_TYPE_INT`	1	Integer literal value
`OP_FLAG_TYPE_FLOAT`	2	Floating-point literal value
`OP_FLAG_TYPE_STRING`	4	String literal value
`OP_FLAG_TYPE_MEM_REF`	8	Memory reference (variable or array index)
`OP_FLAG_TYPE_LINE_LABEL`	16	Line label (used in jump instructions)
`OP_FLAG_TYPE_FUNC_NAME`	32	Function name (used in the `Call` instruction)
`OP_FLAG_TYPE_HOST_API_CALL`	64	Host API call (used in the `CallHost` instruction)
`OP_FLAG_TYPE_REG`	128	A register, which is always the `_RetVal` register in our case

I mentioned originally that these masks don't match up directly with the specific operand types we've established because the parser only needs a general idea of which operands are acceptable, as opposed to the exact type that was used. The XVM, however, will need to know *exactly* what type of operand was actually used at runtime, because variables, arrays indexed with integer literals, and arrays indexed with variables are all handled differently. In fact, you'll need a new set of constants to handle the outgoing operand types that are written to the instruction stream. These will correspond with the operand types we decided upon in the description of the .XSE format. Table 9.19 lists these types.

You can now begin the implementation of your operand parser. Because each instruction can have *N* number of operands, you need to write your parser in the form of a loop. On a basic level, the loop should iterate through each operand specified by the `iOpCount` field we read from `CurrInstr`, and read the `OpList []` array to determine which types are supported by that particular operand.

With the opcode and operand count written to the stream, the next part of the instruction parser is the operand parsing loop. The loop starts by reading out the operand type bitfield, reading in the operand's initial token processing the operand, and ensuring that each operand except for the last is followed by a comma. Here's the general skeleton:

Table 9.19 Operand List Type Constants

Constant	Description
OP_TYPE_INT	Integer literal value
OP_TYPE_FLOAT	Floating-point literal value
OP_TYPE_STRING	String literal index
OP_TYPE_ABS_STACK_INDEX	An absolute stack index (for variables and arrays indexed with integer literals)
OP_TYPE_REL_STACK_INDEX	A relative stack index (for arrays indexed with variables)
OP_TYPE_INSTR_INDEX	An instruction index (used for jump targets)
OP_TYPE_FUNC_INDEX	Function index (used for Call instructions)
OP_TYPE_HOST_API_CALL_INDEX	Host API call index (used for CallHost instructions)
OP_TYPE_REG	A register, which in our case always means the _RetVal register

```
// Loop through each operand, read it from the source and assemble it
for ( int iCurrOpIndex = 0; iCurrOpIndex < CurrInstr.iOpCount;
    ++ iCurrOpIndex )
{
    // Read the operand's type bitfield
    OpTypes CurrOpTypes = CurrInstr.OpList [ iCurrOpIndex ];

    // Read in the next token, which is the initial token of the operand
    Token InitOpToken = GetNextToken ();

    // --- Process the operand

    // Make sure a comma follows the operand, unless it's the last one
    if ( iCurrOpIndex < CurrInstr.iOpCount - 1 )
        if ( GetNextToken () != TOKEN_TYPE_COMMA )
            ExitOnCharExpectedError ( ',' );
}
```

```
// Make sure there's no extraneous stuff ahead
if ( GetNextToken () != TOKEN_TYPE_NEWLINE )
    ExitOnCodeError ( ERROR_MSSG_INVALID_INPUT );

// Copy the operand list pointer into the assembled stream
g_pInstrStream [ g_iCurrInstrIndex ].pOpList = pOpList;

// Move along to the next instruction in the stream
++ g_iCurrInstrIndex;
```

This actually brings you closer than you might think to a working operand parser and assembler. You also might notice that this code listing includes the completion of the instruction; the parser makes sure there's nothing following the end of the instruction on the line, the operand list pointer is copied into the assembled instruction stream, and `g_iCurrInstrIndex` is incremented.

So now, all that's really left is to identify and parse the operands as they exist in the source code. The framework around which this process can be carried out is already in place, so you're only one step away from completion. Once you're inside the operand loop, the next token you read is the first token of the operand. This is like a new "initial token", and so your parsing strategy will be based on whatever its type happens to be.

The easiest operands to parse are of the deterministic, single-token variety. These include:

- Integer literals
- Floating-point literals
- The `_RetVal` register

All of these operands exist as single tokens. The basic strategy here, then, is to read the initial token and determine what its type is. You can use a `switch` construct to compare this type to each of the possible operand types until you find a match. When you find a match, you first validate the operand type against the current instruction; in other words, you make sure that the operand of the instruction you're dealing with supports the operand type you've found. If this checks out, you can proceed to parse and translate the operand into its assembled state and write it to the instruction stream. If it's not supported, you can of course exit on an error.

```
// --- Process the operand
switch ( InitOpToken )
{
    // An integer literal
    case TOKEN_TYPE_INT:

        // Make sure the operand type is valid
        if ( CurrOpTypes & OP_FLAG_TYPE_INT )
```

```
            {
                // Set an integer operand type
                pOpList [ iCurrOpIndex ].iType = OP_TYPE_INT;

                // Copy the value into the operand list from the current
                // lexeme
                pOpList [ iCurrOpIndex ].iIntLiteral = atoi ( GetCurrLexeme () );
            }
            else
                ExitOnCodeError ( ERROR_MSSG_INVALID_OP );

            break;
    }
```

This code implements an integer operand parse-and-assemble sequence. Of course, that leaves a number of other operand types, but you get the idea. The process is virtually the same for all operands; the basic process is to use the same type of parsing strategies you've used for everything else to read out the operand itself. Analysis of the lexemes associated with each token can then be converted to the data that needs to be written out to the executable in the instruction stream.

Rather than just give you a code dump, let's explore the actual process behind parsing each type of operand. These algorithms, when coded, form the remaining `case`s in the `switch` block. Implementation of each of these can be found in the XASM source, of course.

- **Integer literals.** This operand type was also listed in the previous code, but here's the verbal explanation. Because integers are simple, deterministic tokens, you need only read out the initial token. If it's of type `TOKEN_TYPE_INT`, you know the operand is already fully read from the token stream. You then use `atoi ()` to convert the lexeme (which is a string representation of the number) to its numeric equivalent and write that to the operand list. The operand type is set to `OP_TYPE_INT`.
- **Floating-point literal.** Floating-point literals are treated in the exact same way integers are, except you need to read a `TOKEN_TYPE_FLOAT` token. The lexeme is then converted to a floating-point numeric with `atof ()`, which is written to the operand list. The operand type is set to `OP_TYPE_FLOAT`.
- **String literal.** This is a relatively easy operand to parse (which is ironic, given how complicated it was to lex), but it does require more than one token to express. If the initial token is `TOKEN_TYPE_QUOTE`, you know a string is on the way. The next token is read, which should be of type `TOKEN_TYPE_STRING`. This token's lexeme is the string value itself, which is immediately written to the string table. `AddString ()` will return the string's index, which is then written out to the operand list. The operand type is set to `OP_TYPE_STRING`.

- **The `_RetVal` register.** `_RetVal` is another easy one. It exists as a single, deterministic token, which means all you need to do is make sure the initial token is `TOKEN_TYPE_REG`, and write the register code zero to the operand list. The operand type is set to `OP_TYPE_REG`.
- **Line labels.** This is the first operand type that involves an identifier, which makes it non-deterministic. The reason for this is that labels, function names, host API calls, variables, and arrays indices all either begin with identifiers or are solely defined as identifiers. Fortunately, you can easily resolve this situation by checking the supported operand type bitfield for that particular operand. If a line label is accepted, the identifier must be the label. You then get the label's target index from the label table and write this to the operand list. The operand type is set to `OP_TYPE_INSTR_INDEX`.
- **Variables.** Variables are the first operands you need to check for when the operand type bitfield contains an `OP_TYPE_MEM_REF` flag. If the look-ahead character does not reveal an open bracket, you know there's no array reference to worry about. You then use the variable name as a search key for the symbol table to retrieve the variable's stack index, and write that to the operand list. Note also that local variables, global variables, and parameters are all taken care of with this simple process—the only difference between all three of these are their stack indices, which is handled transparently by the symbol table. The operand type is set to `OP_TYPE_ABS_STACK_INDEX`.
- **Array indices.** Arrays can be indexed with both integer literals and variables, two cases that must be handled separately. Array index operands always start out as variables until the open bracket is discovered with the look-ahead. The parser then focuses on the structure of the array index, which is always one of two token sequences, depending on the index type: `TOKEN_TYPE_OPEN_BRACKET`, `TOKEN_TYPE_INT` and `TOKEN_TYPE_CLOSE_BRACKET`, or `TOKEN_TYPE_OPEN_BRACKET`, `TOKEN_TYPE_IDENT`, `TOKEN_TYPE_CLOSE_BRACKET`. In the first case (an integer index), the integer is added to the base address of the array (using the symbol table to find the stack index) and written to the operand as an absolute stack index operand. In the second case (variable index), the arrays base index is written out to the operand list along with the index of the variable and the operand type is set to relative stack index. The operand type is set to `OP_TYPE_REL_STACK_INDEX`.
- **Function names.** Function names are used as operands to the `Call` instruction and are a single `TOKEN_TYPE_IDENT` token. This token's lexeme contains the function name itself, which is used as a search key into the function table to retrieve the function's index. This index is then written to the operand list. The operand type is set to `OP_TYPE_FUNC_INDEX`.
- **Host API calls.** Calls to the host API are treated much like string literal values in the sense that they're added to the host API call table and replaced with an index. `AddHostAPICall ()` is used to add the call, which returns the index that must be written to the operand table. The operand type is set to `OP_TYPE_HOST_API_CALL_INDEX`.

This sums up the operand-parsing process. This list should go hand in hand with a personal examination of the XASM source, which provides a complete explanation of how the assembler works.

Building the .XSE Executable

The source file has been fully assembled, so all that remains is dumping everything into an .XSE file. We already know what the structure of the file is like, so let's look at some code. To get started, the file is opened for binary output (assume `pstrFilename`) contains the name of the executable file):

```
FILE * pExecFile;
if ( ! ( pExecFile = fopen ( pstrFilename, "wb" ) ) )
    ExitOnError ( "Could not open executable file for output" );
```

With the file open, we can begin writing data.

The Header

The header is written first:

```
// Write the ID string (4 bytes)
fwrite ( XSE_ID_STRING, 4, 1, pExecFile );

// Write the version (1 byte for each component, 2 total)
char cVersionMajor = VERSION_MAJOR,
     cVersionMinor = VERSION_MINOR;
fwrite ( & cVersionMajor, 1, 1, pExecFile );
fwrite ( & cVersionMinor, 1, 1, pExecFile );

// Write the stack size (4 bytes)
fwrite ( & g_ScriptHeader.iStackSize, 4, 1, pExecFile );

// Write the global data size (4 bytes )
fwrite ( & g_ScriptHeader.iGlobalDataSize, 4, 1, pExecFile );

// Write the _Main () flag (1 byte)
char cIsMainPresent = 0;
if ( g_ScriptHeader.iIsMainFuncPresent )
   cIsMainPresent = 1;
fwrite ( & cIsMainPresent, 1, 1, pExecFile );

// Write the _Main () function index (4 bytes)
fwrite ( & g_ScriptHeader.iMainFuncIndex, 4, 1, pExecFile );
```

Notice that the function makes a number of local copies of the data before writing it. This is done to ensure that the variable written to the file occupies the exact number of bytes specified by the format. Even though 32-bit integers are used to store most integer values internally, many of these values are represented more efficiently in the file as 8- and 16-bit values. In these cases, the values are temporarily stored in `char` and `short` variables.

Everything beyond that should speak for itself. Each field is written from its structure, one by one.

The Instruction Stream

The instruction stream comes next, and is probably the most complex structure to write. Much like we saw in the parsing phase, the writing of the instruction stream is complicated by the fact that each operand type must be handled differently.

The general strategy when writing the stream is this:

- Start by writing the instruction count.
- Loop through each instruction in the stream and write out its opcode and operand count.
- Loop through each operand in the instruction's operand array and write out its type. Following the type, use a `switch` block to write out the specific operand data based on the type.

Here's the code:

```
// Output the instruction count (4 bytes)
fwrite ( & g_iInstrStreamSize, 4, 1, pExecFile );

// Loop through each instruction and write its data out
for ( int iCurrInstrIndex = 0;
      iCurrInstrIndex < g_iInstrStreamSize;
      ++ iCurrInstrIndex )
{
    // Write the opcode (2 bytes)
    short sOpcode = g_pInstrStream [ iCurrInstrIndex ].iOpcode;
    fwrite ( & sOpcode, 2, 1, pExecFile );

    // Write the operand count (1 byte)
    char iOpCount = g_pInstrStream [ iCurrInstrIndex ].iOpCount;
    fwrite ( & iOpCount, 1, 1, pExecFile );
```

```
// Loop through the operand list and print each one out
for ( int iCurrOpIndex = 0; iCurrOpIndex < iOpCount; ++ iCurrOpIndex )
{
    // Make a copy of the operand pointer for convenience
    Op CurrOp = g_pInstrStream
        [ iCurrInstrIndex ].pOpList [ iCurrOpIndex ];

    // Create a character for holding operand types (1 byte)
    char cOpType = CurrOp.iType;
    fwrite ( & cOpType, 1, 1, pExecFile );

    // Write the operand depending on its type
    switch ( CurrOp.iType )
    {
        // Integer literal
        case OP_TYPE_INT:
            fwrite ( & CurrOp.iIntLiteral, sizeof ( int ), 1, pExecFile );
            break;

        // Floating-point literal
        case OP_TYPE_FLOAT:
            fwrite ( & CurrOp.fFloatLiteral, sizeof ( float ), 1,
                 pExecFile );
            break;

        // String index
        case OP_TYPE_STRING_INDEX:
            fwrite ( & CurrOp.iStringTableIndex, sizeof ( int ), 1,
                pExecFile );
            break;

        // Instruction index
        case OP_TYPE_INSTR_INDEX:
            fwrite ( & CurrOp.iInstrIndex, sizeof ( int ), 1, pExecFile );
            break;

        // Absolute stack index
        case OP_TYPE_ABS_STACK_INDEX:
            fwrite ( & CurrOp.iStackIndex, sizeof ( int ), 1, pExecFile );
            break;
```

```
                // Relative stack index
                case OP_TYPE_REL_STACK_INDEX:
                    fwrite ( & CurrOp.iStackIndex, sizeof ( int ), 1, pExecFile );
               fwrite ( & CurrOp.iOffsetIndex, sizeof ( int ), 1, pExecFile );
                    break;

                // Function index
                case OP_TYPE_FUNC_INDEX:
                    fwrite ( & CurrOp.iFuncIndex, sizeof ( int ), 1, pExecFile );
                    break;

                // Host API call index
                case OP_TYPE_HOST_API_CALL_INDEX:
                    fwrite ( & CurrOp.iHostAPICallIndex, sizeof ( int ), 1,
                        pExecFile );
               break;

                // Register
                case OP_TYPE_REG:
                    fwrite ( & CurrOp.iReg, sizeof ( int ), 1, pExecFile );
                    break;
            }
        }
}
```

The String Table

Immediately following the instruction stream is the string table, which consists almost entirely of raw string data. Since this is the first linked list we'll be writing to a file, we need to create a dummy node pointer to traverse the list. We'll also use this node pointer for the remaining lists in the table.

```
int iCurrNode;
LinkedListNode * pNode;
Now for the table itself:
// Write out the string count (4 bytes)
fwrite ( & g_StringTable.iNodeCount, 4, 1, pExecFile );

// Set the pointer to the head of the list
pNode = g_StringTable.pHead;
```

```
// Create a character for writing parameter counts
char cParamCount;

// Loop through each node in the list and write out its string
for ( iCurrNode = 0; iCurrNode < g_StringTable.iNodeCount; ++ iCurrNode )
{
    // Copy the string and calculate its length
    char * pstrCurrString = ( char * ) pNode->pData;
    int iCurrStringLength = strlen ( pstrCurrString );

    // Write the length (4 bytes), followed by the string data (N bytes)
    fwrite ( & iCurrStringLength, 4, 1, pExecFile );
    fwrite ( pstrCurrString, strlen ( pstrCurrString ), 1, pExecFile );

    // Move to the next node
    pNode = pNode->pNext;
}
```

The table is written in a very straightforward way– the list is traversed from start to finish, and at each node the string is written out. Notice however that we never stored the length of each string in the table itself, which is why it's calculated here.

The Function Table

The next table to write is the function table, which describes each of the script's functions. This is another linked list, so we'll use the same node pointer declared above. Like the string table, it should all be reasonably straightforward:

```
// Write out the function count (4 bytes)
fwrite ( & g_FuncTable.iNodeCount, 4, 1, pExecFile );

// Set the pointer to the head of the list
pNode = g_FuncTable.pHead;

// Loop through each node in the list and write out its function info
for ( iCurrNode = 0; iCurrNode < g_FuncTable.iNodeCount; ++ iCurrNode )
{
    // Create a local copy of the function
    FuncNode * pFunc = ( FuncNode * ) pNode->pData;

    // Write the entry point (4 bytes)
    fwrite ( & pFunc->iEntryPoint, sizeof ( int ), 1, pExecFile );
    // Write the parameter count (1 byte)
    cParamCount = pFunc->iParamCount;
    fwrite ( & cParamCount, 1, 1, pExecFile );

    // Write the local data size (4 bytes)
    fwrite ( & pFunc->iLocalDataSize, sizeof ( int ), 1, pExecFile );

    // Move to the next node
    pNode = pNode->pNext;
}
```

For convenience the function creates a local copy of the function at each iteration of the loop, and once again creates individual local copies of certain fields to ensure that they occupy the proper number of bytes in the output file.

The Host API Call Table

Last in line is the host API call table, which is the third and final linked list to write the file.

```
// Write out the call count (4 bytes)
fwrite ( & g_HostAPICallTable.iNodeCount, 4, 1, pExecFile );

// Set the pointer to the head of the list
pNode = g_HostAPICallTable.pHead;

// Loop through each node in the list and write out its string
for ( iCurrNode = 0; iCurrNode < g_HostAPICallTable.iNodeCount; ++ iCurrNode )
{
    // Copy the string pointer and calculate its length
    char * pstrCurrHostAPICall = ( char * ) pNode->pData;
    char cCurrHostAPICallLength = strlen ( pstrCurrHostAPICall );

    // Write the length (1 byte), followed by the string data (N bytes)
    fwrite ( & cCurrHostAPICallLength, 1, 1, pExecFile );
    fwrite ( pstrCurrHostAPICall, strlen ( pstrCurrHostAPICall ), 1,
        pExecFile );

    // Move to the next node
    pNode = pNode->pNext;
}
```

Since the host API call table is really just a glorified string table, the procedure is more or less identical. Also like the string table, the length of each host API call string is calculated just before being written out.

NOTE

As always, check out the source on the CD!

With this table written, the entire .XSE file is complete, along with the rest of the assembly process for that matter! It's been a pretty long road, but at this point we've seen how almost everything works from the loading of the initial source file to the writing of the executable.

The Assembly Process

Now that you've created all of the internal structures you need, and learned how the lexing and parsing phases are used to interpret and translate the source code, let's apply everything to the big picture and see the process of turning a source script into an assembled executable from start to finish. I'm going to move through this part pretty quickly, so make sure you've paid attention throughout the chapter so far and know your stuff.

This section won't really teach you anything new, but it does illustrate how everything you've learned in this chapter fits together and presents it in a fast-paced manner.

Loading the Source File

The first thing XASM does is validate the command-line parameters and filenames. If everything checks out, the source file is opened; otherwise, an error message is printed and the program exits. An initial scan through the file is performed to count the total number of lines it contains. The source code array is then allocated with a number of strings equivalent to the number of lines in the source file, and a simple loop is executed that loads each line of the script file into its corresponding array index. Check out Figure 9.46.

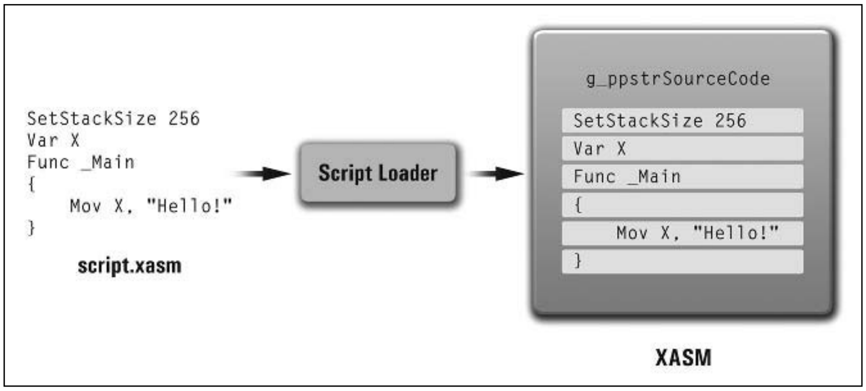

```
SetStackSize 256
Var X
Func _Main
{
    Mov X, "Hello!"
}
```

Figure 9.46

Loading the source file into memory.

The initialization of the program then begins. This is where the master instruction lookup table is initialized. This can either be done in the code itself by an initialization function, or loaded from a file containing a description of the instruction set. The lexer is also reset with a call to `ResetLexer ()`.

NOTE

In the XASM source, you can find both the first and second passes implemented in the `AssmblSourceFile ()` function.

The First Pass

With the source code loaded into memory, the first pass begins. This pass is solely concerned with directives—primarily variables, functions and line labels, although it counts instructions as well. Whether or not the instruction is valid is not checked in this phase.

Variables declared with `Var` can be found both inside and outside of functions. Instances of the directive found outside a function are added to the global symbol table, which also increments the global data size.

Each time a new function is detected, its code is scanned and its local variables and parameters are counted based on the number of `Param` and `Var` directives found within its curly braces (in other words, its scope). This information, along with the function's name and entry point, are saved to the function table. Whenever a function is added to the table, corresponding local symbol and label tables are created as well. Each variable found within the scope is added to the function's local symbol table.

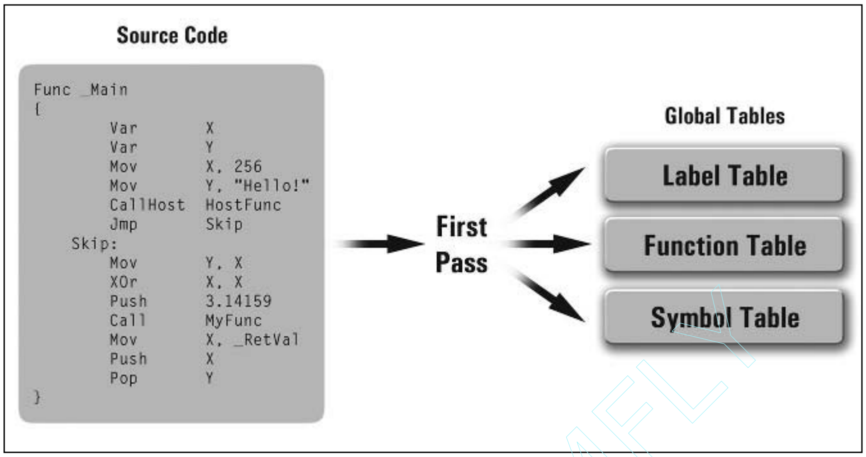
NOTE

Remember, not all assemblers work in two passes. Single-pass assemblers lend themselves well to environments where memory is tight, because they can easily assemble a file sequentially in small chunks. Of course, doing everything in a single pass is considerably more complicated because so much extra information has to be extracted from a single line (since you won't have a chance to see it again). For our purposes, it's not worth the trouble.

Line labels can only be found inside functions, so right off the bat any label declaration encountered in the global scope causes an error and terminates the assembly process. As line labels within functions are found, their names and corresponding instruction indices are written to the function's local label table.

Upon the completion of the first pass, the script's functions, global and local variables, and line labels have been identified and recorded for reference in the second pass. The number of instructions has also been counted. This last piece of information is used to allocate a new array to hold the assembled instruction stream, which will be generated in the second pass.

The first pass is illustrated in Figure 9.47.



```
Func _Main
{
            Var       X
            Var       Y
            Mov       X, 256
            Mov       Y, "Hello!"
            CallHost  HostFunc
            Jmp       Skip
      Skip:
            Mov       Y, X
            XOr       X, X
            Push      3.14159
            Call      MyFunc
            Mov       X, _RetVal
            Push      X
            Pop       Y
}
```

Figure 9.47

The first pass primarily builds up many of XASM's major tables.

The Second Pass

The second pass is responsible for actually assembling the code into an instruction stream capable of being dumped into the executable file. This pass makes heavy use of data collected in the first pass, but, all things considered, is the more vital of the two.

Directives are largely ignored in the second pass, and regardless of function declarations, instructions are almost treated as one contiguous block. In other words, the vertical order in which functions are declared in the file is also the exact order in which the instructions will be found in the assembled executable (see Figure 9.48). The function table is expected to tell the VM where each function's entry point lies, which is why the assembler can collapse the entire script into a single, contiguous stream without worrying about losing track of what code belongs to which function. Among the only real use of directives is tracking the current function to validate the scope of variable and line label references, and handling parameters.

Instructions are read sequentially, and are compared to the master lookup table that contains each instruction's mnemonic, opcode, and operand list. This table gives you the information you need to both assemble and validate the instruction and its operands. Any syntax errors, invalid instructions, or improper operand lists found during this process terminate the assembly process and generate an error message that's displayed for the user.

As each instruction is translated into an opcode and an assembled operand list, the operands are resolved primarily through references to the tables built in the first pass. Parameter, variable, and array references are replaced with their respective absolute or relative stack indices, labels in jump instructions are replaced with instruction indices, and function names in `Call` instructions are replaced with indices into the function table. Any instance of `_RetVal` is also replaced with the

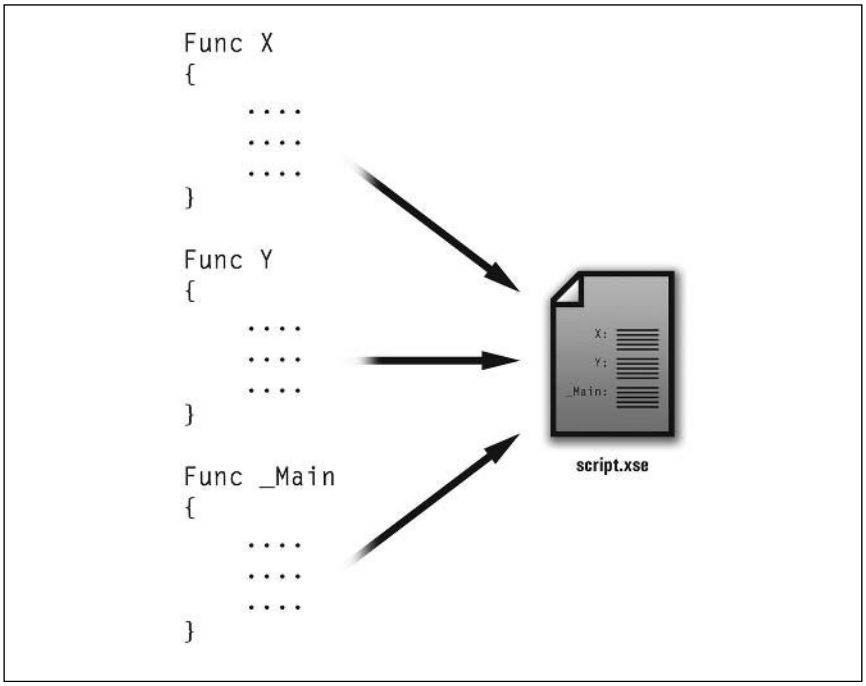


Figure 9.48

The vertical order of functions dictates how the final instruction stream will be ordered.

proper register code. Any reference to a variable, parameter, array, function, or line label that's either not in the current scope or doesn't exist results in an error that terminates the assembly process and is displayed for the user.

That brings you to literal values. Integer and float literals are dumped directly into the instruction stream, whereas strings are identified and added to the string table (note that strings were not collected in the first pass, because that would've involved parsing the instructions in full, which only the second pass is responsible for). The function that adds the string to the table automatically calculates and returns the string's index, which is immediately output to the instruction stream. This allows the conversion of string literal to index to be done quickly and easily.

Lastly, there's the collection of host API calls, which are treated much like string literals in that the string data composing each host API function name is removed from the instruction stream, placed in a separate table, and replaced within the stream as an index into that table.

With the second pass complete, all necessary tables have been filled, and the assembled instruction stream has been generated. The assembled script is complete, albeit in a somewhat disjointed form that resides in memory rather than in a file.

Producing the .XSE

The last step in the assembly process is dumping everything into the executable. This process begins by writing out the main header, including the ID string, major and minor version numbers, requested stack size, and a single integer value representing whether a `_Main ()` method was implemented.

After the main header, the instruction stream is dumped virtually as-is from the global instruction stream array. Followed by the instruction stream is the string table, the function table, and the host API call table. As each table is written to the file, it's prefixed with the proper header data like the number of elements it contains. These structures complete the executable, and leave you with a ready-to-use, assembled XVM script. Check out Figure 9.49.

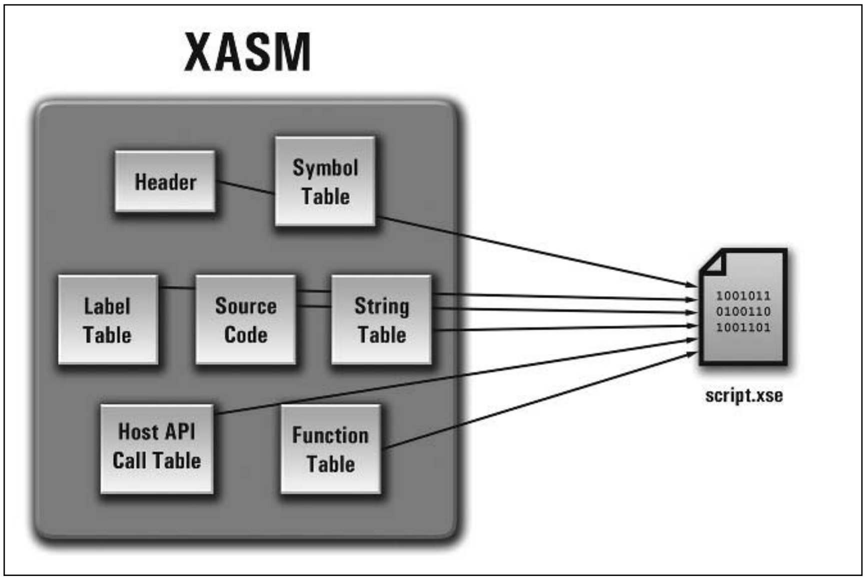


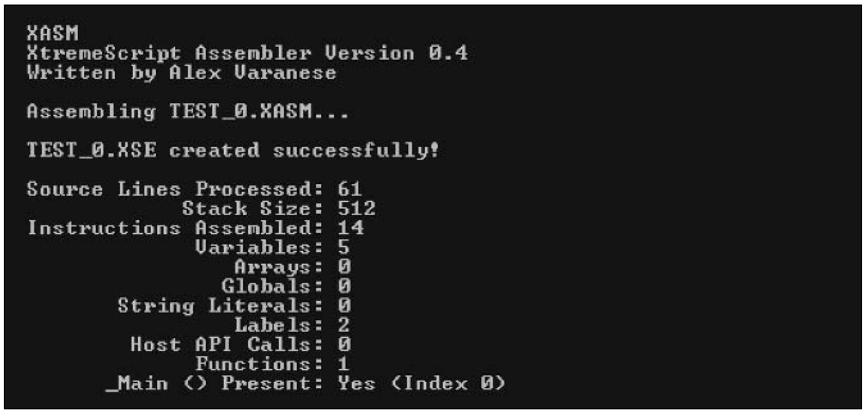
Figure 9.49

The contents of XASM's structures are dumped into the .XSE file like a body into the East River.

The tables are up next: the string table, function table, and host API call table. These can be written to the file almost verbatim.

To finish things up, a small summary of stats collected during the assembly process is displayed for the user (number of lines processed, number of labels, functions and variables, and so on) along with a success message. The output file, either given the same name as the input file or overridden with a user-specified name, can be found in XASM's working directory. Check out Figure 9.50 for a screenshot of these statistics.

The last step involves manually freeing every structure (and nested structure) allocated during the assembly process. Once you've cleaned up, the program can exit and your job is done! Woohoo!



```
XASM
XtremeScript Assembler Version 0.4
Written by Alex Varanese

Assembling TEST_0.XASM...

TEST_0.XSE created successfully!

Source Lines Processed: 61
            Stack Size: 512
Instructions Assembled: 14
             Variables: 5
                Arrays: 0
               Globals: 0
       String Literals: 0
                Labels: 2
         Host API Calls: 0
             Functions: 1
      _Main () Present: Yes (Index 0)
```

Figure 9.50

The statistical summary presented by XASM upon the completion of a successful assembly.

Summary

You've done well, apprentice. Against all odds, you rose to the challenge and took your first *major* step towards attaining scripting mastery by building your own assembler (or, at least, you read about how it's done and hopefully understood it). If you haven't already, I strongly urge you to check out the working XASM implementation on the accompanying CD. Take a look at the source, try assembling some scripts of your own, and, for some real fun, load the resulting .XSE files in a hex editor and see if you can follow the structure.

NOTE
Every time you check the source to XASM, an angel gets its wings!

The assembler is pretty slick, don't you think? You pass it a file containing human-readable code written in its own custom-designed assembly language, and it'll spit out a ready-to-run XVM executable, or print out a reasonably verbose error message explaining what went wrong. How cool is that? Of course, you can't actually *do* anything with the compiled scripts just yet, but the good news is that the next chapter (which begins the next section of the book) will get you started in the construction of the XtremeScript Virtual Machine. By the end of the next section, you'll have both this working assembler, and an embeddable VM that can hold its own against even the existing scripting systems you worked with in Chapter 6. This means that for the first time, you'll be able to do *serious* game scripting with your own home-grown software.

I can't emphasize enough that even without the high-level compiler, the stuff you're building in these chapters alone can be employed as a useful game-scripting system. I don't mean to downplay the importance of the compiler you'll eventually make, of course—that'll still be the coolest part of the whole project by far—but I do want you to understand that you're free to only go as far as you want. If you'd like to jump right into game scripting as soon as the XVM is done and don't mind (or even *enjoy*) coding in assembly, nothing will stop you from immediately putting the system to use. That's why I made sure you designed the language with human coders in mind as well. Remember– the syntax may be a bit funky, but assembly languages can do everything higher level languages can. That means XASM and the XVM alone will be enough to satisfy most, if not all, of your game scripting needs.

On the CD

As I've mentioned numerous times so far, it will be highly beneficial for you to browse the finished, working XASM implementation included on the CD. You can find it in the `Programs/Chapter 9/XASM 0.4/` directory, in both source and executable form.

The program is a simple Win32 console application, so you shouldn't have much trouble compiling it. Simply load the workspace file into Visual C++ and build. For simplicity's sake, and because it really isn't all that big, the entire program is contained in `xasm.cpp`. The source file is highly commented, and I encourage you to try compiling it and even making changes and enhancements. For some specific ideas, try the challenges listed below.

Challenges

- ***Easy:*** Add new instructions to the assembler's vocabulary and compile scripts that use them. Try these for example: `Sqrt` (for computing square roots), `RoL` (for rotating bits to the left) and `RoR` (for rotating bits to the right).
- ***Intermediate:*** Implement the language definition file feature I mentioned in the section on populating the instruction lookup table externally.
- ***Difficult:*** Implement at least a simplified version of the state machine-based lexical analyzer I introduced above. You'll learn how this is done first hand in a few chapters, but it'll be interesting to see how far you can get now.

Part Five

Designing and Implementing a Virtual Machine

CHAPTER 10

Basic VM Design and Implementation

"They're gonna build it."
——Palmer Joss, Contact

XASM is up and running, which means you're now capable of turning XVM Assembly scripts into executables. However, despite your ability to create neat-looking binary files that amaze and confuse your friends, you can't actually do anything with them. Fortunately, this chapter is all about changing that.

An executable produced by the XASM assembler is designed for a runtime environment called the XVM, or *XtremeScript Virtual Machine*. The XVM is designed in many ways to mimic a generic hardware processor, which makes it ideal for executing the sort of code you've just learned to assemble. This chapter is only about the basics, however. You're going to be introduced to the XVM, but won't actually finish it until Chapter 11. Instead, you'll build a small prototype that encapsulates the majority of its overall functionality, but in a simplified way. Don't let your guard down, though—you're still going to cover a lot of important ground in this chapter, including

- How a virtual machine works, and how it fits into the XtremeScript system.
- A detailed structural overview of the XVM prototype's major facilities and structures.
- Step-by-step explanations of how the simplified runtime environment prototype will be built.

This chapter will follow the basic format of the last. Rather than dump page after page of code on you, I'm going to give a detailed tour of how the XVM will be built that incrementally teaches the ins and outs of the machine, with many small code examples. Also, like the last chapter, it's *highly* recommended that you check out the source code to the XtremeScript Virtual Machine on the accompanying CD. This is the best way to solidify the material this chapter teaches.

Ghost in the Virtual Machine

Let's start with an introduction to the theory behind a virtual machine, or VM. A VM is a type of *runtime environment*, which is a piece of software designed to facilitate the execution of some other piece of software or data—usually executable code. Runtime environments come in many forms; for example, 3ds max, a high-end 3D modeling and animation package from Discreet contains a built-in runtime environment for its proprietary scripting language, MAXScript. The Apache Web server can be expanded with runtime environments for a variety of scripting languages, such as Perl and PHP, which can control the server's output for the purpose of generating dynamic responses for HTTP requests. Even Microsoft Word has a built-in runtime environment for its own simple scripting system, WordBASIC (which you can actually write games with!)

The common thread among all of these examples is that without using the hardware processor itself, these pieces of software are capable of executing programs in the form of scripts and providing them with the necessary memory address space and other such facilities. This is exactly what the virtual machine will do. Check out Figure 10.1.

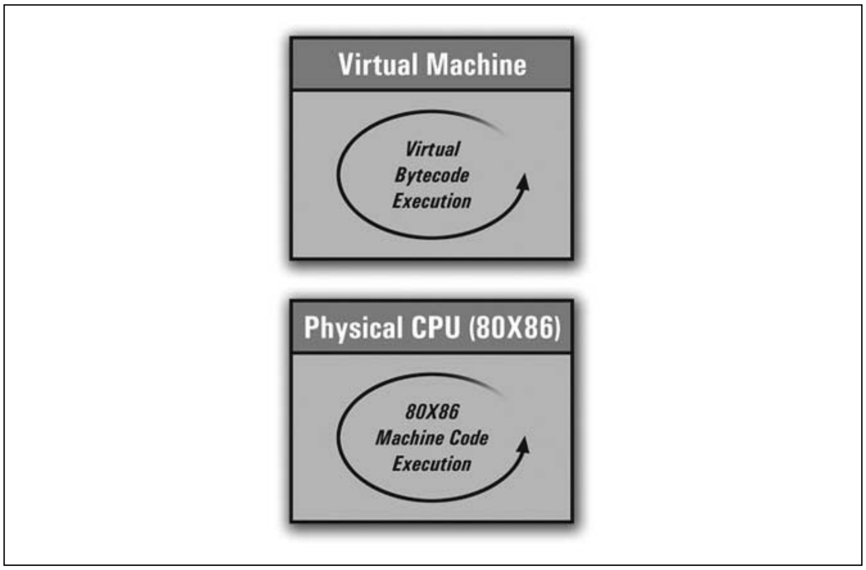


Figure 10.1

The virtual machine is a runtime environment that executes code "above" the physical hardware.

Mimicking Hardware

The distinguishing quality of a virtual machine as opposed to other types of runtime environments is that it's specifically designed to mimic the layout and functionality of a real computer—complete with a virtual processor, virtual memory address space, and even virtual registers in some cases. Just as a real computer streams compiled opcodes through its processor and maintains random-access memory and a runtime stack, so too does the virtual machine. The only difference is that instead of building these components with silicon, you're writing them in C. Check out Figure 10.2 for an example of the VM's layout.

The beauty of the VM approach to a runtime environment is that it automatically comes with countless examples upon which to base your design strategies. Computer architecture has been a rapidly developing field for decades, which means you can leverage the diligent work of thousands of engineers who've found out exactly what works and what doesn't work when implementing a computing system. You can directly apply much of this hard-earned knowledge and perspective in the hopes of quickly building a robust and efficient system for executing code.

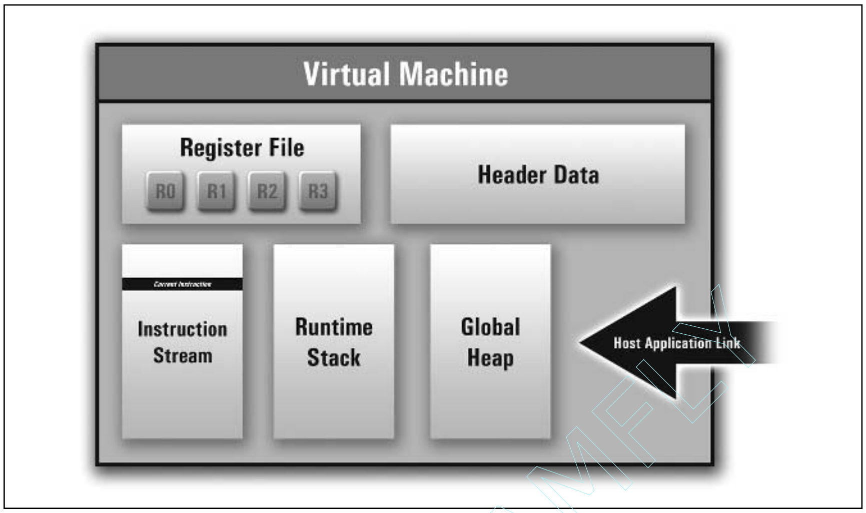


Figure 10.2

A basic virtual machine's architecture.

But much like *Blade*, with his combination of human and vampire blood, your VM will enjoy most of the strengths and few of the weaknesses of a real computing system. On the one hand, you can take advantage of the tried-and-true architecture that already runs so well on real hardware. On the other hand, however, you can discard many of the low-level complexities of real hardware and replace them with high-level abstractions that both enhance the system's ease of use and reduce its tendency for errors. For example, unlike nearly all real hardware, this VM is typeless, allowing you to focus on the logic of your scripts without worrying about data types and compatibility.

Of course, you can't forget the one major weakness of any scripting system—the significant speed overhead. Remember, every instruction that the runtime environment processes will in turn require a much larger number of native instructions to be executed by the actual hardware. For example, a `Mov` instruction running inside your VM will take considerably longer to execute than a `Mov` instruction executed by the physical CPU itself. Scripting can make designing and structuring a large game project orders of magnitude easier and more robust, but it does come at a performance price that shouldn't be taken lightly.

The VM's Major Components

A virtual machine can be thought of primarily as a collection of large, interconnected components. Let's take a brief look at each of the major data structures a virtual machine must maintain in order to execute a script.

The Instruction Stream

The first and most obvious, of course, is the instruction stream— an array of compiled opcodes and operands that describes the logic of the script. The instruction stream embodies the script's runtime activity— as execution progresses, the script's opcodes determine exactly what will happen. Figure 10.3 illustrates the instruction stream.

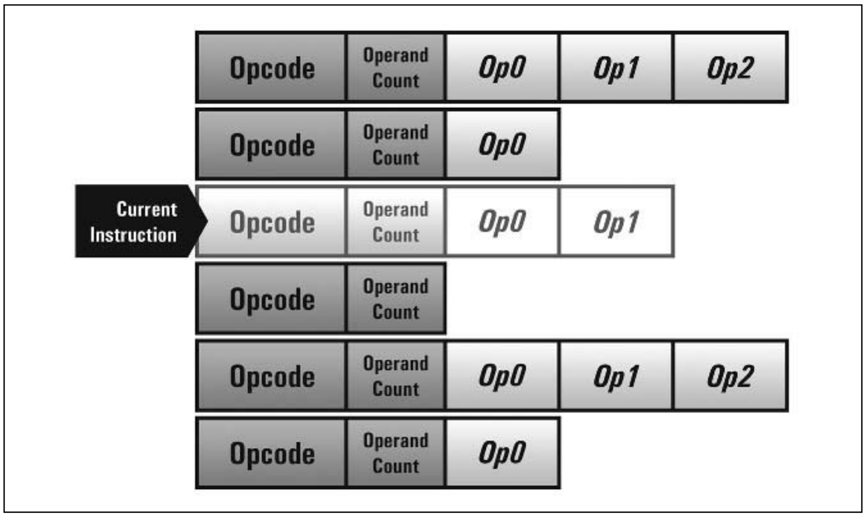


Figure 10.3

The instruction stream.

The Runtime Stack

Another highly dynamic structure is the runtime stack, which is both read from and written to by the instruction stream as the script executes. It grows, it shrinks, its values are in a constant state of change, and thus, the stack is among the most vital components the virtual machine maintains. Without it, function calls and complex expressions would be nearly impossible to implement. Figure 10.4 illustrates the runtime stack.

Global Data Tables

Following these two major structures are the global data tables; namely, tables containing profiles of each function and the host API call table, which consists of strings containing host API function names. These tables are also read from and written to by instructions, and are heavily referenced by the stack. Figure 10.5 illustrates these tables.

Together, these components comprise just about everything you'll need to describe and encapsulate a single script. If, for example, two scripts were loaded at one time, you'd need two instruction streams, two runtime stacks, and two copies of each global data table. These structures are generally not shared; rather, scripts exist within their own self-contained universe, which makes conceptualization and implementation of the system easier, cleaner, and safer. It strongly reduces the possibility of errors and the general corruption of data by buggy script code, because it simulates the memory protected address spaces offered by operating systems like Microsoft Windows and Linux.

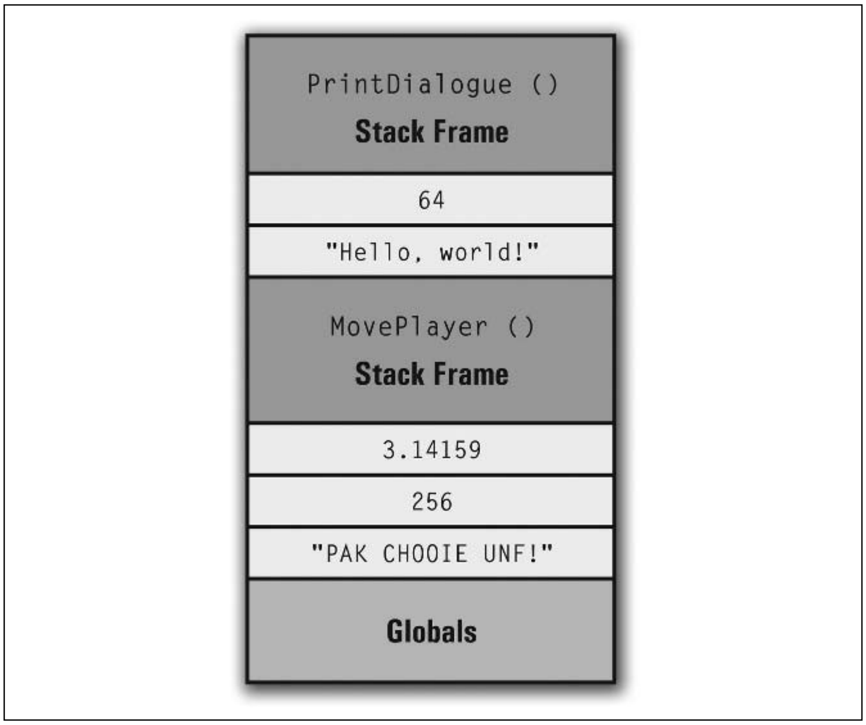
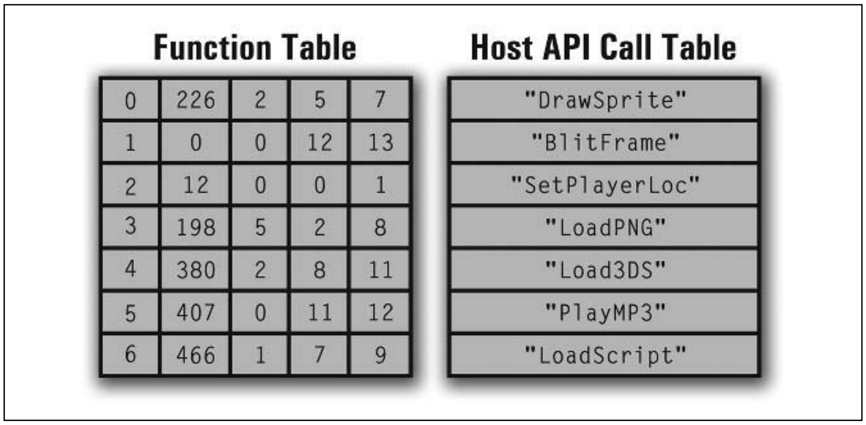


Figure 10.4

A general memory map of the VM's runtime stack. Globals always start at the base, followed by function stack frames. In between frames may exist 0-N elements pushed on by code using the `Push` *and* `Pop` *instructions.*



Function Table

0	226	2	5	7
1	0	0	12	13
2	12	0	0	1
3	198	5	2	8
4	380	2	8	11
5	407	0	11	12
6	466	1	7	9

Host API Call Table

"DrawSprite"
"BlitFrame"
"SetPlayerLoc"
"LoadPNG"
"Load3DS"
"PlayMP3"
"LoadScript"

Figure 10.5

Global data tables.

Multithreading

Especially in the context of game scripting, it's extremely important that a VM support multithreading to allow the concurrent execution of multiple scripts. If each enemy on the screen is controlled by a separate script, and the level environment is scripted as well, it's obvious that all of these entities must be able to execute at once without stepping on each other's toes. Just as any decent modern operating system supports multitasking, a VM should be strongly multithreaded. See Figure 10.6.

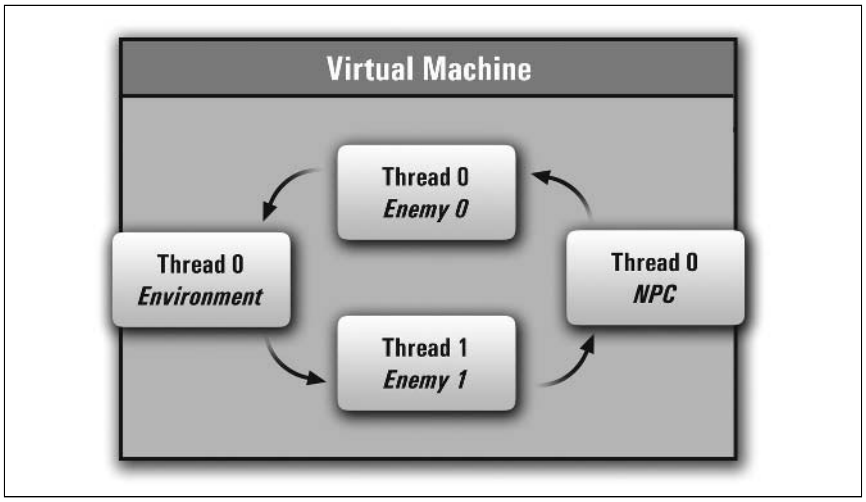


Figure 10.6

A multithreaded VM can run multiple scripts concurrently.

As mentioned, multiple scripts can be loaded into memory at once by duplicating the structures that are used to describe a single script. This usually means that everything a script needs to run—the instruction stream, runtime stack, global data tables, and other miscellanies—is wrapped up into a single, high-level structure. Each thread of execution in the VM can then simply be described by these high-level structures.

I'll discuss multithreading in more detail in the next chapter.

Integration with the Host Application

Of course, no matter how long the feature list of the VM gets, none of it matters if you can't communicate with the host application. After all, the whole purpose of game scripting in the first place is to control a game engine with external script code, so an interface between scripts and the host is of the utmost importance.

As you saw in Chapter 6, this usually comes down to a translation mechanism that can facilitate intra-language function calls—in other words, an abstraction layer that lets the host call script functions, and vice versa, without either side knowing the details of the other. See Figure 10.7.

Like multithreading, I'll also discuss the host/script interface in the next chapter.

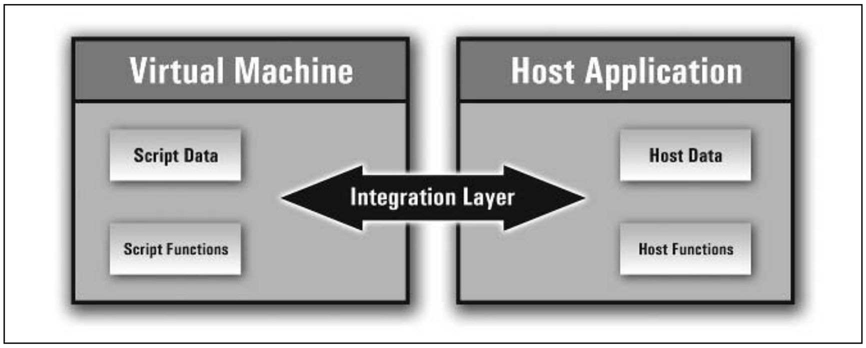


Figure 10.7

The VM allows communication between the script and the host.

A Brief Overview of a VM's Lifecycle

A VM operates much like many other types of programs. It opens a source file, reads in the data, processes that data in some way, and frees its resources before exiting. In this case, the data file is a compiled script, and the "processing" is the execution of its code.

The lifecycle of a VM can be broken into a number of discreet phases. Let's have a look:

- Loading the script and initializing the major data structures.
- Locating the script's entry point and beginning the execution cycle.
- Perpetuating the execution cycle by processing the next instruction in the stream.
- Terminating execution and freeing major data structures.

Nothing too surprising, I hope. Let's dig a little deeper and explore each of these phases in a bit more depth.

Loading the Script

Before anything can happen, a script has to be loaded into memory. This involves locating the file on the disk, reading its contents into memory, and distributing this data among the major data structures.

This process starts by reading the script's header data. In the case of your predefined .XSE executable format, you begin by reading the four-byte ID string and comparing it to `"XSE0"`. This is done to ensure that the file in question is indeed a valid XVM executable. Once the ID string is validated, you can proceed to read out the version number, which lets you know how to process the file specifically. This version information lets you know if the rest of the format should be read and/or executed differently than others. After the version information is confirmed the rest of the header is read—general information about the rest of the script, such as the presence of the `Main ()` function and the stack size, among other things.

With the header read, you're ready to get into the real guts of the executable. You move on to the instruction stream next, which is almost the exact reverse of the process used by XASM to initially dump its assembled code into the file. The VM's instruction stream is composed of the same hierarchical structure—wherein the instruction is the highest level, composed of the opcode, operand count and operand list, which is in turn composed of individual operands defined by a type and data field. The data in the file is loaded directly from the disk into this structure.

With the instruction in memory, a stack is then allocated to the size specified by the executable's header.

This takes care of your two major runtime structures, so you can move on to the global data tables. The string tables and host API call tables are read in similar ways; the string data is loaded into memory and stuffed into a string array and then dispersed throughout the instruction stream's operands. The host API call table is simply loaded into a table and left alone. The function table is loaded a bit more carefully, as it must be loaded into an array of function-defining structures. When this phase is over with, the entire script has been read into memory and is ready for execution. Figure 10.8 illustrates the loading of an executable script.

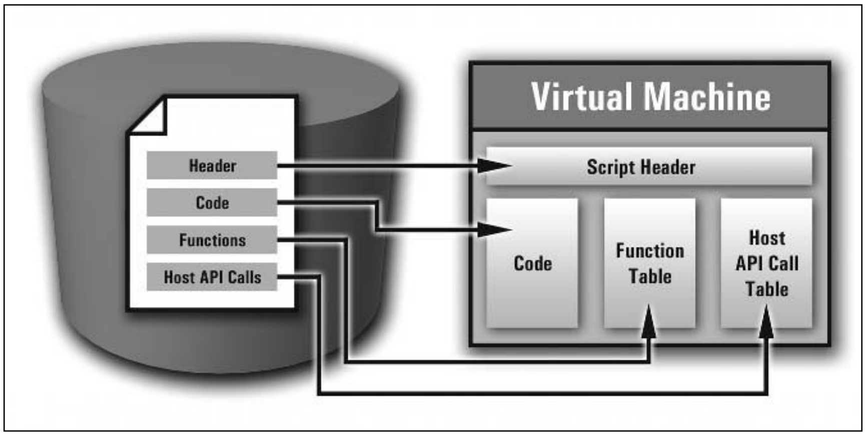


Figure 10.8

Loading an executable.

Beginning Execution at the Entry Point

Every script with a `_Main ()` function has an entry point by nature, whereas those without `_Main ()` do not. This term refers to the first instruction of `_Main ()`, which is where the automatic execution of the script begins. Not every script needs an entry point. In the case of these scripts, execution doesn't begin until a specific function is called.

No matter how execution begins, there is an entry point involved somehow. It's either the entry point of the `_Main ()` function, or that of the one specified by the host in the form of a manual function call. This entry point is used to initialize an *instruction pointer,* which is how you keep track of the currently executing instruction. Once the instruction is executed, the instruction pointer is incremented to point to the next in the stream, and the process continues. This is how scripts are executed in a sequential fashion. Of course, the jump instruction family, as well as `Call`, can be used to cause the pointer to jump around the script non-sequentially, thus enabling conditional logic, iteration, and function calls.

NOTE

The instruction pointer is often referred to simply as *IP*, mainly because `IP` is the name of the 80X86 register used for tracking the current instruction. A common synonym for "instruction pointer" is *program counter*, or *PC*. The latter terminology is used in the Java Virtual Machine, for example.

The Execution Cycle

Once the script is running, either automatically because of the presence of `_Main ()`, or manually through a function call from the host, the execution of the instruction stream begins. At each iteration of the virtual machine's main loop, the current instruction is found by indexing into the instruction stream with the instruction pointer. The instruction at this index is then executed.

The processing of an instruction may seem simple at first, but just as assembling the instruction stream proved complicated, so will its execution. Most instructions are implemented using the same multi-phase process, which is described here:

- **Opcode Identification.** The opcode is first read from the stream, which lets you know which instruction you need to execute. This value can be used as the criteria for a `switch` block, where each `case` implements its own instruction, or perhaps as the index into an array of function pointers, wherein the instructions are implemented as separate functions. Regardless of the implementation, however, interpreting the opcode is the first step.
- **Operand Resolution.** An instruction's operands are necessary to guide its behavior, so you need to read them from the stream next. Reading the instruction's operand list from the stream in its entirety is a rather involved process, which makes this phase one of the most complex in the overall execution of an instruction. For example, because this

language is largely typeless, an `Add` instruction may be required to "add" an integer to a string, because of the data types of the operands. Because of cases like this, the first step in dealing with operands is converting them to a common type. Because the integer and string values can't *actually* be added, you'll need to temporarily cast the string to an integer. Furthermore, operands aren't always immediate values; more often than not an instruction will be presented with variables and array indices, which point to offsets within the stack. This means you also have to locate these values and store them locally before they can be processed. Overall, this process of identifying, locating, and converting operand values is called *operand resolution*, because the operand is *resolved* from a possibly disjointed or scattered form to a much simpler one.

- **Instruction Execution.** Once your operand data is locally stored and ready to go, you can execute the actual instruction's logic. This might mean adding two integers, extracting a character from a string, making a function call, or whatever. Although this is definitely the most important phase of an instruction, it's usually one of the easier to implement.
- **Store Results.** Many instructions produce some sort of results; perhaps most obviously, instructions like `Mov` and the arithmetic family are designed to change the values of their destination operands. This means that the last phase of the execution process is storing the results of the instruction's operation in the specified destinations (usually a stack index).

As is the case with most aspects of computer science, the actual implementation of something that may have initially seemed trivial is, in fact, rather complex. Remember, you may be executing thousands of instructions per second as your script flies through the VM, but each time one of those instructions is processed, this entire process must be completed. Check out Figure 10.9 for a more visual idea of the execution cycle.

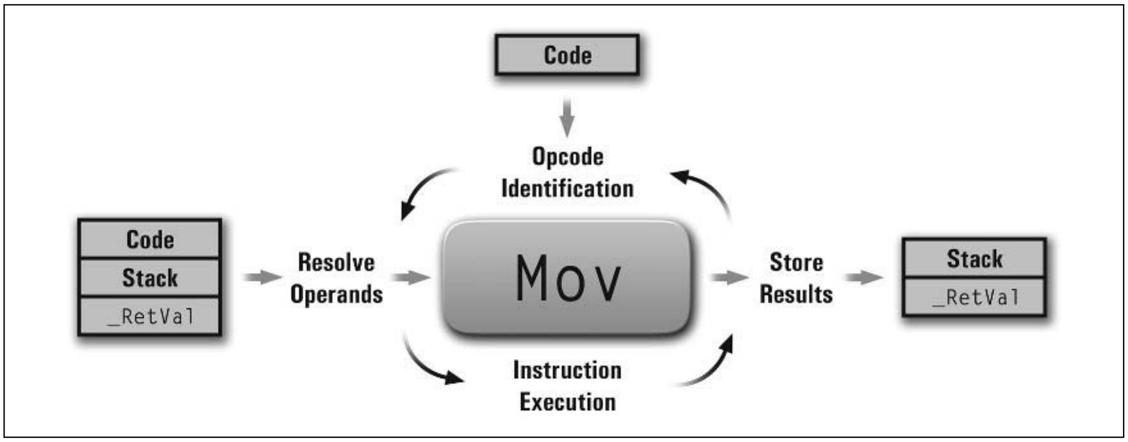


Figure 10.9

The execution cycle.

Function Calls

One major aspect of a script's runtime behavior is the calling of and returning from functions. Naturally, since the XtremeScript system is based around a procedural language, a reliable method of handling function calls is crucial. Up until now, we've learned quite a bit about stack frames, how functions are described and stored in the .XSE executable, and other issues. Now, let's take a general look at how the XVM specifically handles function calls.

Calling a Function

The first step in calling a function is getting its information from the function table. The return address is then pushed onto the stack, followed by the stack frame (whose size is equal to the function's local data). Figure 10.10 illustrates this.

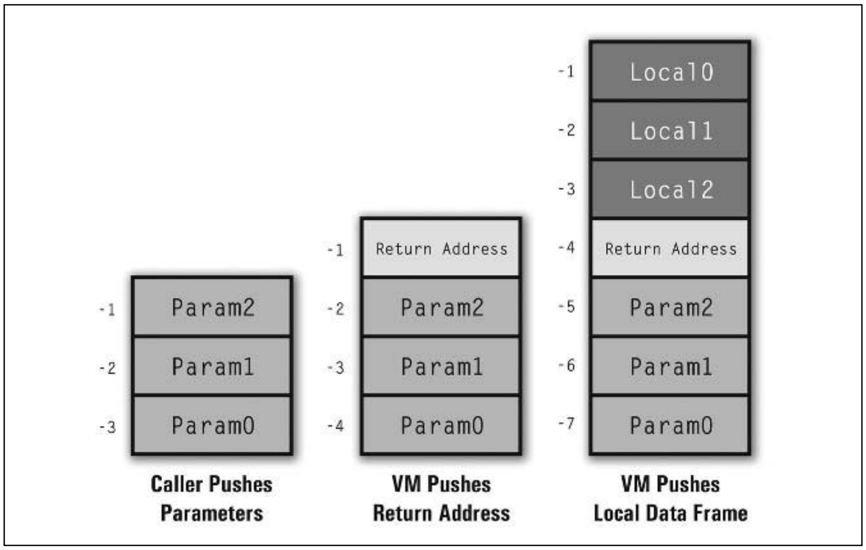


Figure 10.10

A basic function call procedure.

Two problems exist with this method, however. Firstly, remember how a function returns– the `Ret` instruction reads the return address from the stack, clears off the stack frame, and makes a jump back to the caller. The problem is, the return address is buried *N* elements deep into the stack, where *N* is the size of the function's local data. Therefore, the address at which the return address resides is the top of the current stack frame minus *N*. `Ret` can get the index of the current stack frame, but where's it going to get *N*? The only way to get the value of *N* (the function's local data size) is to somehow get the function's information from the function table. The problem is, `Ret` would have no way of knowing which function it is, thereby making the return address unreachable. Check out Figure 10.11.

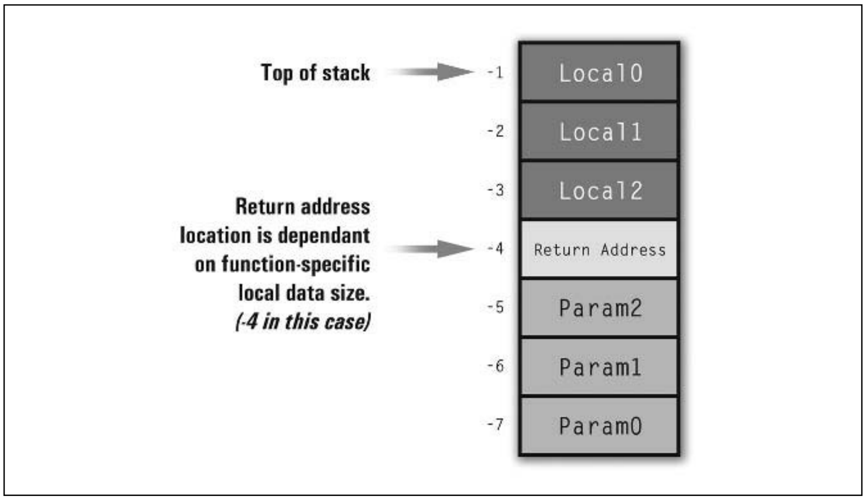

Top of stack
Return address location is dependant on function-specific local data size. (-4 in this case)
-1
-2
-3
-4
-5
-6
-7
Local0
Local1
Local2
Return Address
Param2
Param1
Param0

Figure 10.11

`Ret` can't reach the return address because it doesn't know how far down into the stack it is.

So, we can solve this problem by pushing another stack element on `after` the stack frame. This element will have the index of the function to which the frame belongs written to its `iFuncIndex` field, which means that all `Ret` has to do is read the element at the top of the current stack frame, grab the value of its `iFuncIndex` field, and use that to get the function's `Func` structure from the function table. Once it has this structure, it can determine the size of the function's local data and locate the return address. This also lets it know how large of a frame to pop off the stack. This finally explains what we need that extra element on top of the stack frame for, and in turn explains why local data always starts at -2 rather than -1.

Secondly, when popping the stack frame, the stack structure's `iFrameIndex` pointer has to be updated to point to the location of the previous stack frame. We could assume that after popping the current frame, the new top of the stack will be equal to the top of the last function's stack frame, and this may be correct most of the time. However, if that function's code used the `Push` instruction to push anything onto the stack, and called the current function before `Pop`ping them off, the stack frame will actually reside *N* number of elements *below* the new top of the stack.

The easiest way to resolve this issue is to simply save the current value of `iFrameIndex` on the stack as well before calling the function. That way, `Ret` can be sure that it's restoring the old frame index exactly as it was, and none of the data the function may have pushed onto the stack will be disturbed. And the best part is, we already have a place to store this value– we can just use one of the other fields in the stack element we pushed on to hold the function's index. Of course, we have to be careful not to use one of the other `union` fields, because that would overwrite the function index. Rather, we'll use the `iOffsetIndex` field since that resides outside of the `union` and won't corrupt `iFuncIndex`. This way, this single element stores both the function index of the previous function, and the top of that function's stack frame. This process is illustrated in Figure 10.12.

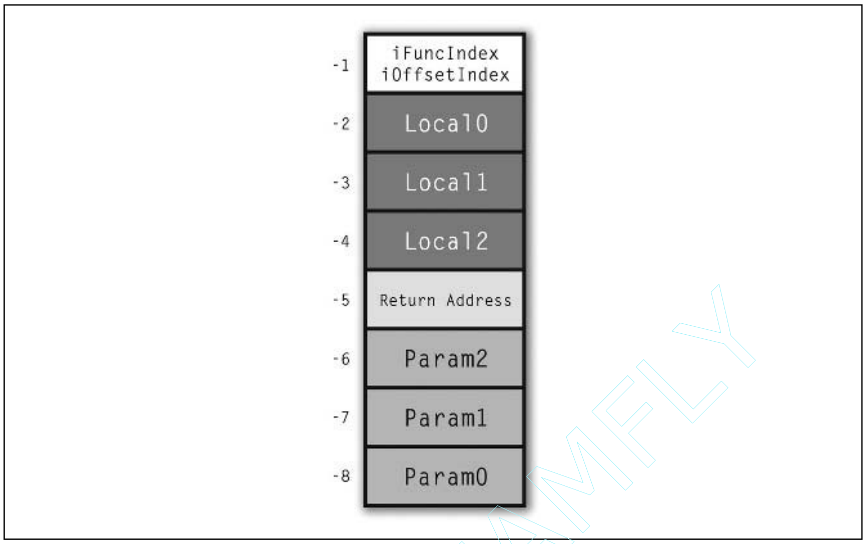


Figure 10.12

Saving the frame index and function index before calling the new function.

We now have all the information we need to safely call a function, so let's review. When calling a function:

- The function's information is retrieved in the form of a `Func` structure from the function table.
- The return address is pushed onto the stack.
- The stack frame is pushed. The size of this frame is large enough to hold the function's local data, as well as one extra element to hold the information `Ret` will need to properly restore the previous function.
- The top element of the stack is filled with two values: `iFuncIndex` is set to the function index of the new function, and `iOffsetIndex` is set to the top index of the previous stack frame.

Returning From a Function

The explanation of how a function is called overlapped pretty heavily with how a function returns, so this will be quick. To return from a function, the top of the stack is popped off. This element contains both the index of the function we're returning from, as well as the location of the previous stack frame. The first of these two pieces of information is used to retrieve the current function's `Func` structure from the function table.

The size of the function's local data is subtracted from the location of the current stack frame to calculate the location of the return address. This is illustrated in Figure 10.13.

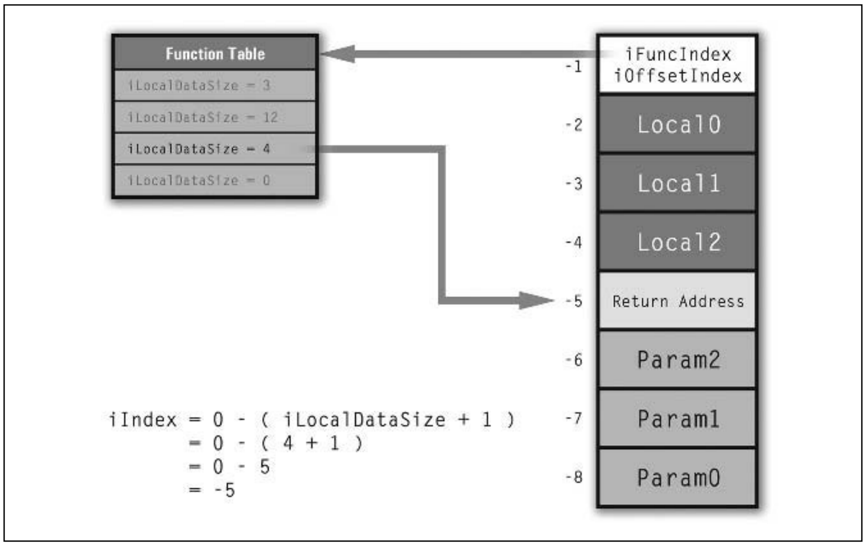


Figure 10.13

Locating the return address on the stack.

With the return address saved, the entire stack frame—meaning the function's local data, return address, and parameters—is popped off. The function's stack frame is now entirely removed, so the stack structure's `iTopIndex` and `iFrameIndex` values are updated. With the stack in the state it was in before the function was called, an unconditional jump is made to the return address, routing the flow of execution back to the caller. The stack is restored, the caller has control of the script again, and the process is complete.

Sounds good, huh? We'll come back to all this when we implement the `Call` and `Ret` instructions, but we've pretty much got it all here.

Termination and Shut Down

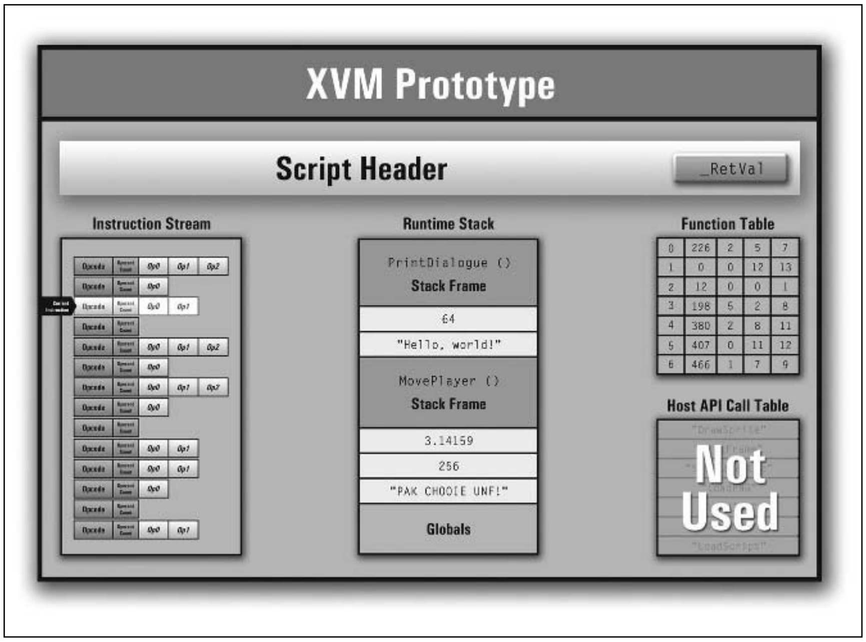
Like all good programs, your VM has to play nice with its operating environment and properly clean up after itself. A script can terminate for a number of reasons, ranging from the last instruction being reached to the game engine sending a specific request to shut down. In both cases, major structures like the instruction stream, stack, and all global data tables must be freed. This of course is one of the easier phases of the VM's lifecycle, but it's extremely important. Remember that a real-world game may load, run, and terminate thousands of scripts as it progresses, which means you can easily clog up the system's resources if each one of these aren't properly removed.

Structural Overview of the XVM Prototype

A VM's structure is extremely important. Because scripting is already daunted by a considerable performance overhead, you should do all you can to design your runtime environment to minimize bottlenecks and maximize efficiency. You've already taken a brief tour of the virtual machine's major components, so let's take a deeper look and explore exactly how they're implemented.

We're now going to examine each major structure that the VM must keep track of in order to encapsulate a single script. As you read, note the similarities between these and the structure of the .XSE file. Also, as you'll see more clearly as the chapter progresses, what I'm describing here is only a prototype version of the XVM. The actual XVM will be considerably more powerful than what's described in this chapter, so its data structures will differ somewhat. However, what you're learning here is mostly a subset of what you'll find in the final XVM, so it's nonetheless important to understand. Your future XVM development will be based on the foundation this information will be used to create.

Figure 10.14 illustrates the final overview of the XVM prototype, which will be explained in more detail in the following subsections.



0	226	2	5	7
1	0	0	12	13
2	12	0	0	1
3	198	5	2	8
4	380	2	8	11
5	407	0	11	12
6	466	1	7	9

Figure 10.14

The architecture of the XVM prototype.

The Script Header

Just as an executable file maintains a script header area, a script's representation in memory will involve a header-like structure that manages miscellaneous high-level attributes. Here's a list of what a script in the XVM prototype will need to properly maintain itself:

- **A Pause Flag.** The `Pause` instruction can be used at any time to temporarily halt the execution of the script, which means you'll need to maintain a flag that tells you, at each iteration of the VM's main loop, whether or not the script should continue executing.
- **The Pause End Time.** Of course, a simple flag isn't enough to implement the `Pause` instruction, because you'd have no idea when the script should resume execution. This is why you also need to maintain a timestamp that can be repeatedly compared to the current time in order to determine whether the pause time has elapsed. This value will always be based on `Pause`'s duration operand.
- **The Presence of `_Main ()`.** Self-explanatory; whether or not the script defines a `_Main ()` function.
- **`_Main ()`'s Function Index.** In addition to knowing whether or not a `_Main ()` function is present, we need to know where it is in the function table.
- **Global Data Size.** Especially during initialization, it's important to know how large the script's global data is. Remember, global data is always stored at the bottom of the stack, which means that all other data on the stack will be stored relative to the end of the global data's block.
- **The `_RetVal` Register.** Because `_RetVal` is global to the entire script, it should also be global within the VM. Let's set aside a special structure within the header specifically for holding its current value.

Runtime Values

Because this language is typeless, you can't just use the built-in C primitive types like `int`, `float` and `char *` to represent your script's data. Instead, even single values must be wrapped in larger structures to allow those values to change from one type to another without the need to reallocate anything. Both immediate operands and the contents of the stack are instances of structures I call *runtime values.*

A runtime value is the term I use to describe any value that exists within the script at runtime; this may be an immediate operand value in the instruction stream, or the value residing in the stack. All of these values are typeless, which means they need the ability to switch from integer to floating-point to string and so on, whenever necessary. This is implemented with a simple `union`, just as you did in XASM. Check it out:

```
typedef struct _Value                   // A runtime value
{
    int iType;                          // Type
    union                               // The value
    {
        int iIntLiteral;                // Integer literal
        float fFloatLiteral;            // Float literal
        char * pstrStringLiteral;       // String literal
        int iStackIndex;                // Stack Index
        int iInstrIndex;                // Instruction index
        int iFuncIndex;                 // Function index
        int iHostAPICallIndex;          // Host API Call index
        int iReg;                       // Register code
    };
    int iOffsetIndex;                   // Index of the offset
}
   Value;
```

The `Value` structure will be the basis for virtually all of the script's data storage.

The Instruction Stream

The structure of the instruction stream within an .XSE executable is rather complex, and its run-time representation is no different. It more or less follows the same structure you created in XASM for holding the instruction stream as it was assembled. Regardless, I'll recap it quickly.

The first aspect of the structure is the instructions themselves, of which a global array is allocated to fit the size of script. Instructions are represented with the `Instr` structure, which looks like this:

```
typedef struct _Instr                     // An instruction
{
    int iOpcode;                          // The opcode
    int iOpCount;                         // The number of operands
    Value * pOpList;                      // The operand list
}
    Instr;
```

The structure contains the code, the operand count, and a pointer to the list of operands. The operand list is now represented with the `Value` structure, which you'll see more of shortly.

You also need to maintain the number of instructions in the stream, so you wrap it in a larger structure. Here's the final instruction stream; note that the script's instruction pointer resides here as well:

```
typedef struct _InstrStream                 // An instruction stream
{
    Instr * pInstrs;                       // The instructions themselves
    int iSize;                             // The number of instructions in the
                                           // stream
    int iCurrInstr;                        // The instruction pointer
}
    InstrStream;
```

Figure 10.15 illustrates the instruction stream.

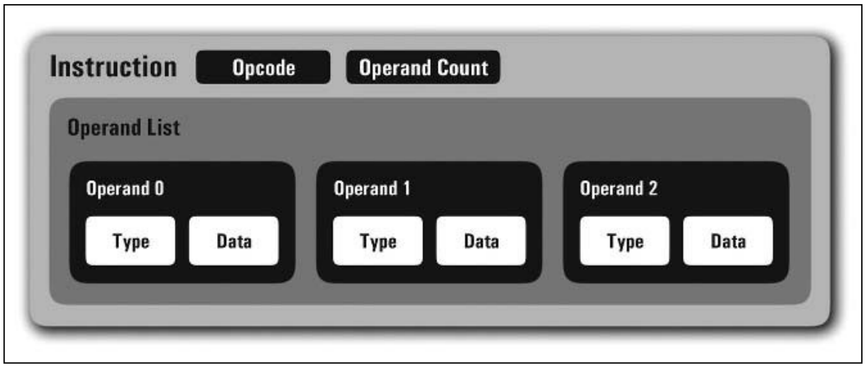


Figure 10.15

An instruction in memory.

The Runtime Stack

The stack is one of the simpler structures your runtime environment will require, as it's really just a dynamically allocated array of runtime values. Each element of the array is a stack element, which makes things rather simple.

Of course, the stack doesn't actually grow and shrink at runtime. Although a truly dynamic runtime stack would make the issue of stack overflow nearly non-existent (as long as system memory holds out, that is), it'd ultimately bring with it a huge performance overhead. Remember that the stack will have to grow literally every time a function is called, and shrink every time a function returns. Because this may happen tens, hundreds, or even thousands of times *per frame* in a game, dynamically allocating even part of the stack would be yet another case of *frame rate homicide.*

Of course, you don't have to worry about this, because it's up to the script itself to provide the ideal stack size. You just give the script the amount it asks for and assume it knows what it's doing. This means you only have to allocate the space once at script load-time, eliminating the performance penalty.

Ultimately, the stack is just an array of runtime values. However, because it doesn't have the ability to physically grow or shrink as the script executes, you must augment this otherwise simple structure with an extra data member— a simple integer value that tracks the current top index. This value will initially be set to zero, as the stack will start off empty. As functions are called and values are pushed on, however, this number will be incremented by the appropriate amount. Likewise, when values are popped off, the value will decrease.

So, the final stack structure contains an array of `Values` and two integer fields (you'll also need to keep track of the stack's size):

```
typedef struct _RuntimeStack                // A runtime stack
{
    Value * pElmnts;                        // The stack elements
    int iSize;                              // The number of elements in the stack
    int iTopIndex;                          // The top index
    int iFrameIndex;                        // Index of the top of the current
                                            // stack frame.
}
    RuntimeStack;
```

The Frame Index

You may be wondering what the `iFrameIndex` field is all about– why do we need to keep track of the top of the current stack frame? To answer this question, consider the following example. Imagine that a function is called, which causes its frame to be pushed on to the stop of the stack. When a variable is manipulated that resides on a stack, say as the result of a `Mov` instruction, the address of that variable will be relative to the top of that function's frame. As we'll see shortly, these addresses always begin at -2 and work their way down from there, which is why local data is always addressed in relative terms.

Now imagine that a `Push` instruction is executed, which pushes a new element onto the stack. -2, relative to the *top* of the stack, is no longer equal to -2 relative to the current *stack frame*. A variable that was located at index -2 before the push is now relative to -3 because of the extra element on top. This is why, even though we conceptually think of negative indices being relative to the top of the stack itself, they're *actually* relative to the top of the current stack frame. Therefore, if `iFrameIndex` is updated each time a new stack frame is pushed, and all negative stack indices are calculated relative to this value, the function can push and pop all it wants and never disturb the locations of its local data.

The Function Table

Fortunately, the function table marks the first of the easy structures. The function table never changes during the execution of the script, which means you can allocate it once at the time the script is loaded and can forget about it. A script won't somehow add, remove, or change its functions, so once it's initialized, the table is good to go throughout the script's lifespan.

The XVM will once again borrow from XASM in its representation of functions. Fortunately, however, the runtime environment only needs a static function table. As a result, you no longer need the `FuncNode` structure, but rather a subset of that structure with the linked-list capabilities removed. Here's the `Func` structure:

```
typedef struct _Func                   // Function table element
{
    int iEntryPoint;                   // The entry point
    int iParamCount;                   // Number of parameters to expect
    int iLocalDataSize;                // Total size of all local data
    int iStackFrameSize;               // Total size of the stack frame
}
    Func;
```

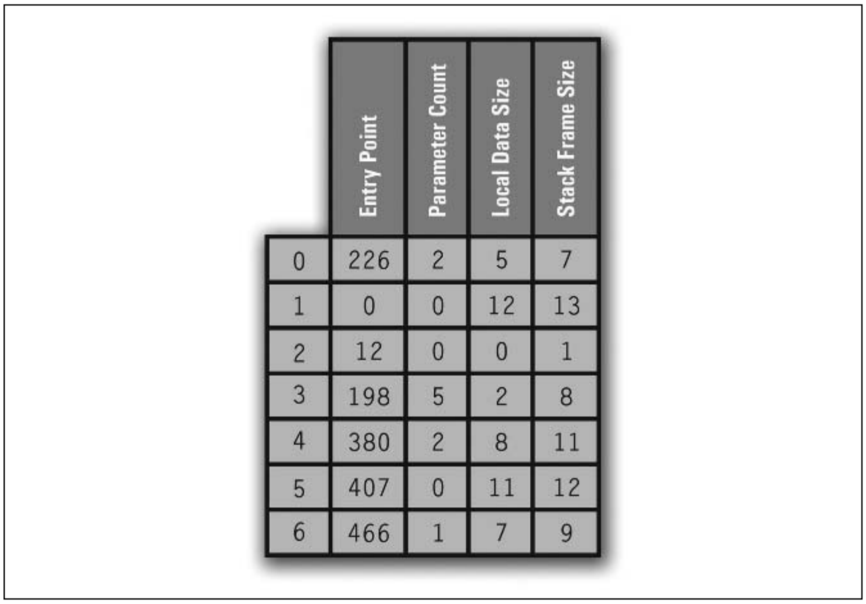
Pretty simple. Notice again that even though the `StackFrameSize` element is always defined as `ParamCount + 1 + LocalDataSize`, you keep it here anyway so you can compute the final stack size at load-time rather than doing it every time a function is called. Given the frequency at which functions will be invoked when scripts are running in an actual game, it's a good idea to have the stack frame size worked out beforehand.

Because you have to allocate the function table only once, there's no need to wrap the `Func` array in a larger structure. Figure 10.16 illustrates the function table.

The Host API Call Table

The host API call table is reasonably simple in that all it really needs to manage is an array of strings. Of course, when we shut everything down, we'll need to know how big this array is in order to free it properly, so the table boils down to a two-field structure:

```
typedef struct _HostAPICallTable        // A host API call table
{
    char ** ppstrCalls;                 // Pointer to the call array
    int iSize;                          // The number of calls in the array
}
    HostAPICallTable;
```



	Entry Point	Parameter Count	Local Data Size	Stack Frame Size
0	226	2	5	7
1	0	0	12	13
2	12	0	0	1
3	198	5	2	8
4	380	2	8	11
5	407	0	11	12
6	466	1	7	9

Figure 10.16

The function table.

The Final Script Structure

All of these structures I've discussed are brought together to describe the script as a whole. It's therefore convenient to wrap them into a single main structure that allows you to refer to each of the script's elements relative to a common name. This structure is simply called `Script`, and looks like this:

```
typedef struct _Script                      // Encapsulates a full script
{
    // Header data
    int iGlobalDataSize;                    // The size of the script's global
                                            // data
    int iIsMainFuncPresent;                 // Is _Main () present?
    int iMainFuncIndex;                     // _Main ()'s function index
    int iIsPaused;                          // Is the script currently paused?
    int iPauseEndTime;                      // If so, when should it resume?

    // Register file
    Value _RetVal;                          // The _RetVal register

    // Script data
    InstrStream InstrStream;                // The instruction stream
    RuntimeStack Stack;                   // The runtime stack
    Func * pFuncTable;                    // The function table
    HostAPICallTable HostAPICallTable;    // The host API call table
}
```

For now, this is just an easy way to refer to your single script, but as you'll see in the next chapter, wrapping everything like this makes multithreading much easier. For the purpose of the following sections, let's assume you declare a global script instance like this:

```
Script g_Script;
```

From here on out, `g_Script` will be the focus of all your script-manipulation tasks.

Building the XVM Prototype

With the structural overview of the XVM over with, you have enough information to start building this thing. Of course, you don't know all of the details of how it'll actually run once the script starts executing, but you can figure that out along the way.

So what exactly is this "XVM prototype" I keep mentioning? Well, to put it simply, it's a command-line application that loads a single script, prints some basic statistics, and then executes the file and prints out the instructions as they're processed. Assuming the script employs some sort of main loop, this output should continue indefinitely until a key is pressed.

Sure, it's not exactly mind blowing, but trust me–it'll be cool when you see your first batch of instructions come scrolling down the screen. What's important, though, is that this lets you develop the core of the XVM without getting too bogged down with other details. You won't have to worry about a host application or multithreading; all you need to worry about is getting the instructions to execute and properly manipulate the VM's structures.

Before getting started, let's run down the major phases of the VM just one more time:

- The script is loaded and its contents are used to initialize the script structure.
- The entry point of the `_Main ()` function is found and the execution cycle begins.
- Execution terminates when a key is pressed, at which point all major structures are freed and the program shuts down.

Notice the second bullet point states that execution will begin in `_Main ()`. This means that in order to get any sort of meaningful results from this program, you'll have to load scripts that define a `_Main ()` function. Scripts without the function won't cause anything bad to happen, but because a function will never be called, they won't do anything.

Loading an .XSE Executable

The first thing to do, naturally, is write a function that will give you the ability to load executable script files and populate the VM script structure's major structures with their data. This will account for the first major phase of the XVM prototype's lifecycle.

An .XSE Format Overview

To get things started, refresh yourself on the details of the .XSE format with a quick overview. Tables 10.1 through 10.11 provide a full .XSE format reference. The contents of each table directly follow the contents of the table that precedes them, which means each element of each table can be read in vertical order and assumed to be one contiguous data stream.

Table 10.1 is the main header.

Tables 10.2 through 10.5 comprise the instruction stream.

Following the instruction stream is the string table, displayed in Tables 10.6 and 10.7.

Next up is the function table, in Tables 10.8 and 10.9.

Last up is the host API call table, in Tables 10.10 and 10.11.

Table 10.1 .XSE Main Header

Name	Size (in Bytes)	Description
ID String	4	Four-character string containing the .XSE ID, "XSE0"
Version	2	Version number; (first byte is major, second byte is minor)
Stack Size	4	Requested stack size (set by `SetStackSize` directive; 0 means use default)
Global Data Size	4	The total size of all global data
Is `_Main ()` Present?	1	Set to 1 if the script implemented a `_Main ()` function, 0 otherwise
`_Main ()` Index	4	Index into the function table at which `_Main ()` resides

Table 10.2 The Instruction Stream Structure

Name	Size (in Bytes)	Description
Size	4	The number of instructions in the stream (not the stream size in bytes)
Stream	N	A variable-length stream of instruction structures

Table 10.3 The Instruction Structure

Name	Size (in Bytes)	Description
Opcode	2	The instruction's opcode, corresponding to a specific VM action
Operand Stream	N	Contains the instruction's operand data

Table 10.4 The Operand Stream Structure

Name	Size (in Bytes)	Description
Size	1	The number of operands in the stream (the operand count)
Stream	N	A variable-length stream of operand structures

Table 10.5 The Operand Structure

Name	Size (in Bytes)	Description
Type	1	The type of operand (integer literal, variable, and so on)
Data	N	The operand data itself, which may be any size

Table 10.6 The String Table Structure

Name	Size (in Bytes)	Description
Size	4	The number of strings in the table (not the total table size in bytes)
Strings	N	String data

Table 10.7 The String Structure

Name	Size (in Bytes)	Description
Size	4	The number of characters in the string
Characters	N	Raw string data itself (*not* null terminated)

Table 10.8 The Function Table Structure

Name	Size (in Bytes)	Description
Size	4	The number of functions in the table
Functions	N	Function data

Table 10.9 The Function Structure

Name	Size (in Bytes)	Description
Entry Point	4	The index of the first instruction of the function
Parameter Count	1	The number of parameters the function accepts
Local Data Size	4	The total size of the function's local data (the sum of all local variables and arrays)

Table 10.10 The Host API Call Table Structure

Name	Size (in Bytes)	Description
Size	4	The number of host API calls in the table (not the total table size in bytes)
Host API Calls	N	Host API calls

Table 10.11 The Host API Call Structure

Name	Size (in Bytes)	Description
Size	1	The number of characters in host API function name
Characters	N	The host API function name string (*not* null terminated)

The Header

The header is probably the easiest part of the executable to load. It's read from the file simply by reading the first four elements and saving a few of them. Here's the XVM prototype's implementation:

```
// Create a buffer to hold the file's ID string
// (4 bytes + 1 null terminator = 5)
char * pstrIDString;
pstrIDString = ( char * ) malloc ( 5 );

// Read the string (4 bytes) and append a null terminator
fread ( pstrIDString, 4, 1, pScriptFile );
pstrIDString [ strlen ( XSE_ID_STRING ) ] = '\0';

// Compare the data read from the file to the ID string and exit on an error
// if they don't match
if ( strcmp ( pstrIDString, XSE_ID_STRING ) != 0 )
    return LOAD_ERROR_INVALID_XSE;

// Free the buffer
free ( pstrIDString );

// Read the script version (2 bytes total)
int iMajorVersion = 0,
    iMinorVersion = 0;
fread ( & iMajorVersion, 1, 1, pScriptFile );
fread ( & iMinorVersion, 1, 1, pScriptFile );

// Validate the version, since this prototype only supports version 0.4 scripts
if ( iMajorVersion != 0 || iMinorVersion != 4 )
    return LOAD_ERROR_UNSUPPORTED_VERS;

// Read the stack size (4 bytes)
fread ( & g_Script.Stack.iSize, 4, 1, pScriptFile );

// Check for a default stack size request
if ( g_Script.Stack.iSize == 0 )
    g_Script.Stack.iSize = DEF_STACK_SIZE;
```

```
// Allocate the runtime stack
int iStackSize = g_Script.Stack.iSize;
g_Script.Stack.pElmnts = ( Value * )
    malloc ( iStackSize * sizeof ( Value ) );

// Read the global data size (4 bytes)
fread ( & g_Script.iGlobalDataSize, 4, 1, pScriptFile );

// Check for presence of _Main () (1 byte)
fread ( & g_Script.iIsMainFuncPresent, 1, 1, pScriptFile );

// Read _Main ()'s function index (4 bytes)
fread ( & g_Script.iMainFuncIndex, 4, 1, pScriptFile );
```

NOTE

These code excerpts are from the XVM prototype's `LoadScript ()` function. This function returns a number of error codes to the caller if something goes wrong during the loading process, like `LOAD_ERROR_UNSUPPORTED_VERS` for example. They should be self explanatory, but check out the XVM source on the accompanying CD for more information if you're interested.

That does it. Notice that I also went ahead and allocated the stack at this stage in the loading process. Now is as good a time as any to take care of it.

The Instruction Stream

Immediately following the header data is the instruction stream. Before reading the instruction data, however, you must first read the instruction count and properly allocate space for it. Here's the code:

```
// Read the instruction count (4 bytes)
fread ( & g_Script.InstrStream.iSize, 4, 1, pScriptFile );

// Allocate the stream
g_Script.InstrStream.pInstrs = ( Instr * )
    malloc ( g_Script.InstrStream.iSize * sizeof ( Instr ) );
```

That was easy, but loading the stream itself is considerably more complex. For the most part, it's just a simple loop, but just like always, the details of the operand lists are going to make things tough. The basic idea is to start a loop that will iterate through each instruction in the stream. At each iteration, the opcode and operand count are read from the file. This is easy enough, but the operands themselves pose a slight problem.

Because operand data is neither of a fixed type (floating-point data can be mixed in with integers), nor is it a constant size, each different operand type must be given its own loading code. This is most easily accomplished with a `switch` block that is evaluated at each iteration of another loop that runs inside the first loop to read each operand.

Check it out:

```
for ( int iCurrInstrIndex = 0;
      iCurrInstrIndex < g_Script.InstrStream.iSize;
      ++ iCurrInstrIndex )
{
    // Read the opcode (2 bytes)
    g_Script.InstrStream.pInstr [ iCurrInstrIndex ].iOpcode = 0;
    fread ( & g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].iOpcode,
        2, 1, pScriptFile );

    // Read the operand count (1 byte)
    g_Script.InstrStream.pInstr [ iCurrInstrIndex ].iOpCount = 0;
    fread ( & g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].iOpCount,
        1, 1, pScriptFile );

    int iOpCount = g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].iOpCount;

    // Allocate space for the operand list in a temporary pointer
    Value * pOpList;
    pOpList = ( Value * ) malloc ( iOpCount * sizeof ( Value ) );

    // Read in the operand list (N bytes)
    for ( int iCurrOpIndex = 0; iCurrOpIndex < iOpCount; ++ iCurrOpIndex )
    {
        // Read in the operand type (1 byte)
        pOpList [ iCurrOpIndex ].iType = 0;
        fread ( & pOpList [ iCurrOpIndex ].iType, 1, 1, pScriptFile );

        // Depending on the type, read in the operand data
        switch ( pOpList [ iCurrOpIndex ].iType )
```

```
{
    // Integer literal
    case OP_TYPE_INT:
        fread ( & pOpList [ iCurrOpIndex ].iIntLiteral,
            sizeof ( int ), 1, pScriptFile );
        break;

    // Floating-point literal
    case OP_TYPE_FLOAT:
        fread ( & pOpList [ iCurrOpIndex ].fFloatLiteral,
            sizeof ( float ), 1, pScriptFile );
        break;

    // String index
    case OP_TYPE_STRING:
        // Since there's no field in the Value structure for string
        // table
        // indices, read the index into the integer literal field
        // and set
        // its type to string index
        fread ( & pOpList [ iCurrOpIndex ].iIntLiteral, sizeof ( int ),
            1, pScriptFile );
   pOpList [ iCurrOpIndex ].iType = OP_TYPE_STRING;
            break;

    // Instruction index
    case OP_TYPE_INSTR_INDEX:
        fread ( & pOpList [ iCurrOpIndex ].iInstrIndex,
            sizeof ( int ), 1, pScriptFile );
        break;

    // Absolute stack index
    case OP_TYPE_ABS_STACK_INDEX:
       fread ( & pOpList [ iCurrOpIndex ].iStackIndex,
          sizeof ( int ), 1, pScriptFile );
       break;

    // Relative stack index
    case OP_TYPE_REL_STACK_INDEX:
        fread ( & pOpList [ iCurrOpIndex ].iStackIndex, sizeof ( int ),
            1, pScriptFile );
```

```
                    fread ( & pOpList [ iCurrOpIndex ].iOffsetIndex,
                        sizeof ( int ), 1, pScriptFile );
                    break;

                // Function index
                case OP_TYPE_FUNC_INDEX:
                    fread ( & pOpList [ iCurrOpIndex ].iFuncIndex, sizeof ( int ),
                        1, pScriptFile );
                    break;

                // Host API call index
                case OP_TYPE_HOST_API_CALL_INDEX:
                    fread ( & pOpList [ iCurrOpIndex ].iHostAPICallIndex,
                        sizeof ( int ), 1, pScriptFile );
                    break;

                // Register
                case OP_TYPE_REG:
                    fread ( & pOpList [ iCurrOpIndex ].iReg, sizeof ( int ),
                        1, pScriptFile );
                    break;
            }
        }

        // Assign the operand list pointer to the instruction stream
        g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].pOpList = pOpList;
    }
```

Each iteration of the loop begins by reading the instruction's opcode and operand count. This count is immediately used to allocate space for the operand's data. Another loop is started, which reads each opcode from the file. The actual opcode reading is handled with a `switch` block that provides code to read each different operand type. Once each operand has been read, the pointer to the operand list is assigned to the instruction stream, and the instruction is fully loaded. Check out Figure 10.17.

Notice that the majority of operands were implemented simply by reading a single integer index. Notice also that string table indices are loaded into the `IntLiteral` field of the `Value` structure. This is because `Value` does not contain a field for storing string table indices, because the string table doesn't exist at runtime. Rather, strings will be stored directly in the structure and as such, you only need to hold onto the string indices temporarily. For that reason, you just stuff them

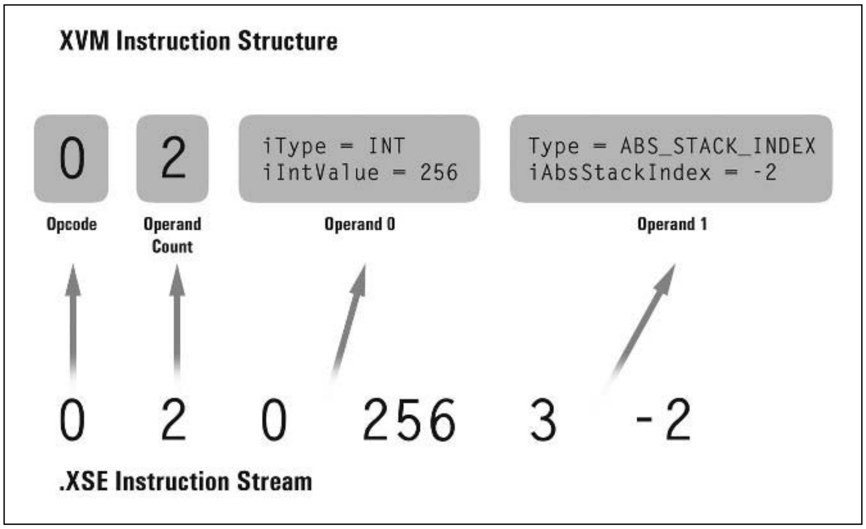


Figure 10.17

Reading instructions from the executable.

into the integer's slot and forget about them it. In the next section, when you load the string table, you'll put this information to use.

The String Table

At runtime, strings are stored directly in the `Value` structure, which is different than their storage on the disk wherein strings are organized in a separate table and only indirectly referenced in the instruction stream. Therefore, once the strings have been read from the file, you need to distribute them throughout the instruction stream so that each operand's `Value` structure contains the string itself.

Basically, the process is as follows: first each string is read from the file into a single array of strings. This creates an in-memory copy of the executable's string table. You then scan through the instruction stream and look for any operand whose type is set for `OP_TYPE_STRING`. Due to the way the file was loaded in the last section, you know that any string operand will have a string table index stored in its `Value` structure's `IntLiteral` field. You just grab this value, use it as an index into the string table, and copy that string literal value into the operand's `StringLiteral` field. You can then delete the table.

Let's begin by allocating the temporary in-memory string table:

```
// Run through each operand in the instruction stream and assign copies
// of string operands' corresponding string literals
for ( int iCurrInstrIndex = 0; iCurrInstrIndex < g_Script.InstrStream.iSize;
    ++ iCurrInstrIndex )
```

```
{
    // Get the instruction's operand count and a copy of its operand list
    int iOpCount = g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].iOpCount;

    Value * pOpList = g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].pOpList;

    // Loop through each operand
    for ( int iCurrOpIndex = 0; iCurrOpIndex < iOpCount; ++ iCurrOpIndex )
    {
        // If the operand is a string index, make a local copy of
        // its corresponding string in the table
        if ( pOpList [ iCurrOpIndex ].iType == OP_TYPE_STRING )
        {
            // Get the string index from the operand's integer literal field
            int iStringIndex = pOpList [ iCurrOpIndex ].iIntLiteral;
            // Allocate a new string to hold a copy of the one in the table
            char * pstrStringCopy;
            pstrStringCopy = ( char * )
                malloc ( strlen ( ppstrStringTable [ iStringIndex ] ) + 1 );

            // Make a copy of the string
            strcpy ( pstrStringCopy, ppstrStringTable [ iStringIndex ] );

            // Save the string pointer in the operand list
            pOpList [ iCurrOpIndex ].pstrStringLiteral = pstrStringCopy;
        }
    }
}
```

With each string in memory, you then run through the instruction stream and replace the `OP_TYPE_STRING` operands:

```
// Loop through each instruction in the stream
for ( int CurrInstr = 0; CurrInstr < g_Script.InstrCount; ++ CurrInstr )
{
    // Get the instruction's operand count
    int OpCount = g_Script.InstrStream.Instrs [ CurrInstr ].OpCount;
    // Loop through each operand in the instruction
    for ( int CurrOp = 0; CurrOp < OpCount; ++ CurrOp )
    {
        // Get the current operand type
        int OpType = g_Script.InstrStream.Instrs    \
            [ CurrInstr ].OpList [ CurrOp ].Type;
        // Is this a string operand?
        if ( OpType == OP_TYPE_STRING )
        {
            // The string index is in the IntLiteral field
            int StringIndex = g_Script.InstrStream    \
                [ CurrInstr ].OpList [ CurrOp ].IntLiteral;
            // Get the string from the table
            string StringOp = StringTable [ StringIndex ];
            // Save the string value in the operand
            g_Script.InstrStream.Instrs [ CurrInstr ].OpList    \
                [ CurrOp ].StringLiteral = OP_TYPE_STRING;
        }
    }
}
```

Of course, we can't just copy the pointers into the instructions' string operands; we have to physically copy the string itself. This is done for two reasons– first, and most obviously, because we're going to free the string table as soon as this loop ends. Also, strings only occur once in the string table; XASM ensures that duplicates are not written to the executable to eliminate needless redundancy. This means that a string literal that appeared four times in the source code will only be represented once in the string table, so each of its four references in the instruction stream will need its own physical copy.

With the strings safely copied to the instruction stream, the string table itself can be disposed of:

```
// Free the original strings
for ( iCurrStringIndex = 0; iCurrStringIndex < iStringTableSize;
    ++ iCurrStringIndex )
    free ( ppstrStringTable [ iCurrStringIndex ] );

// Free the string table itself
free ( ppstrStringTable );
```

The Function Table

The function table contains information about each of the script's functions and is loaded rather easily. First up is the allocation:

```
// Read the function count (4 bytes)
int iFuncTableSize;
fread ( & iFuncTableSize, 4, 1, pScriptFile );

// Allocate the table
g_Script.pFuncTable = ( Func * ) malloc ( iFuncTableSize * sizeof ( Func ) )
```

Next is a loop that reads each function from the file:

```
 // Read each function
for ( int iCurrFuncIndex = 0; iCurrFuncIndex < iFuncTableSize;
    ++ iCurrFuncIndex )
{
    // Read the entry point (4 bytes)
    int iEntryPoint;
    fread ( & iEntryPoint, 4, 1, pScriptFile );

    // Read the parameter count (1 byte)
    int iParamCount = 0;
    fread ( & iParamCount, 1, 1, pScriptFile );

    // Read the local data size (4 bytes)
    int iLocalDataSize;
    fread ( & iLocalDataSize, 4, 1, pScriptFile );

    // Calculate the stack size
    int iStackFrameSize = iParamCount + 1 + iLocalDataSize;

    // Write everything to the function table
    g_Script.pFuncTable [ iCurrFuncIndex ].iEntryPoint = iEntryPoint;
    g_Script.pFuncTable [ iCurrFuncIndex ].iParamCount = iParamCount;
    g_Script.pFuncTable [ iCurrFuncIndex ].iLocalDataSize = iLocalDataSize;
    g_Script.pFuncTable [ iCurrFuncIndex ].iStackFrameSize = iStackFrameSize;
}
```

The Host API Call Table

The last structure to load from the executable is the host API call table. This, like the string table, is simply a sequence of strings and is loaded like virtually everything else you've read from the executable file so far.

I'll just let the code speak for itself. Here's the allocation:

```
// Read the host API call count
fread ( & g_Script.HostAPICallTable.iSize, 4, 1, pScriptFile );

// Allocate the table
g_Script.HostAPICallTable.ppstrCalls = ( char ** )
    malloc ( g_Script.HostAPICallTable.iSize * sizeof ( char * ) );
```

Next is a loop that reads each function from the file:

```
for ( int iCurrCallIndex = 0; iCurrCallIndex < g_Script.HostAPICallTable.iSize;
    ++ iCurrCallIndex )
{
    // Read the host API call string size (1 byte)
    int iCallLength = 0;
    fread ( & iCallLength, 1, 1, pScriptFile );

    // Allocate space for the string plus the null terminator in a
    // temporary pointer
    char * pstrCurrCall;
    pstrCurrCall = ( char * ) malloc ( iCallLength + 1 );

    // Read the host API call string data and append the null terminator
    fread ( pstrCurrCall, iCallLength, 1, pScriptFile );
    pstrCurrCall [ iCallLength ] = '\0';

    // Assign the temporary pointer to the table
    g_Script.HostAPICallTable.ppstrCalls [ iCurrCallIndex ] = pstrCurrCall;
}
```

Structure Interfaces

So you've got the script loaded into memory. Now what? You aren't quite prepared to begin execution just yet, but you're getting there. Let's turn the focus of our discussion to the interfaces you'll need to read and write these major structures you've worked so hard to initialize.

The interfaces to these structures are of prime importance; they'll be the deciding factor in the overall elegance and simplicity of the rest of your VM. The more work and headache involved in interfacing with these structures, the worse your VM's code will ultimately turn out. Priority one is therefore making these interfaces as easy to use as possible.

NOTE

The details and purpose of this section may be somewhat confusing at first, so you might have to take some of this on faith. The following section, "The Execution Cycle," will be considerably easier to understand and implement with this under your belt, however. So, do your best to work through it—if it all makes sense, great, but if you don't get why you're doing everything here, understand that it'll become clear shortly. You may even want to reread this section after you finish the one that follows it.

Figure 10.18 illustrates the concept of adequate interfaces for script structures:

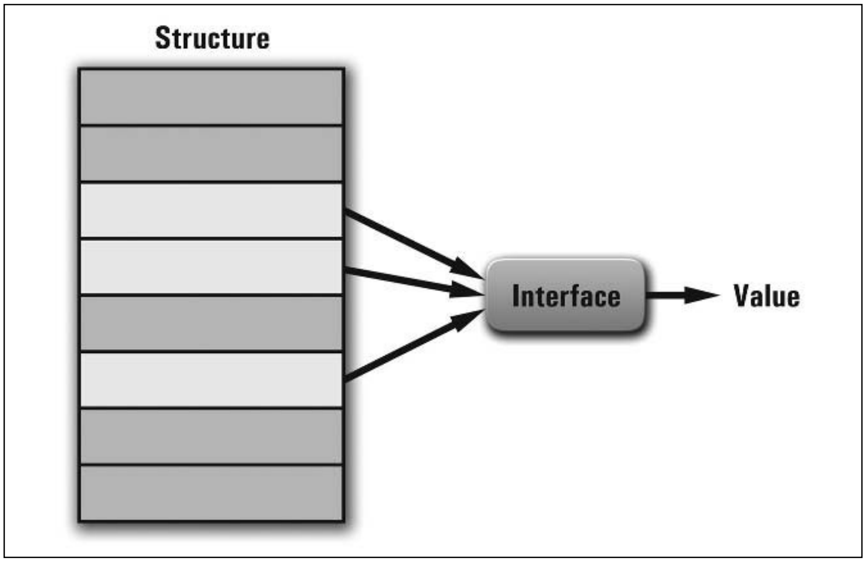


Figure 10.18

Interfaces make structures easy to work with.

The Instruction Stream

As the VM progresses through the instruction stream, it'll frequently need to access and manipulate operand values. Because all instructions (or all of the instructions that take parameters) will need to access their operands in roughly the same way, it'd be silly to duplicate that logic for each instruction handler.

Operands need to be accessed in a number of ways. For example, the code that implements `Mov` will need to determine the stack index pointed to by the destination operand so it knows where to move the source data. It'd be nice to make a single function call that essentially tells the VM "give me the stack index of the first operand". Of course, because the destination may also be the _RetVal register, which doesn't reside on the stack, you might first want to say "tell me the *type* of the first operand." This would just be a simple function that would return constants representing different types of operand values, such as OP_TYPE_STACK_INDEX or OP_TYPE_REG in this case. Once you know the type, you can use the first function to find out where in the stack to copy the data, or just assign it to _RetVal.

Of course, there's also the issue of relative and absolute stack indices. You may want to make another single call that'll fully resolve a relative stack index, because the value of the offset index variable can now be determined. The Mov handler then wouldn't even need to know whether the destination operand was an absolute or relative stack index, because it'd all be handled transparently. The point to all this is again that the more functions you create here, the easier the implementation of your instruction set will be later. Check out Figure 10.19 to see how this automatic index resolution works.

Remember, at any given time, the instruction pointer will tell you where in the instruction stream you are. You can use this to write a set of functions that will return information regarding the operands of the current instruction. Because IP is global it will always track the instruction for you; you can call these functions at any time and be certain you're getting the proper operands.

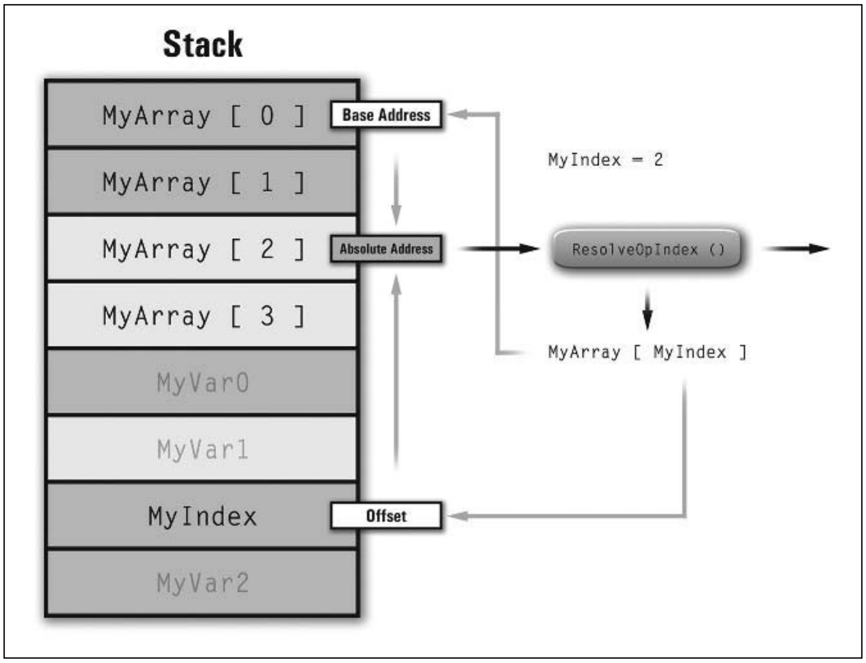


Figure 10.19

A function that automatically resolves relative indices.

First, you'll need a function that will simply return the type of a given operand in the current instruction:

```
int GetOpType ( int iOpIndex )
{
    // Get the current instruction
    int iCurrInstr = g_Script.InstrStream.iCurrInstr;

    // Return the type
    return g_Script.InstrStream.pInstrs
        [ iCurrInstr ].pOpList [ iOpIndex ].iType;
}
```

Simple, huh? All you had to do was grab the `iType` field of the operand in the `pOpList []` array, which resides in the current instruction of the instruction stream, which itself is stored in `g_Script`. Calling this function at any time will return the same constants you defined in XASM for describing operand types. Table 10.12 repeats this list, just for reference:

Table 10.12 Operand List Type Constants

Constant	Description
`OP_TYPE_INT`	Integer literal value
`OP_TYPE_FLOAT`	Floating-point literal value
`OP_TYPE_STRING`	String literal index
`OP_TYPE_ABS_STACK_INDEX`	An absolute stack index (for variables and arrays indexed with integer literals)
`OP_TYPE_REL_STACK_INDEX`	A relative stack index (for arrays indexed with variables)
`OP_TYPE_INSTR_INDEX`	An instruction index (used for jump targets)
`OP_TYPE_FUNC`	Function index (used for `Call` instructions)
`OP_TYPE_HOST_API_CALL`	Host API call index (used for `CallHost` instructions)
`OP_TYPE_REG`	Used for registers references; namely `_RetVal`

So you can read the type of the current instruction's operands. What about the operand values themselves? You can start by writing a function that returns exactly that:

```
int GetOpType ( int iOpIndex )
{
    // Get the current instruction
    int iCurrInstr = g_Script.InstrStream.iCurrInstr;

    // Return the type
    return g_Script.InstrStream.pInstrs
        [ iCurrInstr ].pOpList [ iOpIndex ].iType;
}
```

All you really had to do was take the reference to the `Type` field out, and now it returns the entire `Value` structure. Of course, getting the whole structure is going to be more than you're interested in a lot of situations. For example, consider the index operands of the `GetChar` instruction (see Chapter 8 for a reference). The index operands of this instruction are always integers, which means you'll always want the `IntLiteral` field from the `Value` structure. Let's write a function that'll always return the integer literal component of an operand, regardless of whether it's the active data type:

```
int GetOpAsInt ( int iOpIndex )
{
    // Get the current instruction
    int iCurrInstr = g_Script.InstrStream.iCurrInstr;

    // Return the type
    return g_Script.InstrStream.pInstrs
        [ iCurrInstr ].pOpList [ iOpIndex ].iIntLiteral;
}
```

This is much more convenient. All you have to do now is write versions that do the same thing for each of the other types, which might look like this:

```
// Return a floating-point literal
float GetOpAsFloat ( int OpIndex );
// Return a string literal
string GetOpAsString ( int OpIndex );
// Return a stack index, and automatically resolve relative indices
int GetOpAsStackIndex ( int OpIndex );
// Return an instruction index
int GetOpAsInstrIndex ( int OpIndex );
```

```
// Return a function table index
int GetOpAsFuncIndex ( int OpIndex );
// Return a host API call index
string GetOpAsHostAPICallIndex ( int OpIndex );
// Return a register code
string GetOpAsReg ( int OpIndex );
```

These functions are only so useful, however. Remember, most instructions not only accept literal values, but also _RetVal and variables that refer to values on the stack. For this reason, these functions will return Value structures whose active data types are relative operands and stack indices most often, rather than the actual values themselves. What would be ideal would be a set of functions just like the GetOp* () ones, but instead of just returning whatever operand was found in the instruction stream, would also track down the final values in the case of relative stack values, absolute stack values, and references to _RetVal. This way, a single function call would give us an operand's final, ready-to-use value. Since these functions actually resolve stack indices, they should be called ResolveOp* (), and match the GetOp* () function for function. To get things started, here's the code for ResolveOpValue (), which will return the *final* value of an operand:

```
Value ResolveOpValue ( int iOpIndex )
{
    // Get the current instruction
    int iCurrInstr = g_Script.InstrStream.iCurrInstr;

    // Get the operand type
    Value OpValue = g_Script.InstrStream.pInstrs
        [ iCurrInstr ].pOpList [ iOpIndex ];

    // Determine what to return based on the value's type
    switch ( OpValue.iType )
    {
        // It's a stack index so resolve it
        case OP_TYPE_ABS_STACK_INDEX:
        case OP_TYPE_REL_STACK_INDEX:
        {
            // Resolve the index and use it to return the corresponding
            // stack element
            int iAbsIndex = ResolveOpStackIndex ( iOpIndex );
            return GetStackValue ( iAbsIndex );
        }
        // It's in _RetVal
        case OP_TYPE_REG:
          return g_Script._RetVal;

        // Anything else can be returned as-is
        default:
            return OpValue;
    }
}
```

How cool is this function? Just pass it an operand index, and it'll return the `Value` structure that contains it, no matter where it is– directly in the instruction stream, on the stack via both absolute and relative indices, or in `_RetVal`. The only issue worth mentioning is the call to a yet-undefined function called `GetStackValue ()`. Don't worry, we'll define this function in the next section, and it's extremely simple anyway– all it does is return the stack value at the index you specify. No big deal.

Of course, again, we usually won't want an entire `Value` structure when dealing with operands. Rather, we'd like direct values we can immediately plug into expressions when implementing instructions. So, we'll have to create a whole family of functions that resolve operands as specific data types. Here's an example for resolving operands as integers:

```
int ResolveOpAsInt ( int iOpIndex )
{
    // Resolve the operand's value
    Value OpValue = ResolveOpValue ( iOpIndex );

    return OpValue.iIntLiteral;
}
```

Now that we can leverage `ResolveOpValue ()`, these functions are trivial to say the least. Just resolve the value structure and return the proper field. We'll easily be able to use this framework to create the following:

```
// Return an integer literal
int ResolveOpAsInt ( int OpIndex );
// Return a floating-point literal
float ResolveOpAsFloat ( int OpIndex );
// Return a string literal
char * ResolveOpAsString ( int OpIndex );
```

Now we can resolve operands of any type, which nearly completes the set of functions we'll need when implementing instructions. There is another detail worth exploring, however.

Being able to load a specific data type from any operand with a single call is a great help, but you need to take it one step further for it to do everything you'll ultimately need. In addition to simply reading a given field from an operand's `Value` structure, you'll also need these functions to automatically perform *coercions*. For example, imagine you're executing an `Add` instruction. Now imagine that the source operand is an integer, whereas the destination operand is the string `"256"`. These can't be directly added for obvious reasons, so you might just default to temporarily converting the string to the integer value zero so the two can be added. It won't produce the most meaningful results, but it's not like it was a particularly intelligent instruction to begin with.

You can do better, however. Imagine if `ResolveOpAsInt ()` would *always* produce a valid integer, whether or not the active data type of the operand was an integer. This means that if the operand were the value `256`, you'd get `256` as the return value. If the operand were the floating-point value `256.4`, you'd still get `256`. You'd even get `256` if the operand was the string literal `"256"`. This is an example of data type coercion, and makes your system much more robust by transparently giving instructions exactly the data they need without them having to worry about its original form. Figure 10.20 describes this process visually.

NOTE

The previous reference to the `Add` instruction was just an example. The real `Add` implementation will only be designed for adding numbers.

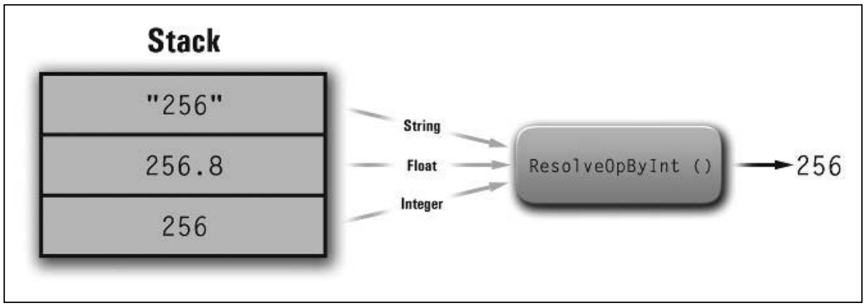


Figure 10.20

Automatic data type coercion and index resolution with a single function call.

The way our `ResolveOpAs* ()` functions are currently implemented, `ResolveOpValue ()` is called first, then the proper field is extracted and returned from the caller. So, rather than directly adding the coercion code to each `ResolveOpAs* ()` function, which would be virtually the same in all cases and therefore redundant, we can create a separate function that coerces `Value` structures to a specified type. We can then use this on the `Value` returned by `ResolveOpValue ()` and nearly complete our set of operand resolution functions. Here's a function for coercing `Value` structures to integer values:

```
int CoerceValueToInt ( Value Val )
{
    // Determine which type the Value currently is
    switch ( Val.iType )
    {
        // It's an integer, so return it as-is
        case OP_TYPE_INT:
            return Val.iIntLiteral;

        // It's a float, so cast it to an integer
        case OP_TYPE_FLOAT:
            return ( int ) Val.fFloatLiteral;

        // It's a string, so convert it to an integer
        case OP_TYPE_STRING:
            return atoi ( Val.pstrStringLiteral );

        // Anything else is invalid
        default:
            return 0;
    }
}
```

This function accepts a single `Value` structure, determines what its active data type is, and coerces it to the specified type. In this case, integers are returned as-is since they're already in the proper form, floats are cast to integers, and strings are converted to numeric values with the ever-handy `atoi ()`. Since these functions are so straightforward and not particularly huge, let's look at the other two we'll need, `CoerceValueToFloat ()` and `CoerceValueToString ()`:

```
float CoerceValueToFloat ( Value Val )
{
    // Determine which type the Value currently is
    switch ( Val.iType )
    {
        // It's an integer, so cast it to a float
        case OP_TYPE_INT:
            return ( float ) Val.iIntLiteral;

        // It's a float, so return it as-is
        case OP_TYPE_FLOAT:
            return Val.fFloatLiteral;
        // It's a string, so convert it to a float
        case OP_TYPE_STRING:
            return ( float ) atof ( Val.pstrStringLiteral );

        // Anything else is invalid
        default:
            return 0;
    }
}
```

Looks simple enough. Here's the string version:

```
char * CoerceValueToString ( Value Val )
{
    char * pstrCoercion;
    if ( Val.iType != OP_TYPE_STRING )
        pstrCoercion = ( char * ) malloc ( MAX_COERCION_STRING_SIZE + 1 );

    // Determine which type the Value currently is
    switch ( Val.iType )
    {
        // It's an integer, so convert it to a string
        case OP_TYPE_INT:
            itoa ( Val.iIntLiteral, pstrCoercion, 10 );
            return pstrCoercion;

        // It's a float, so use sprintf () to convert it since there's
        // no built-in function for converting floats to strings
        case OP_TYPE_FLOAT:
            sprintf ( pstrCoercion, "%f", Val.fFloatLiteral );
            return pstrCoercion;

        // It's a string, so return it as-is
        case OP_TYPE_STRING:
            return Val.pstrStringLiteral;

        // Anything else is invalid
        default:
            return NULL;
    }
}
```

Now this function is a bit different and deserves some explanation. The issue here is that unlike primitive data types `int` and `float`, strings are not allocated statically and therefore, whenever an operand must be converted to a string, its space must be allocated immediately. Unfortunately, we can't very easily tell how long the string needs to be that will hold the converted version of a numeric value. Fortunately, we do know that almost no number will be more than six to ten digits at the most, so allocating even a string as small as 16-24 characters will be enough. I like to play it *really* safe though, so we'll use a default string coercion size of 64 characters, a value stored in `MAX_COERCION_STRING_SIZE`. Sixty-four characters is *way* more than enough, so there shouldn't be any possibility for trouble. The function allocates such a string if the type to which the data needs to be coerced isn't already a string. It then performs the coercion and returns the string's pointer.

TIP

For the sake of performance, you might find that converting strings to integers and back is just needless overhead. In the case of Web scripting like Perl and PHP, this is an invaluable feature, but I must admit it has limited use in the game programming world. My suggestion is to evaluate it on a per-game basis; if you're making a text heavy game that requires a lot of numeric/text conversion, go for it. Otherwise, keep things simple and fast.

The coercion functions can be applied to the operand resolution functions to create some really useful stuff. Let's look at the new version of `ResolveOpAsInt ()`:

```
inline int ResolveOpAsInt ( int iOpIndex )
{
    // Resolve the operand's value
    Value OpValue = ResolveOpValue ( iOpIndex );

    // Coerce it to an int and return it
    int iInt = CoerceValueToInt ( OpValue );
    return iInt;
}
```

Slick, huh? All you have to make is one call, and no matter where the operand resides, and regardless of its data type, you get the optimal integer value. Very cool. Writing one of these for each of the major data types would give you an arsenal of functions making the implementation of your VM's instruction set much easier. All of these instructions will need to be able to easily read operands, and these functions will do exactly that. To wrap this all up, check out Figure 10.21, which illustrates the process of resolving and coercing an operand from start to finish.

For the most part, the `ResolveOp* ()` functions will replace the `GetOp* ()` versions entirely. After all, why waste your time with functions that won't automatically resolve the operand's location?

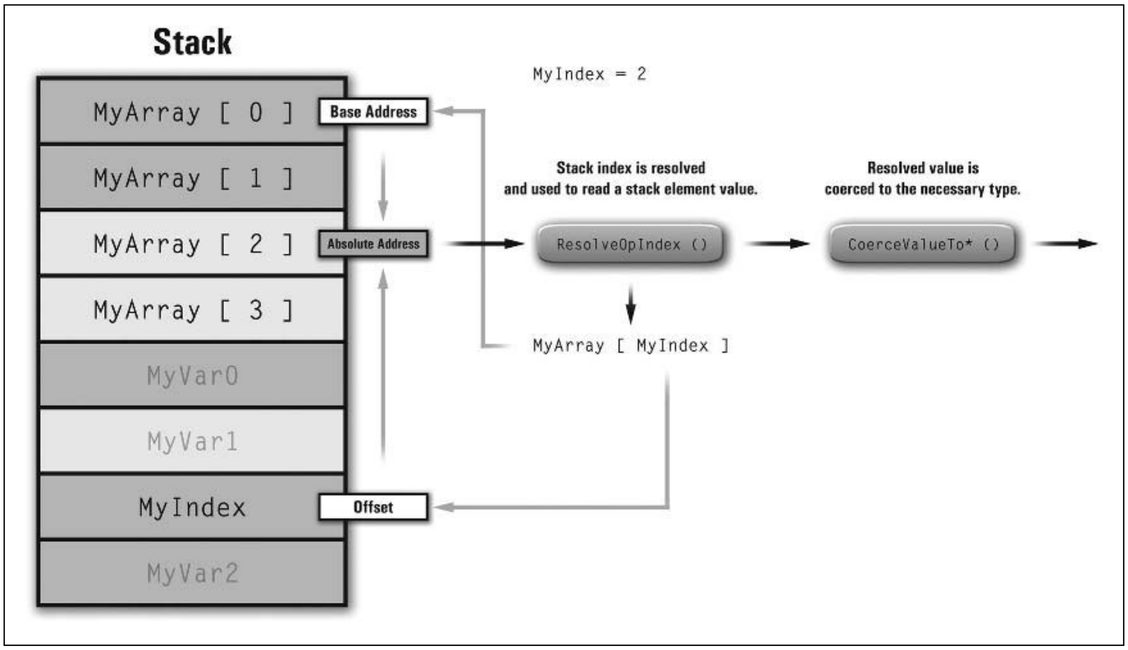


Figure 10.21

The entire process of resolving and coercing an operand.

There is one exception, however, and that's `GetOpType ()`, which must actually exist in two forms. The reason for this is an operand can potentially have two types at once, in a manner of speaking. On the one hand, all values ultimately come down to one of the direct types— integers, strings, line labels, whatever. However, the single level of indirection allowed by your language means that two `Value` objects may be associated with a given operand. The first is the one found in the instruction stream itself, which, in the case of an indirect operand, will be one of the following: a relative stack index, an absolute stack index, or `_RetVal`. This value represents the first "type" of the operand. Once you follow that indirection to the value it points to, however, you find the next "type", which is the value itself. So, for example, one type of operand might be 1) an integer 2) on the stack, whereas another is 1) an integer 2) in `_RetVal`. So, even though both are of the integer data type, their locations differ. This is why you need functions for returning both the operand type as it exists in the stream (the method of indirection), and for returning the *resolved* type (the final value), which is whatever the indirection points to. I'll call them `GetOpType ()` and `ResolveOpType ()`, respectively. Check out Figure 10.22.

There is one last detail, though. We've spent a lot of time writing functions that help us *read* operands, but what about *writing* them? Once an instruction has finished its job and is ready to write the destination, it should have an equally powerful set of functions for making this process

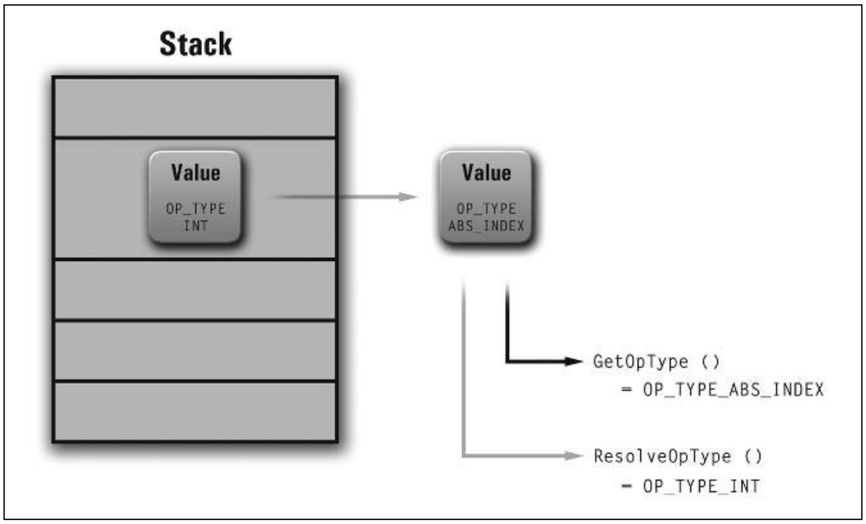


Figure 10.22

The difference between getting an operand type and resolving it.

easy and automated. Fortunately, this part of the job is easier by nature, and we'll only need to write one new function to handle it.

Reading operands is complicated because their location within the runtime environment must be resolved, and their data types must be coerced. Writing them, however, is quite a bit simpler because they can only go to one of two places: the stack or `_RetVal`, and there's no coercion or data type issues to worry about– the destination will take on whatever data type you stuff in it. So, all we really need is an easy way to write a `Value` anywhere, that transparently handles stack indices and `_RetVal`.

I solved this problem by writing a function that simply returns a pointer to wherever the `Value` needs to be written, whether it's on the stack or not. The `Value` object is then written to this pointer, and the job is done. The function is called `ResolveOpPntr ()`, and looks like this:

```
Value * ResolveOpPntr ( int iOpIndex )
{
    // Get the method of indirection
    int iIndirMethod = GetOpType ( iOpIndex );

    // Return a pointer to wherever the operand lies
    switch ( iIndirMethod )
    {
        // It's on the stack
        case OP_TYPE_ABS_STACK_INDEX:
```

```
        case OP_TYPE_REL_STACK_INDEX:
        {
            int iStackIndex = ResolveOpStackIndex ( iOpIndex );
            return & g_Script.Stack.pElmnts
                [ ResolveStackIndex ( iStackIndex ) ];
        }

        // It's _RetVal
        case OP_TYPE_REG:
            return & g_Script._RetVal;
    }

    // Return NULL for anything else
    return NULL;
}
```

With this function, any destination operand can be easily written to by writing a `Value` structure to the pointer it returns. With these functions under our belt, we've mastered the instruction stream and can move on.

The Runtime Stack

The runtime stack is usually manipulated by the script itself, using the `Push` and `Pop` instructions. The VM will have to interface directly with the stack on a frequent basis too, however; namely, when creating and destroying the stack frames that enable your language's nested function calls.

In addition, stack values will be frequently read from and written to by the implementation of various instructions, so you'll need to easily be able to do this. Of course, you can already access the stack with a single line of code, but having to type `g_Script.Stack.Blah.Blah [ iBlah ]` every time gets old after a while. It's just cleaner to wrap stack access in a set of simple functions, and once again, will allow you to add error handling (perhaps to gracefully detect and avoid stack overflow) and other improvements later. Besides, the functions need to be able to automatically interpret negative indices as a sign to index relative to the top of the current stack frame, rather than the bottom of the stack. It'd be a pain to duplicate this logic every time you access the stack. Speaking of which, we should write a macro for resolving stack indices (translating negatives to positives) immediately, since every stack interface function will need to do this:

```
#define ResolveStackIndex( iIndex )    \
    ( iIndex < 0 ? iIndex += g_Script.Stack.iFrameIndex : iIndex )
```

The way this works is simple– if `iIndex` is less than zero, meaning it's a negative stack index and is therefore relative to the top of the current stack frame, it's added to the stack's `iFrameIndex` index. Otherwise, it's left alone because positive indices are already in their fully resolved form. Remember, negative stack indices are relative to the top of the current *stack frame*, not the actual top of the stack (although these two values are often equal). The whole point of negative indices is to easily access a function's local values.

Now that we can translate stack indices painlessly, let's write some general, random access stack manipulation functions:

```
Value GetStackValue ( int iIndex )
{
    // Use ResolveStackIndex () to return the element at the specified index
    return g_Script.Stack.pElmnts [ ResolveStackIndex ( iIndex ) ];
}
void SetStackValue ( int iIndex, Value Val )
{
    // Use ResolveStackIndex () to set the element at the specified index
    g_Script.Stack.pElmnts [ ResolveStackIndex ( iIndex ) ] = Val;
}
```

Simple, but quite useful. This explains the `GetStackValue ()` function from the last section, by the way. Figure 10.23 illustrates its use.

Of course, the real way to access a stack is through the traditional push and pop interface. You'll write two functions, `Push ()` and `Pop ()`, that can push and pop `Value` structures onto and off of the stack. You'll even be able to use these functions directly in the implementation of their corresponding instructions.

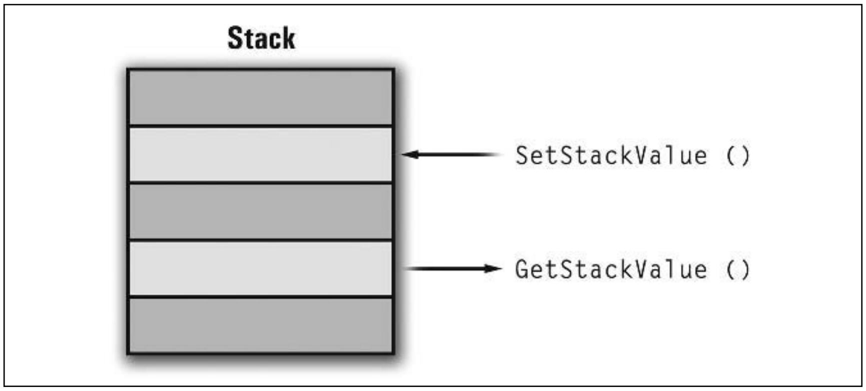


Figure 10.23

Random stack access.

To push a runtime value onto the stack, you copy the `Value` structure into the array index pointed to by the `iTopIndex` field of the `Stack` structure, and then increment that value. Here's an example:

```
void Push ( Value Val )
{
    // Get the current top element
    int iTopIndex = g_Script.Stack.iTopIndex;

    // Put the value into the current top index
    g_Script.Stack.pElmnts [ iTopIndex ] = Val;

    // Increment the top index
    ++ g_Script.Stack.iTopIndex;
}
```

To pop a value off, you need only reverse the process. One thing to note, however, is that you won't actually erase the index. Rather, you'll simply decrement the top index so that the next `Push` operation will overwrite it.

```
Value Pop ()
{
    // Decrement the top index to clear the old element for overwriting
    -- g_Script.Stack.iTopIndex;

    // Get the current top element
    int iTopIndex = g_Script.Stack.iTopIndex;

    // Use this index to read the top element
    Value Val = g_Script.Stack.pElmnts [ iTopIndex ];

    // Return the value to the caller
    return Val;
}
```

So you've got random stack access in addition to the traditional interface. You're almost there, but while you're at it you might as well add two more simple functions for aiding in the construction and destruction of stack frames.

Stack frames can really just be thought of as sequential blocks of stack elements. When a new function is invoked, the parameter list will have already been pushed on by the script, which means all that's left is the return address and local data space. This is handled by the VM, so you need a good way to quickly push a large block of new elements onto the stack. You can create a function called `PushFrame ()` to do the job for you:

```
void PushFrame ( int iSize )
{
    // Increment the top index by the size of the frame
    g_Script.Stack.iTopIndex += iSize;

    // Move the frame index to the new top of the stack
    g_Script.Stack.iFrameIndex = g_Script.Stack.iTopIndex;
}
```

Just pass it the desired stack size with the `Stack` parameter and you're done. But wait a second; is that everything? Yes, all you need to do is increment `iTopIndex` and update `iFrameIndex`, and the frame becomes available on the stack. This is because when dealing with a stack, all that really matters is where these two indices are. Any subsequent calls to `Push ()` or even `PushFrame ()` (as well as any further execution of the `Push` instruction from within the script) will create new elements on *top* of the frame, because their locations will be based on the new value of `iTopIndex`. Therefore, the area within the frame will remain safe to use for your purposes. Of course, what this also means is that your newly allocated stack frame will be filled with potential garbage values, which in turn means that XVM variables are not automatically initialized to zero. You could manually scan through each element of our new frame and clear it, but it's just more overhead you don't need. Again, think about how often functions will be called as a script executes—if you can eliminate the overhead of clearing out every one of those functions' stack frames simply by making sure to initialize your own variables, you can save a lot of time. Figure 10.24 illustrates how this works.

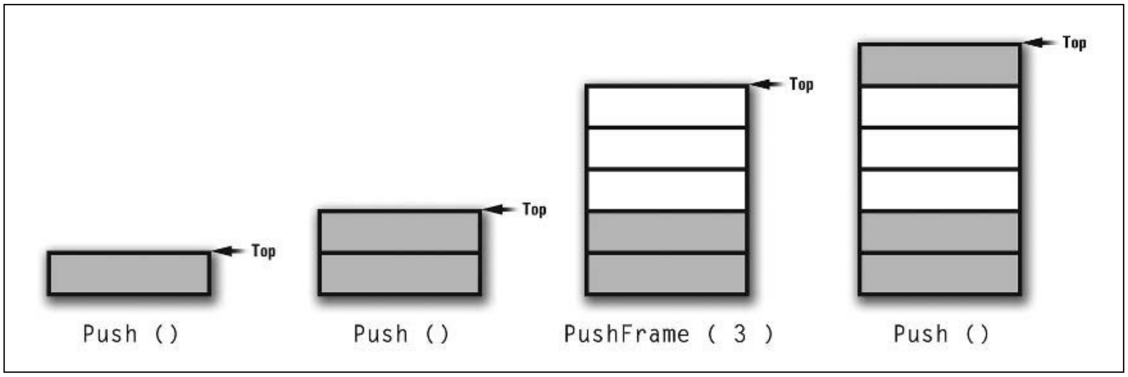


Figure 10.24

Pushing a stack frame simply involves incrementing the top index.

Once an empty frame has been established with PushFrame (), you can use the random access SetStackValue () and GetStackValue () to manipulate its elements.

But, like everything you push onto the stack, stack frames must eventually be popped back off when the function returns. This is just as easy as the PushFrame () function—all you do is decrement TopIndex by the specified frame size, and that entire area of the stack will immediately be cleared for overwriting by the next stack operation. You also won't return any of the frame's data, instead just leaving it up to the caller to use GetStackValue () to save anything important beforehand. Check it out:

```
void PopFrame ( int iSize )
{
    g_Script.Stack.iTopIndex -= iSize;
}
```

Remember also, unlike PushFrame (), PopFrame () shouldn't mess with the stack's frame pointer (iFrameIndex). As we saw in an earlier section, the Call and Ret instructions will manually handle iFrameIndex, so the PopFrame () function itself shouldn't mess with it.

As usual though, there's a very important detail we haven't addressed yet that needs to be dealt with before moving on. This particular issue rears its head initially in the implementation of Push ()– specifically, where the next element of the stack is overwritten by the supplied Value structure. Here's an example to help you understand the problem:

Imagine that a string value is pushed onto the stack. This means that the top element on the stack has a string pointer in its pstrStringLiteral field, which points to a pre-allocated string buffer in memory. Now imagine that this value is popped off along with a stack frame, which means that the string is never freed; rather, the stack's top index is just decremented so that this particular value will eventually be overwritten. The problem is, once this stack element is filled with another Value structure, the XVM will lose track of the string to which it points, preventing it from ever being freed and thus starting a possibly large series of dangling string pointers. If this problem persists, the system's memory will slowly lock up as more and more strings are allocated but never released.

To solve this problem, we need to abstract the process of writing one Value structure to another by wrapping it in a separate function. This way, we can write the function to intelligently handle this string pointer issue, and defuse the situation. This function will be called CopyValue () and will look like this:

```
void CopyValue ( Value * pDest, Value Source )
{
    // If the destination already contains a string, make sure to free it first
    if ( pDest->iType == OP_TYPE_STRING )
        free ( pDest->pstrStringLiteral );
    // Copy the object
    * pDest = Source;

    // Make a physical copy of the source string, if necessary
    if ( Source.iType == OP_TYPE_STRING )
    {
        pDest->pstrStringLiteral = ( char * )
            malloc ( strlen ( Source.pstrStringLiteral ) + 1 );
        strcpy ( pDest->pstrStringLiteral, Source.pstrStringLiteral );
    }
}
```

Cool, huh? Now, instead of directly assigning anything to the stack or _RetVal, we just pass the source Value, and a pointer to the destination Value, and we'll be guaranteed a safe copy.

This should be everything you need to intelligently handle the script's runtime stack, so let's move on.

The Function Table

Finally, an easy structure to work with! Unlike everything else you've seen so far, the function table is extremely simple and only requires a single function. The function table is an entirely static structure—it doesn't change in any way during the runtime of a script. This must mean the script never writes to it, which in turn means you only need to create a function for reading functions from the table. Here it is:

```
Func GetFunc ( int iIndex )
{
    return g_Script.FuncTable [ iIndex ];
}
```

The Host API Call Table

I won't be discussing communication with the host application until the next chapter, but you'll create the necessary host API call table interface now due to its simplicity. Much like the function table interface, all you need is the ability to read a host API call. There won't be any time you need to make changes to this table, so this single function will suffice. Here it is:

```
char * GetHostAPICall ( int iIndex )
{
    return g_Script.HostAPICallTable.ppstrCalls [ iIndex ];
}
```

Summary

Just to round out the discussion and provide a reference, here are all of the functions you've created (directly or indirectly) in this section:

The Instruction Stream

The following code returns the type of the specified operand in the current instruction. Note the difference between `GetOpType ()` and `ResolveOpType ()`. The first returns the type of the operand as it exists in the instruction stream, which may simply be a stack index or reference to `_RetVal`. `ResolveOpType ()`, however, always returns the final type of the value itself.

```
int GetOpType ( int OpIndex );
int ResolveOpType ( int OpIndex );
```

The following function returns a `Value` structure representing the specified operand in the current instruction. The returned `Value` structure is always the actual value itself; if the operand references it in `_RetVal` or the stack, this function will locate it. This process is called *resolving*.

```
Value ResolveOpValue ( int OpIndex );
```

The following code returns the value of the specified operand in the current instruction in a specific data type. It automatically performs coercions to ensure that the returned value is always optimal given the operand's active data type. These functions use `ResolveOpValue ()` to initially locate the real `Value` structure, which means they too always return the real value in the case of indirection.

```
int ResolveOpAsInt ( int OpIndex );
float ResolveOpAsFloat ( int OpIndex );
string ResolveOpAsString ( int OpIndex );
int ResolveOpAsStackIndex ( int OpIndex );
int ResolveOpAsInstrIndex ( int OpIndex );
int ResolveOpAsFuncIndex ( int OpIndex );
string ResolveOpAsHostAPICallIndex ( int OpIndex );
string ResolveOpAsReg ( int OpIndex );
```

These functions also make use of the `Value` structure coercion functions:

```
int CoerceValueToInt ( Value Val );
float CoerceValueToFloat ( Value Val );
char * CoerceValueToString ( Value Val );
```

Lastly, once we've done all of our operand reading, it's time to do some writing. We can do this easily with `ResolveOpPntr ()`, which returns a pointer to the `Value` structure of any operand:

```
Value * ResolveOpPntr ( int iOpIndex );
```

The Runtime Stack

Above all else, stack indices need to be interpreted properly since they can come in positive and negative forms. This is handled via the `ResolveStackIndex ()` macro.

The following functions set and return the value of specific stack indices, thus providing random access to the runtime stack.

```
void SetStackValue ( int iIndex, Value Val );
Value GetStackValue ( int iIndex );
```

These functions provide a traditional stack interface by allowing `Value` structures to be pushed on and popped off.

```
void Push ( Value Val );
Value Pop ();
```

The following functions are used to push and pop variable-sized blocks of elements without initializing or clearing them. They're primarily used when constructing and destructing a function call's stack frame, but can be used any time the creation or destruction of a contiguous block of stack elements relative to the top of the stack is necessary.

```
void PushFrame ( int iSize );
void PopFrame ( int iSize );
```

Lastly, in order to safely move one `Value` structure into another, use this:

```
void CopyValue ( Value * pDest, Value Source );
```

The Function Table

This returns a `Func` structure describing the specified function.

```
Func GetFunc ( int Index );
```

The Host API Call Table

This returns the host API function name at the specified index.

```
char * GetHostAPICall ( int iIndex );
```

This wraps up the interfaces the XVM prototypes major structures will need. With these in place, we can get back to executing scripts.

Initializing the VM

Before the script can begin execution, the runtime environment must be prepared, which is a simple but vital process. Here's a rundown of what must be done to set the stage for the script to run:

- The script's entry point must be found and placed in the instruction pointer. Check out Figure 10.25.
- The stack must be cleared; in other words, the stack's top index and frame index must both be set to zero.
- Each element of the stack must be nulled out.
- The script's pause flag must be cleared by explicitly setting it to `FALSE`.
- Space for the script's global variables must be allocated by pushing a frame equal to the script's global data size onto the stack.
- `_Main ()`'s stack frame must be pushed onto the stack as well, to provide space for its local variables.

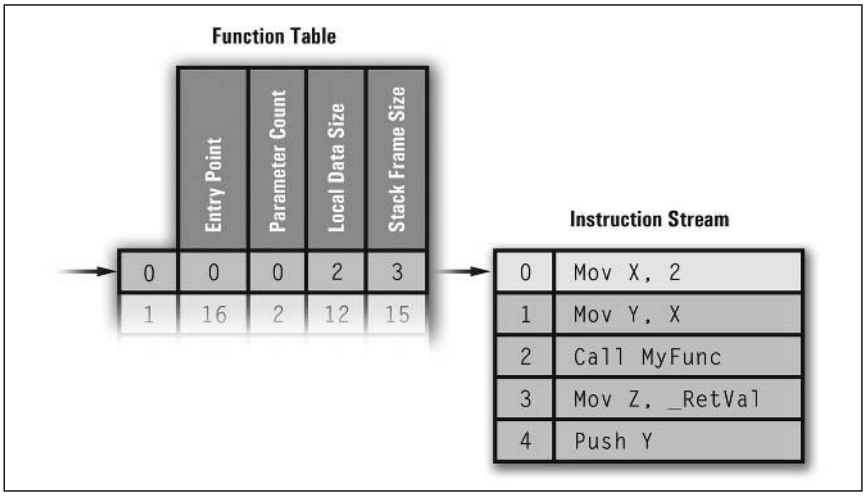


Figure 10.25

Using the function table to determine `_Main ()`'s entry point.

Once these steps are completed, the VM will be ready to roll. Let's take a look at `ResetScript ()`, an XVM prototype function used to do exactly this:

```
void XS_ResetScript ()
{
    // Get _Main ()'s function index in case we need it
    int iMainFuncIndex = g_Script.iMainFuncIndex;
    // If the function table is present, set the entry point
    if ( g_Script.FuncTable.pFuncs )
    {
        // If _Main () is present, read _Main ()'s index of the function
        // table to get its entry point
        if ( g_Script.iIsMainFuncPresent )
        {
            g_Script.InstrStream.iCurrInstr = g_Script.FuncTable.pFuncs
               [ iMainFuncIndex ].iEntryPoint;
        }
    }

    // Clear the stack
    g_Script.Stack.iTopIndex = 0;
    g_Script.Stack.iFrameIndex = 0;

    // Set the entire stack to null
    for ( int iCurrElmntIndex = 0; iCurrElmntIndex < g_Script.Stack.iSize;
        ++ iCurrElmntIndex )
        g_Script.Stack.pElmnts [ iCurrElmntIndex ].iType = OP_TYPE_NULL;

    // Unpause the script
    g_Script.iIsPaused = FALSE;

    // Allocate space for the globals
    PushFrame ( g_Script.iGlobalDataSize );

    // If _Main () is present, push its stack frame (plus one extra stack
    // element to compensate for the function index that usually sits on top
    // of stack frames and causes indices to start from -2)
    PushFrame ( g_Script.FuncTable.pFuncs
        [ iMainFuncIndex ].iLocalDataSize + 1 );
}
```

Just as I described in the list above, this code begins by locating the script's entry point and initializing the instruction pointer to point to it. The stack's `iTopIndex` and `iFrameIndex` fields are then zeroed out. The stack structure itself is then looped through and set to the `OP_TYPE_NULL` operand type, which is a new constant added to the XVM that was not present in XASM and should be reasonably self explanatory. The script is then explicitly unpaused.

The next two sections require a bit more explanation. The global data in a script always resides at the bottom, which means that if there are four global variables and a global array of 12 elements, declared like this:

```
Var GlobalVar0
Var GlobalVar1
Var GlobalVar2
Var GlobalVar3
Var GlobalArray [ 12 ]
```

The script will need to maintain a total of 16 stack indices, relative to the bottom, to hold them (0-15). This is accomplished by pushing a stack frame equal in size to the script's global data, which explains this line:

```
PushFrame ( g_Script.iGlobalDataSize );
```

Figure 10.26 illustrates the space set aside from globals on the stack.

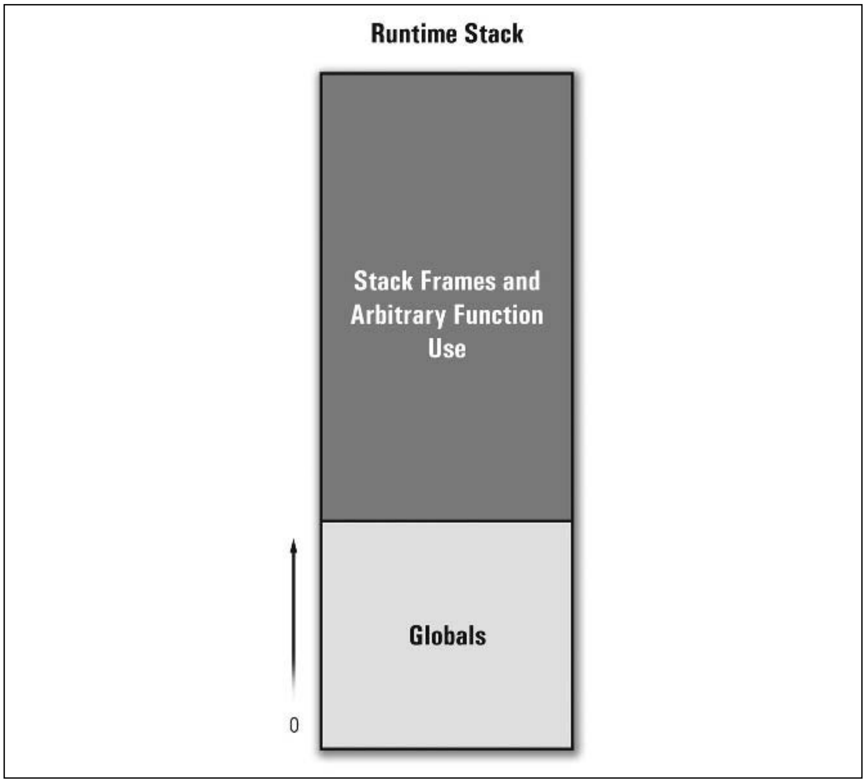


Figure 10.26

Global data resides in a contiguous region at the bottom of the stack.

Once the global data region has been added, the stack is almost ready to go. The only detail that remains is the `_Main ()` function's stack frame. `_Main ()` may be a special function, but it needs a stack frame just like any other function the script may define. The stack frame itself is used for slightly simpler purposes, however. Since `_Main ()` doesn't have to "return" to anything, there's no need to make room for a return address. Also, you can't pass `_Main ()` parameters, so parameter space isn't necessary either. All you really need is room for its local data, hence the following line:

```
PushFrame ( g_Script.FuncTable.pFuncs [ iMainFuncIndex ].iLocalDataSize + 1 );
```

Wait a second, though, what's with the `+ 1`? We need to make room for an extra stack index because, even though `_Main ()` doesn't use it, every function's local data is indexed with -2 because any non-`_Main ()` function requires the extra function index pushed onto the stack just after the frame. Because of this, even though it doesn't apply to `_Main ()`, all of its local variables will access the stack relative to the same -2 index. Rather than rigging XASM to handle this as a special case when parsing variable declarations, we can solve the problem much more easily by just pushing on a dummy stack element. This is all explained graphically in Figure 10.27.

The Execution Cycle

After much planning, the time is finally upon you. You've seen everything (more or less) this initial XVM prototype will have to manage, and are finally ready to explore the implementation of its execution cycle.

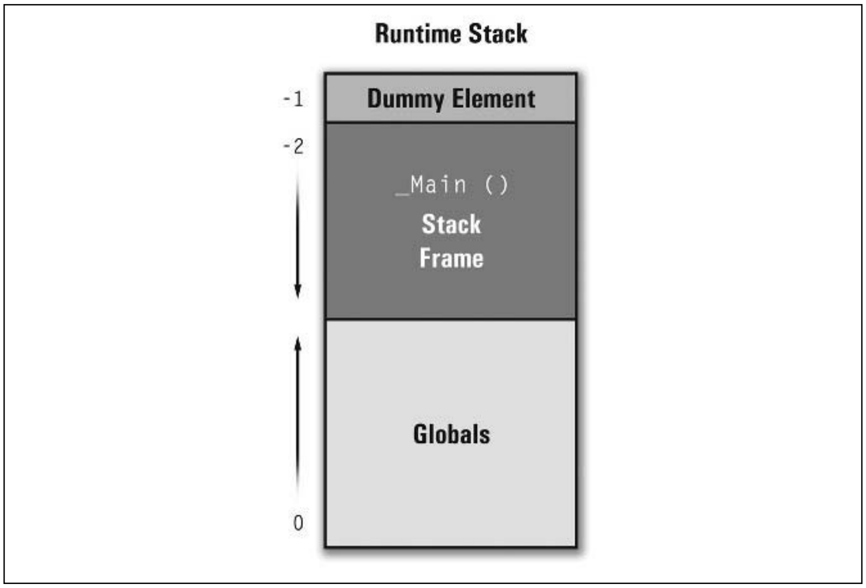


Figure 10.27

An extra dummy element must be pushed onto the stack after `_Main ()`'s frame to align its local variable stack references, even though it won't be used.

On a basic level, this primitive version of the VM will consist mainly of a `while` loop that encapsulates the entire execution cycle and runs until a key is pressed. At each iteration of the loop, a new instruction is processed in full; its effects on the stack and string table are managed and any jumps or function calls it makes are handled. After executing the instruction, its instruction mnemonic and operands are printed to the screen so you can watch the flow of execution progress.

The loop itself will of course be simplistic. All you really need is this:

```
// Loop until a key is pressed
while ( ! kbhit () )
{
    // Handle next instruction
}
```

Of course, it's the guts you're really interested in. The following sections deal with what will go on inside this loop as it's executing. It's inside this loop that script execution finally gets off the ground; this is really one of the major moments you've been working your way up to.

Figure 10.28 will help you visualize the execution cycle.

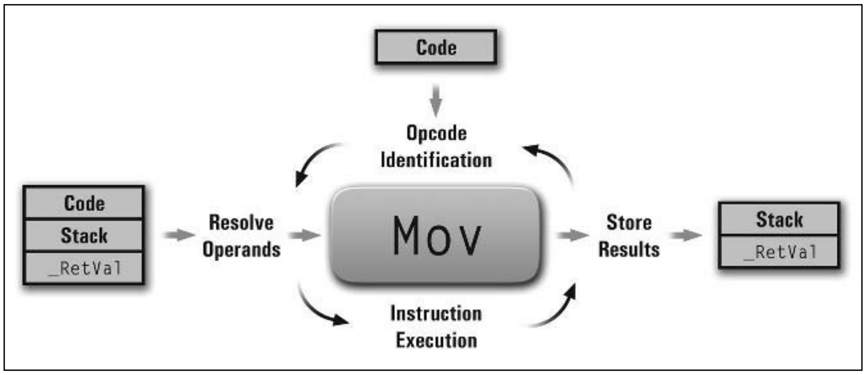


Figure 10.28

The execution cycle.

Instruction Set Implementation

The most important part of each iteration of the main loop is the execution of the next instruction. Given the opcode of the current instruction, there are a number of ways to vector to the proper instruction handler. The first and most obvious is simply a giant `switch` block, like you saw earlier. Each `case` of the block implements a specific instruction in full. Another popular method is to write individual functions for each instruction and group their pointers in an array that's indexed by their opcode. Figure 10.29 illustrates this.

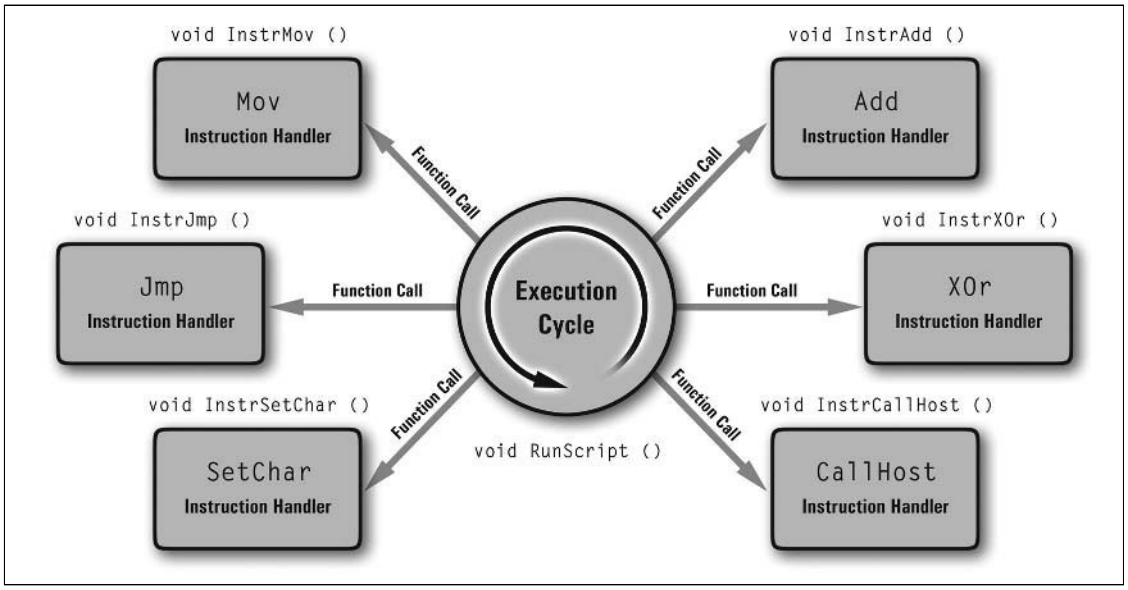


Figure 10.29

Writing separate functions for each instruction handler.

In a lot of ways the function method is more flexible; for example, `DLL`s or other forms of dynamic libraries could be written that allow the VM to "swap out" entire instruction sets. It also provides better overall encapsulation, because each instruction is in an isolated scope. However, I prefer the `switch` method for smaller languages like this one and mostly for teaching purposes because it's easier to visualize and implement. One important advantage to this method is that state information is easier to manage. In other words, a number of important variables must be tracked during the progression of the main loop, variables that are often important to each instruction implementation. If these are defined in the main loop's local scope, the `switch` block will have automatic access to all of them. However, in order for separate instruction-implementing functions to access them, they must either be passed every time as a function or made global. Check out Figure 10.30.

You may be wondering, however, why I suddenly recommend using a giant `switch` block when I said just the opposite during the construction of XASM. This is because even though the two `switch`es are both concerned with handling instructions, they're implemented in very different ways. In XASM, the assembly of an instruction doesn't vary much from one to the next, and what does vary can be stored in an array or other similar structure. This isn't the case at runtime. Obviously, the functionality of one instruction will be considerably different than another. `Add` and `CallHost` may be assembled in the same way, but they behave totally differently and are designed for completely unrelated purposes.

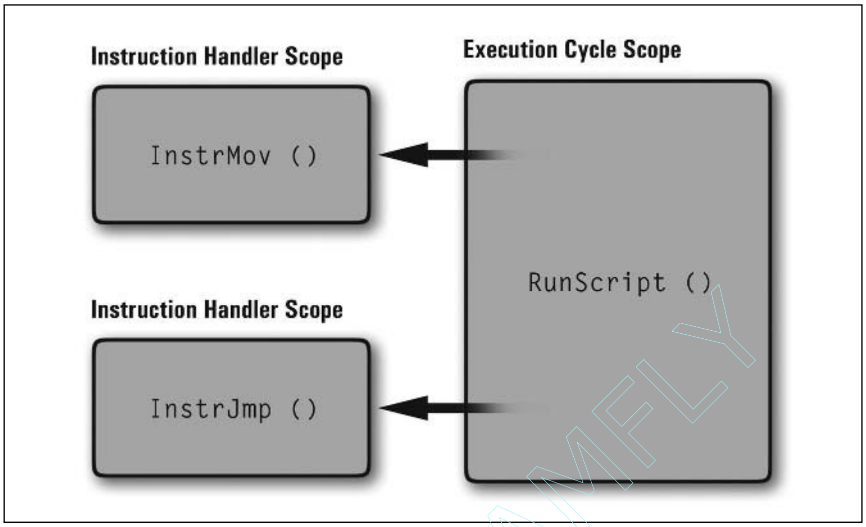


Figure 10.30

Instruction handlers are awkward to write as functions because their scope isolates them from `RunScript ()`*'s local variables.*

In order to `switch` to the proper instruction, it helps to assign each opcode to a constant that gives it a more intelligible name. The code then becomes much more readable. Consider this:

```
switch ( Opcode )
{
    case 0:
        // Implement Mov
        break;
    case 1:
        // Implement Add
        break;
    case 2:
        // Implement Sub
        break;
}
```

And compare it to this:

```
switch ( Opcode )
{
    case INSTR_MOV:
        // Implement Mov
        break;
    case INSTR_ADD:
        // Implement Add
        break;
    case INSTR_SUB:
        // Implement Sub
        break;
}
```

The latter is obviously a lot easier to follow and understand. Table 10.13 lists these constants.

Table 10.13 Instruction Opcode Constants

Mnemonic	Opcode	Constant
Mov	0	INSTR_MOV
Add	1	INSTR_ADD
Sub	2	INSTR_SUB
Mul	3	INSTR_MUL
Div	4	INSTR_DIV
Mod	5	INSTR_MOD
Exp	6	INSTR_EXP
Neg	7	INSTR_NEG
Inc	8	INSTR_INC
Dec	9	INSTR_DEC
And	10	INSTR_AND
Or	11	INSTR_OR
XOr	12	INSTR_XOR
Not	13	INSTR_NOT
ShL	14	INSTR_SHL
ShR	15	INSTR_SHR
Concat	16	INSTR_CONCAT
GetChar	17	INSTR_GETCHAR
SetChar	18	INSTR_SETCHAR
Jmp	19	INSTR_JMP

Table 10.13 Continued

Mnemonic	Opcode	Constant
JE	20	INSTR_JE
JNE	21	INSTR_JNE
JG	22	INSTR_JG
JL	23	INSTR_JL
JGE	24	INSTR_JGE
JLE	25	INSTR_JLE
Push	26	INSTR_PUSH
Pop	27	INSTR_POP
Call	28	INSTR_CALL
Ret	29	INSTR_RET
CallHost	30	INSTR_CALLHOST
Pause	31	INSTR_PAUSE
Exit	32	INSTR_EXIT

With this table, you can easily set up a basic instruction-handling skeleton, like so:

```
// Check the current opcode value
switch ( iOpcode )
{
    case INSTR_MOV:
        // Implement Mov
        break;

    case INSTR_ADD:
        // Implement Mov
        break;

    case INSTR_SUB:
        // Implement Mov
        break;
    // ...

    case INSTR_PAUSE:
        // Implement Pause
        break;

    case INSTR_EXIT:
        // Implement Exit
        break;
}
```

As you can see, you're working your way in from the outside. You started with nothing but data structures, and then created a main loop, and now you have an instruction-handling skeleton. The next stop is each instructions' behavior. But first, let's take a quick detour into a few loose ends that need to be tied up before jumping in.

Handling Script Pauses

Our execution cycle skeleton is starting to take shape, but we can't get to the implementation of instructions just yet. Remember, scripts can pause themselves for specified durations with the `Pause` command. The actual `Pause` instruction handler can't perform this delay itself, however, because the loop needs to continually execute until the pause duration elapses. The XVM prototype really gains nothing from this, but the final version of the XVM, which is both multithreaded and has to run smoothly alongside a host application, *must* be able to handle script pauses synchronously (meaning, without stalling the rest of the game).

Because of this, the main execution loop must begin with a check for the script's pause flag. If the script is currently paused, the current time is compared to the time at which the pause is scheduled to end. If these times are equal, or if the current time is greater, we know the pause has elapsed and can clear the pause flag. Here's some code:

```
// Update the current time
int iCurrTime = GetCurrTime ();

// Check the script's pause flag
if ( g_Script.iIsPaused )
{
    // Has the pause duration elapsed yet?
    if ( iCurrTime >= g_Script.iPauseEndTime )
    {
        // Yes, so unpause the script
        g_Script.iIsPaused = FALSE;
    }
    else
    {
        // No, so skip this iteration of the execution cycle
        continue;
    }
}
```

Simple, huh? Either the pause is over and the flag is cleared, or we just skip this iteration of the loop with `continue`. You may be wondering where the `iCurrTime` variable gets its value, however. At each iteration of the execution loop, `iCurrTime` is updated to contain the current time, so that any code within the loop can refer to it. Its apparently gets this value from a function called `GetCurrTime ()`, but we haven't seen that one yet.

GetCurrTime ()

At any point, the current time in milliseconds can be determined with `GetCurrTime ()`. This isn't a platform-specific API call, however; it's defined by the XVM. The implementation, however, is completely platform dependent, which is why I created this function in the first place. It's designed to wrap whatever function the platform provides for getting the current time in milliseconds, so Windows-specific API calls wouldn't have to be hard coded into the system. For example, if you're a Windows user and don't mind a little inaccuracy, (and by "a little" I mean "up to 55 milliseconds") you can use `GetTickCount ()`:

```
int GetCurrTime ()
{
    return GetTickCount ();
}
```

If you're on another platform, you can fill this with whatever it provides.

Incrementing the Instruction Pointer

Naturally, the instruction pointer has to be incremented after the execution of each new instruction so that it points to the next and the process can repeat. At first, this seems like such a trivial issue that you're wondering why I've even bothered dedicating a section to it.

The instruction pointer is indeed easy to handle in the case of most instructions. However, instructions like `Call` and the jump family cause the pointer to move around irregularly. After executing `Call` or `Jmp` (for example), IP will point to the function's entry point or the jump's target instruction. This means that IP shouldn't be changed before the next instruction is executed, because it's already where it needs to be for the next cycle. However, if our code blindly increments IP after executing *all* instructions, we're going to run into some problems because the entry points and jump targets of these instructions will be skipped by one.

So, we need a way to know whether or not IP has changed during the execution of the instruction. If it has, it can be left alone. Otherwise, it needs to be incremented. The simplest way to do this is to save the state of IP in a local variable before the instruction is executed, then compare it to that variable afterwards. If the two values are equal, we know IP hasn't been changed by the instruction, and can be incremented safely. Otherwise, we ignore it and assume that the instruction has moved it to a location we shouldn't mess with. Now that you understand the process, here's the code:

```
// Save IP
int iCurrInstr = g_Script.InstrStream.iCurrInstr;
// Execute the current instruction
switch ( iOpcode )
{
    case INSTR_MOV:
        // This instruction does not alter IP.
        break;

    case INSTR_JMP:
        // This instruction DOES alter IP.
        break;

    case INSTR_CALL:
        // This instruction DOES alter IP.
        break;

    case INSTR_PAUSE:
        // This instruction does not alter IP.
        break;
}

// Has IP changed during the instruction's execution?
if ( iCurrInstr == g_Script.InstrStream.iCurrInstr )
    // No, so increment it
    ++ g_Script.InstrStream.iCurrInstr;
```

With this final detail out of the way, the skeleton of the execution cycle is pretty much taken care of, so we can get back to the real meat of things– implementing the instruction set.

Operand Resolution

As you saw, each instruction's implementation resides in a `case`. Within this `case`, you can break the implementation into phases, as discussed earlier, like this:

```
case INSTR_MOV:
    // Resolve operands
    // Execute instruction logic
    // Store results
    break;
```

NOTE

Like I mentioned earlier, you might want to reread or at least skim over the contents of the last section, *Structure Interfaces*, due to its significant relevance here.

Note that the first and last phases, resolving operands and storing the results, involve interaction with the script's structures like the stack and its global data tables. These phases will be particularly easy to handle due to the set of functions created in the last section ("Structure Interfaces"). You took the time to wrap otherwise complex and inconvenient processes in single functions that will prove more than beneficial in the following subsections.

It's the first of these phases that concerns you now. *Resolving* an instruction's operand is the term I use to refer to locating their its values (whether it's immediately in the instruction stream, in the stack, or in a register like `_RetVal`), bringing a copy of these values into the local scope, and coercing their data types into compatibility with one another. Fortunately, this is almost entirely handled by the set of functions you built specifically for this task in the last section. The `ResolveOp* ()` functions in particular will come in quite handy.

Let's imagine the `Add` instruction. It takes two operands, `Source` and `Destination`. `Source` is then added to `Destination` to compute and store the sum. The problem is an issue of data types—because the language is typeless, the assembler won't speak up if you try adding a string to an integer, or an integer to a float, or whatever. It's therefore up to the VM to straighten any incompatibilities between the source and destination operands, perform the necessary coercion, and continue with the instruction's logic.

The solution to this problem is quite simple: what really matters is the data type of the destination. If, for instance, the VM finds itself adding a string to an integer, the integer destination is most likely what the user found more important (after all, it's the operand that will be changed after the instruction has executed; `Source` will remain unaffected). Therefore, you simply need to use `ResolveOpAsInt ()` when resolving the first operand to automatically cast it from a string to an integer.

Instruction Execution and Result Storage

You've seen a generic method for resolving operands, so you're ready to move into the next phase of the instruction's implementation, which is the execution of its logic and the storage of its results. As you'll see, storing the results of an instruction is so simple it barely deserves its own phase, so it'll be almost implicitly mentioned from here on out.

The following sections each discuss the overall implementation of a major instruction family. I won't cover how every last member of the XVM instruction set works, but understanding the following will give you enough knowledge to implement the rest on your own. Of course, the XVM prototype source code is also available on the accompanying CD, which contains a full implementation of all instructions. As a friendly reminder, don't forget to check it out!

Lastly, I'd just like to point out that you're almost done here—your VM is capable of quite a few things, is heavily structured, and is ready to move forward with actual instructions. You've worked your way through some of the more mundane planning phases, and have worked your way down to the heart of it all. Instruction implementations really are the soul of a virtual machine, so keep that in mind as you read.

Mov

Let's get things started by taking a look at the quintessential instruction. `Mov` embodies virtually everything the average instruction does—it accepts operands, performs logic, and produces output. It's an incredibly simple and generic instruction by nature, however, which makes it the perfect jumping-off point. Besides, `Mov` is generally the most commonly used instruction in assembly language programming, so it's always the one you should be most familiar with.

I almost feel kinda bad after that long-winded build-up, however, because the code behind `Mov` is nothing short of anti-climactic. In fact, I'll just shut up and show it to you:

```
case INSTR_MOV:
    // Mov   Source, Destination

    // Get a local copy of the destination operand (operand index 0)
    Value Dest = ResolveOpValue ( 0 );

    // Get a local copy of the source operand (operand index 1)
    Value Source = ResolveOpValue ( 1 );

    // Skip cases where the two operands are the same
    if ( ResolveOpPntr ( 0 ) == ResolveOpPntr ( 1 ) )
        break;
    // Copy the source operand into the destination
    CopyValue ( & Dest, Source );

    // Use ResolveOpPntr () to get a pointer to the destination Value
    // structure and move the result there
    * ResolveOpPntr ( 0 ) = Dest;

    break;
```

Figure 10.31 illustrates how `Mov` works.

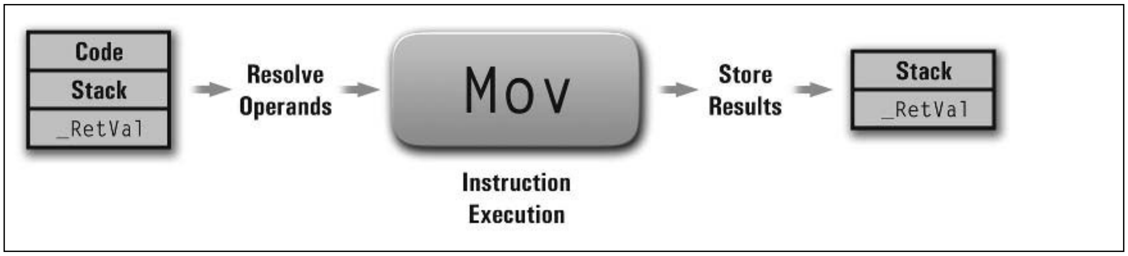


Figure 10.31

`Mov` in action.

Pretty simple, huh? All it does is the following:

- Resolves local copies of the source and destination operands.
- Uses `CopyValue ()` to safely write the source operand to the destination.
- Writes the destination back out to memory (either the stack or `_RetVal`) using the pointer returned by `ResolveOpPntr ()`.

Binary Operation Implementation

Immediately following `Mov` are the binary operation instructions, because they follow a similar pattern. This family of instructions includes arithmetic like `Add` and `Exp`, and bitwise operations like `And` and `XOr`. As you'll see, they follow a very similar pattern to `Mov` in that they accept source and destination parameters and place the result in the destination.

Also like `Mov`, their implementation is reasonably simple and tends to speak for itself. So, I'll once again step back for the moment and let the code do the talking. Check out the implementation of `Add`:

```
case INSTR_ADD:
    Add   Op0, Op1

    // Get a local copy of the destination operand (operand index 0)
    Value Dest = ResolveOpValue ( 0 );

    // Add the source to the destination
    if ( Dest.iType == OP_TYPE_INT )
        Dest.iIntLiteral += ResolveOpAsInt ( 1 );
    else
        Dest.fFloatLiteral += ResolveOpAsFloat ( 1 );

    // Use ResolveOpPntr () to get a pointer to the destination Value
    // structure and move the result there
    * ResolveOpPntr ( 0 ) = Dest;

    break;
```

Just about as easy, huh? The only difference between this and `Mov` is that it adds the source and destination rather than simply performing copying. Also, the addition is of course broken down by data type, since the final, raw values are not typeless like XtremeScript is.

The cool thing here is that all binary operation instructions follow the same format. The only place they differ is the actual operation itself. Because of this, however, you can end up with a lot of redundant code because the only major change you're making is a single-character operator. I generally like to condense all of the binary operation instructions into a single `case` and use another, larger switch to determine which operation to perform once the operands have been resolved. As you'll see in the XVM source, all of the binary instructions, from `Mov` to `XOr`, are implemented in a single instruction handler.

Figure 10.32 illustrates `Add`.

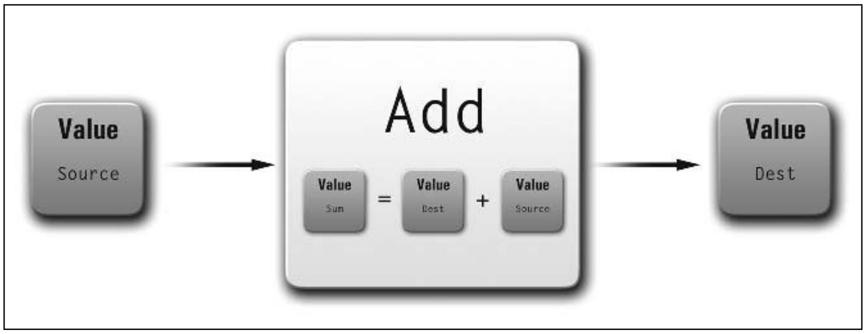


Figure 10.32

How `Add` works.

Conditional Branching Implementation

The jump instructions are a little bit different than Mov and the binary operations, but they're nothing you can't handle. To start things off, let's look at what's by far the simplest branch instruction, Jmp— the unconditional jump.

```
case INSTR_JMP:
{
    // Jmp    Label

    // Get the index of the target instruction (opcode index 0)
    int iTargetIndex = ResolveOpAsInstrIndex ( 0 );

    // Move the instruction pointer to the target
    g_Script.InstrStream.iCurrInstr = iTargetIndex;

    break;
}
```

Tough stuff, huh? Seriously, this is about as simple as instructions get. All we have to do is resolve the first operand (operand index 0) as an instruction index, and we immediately have the jump target. We then set IP to this value and our job is done.

Moving on, the complexity increases when you get into the conditional jumps. Like most instruction families, however, all conditional jumps are coded in the same way, so once you get one figured out the rest come easily. Here's the implementation for JE—jump if equal. As a quick refresher, this instruction compares two operands, Op0 and Op1, and jumps to a target instruction if their values are equal.

```
case INSTR_JE:
    // JE    Op0, Op1, Target

    // Get the two operands
    Value Op0 = ResolveOpValue ( 0 );
    Value Op1 = ResolveOpValue ( 1 );

    // Get the index of the target instruction (opcode index 2)
    int iTargetIndex = ResolveOpAsInstrIndex ( 2 );

    // Perform the specified comparison and jump if it evaluates to true
    int iJump = FALSE;
    switch ( Op0.iType )
    {
        case OP_TYPE_INT:
            if ( Op0.iIntLiteral == Op1.iIntLiteral )
                iJump = TRUE;
            break;

        case OP_TYPE_FLOAT:
            if ( Op0.fFloatLiteral == Op1.fFloatLiteral )
                iJump = TRUE;
            break;

        case OP_TYPE_STRING:
            if ( strcmp ( Op0.pstrStringLiteral, Op1.pstrStringLiteral ) == 0 )
                iJump = TRUE;
            break;
    }

    // If the comparison evaluated to TRUE, make the jump
    if ( iJump )
        g_Script.InstrStream.iCurrInstr = iTargetIndex;
    break;
```

Things are still pretty straightforward. The two operands are read in, and the data type of the first (`Op0`) is used as the basis for the comparison. You set a flag to `FALSE` beforehand that is only changed to `TRUE` if the comparison evaluates to equality. You then use this flag to determine whether to make the jump at the end of the instruction. Like I said, it's not hard to do and once you've got one conditional working, you can code the rest of them just as easily.

NOTE

It's true that pretty much all of the conditional jump instructions can be coded with the same basic framework, and in that regard, should probably be condensed into a single `case` like I mentioned in the section on binary operations. However, remember that only `JE` and `JNE` need to support strings; there's really no such thing as a string that's "greater than" or "less than" another string, so `JG`, `JL`, `JGE`, and `JLE` can be written to only work with numeric operands (integers and floats). Check the XVM source for more information on how jump implementations can be organized.

Function Call Implementation

After all you've seen, you may be under the impression that the implementation of your function call system will be right up there with the more complex aspects of your virtual machine. Fortunately, this is not the case. You've written such a powerful base of helper functions already for working with the stack and routing the flow of execution that `Call` and `Ret` will be borderline trivial. Besides, we've already been through virtually the entire function call and return process, so this is just an application of that material.

The implementation of function calls lies in two instructions: `Call` and `Ret`, which call and return from functions, respectively. The following two subsections explain these instructions' implementation.

Call

`Call` is actually a reasonably simple instruction. Remember, all it does is fill out the remaining components of the stack frame and make an unconditional jump to the function's entry point. The script itself will have already pushed the parameters onto the stack, so it's just up to you to push the instruction pointer's value as an integer (the return address) and use `PushFrame ()` to allocate the necessary space for local data. You then use `Jump ()` to enter the function.

Let's take a look at the code:

```
case INSTR_CALL:
{
    // Call    Func

   // Get a local copy of the function index
   int iFuncIndex = ResolveOpAsFuncIndex ( 0 );

    // Get the destination function's info
    Func DestFunc = GetFunc ( iFuncIndex );

    // Save the current stack frame index
    int iFrameIndex = g_Script.Stack.iFrameIndex;

    // Advance the instruction pointer so it points to the instruction
    // immediately following the call
    ++ g_Script.InstrStream.iCurrInstr;

    // Push the return address, which is the current instruction
    Value ReturnAddr;
```

```
        ReturnAddr.iInstrIndex = g_Script.InstrStream.iCurrInstr;
        Push ( ReturnAddr );

        // Push the stack frame + 1 (the extra space is for the function index
        // we'll put on the stack after it)
        PushFrame ( DestFunc.iLocalDataSize + 1 );

        // Write the function index and old stack frame to the top of the stack
        Value FuncIndex;
        FuncIndex.iFuncIndex = iFuncIndex;
        FuncIndex.iOffsetIndex = iFrameIndex;
        SetStackValue ( g_Script.Stack.iTopIndex - 1, FuncIndex );

        // Let the caller make the jump to the entry point
        g_Script.InstrStream.iCurrInstr = DestFunc.iEntryPoint;
        break;
    }
```

This instruction gives you the ability to call functions, and thanks to its use of the runtime stack, it has automatic support for nesting and recursion. Despite its incredible utility value, however, all its implementation took was a few calls to your helper functions. In that regard, building an otherwise complex instruction was just like snapping together a couple Legos. See how easy these helper functions have made things?

The instruction begins by reading a `Func` structure from the function table using the single operand as the index. Once you have this structure, you have the information you need to complete the stack frame and make the jump to the entry point. You then save the create a new `Value` structure, set its integer literal field to the current instruction pointer, and push it onto the stack. `Ret` will need this in order to find its way back to the caller. The next step is to get the target function's local data size and complete the stack frame by allocating a contiguous region of space for it with a call to `PushFrame ()`.

Before making the jump, however, you need to also save the function's index and the location of the current stack frame, which Ret will need later on. Once again, this is why your assembler always generated local data stack indices starting from -2; the element at index -1 contains this value which cannot be disturbed. Lastly, you finish it all up by using the function's entry point as the target for the jump. Figure 10.33 depicts an XVM stack frame.

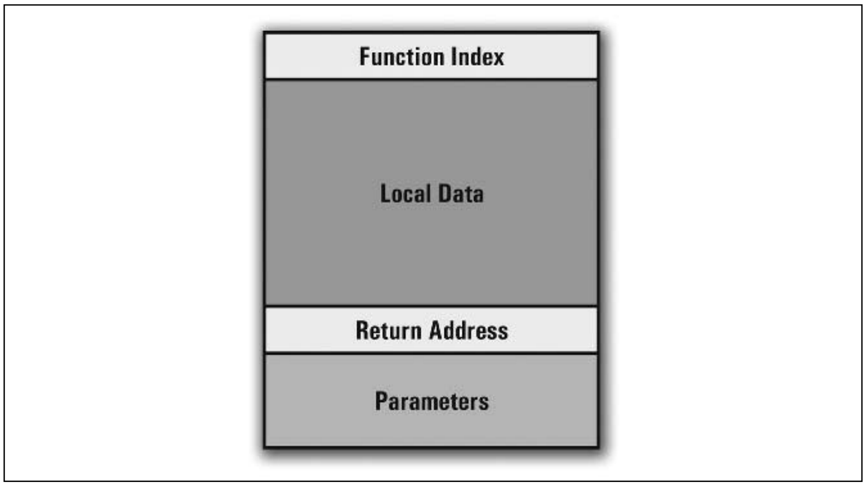


Figure 10.33

An XVM stack frame.

Ret

Of course, you don't want to strand your script inside a function. Once a Ret instruction is encountered by the VM, it's time to go home. Take a look at the implementation:

```
case INSTR_RET
    // Ret

    // Get the current function index off the top of the stack and use it to
    // get the corresponding function structure
    Value FuncIndex = Pop ();

    Func CurrFunc = GetFunc ( FuncIndex.iFuncIndex );
    int iFrameIndex = FuncIndex.iOffsetIndex;

    // Read the return address structure from the stack, which is stored one
    // index below the local data
    Value ReturnAddr = GetStackValue ( g_Script.Stack.iTopIndex -
        ( CurrFunc.iLocalDataSize + 1 ) );

    // Pop the stack frame along with the return address
    PopFrame ( CurrFunc.iStackFrameSize );

    // Restore the previous frame index
    g_Script.Stack.iFrameIndex = iFrameIndex;
    // Make the jump to the return address
    g_Script.InstrStream.iCurrInstr = ReturnAddr.iInstrIndex;

    break;
```

The instruction begins by popping the function table index off the top of the stack that `Call` placed there just before it invoked the function. Remember, this value *must* be on top of the stack when `Ret` is called, or else none of its logic will work. Because of this, functions must always remember to preserve the structure of the stack by popping everything they push. This index is required to complete the rest of the implementation; you need to know which function you're returning from in order to get its relevant information from the function table. This element also contains the previous frame index, which is saved as well.

Once you have the function structure, you can use the information it contains to locate the return address on the stack. The distance of the return address from the top of the stack is always the size of the local data, so you just use that size as a negative offset to obtain it. You then save the return address in a local integer variable.

The stack frame is then taken down: parameters, the return address, the local data, everything. This is done with a single call to `PopFrame ()`, using the `StackFrameSize` field of the `Func` structure. The stack's `iFrameIndex` is then restored to its previous value. The function no longer exists on the stack at this point, so all that's left to do is jump back to the caller using the return address you saved. Figure 10.34 sums up the logic behind `Ret`.

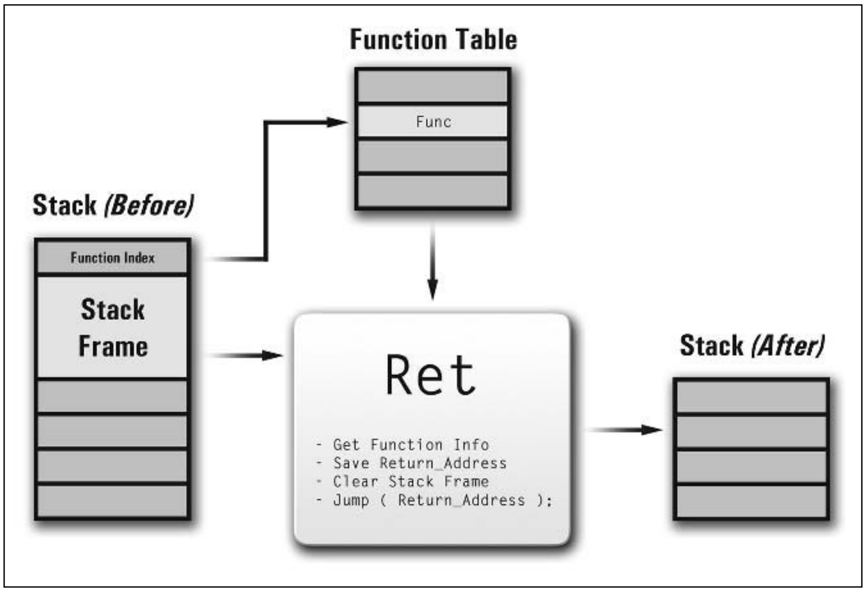


Figure 10.34

The logic behind `Ret`.

Pause Implementation

The last instruction I want to take a look at is `Pause`, because it has more of an effect on the main loop of the virtual machine than the other functions. Once `Pause` is called, the execution cycle will ignore the current instruction until the pause duration has elapsed. Here's the implementation:

```
case INSTR_PAUSE:
{
    // Pause    Duration

    // Get the pause duration
    int iPauseDuration = ResolveOpAsInt ( 0 );

    // Determine the ending pause time
    g_Script.iPauseEndTime = iCurrTime + iPauseDuration;

    // Pause the script
    g_Script.iIsPaused = TRUE;

    break;
}
```

We've already seen how the VM's execution cycle handles script pauses, so we're good to go; executing this instruction will cause the script's activity to halt for the specified duration, but in a synchronous manner that doesn't cause the overall program to stall in an empty loop.

The Rest

I haven't covered every instruction here, but you're by no means on your own. The first and most important thing to do is check out the XVM prototype source. This contains a working implementation of *every* instruction, as well as full commenting, so that alone should be all you need. But even without that, the techniques and principals you've already learned will provide enough of a foundation to implement anything. Remember, once you've resolved your operands, the implementation of an instruction can basically be thought of as writing a function. Just code the logic while taking the operands into account and you're done.

Termination and Shut Down

There's not a whole lot to say about the termination phase, because it's pretty easy in the XVM prototype. Currently, the script will run until an `Exit` instruction is processed, or until a key is pressed.

The only real job left at this point is to free the dynamically allocated data structures. This includes the following:

- The instruction stream and each instruction's operand list.
- The runtime stack.
- The function table.
- The host API call table.

Note that some structures like the script header can be ignored in this phase due to their static allocation.

One major caveat here is the freeing of string literals. Remember, between the stack and the instruction stream, you've got a significant amount of strings allocated that all need to be individually released. Failure to do this will eat up memory extremely quickly. The strategy for handling these strings is simple; first, scan through the instruction stream and check the operand type. Anything set to `OP_TYPE_STRING` contains a string that must be freed. The same goes for the stack.

The following is the implementation of `ShutDown ()`, an XVM prototype for freeing the script's resources:

```
// ---- Free The instruction stream
// First check to see if any instructions have string operands, and free them
// if they do

for ( int iCurrInstrIndex = 0; iCurrInstrIndex < g_Script.InstrStream.iSize;
    ++ iCurrInstrIndex )
{
    // Make a local copy of the operand count and operand list
    int iOpCount = g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].iOpCount;
    Value * pOpList = g_Script.InstrStream.pInstrs [ iCurrInstrIndex ].pOpList;

    // Loop through each operand and free its string pointer
    for ( int iCurrOpIndex = 0; iCurrOpIndex < iOpCount; ++ iCurrOpIndex )
        if ( pOpList [ iCurrOpIndex ].pstrStringLiteral )
            pOpList [ iCurrOpIndex ].pstrStringLiteral;
}

// Now free the stream itself
if ( g_Script.InstrStream.pInstrs )
   free ( g_Script.InstrStream.pInstrs );

// ---- Free the runtime stack
// Free any strings that are still on the stack
for ( int iCurrElmtnIndex = 0; iCurrElmtnIndex < g_Script.Stack.iSize;
    ++ iCurrElmtnIndex )
    if ( g_Script.Stack.pElmnts [ iCurrElmtnIndex ].iType == OP_TYPE_STRING )
        free ( g_Script.Stack.pElmnts [ iCurrElmtnIndex ].pstrStringLiteral );

// Now free the stack itself
if ( g_Script.Stack.pElmnts )
   free ( g_Script.Stack.pElmnts );

// ---- Free the function table

if ( g_Script.FuncTable.pFuncs )
   free ( g_Script.FuncTable.pFuncs );

// --- Free the host API call table

// First free each string in the table individually
for ( int iCurrCallIndex = 0; iCurrCallIndex < g_Script.HostAPICallTable.iSize;
    ++ iCurrCallIndex )
    if ( g_Script.HostAPICallTable.ppstrCalls [ iCurrCallIndex ] )
        free ( g_Script.HostAPICallTable.ppstrCalls [ iCurrCallIndex ] );

// Now free the table itself
if ( g_Script.HostAPICallTable.ppstrCalls )
   free ( g_Script.HostAPICallTable.ppstrCalls );
```

Summary

You're on your way now, my young Padawan. The XVM prototype you built in this chapter marks the first time you've successfully executed your own bytecode, which means you're on the threshold of a finished, working virtual machine. Of course, I've left out all the *real* fun, like multithreading and communication with the host application. But worry not, because they're the focus of the next chapter.

That's right, by the end of the next chapter, you'll be two thirds of the way through this quest for enlightenment of yours, bringing you ever closer to scripting mastery. Out of the compiler, assembler, and virtual machine, the last two components will be finished and ready to go. The next chapter will see you through the completion of the XVM, which will be nothing short of awesome, and will give you plenty of power to play with for a while. The finished XtremeScript Virtual Machine will be a fast, powerful, and best of all, multithreaded virtual machine that can communicate easily with the host application.

Once the next chapter is finished, ending this section of the book, you'll be rounding the home stretch and find yourself hip-deep in the ultimate test: compiling the high-level XtremeScript scripting language. As you build the XtremeScript compiler, you'll use the tools you've developed here—XASM and the XVM—to test and examine its output. As you'll see, the order in which you're developing the system's components (the assembler, and then the VM, and then the compiler) will come in quite handy.

On the CD

The XVM prototype is available on the CD, all greased up and ready to go. Check it out in the `Programs/Chapter 10/XVM Prototype/` directory. As always, it's available in both source and executable form, so you can play with it right away and browse the code at your leisure.

Like XASM, the XVM Prototype is a simple console application which makes things very easy. Simply load the workspace file into Visual C++ and build. The only snag this time is that `GetCurrTime ()` is implemented in a Win32-specific way, so users of other platforms will have to replace the Win32 API calls with corresponding ones from their own platform. All that's necessary is any function that returns the current time in milliseconds, so it shouldn't be too much of an issue.

Challenges

- ***Easy:*** Add more output information for each instruction; for example, arithmetic instructions could be printed with both operands, the operator they represent, and the resulting value.
- ***Intermediate:*** This one relates to the easy challenge from the last chapter. Implement the new instructions you added to XASM and see if you can get them to actually function. The example instructions I suggested where `Sqrt` (for computing square roots), `RoL` (for rotating bits to the left), and `RoR` (for rotating bits to the right).
- ***Difficult:*** Using a graphics API of some sort (like DirectX, or the Wrapppuh API provided with this book), write a graphical-front end for the VM that displays a constantly updated memory map (showing the stack, `_RetVal` and the instruction stream) that allows you to watch the exact behavior of the script as it executes, visually. Finishing this challenge would actually leave you with a powerful low-level debugger.

This page intentionally left blank

CHAPTER 11

Advanced VM Concepts and Issues

"After Fleet gasses the planet, M.I. mops up."
——*Lieutenant Rasczak,* Starship Troopers

It's *on* now. Chapter 10 introduced you to the design and implementation of a virtual machine's core logic, and now you're going to finish the job by adding the much-needed features that will allow your runtime environment to fully integrate itself with a game engine. By the time this chapter is through, the XtremeScript Virtual Machine (XVM) will be finished and ready to go. From there, all that will remain is the design and implementation of the high-level XtremeScript compiler. Throughout the development of that final project, you'll have the XVM to test your results at every step of the way. This should help you understand why you're developing the scripting system's components in this order.

In this chapter, you're going to

- Add the ability to run multiple scripts concurrently, in a priority-based multithreaded environment.
- Add functions for fully integrating the virtual machine with the host application, allowing scripts to call game engine functions and vice-versa.
- Discuss other VM issues, such as basic security and porting.

A Next Generation Virtual Machine

The virtual machine developed in Chapter 10 was definitely a worthwhile project. It could load formatted .XSE executables, implement every instruction (except for `CallHost`), and was capable of running scripts in their entirety. The only real issues were that it couldn't handle more one script at a time, and was a standalone program—there was no way to embed it in a larger program and allow the two entities to easily communicate. This chapter will fill in these blanks, to create the next generation of the virtual machine.

Two Versions of the Machine

Throughout the course of this chapter, you're actually going to create two new virtual machines; the first will demonstrate the basics of multitasking, whereas the second will make a few small detail changes and introduce a host application interface. You can find both of these virtual machine versions on the accompanying CD in the `DIRECTORY_NAME_HERE` directory.

MULTITHREADING

The current VM is single-threaded, which means that only one script's bytecode can be executed at once. Furthermore, the runtime environment's internal structures only allow for a single script to be stored in memory at any given time, using the `g_Script` structure. However, because games are naturally based around large numbers of autonomous entities that all seem to move and exist in parallel, this VM will need the capability to both store and execute as many scripts at one time as the game demands, as shown in Figure 11.1.

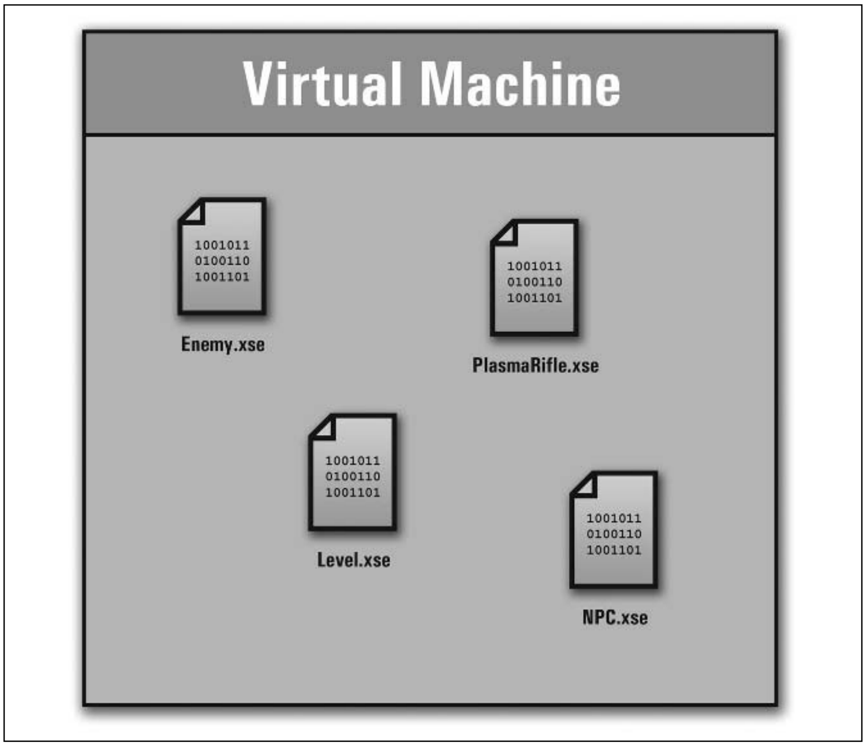


Figure 11.1

Most games require a large number of entities to exist concurrently.

You could add multithreading capabilities by directly using the threading system provided by Windows (or your OS of choice), but this would force the otherwise virtual and platform-neutral runtime environment into a platform-dependent solution. Furthermore,

NOTE

Using an operating system's built-in thread functionality, such as Windows threads, would have virtually no discernable advantage over the custom-built solution on a single-processor system, but it *would* transparently run faster on a multiprocessor system.

these scripts are extremely lightweight—so much so, in fact, that a custom-built threading system would be the best way to capitalize on their small footprints and maximize efficiency. Besides, actually implementing threads is a far better learning experience.

Multithreading Fundamentals

Let's start at the beginning. Virtually all operating systems these days are *multitasking* operating systems. This means that they can distribute the workload of multiple programs evenly across the system's speed and memory resources, and across multiple physical processors if available. In the case of single-processor systems, however, the concept of the multiple programs running in parallel is an illusion made possible by the sheer speed of today's processors. Naturally, a single processor machine can only do one thing at once, but by executing each running program for a *very* brief period of time, in sequence, the user will perceive concurrent execution. Figure 11.2 illustrates the process of running multiple tasks in simulated parallel.

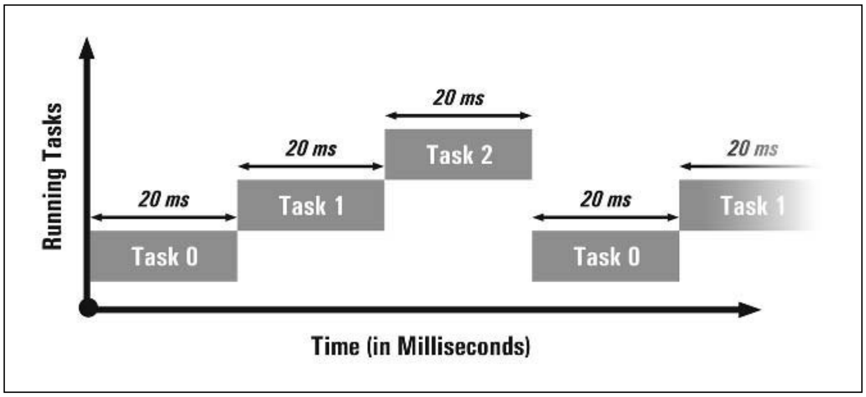


Figure 11.2

When each task runs in sequence for a very brief time, the sheer speed of the processor will make them seem concurrent.

Cooperative vs. Preemptive Multitasking

Generally speaking, multitasking can be implemented in one of two fundamental ways—*cooperative* or *preemptive*. In a cooperative multitasking system, like Windows 3.x, individual programs are allowed to run for as long as they feel is necessary before relinquishing control back to the operating system, and subsequently, to the next program waiting to run. For example, this means that a rendering program may choose to render an entire scanline or portion of an image each time the operating system gives it control, during which time the rest of the system is essentially frozen. When this process is complete, the operating system will move on to the next task, known as a *context switch*, which may be a text editor like notepad. This program, because it's obviously less intensive than the renderer, will probably just idle for a few milliseconds to give the users a chance to input some text, and immediately let the operating system once again switch tasks. The problem with the cooperative approach is that it relies on programs to govern themselves. If you've ever read *Lord of the Flies,* you know this can only end badly. Figure 11.3 displays the uneven behavior of a cooperative multitasking system.

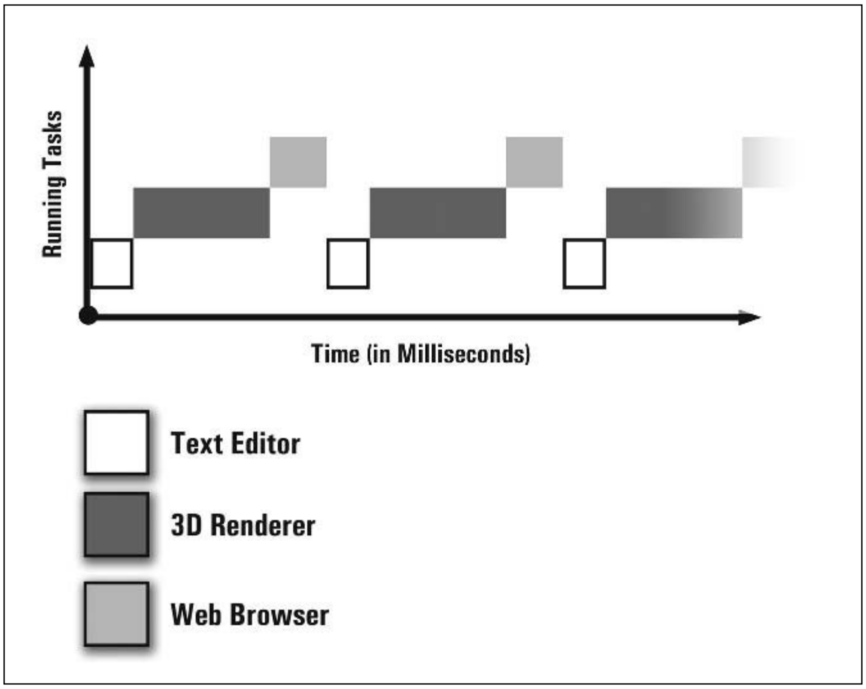


Figure 11.3

Cooperative multitasking leads to an uneven distribution of processor time.

NOTE

The term *context switch* comes from the fact that in a real hardware system, the currently active task must be saved before the next one can be invoked. This means storing the status of each register, along with the tasks instruction pointer and stack pointers. This information is vital—the thread can't be restored without it. This information—the registers, instruction and stack pointers, and so on—is known as a *context*, because it more or less defines the state of the system at a given moment. Therefore, switching from one task to another means switching the context. Fortunately, in the case of the XVM, you don't have to worry about this quite as much. Because the `Script` structure automatically stores all of this information for you, a script's context is implicitly saved at all times.

Because of this lack of equality among tasks, a cooperative multitasking system tends to lag and feel noticeably uneven. This is brought on by the fact that each program in memory can potentially run at wildly varying intervals, resulting in certain programs with perfect responsiveness and others that feel sluggish and jerky. This issue is significant when dealing with business applications, but it's completely unacceptable when writing a game. Games need a liquid-smooth consistency that maintains the players' suspension of disbelief, constantly reassuring their subconscious that they're visiting a convincing alternate reality. Games need to mimic the real-world's ability to run everything within it at a constant rate—just because a powerful car drives by your house doesn't mean that your pets suddenly slow down or start skipping. Figure 11.4 illustrates the even thread execution a game requires.

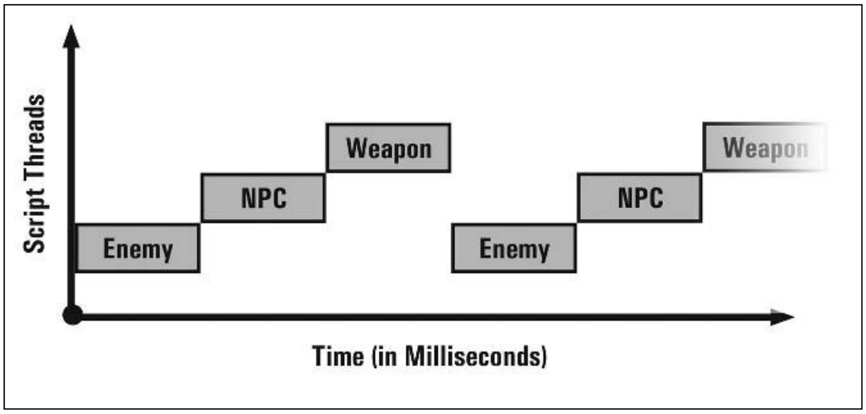


Figure 11.4

The smooth and even thread execution a game requires.

Preemptive multitasking solves this problem. Rather than allow programs to decide their own level of importance, the OS distributes very small, nearly uniform *time slices* among all running tasks. A time slice is a very brief period of time, usually measured in milliseconds, that ensures that all tasks will be evenly distributed across the processor's capabilities. Within a preemptive system, *priorities* can be assigned to tasks that increase or decrease their time slice, giving them a relative advantage or disadvantage based on their importance. This allows for a more intelligent distribution of processor power, because certain programs inevitably require more than others. Of course, priorities are specifically designed to be subtle; only over time will a higher or lower priority task appear to run at a different rate than others. This allows a preemptive system to maintain its smooth flow of execution while still providing more power to programs that need it and less to those that don't.

This approach to priorities varies the size of certain tasks' time slices, but doesn't affect the order in which they execute. Assuming the system is currently running four tasks, numbered 0 to 3, the system will always run the tasks in order, like this:

```
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

This is known as *round-robin scheduling*, because each thread is executed in the same sequence every time, as illustrated in Figure 11.5. The mechanism within the operating system that manages context switches among tasks and threads is known as the *scheduler*.

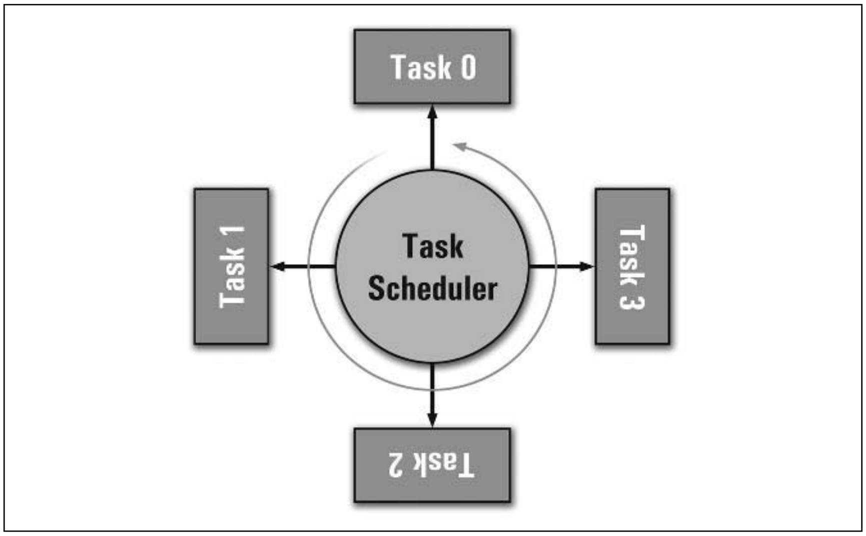


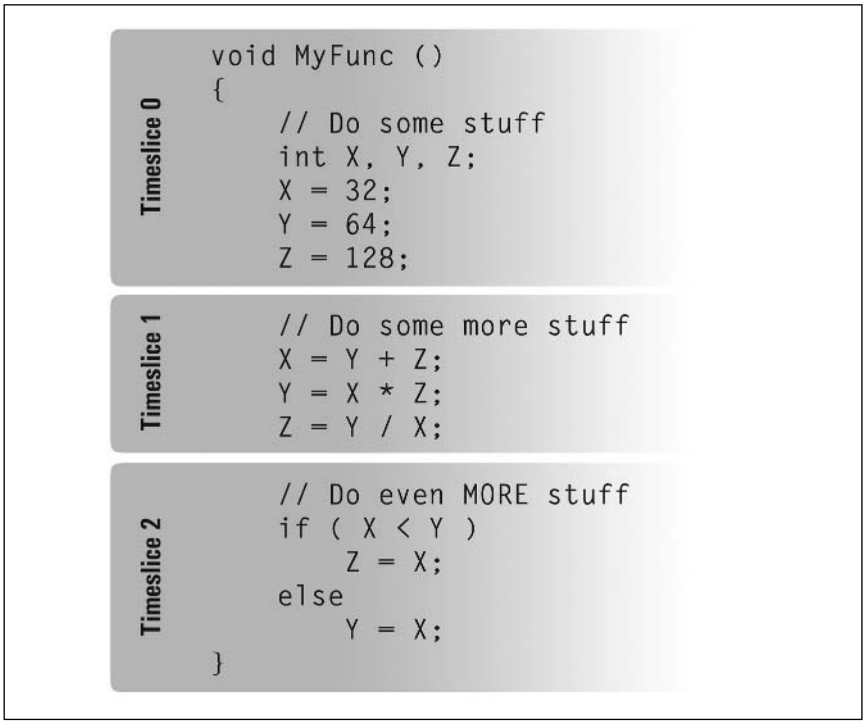
Figure 11.5

Round-robin time slice scheduling.

NOTE

The actual definition of a task's priority can vary. Some implementations may define priorities as I have here—an increase or decrease in the allotted time slice that gives a task more or less time to do its job than others. Other implementations may give all tasks the same time slice and instead vary the *frequency* at which a task is given control based on its priority. In this case, high priority tasks may execute multiple times during an interval in which all other tasks only run once. No matter how you approach the problem however, the overall result is the same—high priority tasks are capable of accomplishing more in a shorter time.

It's important to understand that a time slice is in no way a guarantee that the program will get a chance to finish what it's doing before the next context switch occurs. In fact, programs rarely start and finish even small tasks within their allotted time slices; rather, it's the norm for programs to be constantly interrupted by context switches. Of course, multitasking systems are designed to be transparent to everyone but the scheduler, meaning the program never actually knows it's being interrupted. Figure 11.6 illustrates how a single function or procedure can be transparently broken into multiple time slices.



```
void MyFunc ()
{
    // Do some stuff
    int X, Y, Z;
    X = 32;
    Y = 64;
    Z = 128;

    // Do some more stuff
    X = Y + Z;
    Y = X * Z;
    Z = Y / X;

    // Do even MORE stuff
    if ( X < Y )
        Z = X;
    else
        Y = X;
}
```

Figure 11.6

A single function can be executed over the course of multiple time slices without the program's knowledge.

From Tasks to Threads

Multitasking is great, but modern applications need even more flexibility from the operating system. Just as the OS can split itself up into multiple programs, many of these programs need the capability to further split *themselves* up into concurrently executing chunks. These are known as *threads*, and are shown in Figure 11.7.

Because the VM will ultimately integrate itself with the host application to form a complete game, you can consider the game as a whole to be a single operating system task. Within this task, however, multiple scripts need to coexist and appear to run in parallel. This is why the formerly singular game then needs to be split into multiple threads of execution. One thread will be set aside for the game loop, and the rest will be divided among the currently loaded scripts. By assigning fine-grained time slices to these threads, the game engine and each of its scripts will appear to run at the same time. The result is a game engine with direct support for fully autonomous entities that manage their own behavior.

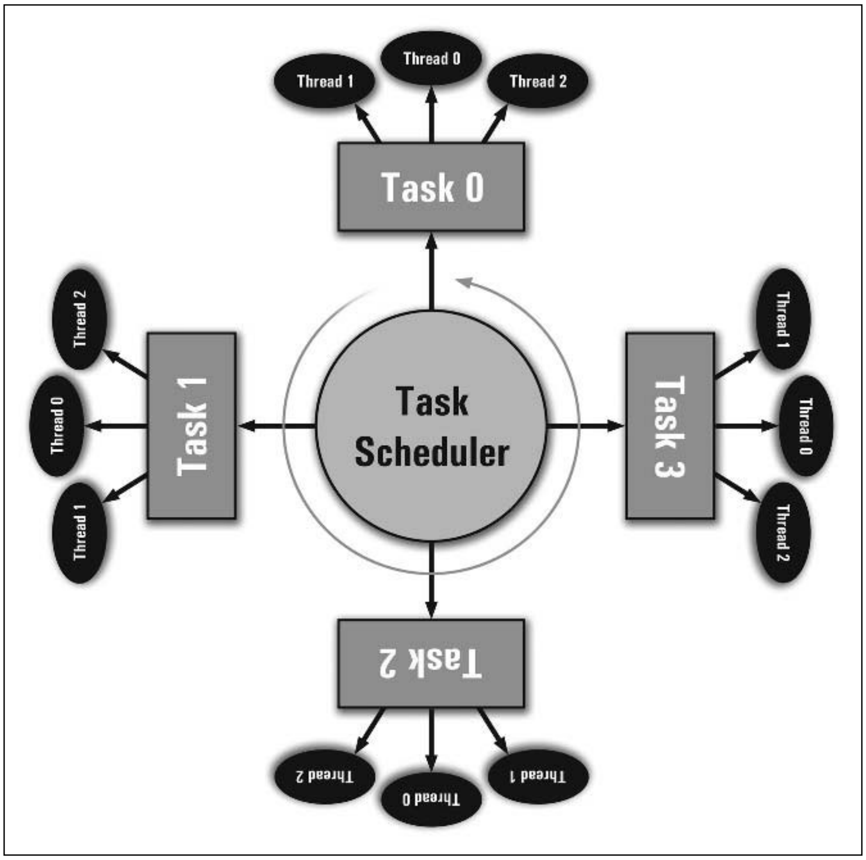


Figure 11.7

Within each task, individual threads can be spawned that further divide the allocated processing time.

Concurrent Execution Issues

Despite its obvious utility value and necessity for game development, multithreading is a technology that brings with it a number of serious issues and caveats. Just as roommates sharing a single bathroom and refrigerator tend to get in each other's way, threads that share common or global data run a significant risk of stepping on one another's toes and causing problems for the system as a whole. The inherent issues involved with multiple threads sharing common resources like data, input devices and so on, all fall under the topic of *synchronization.* The following sections are provided to quickly bring you up to speed on the key concepts behind thread synchronization, starting with the crux of the matter—race conditions.

Race Conditions

Games consist of huge amounts of data. Aside from raw media like sprites, textures, sounds, and 3D meshes, games do huge amounts of bookkeeping, ranging from the location of enemies within the game world to the player's statistics like the amount of damage the ship has taken or how much ammo is left in the sniper rifle. All of this data is vital to a game's execution—if the player's on-screen Y-location were to suddenly jump 400 pixels, for example, it would have a significant effect on the game's overall playability.

Naturally, threads will need to access and modify this data, and on a frequent basis. A script responsible for controlling a player-tracking enemy will need to constantly access both the player's and enemy's X, Y position, whereas another script designed to handle an in-flight rocket will need to constantly monitor and update the weapon's velocity and location. The situation I'm describing here is one in which multiple scripts share common data. This is where synchronization becomes a top priority for the threading system. Check out Figure 11.8.

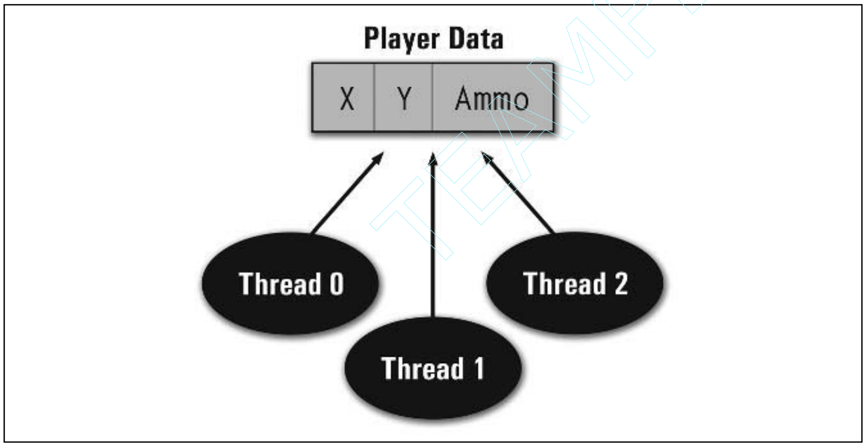


Figure 11.8

Multiple threads sharing common data.

Imagine if, within the same frame, two threads attempt to read and modify the player's on-screen X, Y location. Because each thread runs for a brief time slice wherein the context switch will almost invariably interrupt whatever operation is currently being performed, it won't be long before one thread's modification of the shared data is only partially complete when the next thread is invoked. The second thread will now be working with partially updated data because the first thread hasn't yet finished its job—a serious problem known as *data corruption*. Simply put, data corruption becomes a risk whenever two or more threads attempt to operate on the same data, an event known as a *race condition*. Figure 11.9 demonstrates data corruption over the course of three time slices.

Race conditions are analogous to multiple users on a network attempting to modify the same file. If each user were free to do whatever he or she liked at any time, the file would soon become heavily corrupted by partial modifications that were interrupted by other users' requests and changes. Because of this, networked operating systems enforce strict file sharing rules, wherein only one user can have a file open at one time. Although it's fine for multiple users to *read* from a file simultaneously, a file can only be open for writing by one user at once.

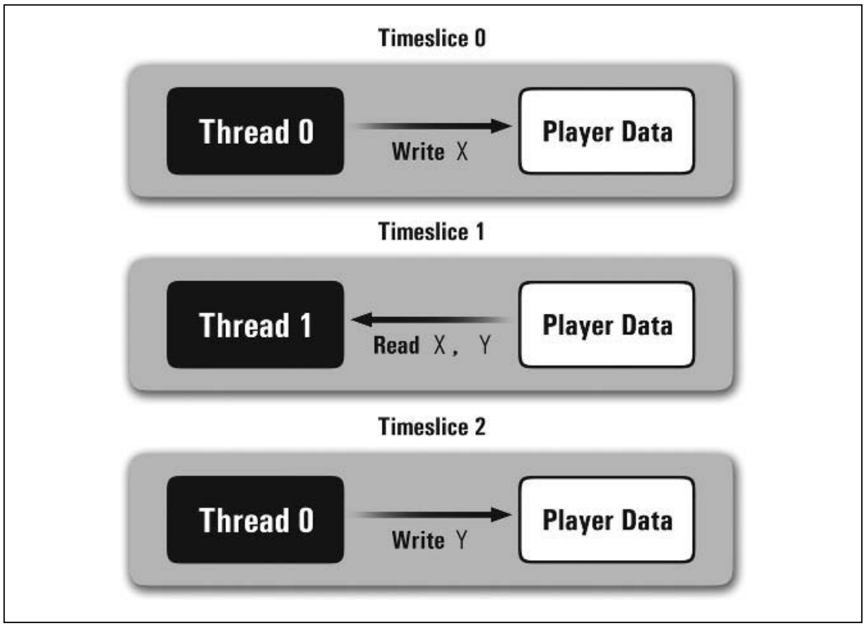


Figure 11.9

Data corruption at work.

Atomic Operations

One approach to the problem presented by race conditions is to wrap all modifications of shared data in *atomic operations*. An atomic operation is a block of code that is guaranteed to execute in full without fear of a context switch occurring. Atomic operations are implemented in many ways, varying from one platform to the next, but I'll discuss a highly simplified approach to better illustrate the concept.

Imagine that the following block of generic code is a script running in a virtual machine with direct access to the game's player data. If the script wanted to update the player's X, Y location, the code might look like this:

```
g_Player.iX += iXDiff;   // Add the X-axis differential
g_Player.iY += iYDiff;   // Add the Y-axis differential
```

As long as this script runs on its own, everything should be fine. Imagine introducing another script, however, that runs in parallel to the first and tracks the player by moving an enemy closer to the player's location at each frame. Here's how it might look:

```
// Move the enemy closer on the X-axis
if ( g_Enemy.iX < g_Player.iX )
    ++ g_Enemy.iX;
```

```
if ( g_Enemy.iX > g_Player.iX )
    -- g_Enemy.iX;

// Move the enemy closer on the Y-axis
if ( g_Enemy.iY < g_Player.iY )
    ++ g_Enemy.iY;
if ( g_Enemy.iY > g_Player.iY )
    -- g_Enemy.iY;
```

With these two threads running concurrently, it won't be long before they slip out of sync (if they're even in sync to begin with, which is unlikely). When this happens, the comparisons and updates made by the enemy's script will take place after only partial updates are made to the player's position, which can result in all sorts of imperfections in the enemy's ability to smoothly track the player. The enemy may end up making too many comparisons to partially updated player data, resulting in jagged and overcorrected movement.

The problem is that the tasks performed by these scripts must be executed in full, regardless of context switches. Each script must be sure that the other was able to finish its job, resulting in completely updated data to use as the basis for its own purposes. Imagine now that this generic language offers an `atomic` keyword that can mark entire blocks of code as atomic operations. Here's the updated version of the first script:

```
atomic
{
    g_Player.iX += iXDiff;    // Add the X-axis differential
    g_Player.iY += iYDiff;    // Add the Y-axis differential
}
```

And here's the second:

```
atomic
{
    // Move the enemy closer on the X-axis
    if ( g_Enemy.iX < g_Player.iX )
        ++ g_Enemy.iX;
    if ( g_Enemy.iX > g_Player.iX )
        -- g_Enemy.iX;

    // Move the enemy closer on the Y-axis
    if ( g_Enemy.iY < g_Player.iY )
        ++ g_Enemy.iY;
    if ( g_Enemy.iY > g_Player.iY )
        -- g_Enemy.iY;
}
```

The scripting system knows now that both of these blocks are critical to the integrity of the game engine's data overall and will allow them to run in full before a pending context switch can take effect. Figure 11.10 illustrates atomic operations.

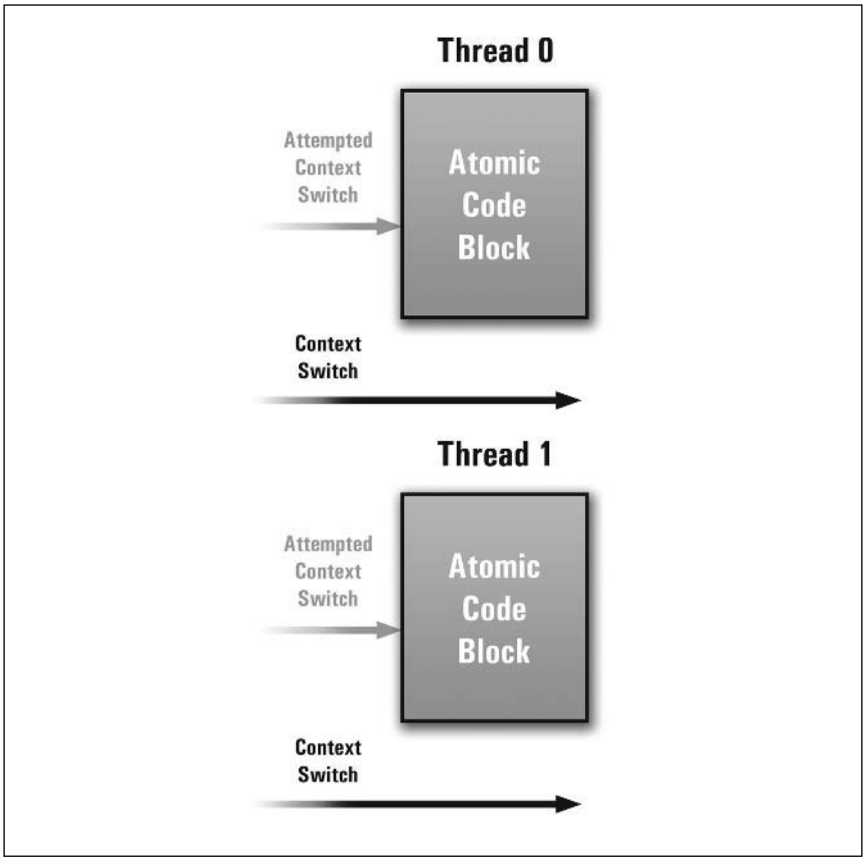


Figure 11.10

Atomic operations allow code blocks to execute in full before the context switch takes effect.

Critical Sections

In the previous examples, the two scripts both attempted to access a shared resource—in this case, the player's X, Y location within the game world—and are therefore examples of a *critical section.* A critical section is the sum of all code blocks across all scripts that attempt to access the same resource. Because shared resources cannot be modified by multiple threads at once, a critical section must enforce a *mutual exclusion.* Even though there were two separate blocks of code in the last example, neither of them can be active at the same time as the other. If there were three such blocks in the example, two of them would have to remain inactive while the third was performing its operation. No matter how many blocks of code attempt to access a single shared resource, they're all part of the same critical section and therefore cannot run in parallel with one another. This is demonstrated in Figure 11.11.

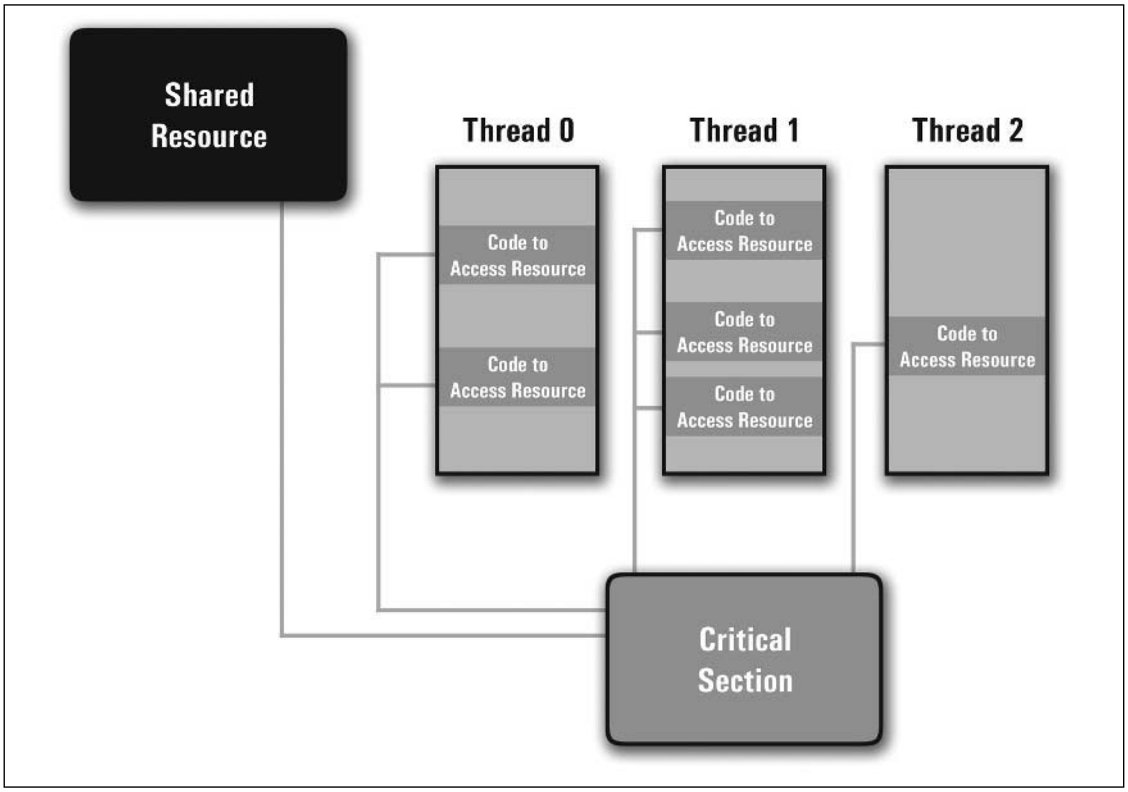


Figure 11.11

A critical section.

Mutexes

A *mutex* is a simple way to regulate critical sections. The term "mutex" is an abbreviation of "**Mut**ual **Ex**clusion", which is exactly what it provides. When a mutex is applied to a critical section, it can be guaranteed that no thread will enter the section at the same time as another.

A mutex is really just a globally defined flag that is accessible from all scripts and is associated with a particular critical section. Whenever a thread attempts to access a shared resource, an operation that takes place within its particular part of the critical section, this flag is read. If it's clear, the thread sets the flag and begins its operation. During this time, context switches will regularly occur and interrupt the thread with the time slices of other threads. These other threads may themselves attempt to access the same resource, and therefore will enter their own parts of the critical section. They too will check the mutex flag, which will now be set. Whenever the flag is set, the thread that's attempting to access it will enter an empty loop and wait until the flag is cleared before entering. When all threads adhere to this policy, the shared resource will never be accessed by more than one thread at a time.

Let's look at an example of using a mutex with the first script in the previous example:

```
// If the mutex is currently locked, wait until it's unlocked
while ( g_iPlayerMutex )
    ;

// Now lock the mutex so other threads
// won't access the resource
g_iPlayerMutex = TRUE;

// Modify the shared resource safely
g_Player.iX += iXDiff;    // Add the X-axis differential
g_Player.iY += iYDiff;    // Add the Y-axis differential

// Unlock the mutex to restore access to the resource
g_iPlayerMutex = FALSE;
```

Astute readers may have already noticed a flaw in this approach, however. The actual process of checking the status of the mutex and locking it can itself be interrupted by a context switch, which would invalidate the whole process. It's important to remember that even locking and unlocking a mutex can be easily interrupted and therefore must be treated as an atomic operation. Because of this, the actual implementation of mutexes is done on the OS level——at the same level as the scheduler—where it can be ensured that mutex operations will be performed without interruption. Check out Figure 11.12 for a visual explanation of a mutex.

Semaphores

Semaphores are like mutexes, but are designed to support an aggregate of generic resources as opposed to just one. I say generic because semaphores are used when multiple copies of a resource are available, and it doesn't matter which thread uses which copy as long as only a certain number of threads are allowed access at once. In other words, the only difference between a

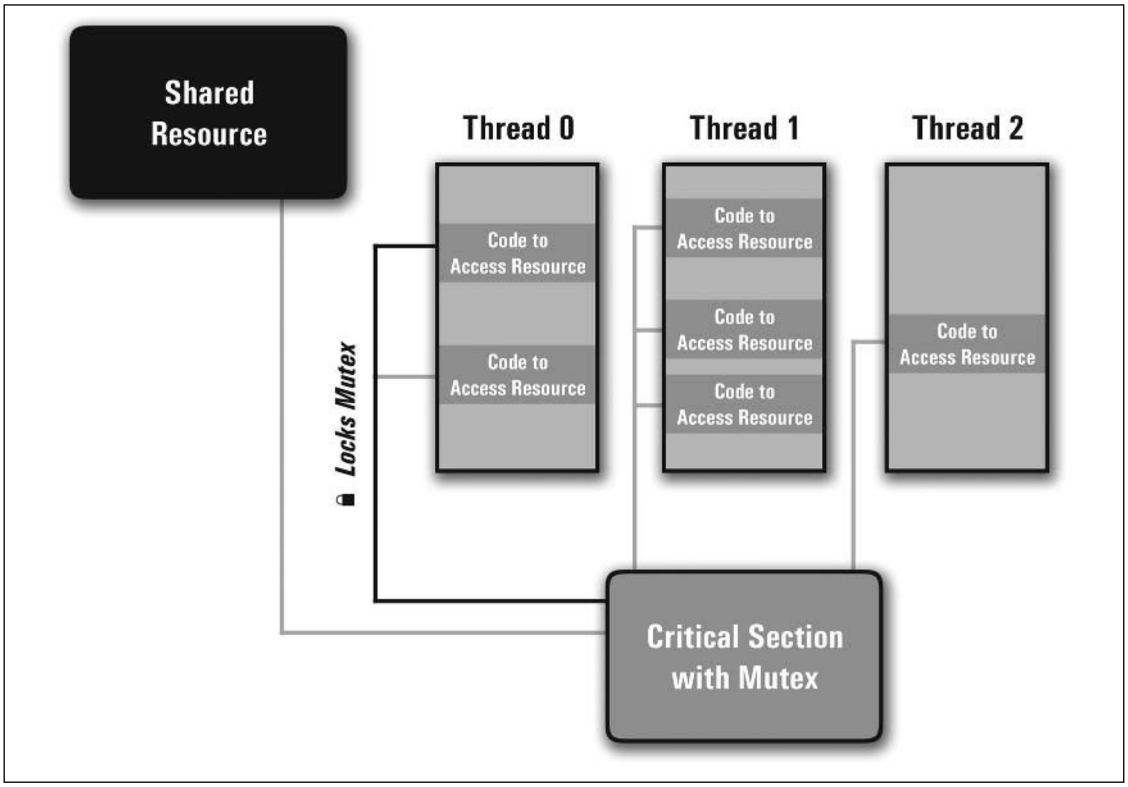


Figure 11.12

Once a mutex is locked by any of the blocks in a critical section, all other blocks must wait until it's unlocked before they can access the resource.

semaphore and a mutex is that a mutex treats a resource as either locked or unlocked, thereby allowing only a single thread access to a resource at one time. A semaphore, on the other hand, lets a specific number of threads access the resource concurrently before it denies subsequent requests. Because of this, mutexes are often known as *binary semaphores.*

Race Conditions in the XVM

As you'll see later in this chapter, race conditions won't be a particularly serious issue in the XVM because scripts can't share data. Through the host API, however, it will become possible for multiple scripts to attempt to change game engine data concurrently, which can result in race conditions. I'll revisit this issue later in the chapter.

Loading and Storing Multiple Scripts

Now that you have a basic understanding of the concepts behind multithreading, it's time to get back to reality. Before I get into the serious stuff, I still have to address the basic issue of loading and storing multiple scripts at once. All the multithreading theory in the world won't matter if you can't even get more than one script into memory at one time, so expanding the XVM's architecture is an important first step.

The `g_Script` Structure

The main reason you can't load more than one script at one time is because you're only declaring a single `g_Script` structure. The obvious solution, then, is to replace this with an array or linked list of `g_Scripts`, right? The question is, which type of aggregate structure is best?

Arrays or Linked Lists?

Scripts can be internally stored using any number of structures. But the question is, does the structure need to be dynamic? If the answer is no, you can slap in a static array and be done with it. You should be careful in answering this question, however.

Many programmers these days would simply go with a linked list because it theoretically offers improved flexibility by supporting virtually unlimited numbers of elements and never using more memory than it needs. Arrays, on the other hand, are just the opposite—they can only support a fixed number of elements and are often using far more memory than is necessary to store a quantity of items that is well below its limit.

Of course, the attitude that complex structures are always better than simpler ones can get you in a lot of trouble, so let's look at the facts. Storing your scripts in a linked list offers the following advantages:

- The ability for the game engine to load a virtually limitless number of scripts, resulting in maximum flexibility—especially for games with lots of separate entities.
- Efficient memory usage wherein script structures are allocated and freed on the fly to adjust to the number of scripts in memory at the moment.

Of course, it also suffers from the following disadvantages:

- Slow random access times, because linked lists must be partially or fully traversed in order to reach specific elements.
- Increased implementation complexity.

Straight C arrays, on the other hand, offer the following advantages:

- Very easy implementation.
- Extremely fast and simple random or sequential access.

And, as expected, the following disadvantages:

- General inflexibility due to a limit being placed on the number of scripts that can theoretically be in memory at once.
- Inefficient memory usage that doesn't attempt to adjust allocated space to match or approximate its contents.

So what's it gonna be? Both approaches seem to make a strong case for themselves and against the other. I personally have to side with arrays on this one, however, as shown in Figure 11.13. Why? For starters, the `g_Script` structure is rather lightweight which means that even in the worst case scenario, a large static `g_Script []` array will really never be a "waste" of memory. To prove this, let's do some basic analysis. You can determine the total size of a single `g_Script` structure by adding up the respective sizes of each of its fields, as long as you assume a 32-bit Windows environment.

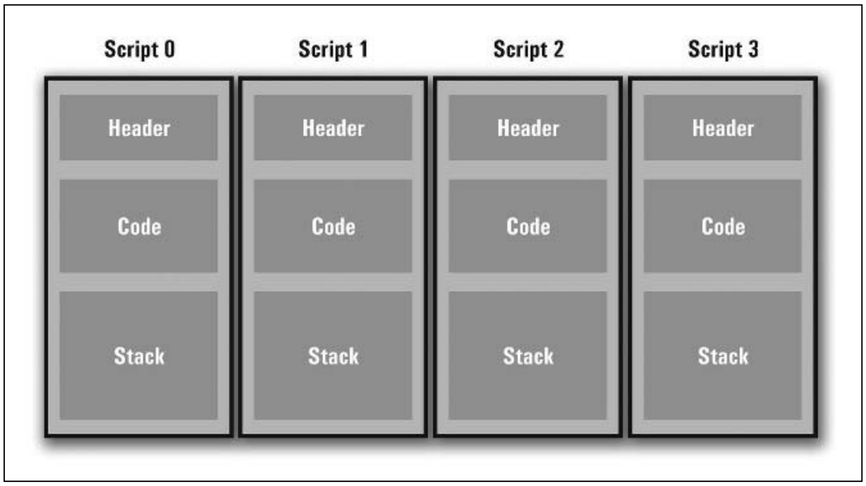


Figure 11.13

Storing multiple scripts in an array.

The `g_Script` structure looks like this:

```
typedef struct _Script          // Encapsulates a full script
{
    // Header data
    int iGlobalDataSize;        // The size of the script's global data
    int iIsMainFuncPresent;     // Is _Main () present?
    int iMainFuncIndex;         // _Main ()'s function index
    // Runtime tracking
    int iIsPaused;             // Is the script currently paused?
    int iPauseEndTime;         // If so, when should it resume?

    // Register file
    Value _RetVal;             // The _RetVal register

    // Script data
    InstrStream InstrStream;  // The instruction stream
    RuntimeStack Stack;       // The runtime stack
    Func * pFuncTable;        // The function table
    HostAPICallTable HostAPICallTable;   // The host API call table
}
    Script;
```

Right off the bat, you can see five ints, each of which occupies four bytes for an initial total of 20 bytes. The rest of the structure consists of other, nested structures, which will have to be added up individually. Let's start with the Value structure, of which the _RetVal field is an instance:

```
typedef struct _Value              // A runtime value
{
    int iType;                     // Type
    union                          // The value
    {
        int iIntLiteral;           // Integer literal
        float fFloatLiteral;       // Float literal
        char * pstrStringLiteral;  // String literal
        int iStackIndex;           // Stack Index
        int iInstrIndex;           // Instruction index
        int iFuncIndex;            // Function index
        int iHostAPICallIndex;     // Host API Call index
        int iReg;                  // Register code
    };
    int iOffsetIndex;              // Index of the offset
}
      Value;
```

iType and iOffsetIndex are both ints, starting you off at eight bytes. The union adds another four bytes (it's composed of 4-byte integers, a 4-byte float, and a 32-bit (4-byte) char pointer). This means the Value structure is 12 bytes, which, when added to the existing size of g_Script, takes the structure to a total of 32 bytes. Moving along, the InstrStream structure is next:

```
typedef struct _InstrStream   // An instruction stream
{
    Instr * pInstrs;          // The instructions themselves
    int iSize;                // The number of instructions in the
                              // stream
    int iCurrInstr;           // The instruction pointer
}
    InstrStream;
```

Two ints and a 32-bit pointer add up to another 12 bytes for this structure, thereby bringing g_Script from 32 to 44 bytes. Next up is the runtime stack:

```
typedef struct _RuntimeStack   // A runtime stack
{
    Value * pElmnts;           // The stack elements
    int iSize;             // The number of elements in the stack

    int iTopIndex;         // The top index
    int iFrameIndex;      // Index of the top of the current
                           // stack frame.
}
     RuntimeStack;
```

One 32-bit Value pointer plus three integers means 16 bytes in total for RuntimeStack, bringing g_Script up to 60 bytes. The function table is up next, but because it's just a single Func pointer, it only adds a single 32-bit pointer. g_Script is now 64 bytes. The last aspect of the structure is the host API table, which is defined as follows:

```
    typedef struct _HostAPICallTable  // A host API call table
    {
        char ** ppstrCalls;      // Pointer to the call array
        int iSize;               // The number of calls in the array
    }
        HostAPICallTable;
```

A 32-bit pointer to a pointer and the iSize integer field add up to eight bytes. This, being the last of g_Script's members, means the total size of an unused script structure is 72 bytes, which is nothing on today's machines. So you now know that a single unused script isn't going to make a noticeable difference in a game's available memory, but what about an entire array of them? To answer that question, it helps to have an idea of how many scripts your game will need active at once. Check out Table 11.1 to find out the total amount of memory required.

Table 11.1 Static `g_Script []` Array Sizes

Scripts	Size (in Bytes)	Size (in Kilobytes)
32	2304	2KB
64	4608	4.5KB
128	9216	9KB
256	18432	18KB
512	36864	36KB
1024	73728	72KB

And there you have it. For only 72KB, which isn't even a tenth of a megabyte, you can support up to *1024* scripts at once—more than enough for most games. So, the first moral of the story is that arrays will hardly waste memory. Secondly, 1024 script structures is huge, which is hardly limiting either. Chances are your game will never even approach that limit, so why worry about the "infinite expansion" of linked lists? With both the memory and flexibility issues debunked, it's safe to say that arrays are the way to go.

So, the first order of business is expanding `g_Script` structure to an array called `g_Scripts []`:

```
Script g_Scripts [ MAX_THREAD_COUNT ];
```

Of course, `MAX_THREAD_COUNT` can be set to anything you want; I've chosen 1024.

Loading Scripts

Now that you can store multiple scripts, `LoadScript ()` needs to be reworked enough to support this. Rather than pass `LoadScript ()` the index of `g_Scripts []` you'd like to load the script into, however, it'd be a nice touch if the function would automatically determine the next free script index, automatically use it, and return it to the caller (like in Figure 11.14). Of course, you're already returning an integer error code, so the index can't be directly returned. Rather, the function will accept an integer pointer and write the index to that. Here's the new prototype:

```
int LoadScript ( char * pstrFilename, int & iThreadIndex );
```

Aside from this change, there isn't much difference in the function's definition, minus the repeated use of `g_Scripts [ iThreadIndex ]` as opposed to `g_Script`.

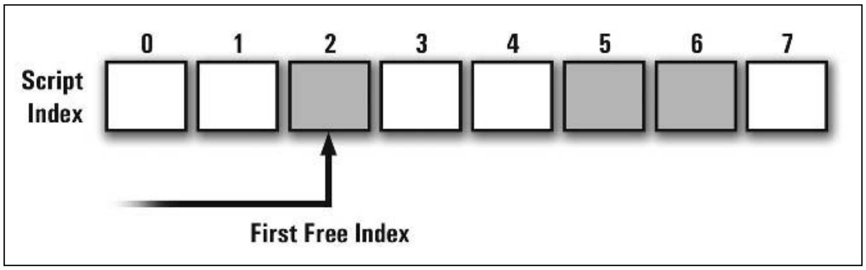


Figure 11.14

Determining the next free script index.

More Robust Error Handling

`LoadScript ()` has always returned an error code to the caller in the event that something went wrong, but has glossed over the potential memory allocation errors that can occur when using `malloc ()`. For the time being this wasn't an issue, but the XVM will soon be an embeddable module, and therefore have a public interface. A module's public interface should also feature robust error handling, especially in the case of memory allocation. Furthermore, it's entirely possible that the `g_Scripts []` array will become full, however unlikely, so an additional error code for this situation will be necessary as well. Once you build your virtual machine, it'll be nice to know that it'll run in any conditions and gracefully handle such problems by returning an error code for all contingencies. Besides, you never know—after developing your ultimate scripting system, you may want to make it publicly available like Lua and Python. In this case, stable error detection is a must.

This is accomplished by first creating new error code constants for memory allocation errors and a lack of available threads:

```
#define LOAD_ERROR_OUT_OF_MEMORY     4
#define LOAD_ERROR_OUT_OF_THREADS    5
```

Allocation error detection is simply a matter of checking the parameter returned by `malloc ()` to make sure it's not `NULL`. For example, the following block of code from the original `LoadScript ()`:

```
// Allocate the runtime stack
int iStackSize = g_Script.Stack.iSize;
g_Script.Stack.pElmnts =
    ( Value * )malloc ( iStackSize * sizeof ( Value ) );
```

Has been changed to:

```
// Allocate the runtime stack
int iStackSize = g_Scripts [ iThreadIndex ].Stack.iSize;
```

```
if ( ! ( g_Scripts [ iThreadIndex ].Stack.pElmnts =
    ( Value * ) malloc ( iStackSize * sizeof ( Value ) ) ) )
    return LOAD_ERROR_OUT_OF_MEMORY;
```

Note again the transition from `g_Script` to `g_Scripts []`. Let's now take a look at the code for determining the next free thread index:

```
// ---- Find the next free script index
int iFreeThreadFound = FALSE;
for ( int iCurrThreadIndex = 0;
    iCurrThreadIndex < MAX_THREAD_COUNT; ++ iCurrThreadIndex )
{
    // If the current thread is not in use, use it
    if ( ! g_Scripts [ iCurrThreadIndex ].iIsActive )
    {
        iThreadIndex = iCurrThreadIndex;
        iFreeThreadFound = TRUE;
        break;
    }
}
// If a thread wasn't found, return an out of threads error
if ( ! iFreeThreadFound )
    return LOAD_ERROR_OUT_OF_THREADS;
```

The process is simple; each element of the array is scanned to determine whether it's free. Upon encountering the first free index, the loop sets a flag indicating the find and breaks. Just outside the loop, the flag is checked to determine whether an index was found. If not, the `LOAD_ERROR_OUT_OF_THREADS` error code is returned. Otherwise, `iThreadIndex` contains the valid index and the loading procedure continues.

TIP

If you do plan on either releasing your scripting system for public use, or would just like to maximize its flexibility for your own use, it might be a good idea to dynamically allocate the `g_Scripts []` array, perhaps based on a parameter specified to the `Init ()` function. This allows the host to define the maximum number of scripts that can be loaded on a per-game basis without the need to recompile anything.

The rest of the source to `LoadScript ()` is the same as it was before the aforementioned changes, so I decided not to waste the space it'd take to print it here. You're encouraged to check out the source on the accompanying CD, however, in the `DIRECTORY_NAME_HERE` directory.

Initialization and Shutdown

In addition to `LoadScript ()`, it's now necessary to make some changes to the `Init ()` and `ShutDown ()` functions. Because these functions are primarily responsible for initializing the script structure to the proper default values and freeing it when the XVM exits, they'll have to be rewritten to work with the entire `g_Scripts []` array. Here's the new `Init ()`:

```
void Init ()
{
    // ---- Initialize the script array
    for ( int iCurrScriptIndex = 0;
        iCurrScriptIndex < MAX_THREAD_COUNT;
          ++ iCurrScriptIndex )
    {
      g_Scripts [ iCurrScriptIndex ].iIsMainFuncPresent = FALSE;
      g_Scripts [ iCurrScriptIndex ].iIsPaused = FALSE;
      g_Scripts [ iCurrScriptIndex ].InstrStream.pInstrs = NULL;
      g_Scripts [ iCurrScriptIndex ].Stack.pElmnts = NULL;
        g_Scripts [ iCurrScriptIndex ].pFuncTable = NULL;
      g_Scripts [ iCurrScriptIndex ].HostAPICallTable.ppstrCalls = NULL;
    }

    // ---- Set the current thread to index zero
    g_iCurrThread = 0;
}
```

As you can see, it's not much different than the original version; it's all just taking place inside a loop. The current thread index is then set to zero, and the stage is set. `ShutDown ()` works in the same way, and because it's a much larger function, I won't bog you down with a code dump.

Handling a Script Array

So you've got an array of scripts and a function for automatically populating that array as scripts are loaded. There's just one problem—every script-related function you wrote in Chapter 10 was designed with a single, global script structure in mind. Do you have to go through *every one* of those functions, add a thread index parameter to specify which thread to work with, and then go through *every one* of the functions' references and change the calls to reflect the new parameter list? Figure 11.15 shows this type of function interface.

Well, you could. However, there's a much easier way to alleviate the problem that can be determined by simply recognizing one key fact—virtually every one of the script-related functions, like

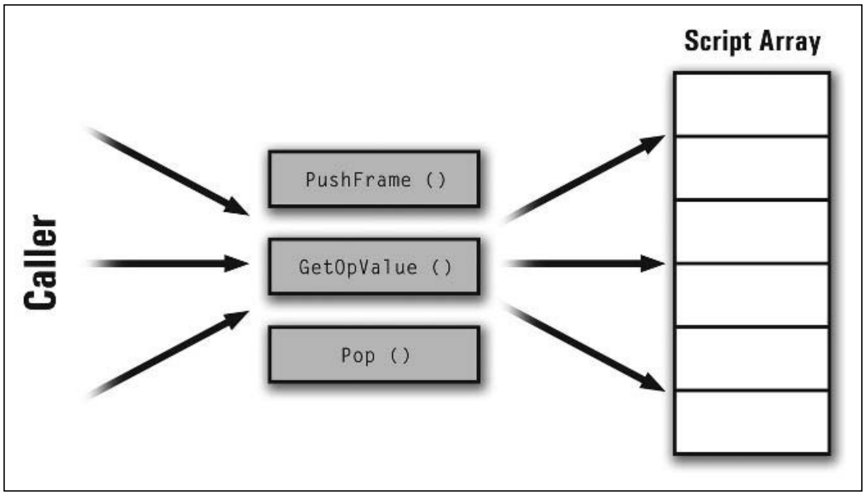


Figure 11.15

The capability to access any script from the script interface functions.

`PushFrame ()`, `ResolveOpAsInt ()`, and so on and so forth, are designed to work with the same script. I don't mean the same script in the sense that they all work with the `g_Script` structure. Rather, I mean that they all work with the script that is currently executing, which could be any of the scripts in the new `g_Scripts []` array. What this means is that instead of changing each function's parameter list and subsequently all of its calls, you can instead replace instances of `g_Script` with `g_Scripts [ g_iCurrThread ]`, where `g_iCurrThread` is a global that tracks the currently active thread. Every time a context switch occurs, `g_iCurrThread` is updated, and every function automatically performs its task on the proper script. Check out Figure 11.16 to see this explained visually. Sounds much easier, right?

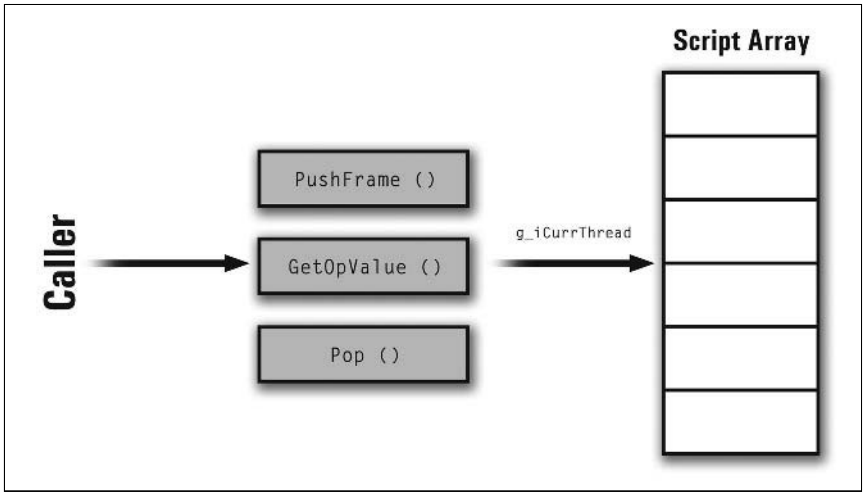


Figure 11.16

Relying on `g_iCurrThread` to determine the proper script to work with.

As an example, here's the old version of `PushFrame ()`:

```
void PushFrame ( int iSize )
{
    // Increment the top index by the size of the frame
    g_Script.Stack.iTopIndex += iSize;

    // Move the frame index to the new top of the stack
    g_Script.Stack.iFrameIndex = g_Script.Stack.iTopIndex;
}
```

Here's the updated version:

```
void PushFrame ( int iSize )
{
    // Increment the top index by the size of the frame
    g_Scripts [ g_iCurrThread ].Stack.iTopIndex += iSize;

    // Move the frame index to the new top of the stack
    g_Scripts [ g_iCurrThread ].Stack.iFrameIndex =
        g_Scripts [ g_iCurrThread ].Stack.iTopIndex;
}
```

See how much simpler it is to fix the problem at the root? Now, the majority of the VM will run unaltered, without even knowing that these functions have been changed. Remember, these changes need to be made to all functions that directly access script data, which include the operand interface:

```
int GetOpType ( int iOpIndex );
int ResolveOpStackIndex ( int iOpIndex );
Value ResolveOpValue ( int iOpIndex );
int ResolveOpType ( int iOpIndex );
int ResolveOpAsInt ( int iOpIndex );
float ResolveOpAsFloat ( int iOpIndex );
char * ResolveOpAsString ( int iOpIndex );
int ResolveOpAsInstrIndex ( int iOpIndex );
int ResolveOpAsFuncIndex ( int iOpIndex );
char * ResolveOpAsHostAPICall ( int iOpIndex );
Value * ResolveOpPntr ( int iOpIndex );
```

The runtime stack interface:

```
Value GetStackValue ( int iIndex );
void SetStackValue ( int iIndex, Value Val );
void Push ( Value Val );
Value Pop ();
void PushFrame ( int iSize );
void PopFrame ( int iSize );
```

And the function table/host API call table interface:

```
Func GetFunc ( int iIndex );
char * GetHostAPICall ( int iIndex );
```

There are, however, cases where `g_iCurrThread` won't be enough, and a specific script index must be acted upon arbitrarily. For example, `ResetScript ()` needs to reset scripts as they're loaded, because you no longer have a single script structure to reset. In this case, the desired thread index must be passed as a parameter, so its new prototype looks like this:

```
void ResetScript ( int iThreadIndex );
```

Once again, the changes are so minute and self-explanatory that it'd be a huge waste of pages to print them all. Be sure to check them out on the accompanying CD instead in the `DIRECTORY_NAME_HERE` directory.

Executing Multiple Threads

With the major structures and functions upgraded to the new multithreaded design, the last major step is to revamp `RunScript ()` as well. The first change, as you may have guessed, is changing the name to `RunScripts ()` to reflect the fact that it now executes multiple scripts in (simulated) parallel. This first version of the multithreading scheduler will not support thread priorities.

The implementation of concurrent thread execution will actually be quite simple. Here's the process in a nutshell (see Figure 11.17 as well):

- `RunScripts ()` begins by saving the current time in a variable to represent the point at which the first thread began execution.
- At each iteration of the execution cycle, the difference between the current time and the time saved in the first step is compared to a constant that determines the length of a time slice. If the time slice hasn't ended yet, the execution cycle iterates within the current script, thereby executing its next instruction.
- If the time slice has elapsed, the scheduler loops through each thread in the `g_Scripts []` array to find the next occupied script and sets that to the new active thread. The current time is once again saved, representing the thread's activation time.

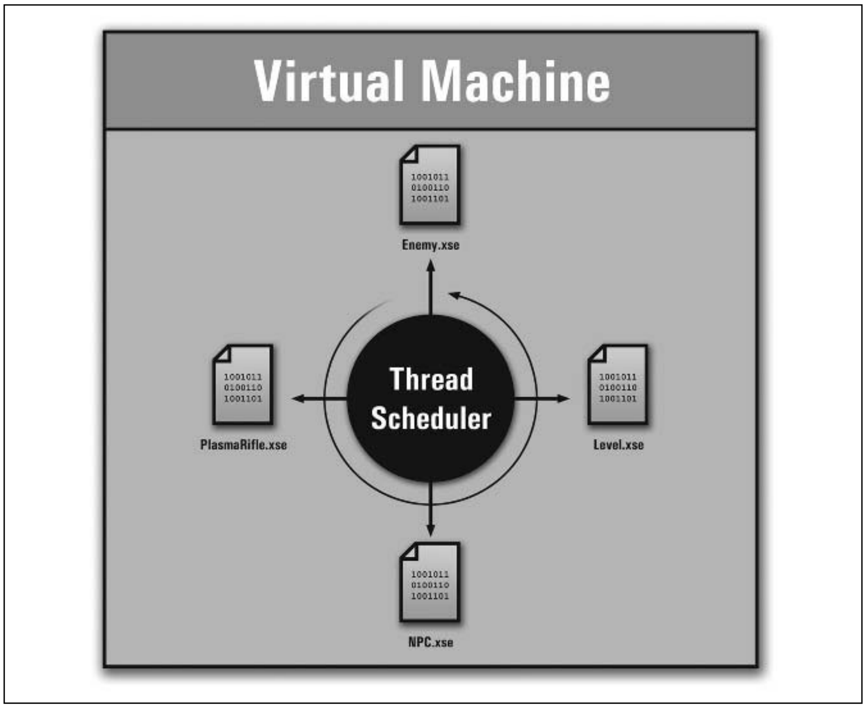


Figure 11.17

Multithreading in the XVM.

This process loops until either a key is pressed or every thread exits by reaching an `Exit` instruction. As you can see, this custom-built multithreading system is really quite simple; all it takes is the capability to maintain a thread index and a time slice timer. Now that you understand the overall strategy, let's break down the details.

Tracking Active Threads

Before you discuss the implementation of time slicing, there's one important detail worth mentioning. The problem with your current `g_Scripts []` array is that there's no explicit way to know whether a given thread is in use. This is important information for the scheduler, which needs to know where in the array the next occupied script structure can be found when a context switch occurs.

Although it's true (more or less) that the fields of a C `struct` are initialized to zero at runtime, I prefer creating an explicit flag within the structure that can be used to track active threads ("active threads" being defined as `Script` structures that have had an .XSE loaded into them). In addition, it's important to know which threads among the active ones are still running. Even if a `Script` structure has been loaded with a script, that function needs to stop executing if it encounters an `Exit` instruction. So, you'll add two new fields to the `Script` structure to track these events. Here's the new structure definition with the added fields in bold:

```
typedef struct _Script       // Encapsulates a full script
{
    int iIsActive;           // Is this script structure in use?

    // Header data
    int iGlobalDataSize;     // The size of the script's global data
    int iIsMainFuncPresent;  // Is _Main () present?
    int iMainFuncIndex;      // _Main ()'s function index

    // Runtime tracking
    int iIsRunning;          // Is the script running?
    int iIsPaused;           // Is the script currently paused?
    int iPauseEndTime;       // If so, when should it resume?

    // Register file
    Value _RetVal;           // The _RetVal register

    // Script data
    InstrStream InstrStream; // The instruction stream
    RuntimeStack Stack;      // The runtime stack
    Func * pFuncTable;       // The function table
    HostAPICallTable HostAPICallTable;
                       // The host API call table
}
    Script;
```

Along with the addition of these two fields, it's important to make changes to `Init ()` and `LoadScript ()` to take them into account. `Init ()` needs to set both `iIsActive` and `iIsRunning` to `FALSE`, whereas `LoadScript ()` needs to set them both to `TRUE` so the scheduler will know that not only is the script structure loaded, but the script is ready to execute when `RunScripts ()` is called.

The Scheduler

All that remains now is managing context switches as `RunScripts ()` executes. This is accomplished by following the previous steps, so let's go over them now in more detail.

Initializing the Time Slice Timer

In order to track the current time slice, the current time has to be recorded when the time slice is invoked using `GetCurrTime ()`. This is initially done outside of the main loop, like so:

```
// Set the activation time for the current thread
// to get things rolling
g_iCurrThreadActiveTime = GetCurrTime ();
```

Now that the first time slice has been invoked, the main loop can begin.

Performing a Context Switch

At each iteration of the main loop, the first order of business is to determine whether the current time slice has elapsed, and perform a context switch if so. As explained previously, the end of a time slice is detected when the difference between the current time and the time at which the time slice was invoked is greater than some constant. This constant is called `THREAD_TIMESLICE_DUR` and defines the standard duration of an XVM time slice, which I like to set to 20 milliseconds:

```
#define THREAD_TIMESLICE_DUR        20
```

Here's the code for using this constant to detect the end of a time slice:

```
// Update the current time
iCurrTime = GetCurrTime ();

// If the current thread's time slice has elapsed, switch to the next
// valid thread
if ( iCurrTime > g_iCurrThreadActiveTime + THREAD_TIMESLICE_DUR )
```

As you can see, the actual code here is a somewhat backwards version of the previous explanation, but it's the same idea. Assuming the time slice has indeed elapsed, the next active thread in the `g_Scripts []` array must be found and invoked:

```
// Loop until the next thread is found
while ( TRUE )
{
    // Move to the next thread in the array
    ++ g_iCurrThread;
    // If you're past the end of the array, loop back around
    if ( g_iCurrThread >= MAX_THREAD_COUNT )
        g_iCurrThread = 0;
    // If the thread you've chosen is active and running, break the loop
    if ( g_Scripts [ g_iCurrThread ].iIsActive &&
        g_Scripts [ g_iCurrThread ].iIsRunning )
        break;
}
// Reset the time slice
g_iCurrThreadActiveTime = iCurrTime;
```

A while loop is entered that cycles through each element of the array. Notice that the current thread is incremented at the top of the loop rather than the bottom; this is because when the loop initially starts, g_iCurrThread will point to the thread that is currently ending, so you need to immediately move past it. The thread index then wraps around to zero if it's passed the end of the array. This has to be done because unless the currently ending thread resides at index 0, the next thread to be executed may very well come *before* it in the array. Finally, the loop analyzes the new thread index to determine if it's both active and running. If so, it's the next thread to be executed and the loop breaks with g_CurrThread set to its index. After the loop completes, the new thread begins executing, so you reset the time slice timer to the current time in order to give it the full duration.

Checking Thread Activity

Lastly, this particular XVM demo is designed specifically to run until either a key is pressed or all threads stop running (which will only occur if none of the loaded threads define infinite loops). To implement this, the Exit instruction should determine whether the iIsRunning field in every currently active thread is clear. If so, the main loop can break. Here's the entire implementation of the Exit instruction:

```
case INSTR_EXIT:
    // Resolve operand zero to find the exit code
    Value ExitCode = ResolveOpValue ( 0 );

    // Get it from the integer field
    int iExitCode = ExitCode.iIntLiteral;

    // Tell the XVM to stop executing the script
    g_Scripts [ g_iCurrThread ].iIsRunning = FALSE;

    // Check to see if all threads have terminated, and if so,
    // break the execution cycle
    int iIsStillActive = FALSE;
    for ( int iCurrThreadIndex = 0;
        iCurrThreadIndex < MAX_THREAD_COUNT;
        ++ iCurrThreadIndex )
    {
        if ( g_Scripts [ iCurrThreadIndex ].iIsActive &&
            g_Scripts [ iCurrThreadIndex ].iIsRunning )
            iIsStillActive = TRUE;
    }
    if ( ! iIsStillActive )
        iExitExecLoop = TRUE;

    // Print the exit code
    PrintOpValue ( 0 );
    break;
```

After extracting the exit code operand as usual, the instruction handler sets the current thread's `iIsRunning` flag to `FALSE`. It then creates a flag variable called `iIsStillRunning`, sets it to `FALSE`, and loops through each thread in the `g_Scripts []` array to find out if any of them are still running. If so, the flag is set to `TRUE` and the loop breaks. Otherwise, the flag remains clear. After the loop, this flag is checked, and unless it's been set, the execution cycle ends.

The First Completed XVM Demo

This wraps up everything your first stab at a next-generation XVM is concerned with. You've added a multitasking scheduler capable of handling an arbitrary number of scripts, which is a great first step towards finishing the runtime environment once and for all. To demonstrate this functionality, the new XVM demo allows you to specify any number of scripts on the command line, which it'll load and run concurrently. Just like the last demo, it'll print each instruction to the screen (along with a thread index), so you can see how it all works firsthand.

To help illustrate the difference between the two, I've included the same two .XSE files I used in the Chapter 10 demo. Now you can load them at the same time and watch them run in parallel. The demo can be found in `Programs/Chapter 11/XVM Demo/` on the accompanying CD.

With this version of the XVM finished, you're ready to move on to the next and final one. In the following sections, you're going to learn how to expand the multitasking system to support thread priorities, which will allow you to balance the XVM's processing load more intelligently among its scripts. In addition, you'll tackle the significant challenge of setting up a powerful interface between the host application and the runtime environment, fully supporting inter-language function calls—both from C to XtremeScript and vice-versa.

Host Application Integration

The focus of the second version XVM, which actually isn't a "version" at all but rather the finished, embeddable module, will revolve around the multithreading scheduler developed in the last section and the host application interface you'll implement here.

Running Scripts in Parallel with the Host

So far, every incarnation of the XVM has been a standalone program that executes scripts in an uninterrupted loop until they terminate, or until the user presses a key. This is fine for demos, as well as standalone virtual machines, but it's not particularly conducive to embeddable runtime environments that need to execute in parallel with their host applications.

The XVM is designed to run *alongside* the main game loop. This means that, at each iteration, the game is updated, the next frame is drawn and blit to the screen, and a small time slice is set aside for the scripts to partially execute. Check out Figure 11.18 to see this expressed visually. The advantage to this approach is that scripted game entities can execute in a much more natural form; rather than the host calling a specific script function at each iteration of the game loop to update all of the script's entities (like you saw in Chapter 6), these entities can be in a constant state of motion and action. In other words, scripts can be written without any knowledge of other scripts or the host—you write them as if they were the only thing executing, which works out fine when they're running in parallel with everything else.

NOTE

Remember, just as you learned earlier in the multithreading system, there's no *true* parallel execution going on here. Everything is split into time slices that are so small and execute so fast, that they appear to be concurrent.

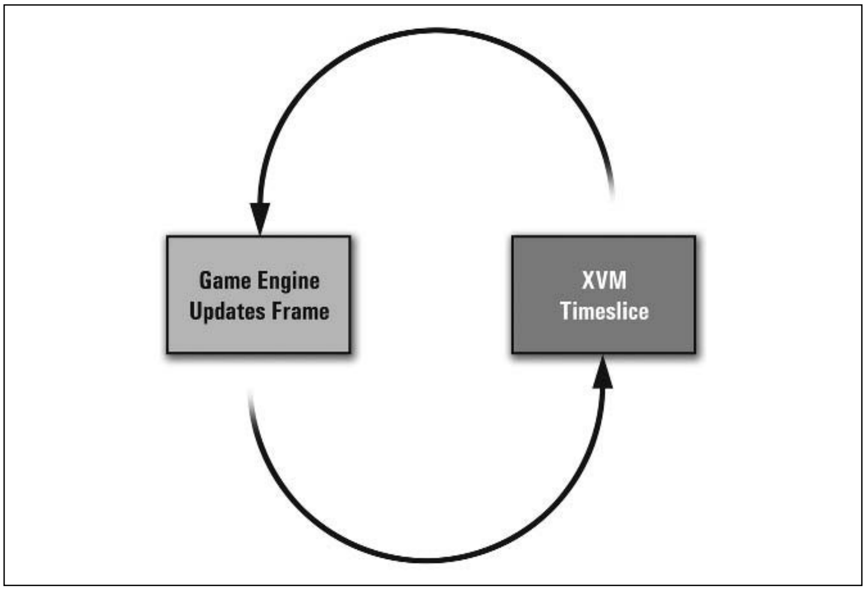


Figure 11.18

The XVM and the game engine share each iteration of the main loop.

Manual Time Slicing vs. Native Threads

There are two ways to go about implementing this approach. You could use the operating system's native threading system to physically run the game engine and virtual machine in separate threads, allowing you to leave the XVM's design as it is and forget about it entirely, or you can do everything yourself and manually implement a time slicing system to do the same thing.

The pros and cons here are the same as they were earlier when developing the XVM's multithreading scheduler. On the one hand, native threads may ultimately be easier to implement (assuming you're familiar with them), and always boast the advantage of providing true parallel execution if you can run your game on a multiprocessor machine. On the other hand, managing time slices on your own will be a better learning experience, illustrates the process more intuitively, and saves me the concern of alienating a sizable portion of the audience who happen to be running on a non-Windows platform. So, I'll go with the latter and show you how to do it all on your own.

A New `RunScripts ()` Function

The only real change that must be made to the XVM in order to allow scripts to be run in a time sliced manner is that `RunScript ()` can no longer enter an indefinite loop that hogs control of the process until its scripts terminate. Rather, the function now needs to accept a time slice parameter that tells it how many milliseconds it should run, and do everything within that duration. Fortunately, all this really means is changing definition of the loop; all of the actual script execution logic you've already written can remain unchanged. You'll see the details behind this process later. What's important now is that you understand that the scripting system will no longer be one continuous loop; instead, it'll run in small time slices defined by whoever calls `RunScripts ()` (which will invariably be the host application). Because of this, the XVM relies on the host's game loop to keep the scripts going; unless `RunScripts ()` is called on a regular interval, nothing will happen.

NOTE

Ideally, I'd recommend using the native threading system of your operating system. Doing so allows `RunScripts ()` to once again run in a single, continuous loop, because it can only dominate the execution of its particular thread, rather than the game engine's entire process. Writing your scripting system's execution cycle this way is a much cleaner solution, and is less error-prone because the time slicing will be handled automatically by the OS. And once again, if the opportunity to run your game on a multiprocessor system ever comes along, the entire overhead of script processing can be offloaded to a separate processor, allowing your game to run at full speed.

Thinking in Multiple Dimensions

It's extremely important that you *not* confuse the XVM's time slice with the time slices assigned to each script. Remember, regardless of how many scripts are in memory, or what their time slices may be, the XVM itself will *only* run for the duration specified by `RunScripts ()'s` caller. Within the XVM's overall time slice of the game loop, context switches may be performed to halt one script and invoke another, but this is entirely unrelated to the larger time slice's duration and will not affect it in any way. Figure 11.19 demonstrates XVM time slices and their relationship to individual script time slice.

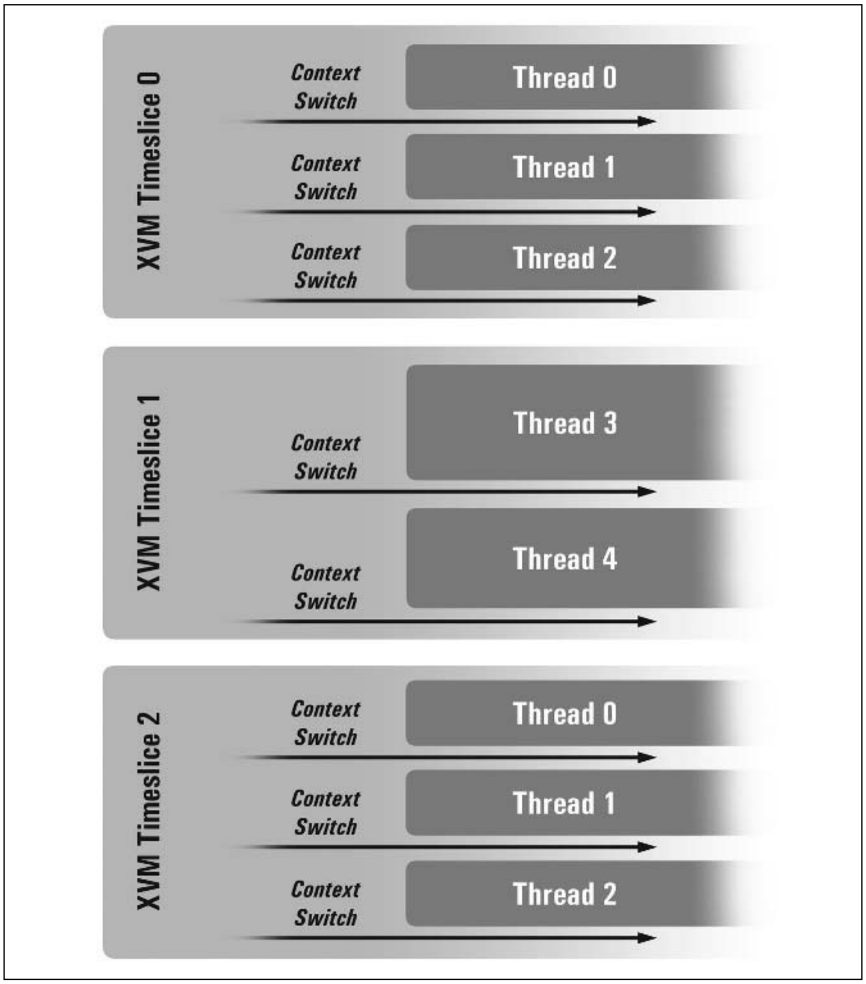


Figure 11.19

There is not necessarily any correlation between the XVM's overall time slice of the game loop and the time slice of each thread.

Introducing the Integration Interface

The integration interface between the host application and the scripts running inside the VM comes down to two major aspects in most scripting systems—the capability to make inter-language function calls, as well as the capability to "track" global variables.

Function calls are the most obvious way to communicate, because they allow you to directly set values, read values, and perform actions. Global variable tracking is also useful, but in more subtle ways; it's best to track a variable when you plan on constantly referring one of the host applications internal values, but don't want to bog everything down with overhead of repetitive function calls. You won't actually implement variable tracking in the XVM's host interface, but general implementation ideas will be discussed.

Calling Host API Functions from a Script

Calling the host API from within a script is facilitated by the `CallHost` instruction, which has gone unimplemented until now. From the perspective of the script, the only difference in the way the call is made is the fact that `CallHost` is used instead of `Call`. Aside from that, parameters are pushed via the stack and return values are stored in the `_RetVal` register. On the script side of things, it's just another function call. Figure 11.20 illustrates this.

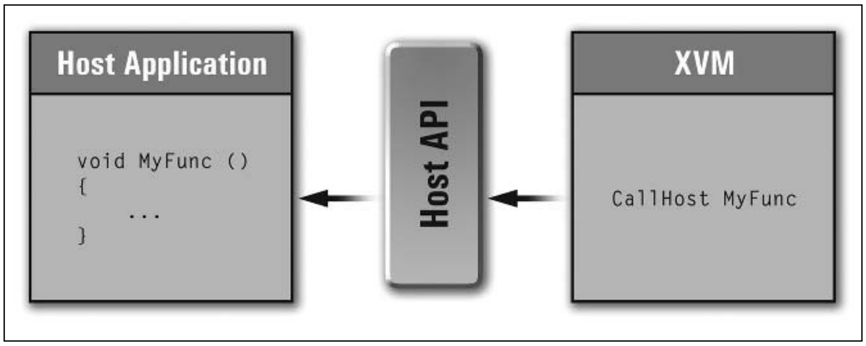


Figure 11.20

Calling a host API function from the script.

On the host side, however, things are a bit more complicated. Host API functions can't be written exactly like typical C functions; rather, they must conform to a specific prototype and deal with parameters in a very particular way.

Functions must follow a specific prototype because the XVM stores the host API internally as an array of function pointers, and it's much easier to call them when the prototype has been decided upon ahead of time. When a host API call is made, this pointer is used to invoke the function.

Host API functions need to read parameters just like any other function does, but they can't use C's parameter passing syntax directly, because the parameters lie on the XVM's runtime stack, not the host's. Furthermore, because these parameters have no explicit type, special functions must be used to read parameters from a specific stack index and with a specific data type in mind.

Return values are much easier; all that's necessary is to set the value of the `_RetVal` register stored within the script's `Script` structure.

Calling Script Functions from the Host

Calling a host API function from the script is one thing; calling a script-defined function from the host is a much more delicate matter. To understand why, it's first important to understand that such calls can be broken down into two categories: *synchronous* and *asynchronous*.

Asynchronous Calls

As you learned earlier, the final version of the XVM will be run in time slices alongside the game engine's main loop. Within this loop, if the host were to call another C-defined function, the main loop would halt until the function returned, which is why calling a particularly slow or processor-intensive function inside your main loop has such a noticeable effect on your frame rate. Of course, it's often necessary to make such calls, usually because the result of the function—either its return value or simply the action it performs—must be completed within the current frame. This even applies to functions defined within scripts, which is where asynchronous calls come in.

Simply put, an asynchronous call to a script-defined function will execute immediately and directly return to the caller. This means that if an asynchronous call is made to a script during the main loop of the game, both the main loop *and* the script will halt until the function returns, at which point execution will resume as normal. Asynchronous calls are made when something needs to be done immediately or before anything else. Check out Figure 11.21 for a visual.

One important detail about making an asynchronous call is that the script's runtime stack and instruction pointer must be restored to the *exact* state they were in before the call is made. In other words, the script shouldn't have any idea that the host called one of its functions when it begins executing again in its next time slice.

Synchronous Calls

Synchronous calls are more or less the opposite of asynchronous calls. A synchronous call will "invoke" a script's function in the same way that a `Call` instruction would, if it were made inside the script itself. Rather than halting the game loop, a synchronous function call won't even take effect until the scripting system enters its next time slice. Furthermore, a synchronous call most likely won't return within a single time slice, but will rather execute over time, as shown in Figure 11.22.

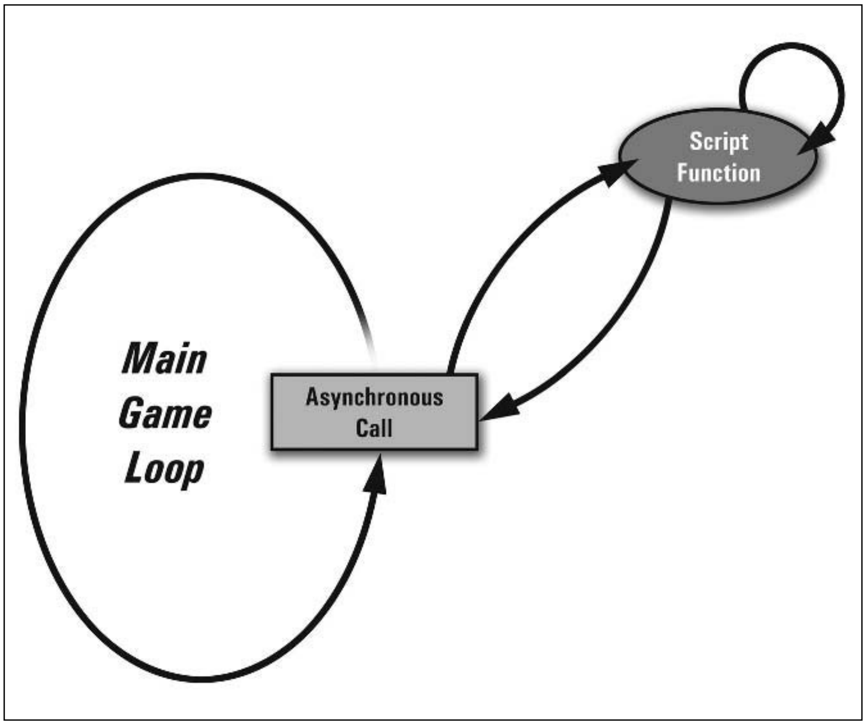


Figure 11.21

Asynchronous function calls interrupt the flow of execution for both the game engine and the script.

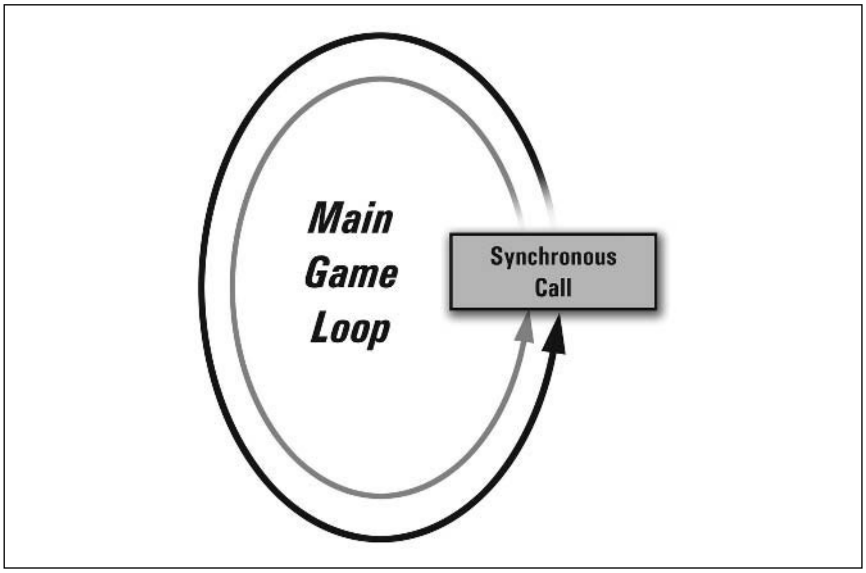


Figure 11.22

Synchronous function calls follow the existing flow of the scripts and game engine, and therefore execute over time as opposed to immediately.

To put it another way, synchronous calls are a way for the host application to simulate the `Call` instruction. If a script were to call one of its own functions just before the XVM's time slice ended, the called function wouldn't begin executing until the next time slice rolled around. Also, unless it was extremely small, it probably wouldn't return for at least a few time slices, because the XVM is only able to use a small portion of the overall length of each game loop iteration. This is exactly how synchronous calls behave—they execute gradually over the course of 1-N XVM time slices, and mimic functions called with the `Call` instruction exactly. This also makes it extremely difficult to retrieve the function's return value (if any).

CAUTION

As you may have already guessed, synchronous calls should be made with caution. Remember, a synchronous will behave exactly like a function called directly from the script with `Call`, and will interrupt whatever the script was already doing. This isn't a problem, unless the function returns a value or modifies global variables. In these cases, if the code that was executing within the script just before the call referred to either `_RetVal` or the same globals used by the function, these values will seem to suddenly change without warning. Because of this, it's best to know that a script function will be called synchronously from the host as you write it, so you can specifically design it to leave globals and `_RetVal` alone.

Synchronous calls are most useful for performing large-scale actions like changing a script's overall state or altering the behavior of a scripted game entity. Remember, because the effects of the called function will take place over time, rather than within a single frame, they can have a longer, more gradual effect on game play overall.

Figure 11.23 provides a more geometric way to visualize this difference; synchronous calls run parallel to the execution of the script, whereas asynchronous calls are perpendicular.

Tracking Global Variables

Lastly, there's the issue of global variable tracking. I should mention right off the bat the XVM won't support this feature, because I personally don't find it useful enough to justify the added complexity to the system overall. Of course, you may feel differently, so let's discuss the general theory behind the implementation of this integration feature.

NOTE

The technique described here can actually be used to track host variables from the script, as well as script variables from the host.

To track the value of a host application variable, the script defines a variable of its own and "binds" it to the specified

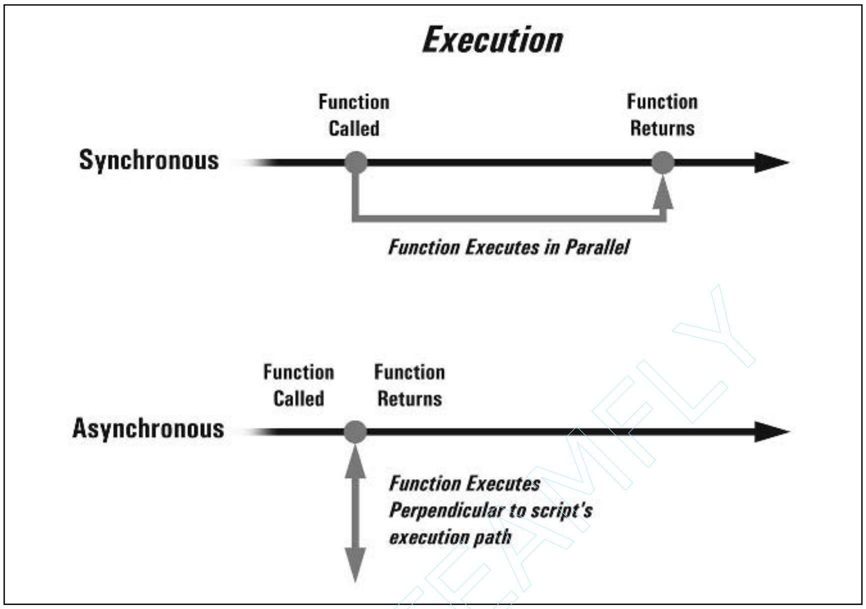


Figure 11.23

One way to visualize the difference between synchronous and asynchronous calls.

host global, such that the script-defined variable always mirrors its value. This way, if the script wants to constantly refer to a host application variable's value, whether for the purpose of reading, writing, or both, it can do so in a more natural way without making a ton of function calls. Figure 11.24 illustrates this concept.

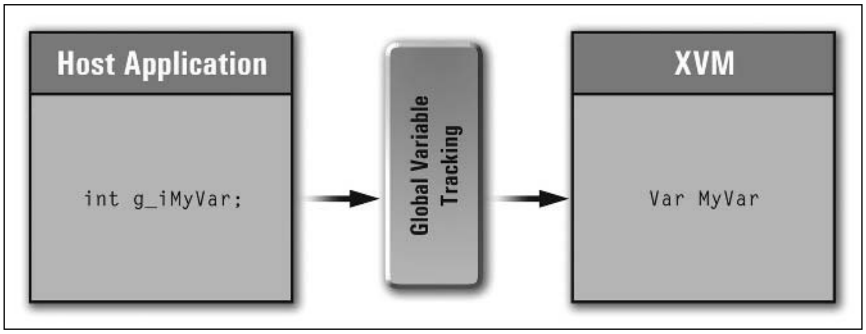


Figure 11.24

Tracking global variables.

The main problem with this approach is that the identifier of the host application variable isn't known to XASM at the time at which the script is assembled. For example, if the host application defines a global integer value called `g_iGlobalInt`:

```
int g_iGlobalInt;
```

you can't just refer to it like this within the assembler (assume `BindToHostVar` is an XASM directive for binding script variables to host variables):

```
Var MyVar
BindToHostVar MyVar, g_iGlobalInt
```

One solution to this problem is to give the host application the capability to assign a numeric index to each of the globals it'd like to expose to the script, perhaps with a function called `BindVarToIndex ()`:

```
BindVarToIndex ( g_iGlobalInt, 0 );
```

Now, assuming the indexes to which each host application global is assigned is known when the script is written, the `BindToHostVar` directive can instead allow scripts to bind their variables to these same indexes as well:

```
BindToHostVar MyVar, 0
```

Now, `g_iGlobalInt`, defined in the host application, and `MyVar`, defined in the script, have the `0` index in common. The XVM can now establish a connection between the two. With this out of the way, let's talk about how to actually make the values of these two variables mirror each other.

Tracking the Bound Variables

The key to tracking variables properly is keeping the values updated and in sync. The first step in doing this is creating an array of `void` pointers that correlate to the host's globals. In this example, you'll create a static array large enough to hold a reasonable number of host-defined globals, like so:

```
#define MAX_TRACKED_VAR_COUNT    1024
void * g_pGlobalVars [ MAX_TRACKED_VAR_COUNT ];
```

Whenever the host binds a variable to an index, its pointer should be stored in this array at the index specified (see Figure 11.25). So, going back to the example from before, the following line of code would store `g_iGlobalVar`'s pointer at index 0 of `g_pGlobalVars []`:

```
BindVarToIndex ( g_iGlobalInt, 0 );
```

After the host binds each of its globals to an index, you'll have an array containing all the pointers you need to track them. The only issue remaining is how the values these globals contain can be accessed from the script.

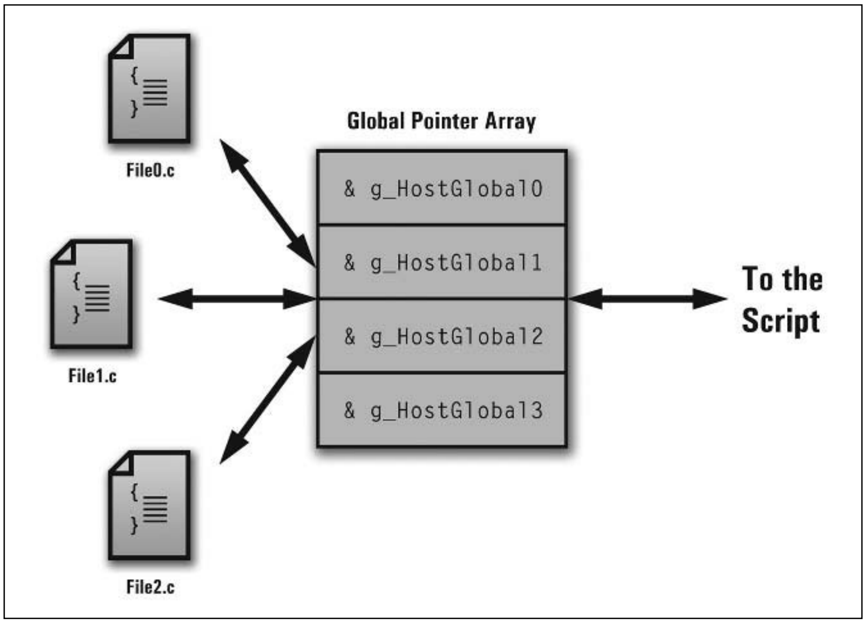


Figure 11.25

Globals from all over the program can be stored as pointers in a single array.

Binding Stack Indexes to the Pointer Array

Even variables that the script binds to the host application reside somewhere on the stack. This stack index, therefore, is all you need to keep the script's variable in sync with the global defined by the host. Therefore, the `BindToHostVar` directive discussed earlier needs to save the specified variable's stack index in a table that can be written to the .XSE file for use by the XVM at run-time. As long as the variables defined by both the script and host are assigned the same index, you'll be able to tell which pointers are assigned to which stack indices. To store these in memory at runtime, the XVM will need another new array, this one within the `Script` structure (see Figure 11.26):

```
int iBoundStackIndices [ MAX_TRACKED_VAR_COUNT ];
```

Now, each script can keep track of which of its stack indices are bound to which host application globals. In case you're wondering why we don't just merge the `g_pGlobalVars []` array with this one, to keep the global pointers and their associated stack indices in the same place, remember that your new multithreaded version of the XVM should allow multiple scripts to track the same globals. Because it's highly unlikely that all of these scripts will just happen to bind the same stack indices to the same global, you need to keep them separate.

Now that you have pointers to the host's globals and the stack indices of the script variables you've bound to them, you have all the information you need to keep their values in sync.

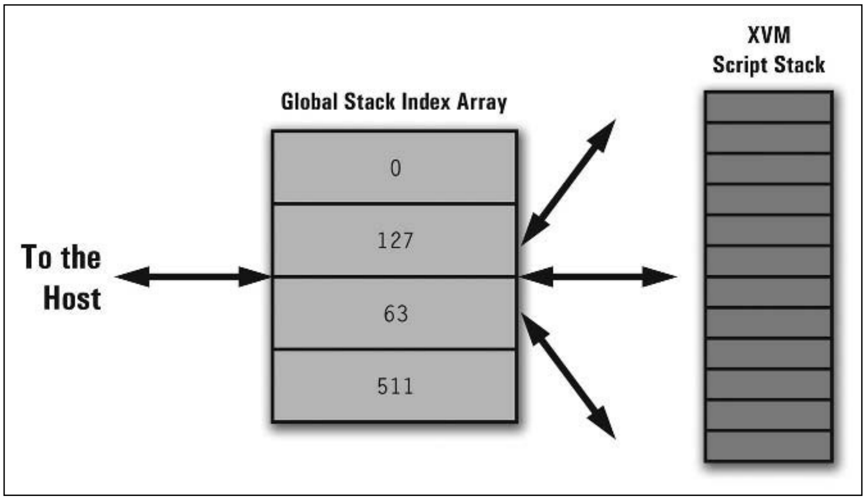


Figure 11.26

A new array, much like the first, maintains the stack indices at which tracked script globals reside.

Keeping the Values Synchronized

At each frame of the game loop, the game engine and scripting system will execute in almost entirely separate phases. With the exception of inter-language function calls, which I won't be addressing in this section, the game engine will be entirely halted while `RunScripts ()` is running. For the rest of the frame, the scripting system is halted while the game engine runs. The point is that at any given time, these two separate entities will never be running at the same time. So, in order to keep bound variables in sync with one another, all you have to do is update them just before and just after `RunScripts ()`'s time slice executes.

Just before the XVM's time slice begins, it's possible that the game engine's globals will be set to new values that the script's bound variables won't reflect. So, the system loops through each pointer stored in the `g_pGlobalVars []` array and writes their values to the corresponding stack indices in scripts `iBoundStackIndices []` array. Now, when the XVM begins its time slice, the script's runtime stack will contain the current value of the bound globals, which can be freely referenced by the script.

After the time slice, it's possible that the script will have made changes to its bound variables, which need to be written to the globals before the game engine proceeds. This time, the update loop writes the values from the stack indices into the pointers of the `g_pGlobalVars []` array, transferring the script's modifications to the game engine. As you can see, by updating each set of variables before and after the XVM's time slice, they'll stay synchronized at all times.

You may have already noticed one flaw in this solution, however; because the XVM is typeless and the game engine is not, how exactly will these values be transferred between them? To address this issue, the `BindVarToIndex ()` function needs to accept an additional parameter that specifies the variable's type:

```
#define HOST_VAR_TYPE_INT    0
#define HOST_VAR_TYPE_FLOAT  1
#define HOST_VAR_TYPE_STRING 2

BindVarToIndex ( g_iGlobalInt, 0, HOST_VAR_TYPE_INT );
```

Of course, this also means the `g_pGlobalVars []` array needs to become an array of structures, wherein each element stores both the pointer and its type:

```
typedef struct TrackedVar
{
    void * pVar;
    int iType;
};

TrackedVar g_TrackedVars [ MAX_TRACKED_VAR_NUM ];
```

The system will now have enough information to transfer values from host variables to script variables, and vice versa.

CAUTION

Remember, in both the script and host application, the bound variables must *always* be global. Because the stack frame in which a local resides is destroyed when its function returns, they'll immediately begin returning garbage values and lead to unexpected results.

The XVM's Public Interface

In order for the XVM to be embedded in a host application, it needs to expose a *public interface*, or collection of functions, that the host can call to control it. For example, the host application will need functions for loading and unloading scripts, as well as functions for starting, stopping, pausing and unpausing them. Also, as you explore the development of an inter-language function call interface, you'll need to add public interface functions to handle that as well.

Currently, all of the XVM's functionality has been stored in a single `.cpp` file for simplicity's sake, but you'll of course need to create an adequate header file for inclusion in host applications. This header file will be called `xvm.h`, and will be built incrementally in the following sections. Figure 11.27 illustrates how these files will interact.

Which Functions Should Be Public?

The first order of business is determining which functions the host application needs to call in order to control the XVM. Generally speaking, all the host application needs to do is initialize and shut down the runtime environment, load and unload scripts, and call `RunScript ()` to keep everything moving. So, the first additions to `xvm.h` will be the following prototypes:

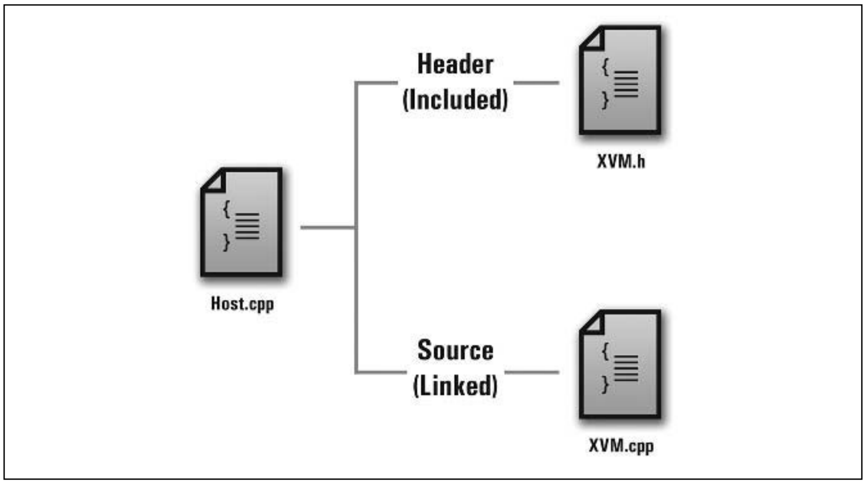


Figure 11.27

The XVM header provides the public interface to the host application.

```
void Init ();
void ShutDown ();

int LoadScript ( char * pstrFilename,
                 int & iScriptIndex,
                 int iThreadTimeslice );
void UnloadScript ( int iThreadIndex );
void ResetScript ( int iThreadIndex );

void RunScripts ( int iTimesliceDur );
```

With these functions, the host application can initialize and shut down the system, load and unload scripts, reset them arbitrarily, and execute a multithreaded time slice (most likely at each frame of the main loop).

Name Clashes

The initial public API is good, but it suffers from one major problem. Function names such as `Init ()` and `ShutDown ()` could be applied to any number of programs or libraries, which means it's entirely possible that the host application has already defined such functions. In such cases, a *name clash* will occur, which of course results in an immediate compile-time or linker error.

If you plan on using your scripting system only for personal projects, you can go ahead and use any naming conventions you want. However, if your goal is to also create a scripting system that you can share with friends and fellow developers, use in a professional environment, or distribute over the Internet, name clashing can ruin an otherwise good product.

Because of this, it's important to transform or mangle your function names in such a way that they're less likely to step on the host application's toes. The easiest way to do it is to follow the time-honored tradition of prefixing your function names with a brief abbreviation of your scripting system's name (usually two letters) and an underscore. So, in the case of XtremeScript, `XS_` would appear before all publicly defined entities, thereby transforming the function prototypes to this:

```
void XS_Init ();
void XS_ShutDown ();

int XS_LoadScript ( char * pstrFilename,
                    int & iScriptIndex,
                    int iThreadTimeslice );
void XS_UnloadScript ( int iThreadIndex );
void XS_ResetScript ( int iThreadIndex );
void XS_RunScripts ( int iTimesliceDur );
```

TIP

For all you C++ coders, this is a great application of namespaces. I personally find namespaces to be cleaner than physically renaming functions with a prefix, so I strongly suggest you go with that particular solution to the name clashing issue.

Sure, it's *possible* that the host has defined a function called "`XS_UnloadScript ()`", but it's a lot less likely.

Public Constants

In addition to functions, the host application will need access to a few of the constants the VM has been using internally thus far; namely, `XS_LoadScript ()`'s error codes:

```
#define XS_LOAD_OK                        0  // Load successful
#define XS_LOAD_ERROR_FILE_IO             1  // File I/O error (most likely
                                             // a file not found error
#define XS_LOAD_ERROR_INVALID_XSE         2  // Invalid .XSE structure
#define XS_LOAD_ERROR_UNSUPPORTED_VERS    3  // The format version is
                                             // unsupported
#define XS_LOAD_ERROR_OUT_OF_MEMORY       4  // Out of memory
#define XS_LOAD_ERROR_OUT_OF_THREADS      5  // Out of threads
```

Note that of course, constants need the `XS_` prefix too; name clashing isn't just for functions to worry about.

Implementing the Integration Interface

It'd be nice if the preparation of a simple header file was all you needed to do to fully integrate the VM with the host application. The real work, of course, lies ahead—the actual implementation of the integration interface. This will mostly boil down to the ability to make inter-language function calls, but as you'll see, this is hardly a trivial matter.

Basic Script Control Functions

Just before getting into the nitty-gritties of the host API and other such issues, however, let's start off with something simple and talk about the functions the host will need to leverage a basic control of its scripts.

Loading and Running

Right off the bat, you've already seen the functions for loading and unloading scripts. Once in memory, scripts can be reset with `ResetScript ()` (although `LoadScipt ()` will do this automatically), and `XS_RunScripts ()` is called periodically to keep everything in motion. These are the bare-minimum functions; they allow the host to read in scripts and unload them, as well as run them with a reasonable level of flexibility, but what about more subtle operations?

For example, a game engine will invariably want to start and stop scripts arbitrarily, usually in reaction to various game events or entity behavior. In these cases, it could use the existing `XS_LoadScript ()` and `XS_UnloadScript ()`, but this is definitely using a hatchet for a scalpel's job—there's no need to physically load the script from the disk and clear it from memory every time it needs to start and stop. Furthermore, it may also be necessary to frequently pause and unpause scripts, which is entirely impossible with the current set of functions we have. To address these issues, you need finer control.

Finer Script Execution Control

The two major features your current set of script control functions doesn't support are the capability to start and stop scripts without physically loading and unloading them from memory, as well as pausing and unpausing them forcibly; in other words, without relying on the script itself to execute a `Pause` instruction.

Let's start with the first issue:

```
void XS_StartScript ( int iThreadIndex );
void XS_StopScript ( int iThreadIndex );
```

These two new functions will enable you to start and stop scripts on a dime. It's also important at this point to rework `XS_LoadScript ()` so that it doesn't automatically start the script it loads—this should instead be at the sole discretion of the host and `XS_StartScript ()`. Let's see the code behind `XS_StartScript ()`:

```
void XS_StartScript ( int iThreadIndex )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // Set the thread's execution flag
    g_Scripts [ iThreadIndex ].iIsRunning = TRUE;

    // Set the current thread to the script
    g_iCurrThread = iThreadIndex;

    // Set the activation time for the current
    // thread to get things rolling
    g_iCurrThreadActiveTime = GetCurrTime ();
}
```

The function begins by calling a macro called `IsThreadActive ()` (which I'll discuss in a second), and then sets the `iIsRunning` flag to `TRUE`. This lets the XVM know that the script is in a state of execution, which is what this particular function is primarily responsible for invoking. In addition, the call automatically preempts the currently running script in favor of the newly executed one, and resets the time slice to reflect this.

`XS_StopScript ()` is even simpler, and pretty much self-explanatory; all it's concerned with is clearing the `iIsRunning` flag:

```
void XS_StopScript ( int iThreadIndex )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // Clear the thread's execution flag
    g_Scripts [ iThreadIndex ].iIsRunning = FALSE;
}
```

As for the `IsThreadActive ()` macro, all it does is ensure that the specified thread index refers to a currently active thread (the term "active" means any thread structure that's been populated with a script; not to be confused with a "running" thread, which is actually executing). Here's all it does:

```
#define IsThreadActive( iIndex )    \
    ( IsValidThreadIndex ( iIndex ) &&
        g_Scripts [ iIndex ].iIsActive ? TRUE : FALSE )
```

Of course, this macro calls another macro, IsValidThreadIndex (). This one just makes sure that the specified thread index is within the proper range:

```
#define IsValidThreadIndex( iIndex )    \
( iIndex < 0 || iIndex > MAX_THREAD_COUNT ? FALSE : TRUE )
```

Together, these two macros provide an easy and quick way to make the public script control functions more robust. The last set of script control functions to discuss is used to pause and unpause scripts. Let's start with XS_PauseScript ():

```
void XS_PauseScript ( int iThreadIndex, int iDur )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // Set the pause flag
    g_Scripts [ iThreadIndex ].iIsPaused = TRUE;

    // Set the duration of the pause
    g_Scripts [ iThreadIndex ].iPauseEndTime =
        GetCurrTime () + iDur;
}
```

All that's necessary (aside from validating the thread index as always) is to set the iIsPaused flag to TRUE and set the pause end time to the current time (as returned by GetCurrTime ()) plus the specified duration. To unpause the script before the original duration elapses, call its sister function; XS_UnpauseScript ():

```
void XS_UnpauseScript ( int iThreadIndex )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // Clear the pause flag
    g_Scripts [ iThreadIndex ].iIsPaused = FALSE;
}
```

Even easier, eh? Once the iIsPaused flag is cleared, the script resumes execution.

NOTE

Remember, the pausing and unpausing of a script has no effect on any other scripts that may be running concurrently. Whether or not a script is paused, its time slice will come and go just like any other. Pausing one script won't free up time for anyone else, or change the round robin scheduling cycle.

Host API Calls

You can begin your descent into the maddening world of the integration layer with host API calls. Host API calls are made from the script, and allow it to call functions written in C (or whatever the host application is written with) just like it'd call a typical script-defined function. The only difference is the use of the `CallHost` instruction instead of `Call`. Aside from that, the procedure is the same—parameters are pushed onto the stack, and the return value (if applicable) is found in `_RetVal`.

Representing the Host API Internally

Before anything can happen, the host application needs to define its API. The actual representation of this API is an internal structure within the XVM—an array, specifically—consisting primarily of a name string and a function pointer. The name string allows scripts to refer to these functions by name, rather than an arbitrary numeric index or other such method of identification. The function pointer, of course, is how the XVM physically invokes the function to complete the call.

Because the `CallHost` requires only the name of a function, as in the following example:

```
CallHost MyHostFunc
```

these two pieces of data are all you really need. In this case, the host API array would contain an element in which the name string was `"MyHostFunc"`. The `CallHost` would search this array until it matched the specified operand with this string. The corresponding element's function pointer would then be used to call the function.

NOTE

Remember, the host API is global throughout the system; in other words, all scripts have access to it in some form (you'll see what I mean by this later on). This makes things easier on you, because you only have to manage a single structure.

CAUTION

Remember, don't confuse the *host API call table* stored within each script with the *host API*. The host API call table is simply an array of strings; each string corresponds to one of the function names specified as an operand to the script's `CallHost` instructions. In other words, this structure is a record of the script's *calls* to the host API, hence the name. The host API structure actually stores the functions themselves. There's only one copy of the host API, but each script has its own host API call table.

THE STRUCTURE

The first order of business is creating a structure to store the API within the XVM. As mentioned, this is really just an array of structures, wherein each structure represents a single API function. Let's start with this structure's definition:

```
typedef struct _HostAPIFunc      // Host API function
{
    int iIsActive;               // Is this slot in use?
    int iThreadIndex;            // The thread to which this function
                                 // is visible
    char * pstrName;             // The function name
    HostAPIFuncPntr fnFunc;      // Pointer to the function definition
}
    HostAPIFunc;
```

The first field, `iIsActive`, is just a simple flag to determine whether this particular structure has been initialized with an actual API function. Sure, I could just check to see whether the `pstrName` string pointer is `NULL`, but I wanted something a bit more explicit. The next field is `iThreadIndex`, which tells the XVM which threads this function can be called from. Setting this value to -1 makes a function available to all threads. The last two functions, `pstrName` and `fnFunc`, are the name string and function pointer fields discussed earlier.

These structures are stored in a static array called `g_HostAPI`. Here's the declaration:

```
HostAPIFunc g_HostAPI [ MAX_HOST_API_SIZE ];
```

`MAX_HOST_API_SIZE` can of course be anything; I have it set for 1024. Figure 11.28 presents a visual of the host API array.

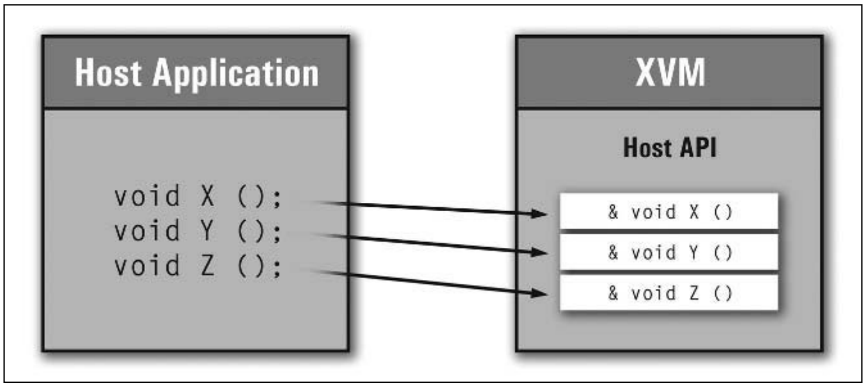


Figure 11.28

The host API resides in an array of structures within the XVM.

TIP

Once again, you're faced with the opportunity to use dynamic structures for flexibility or static arrays for speed and simplicity. As usual, I'm sticking to the static stuff to keep things easy and straightforward for the purpose of the book, but you're always encouraged to make your own decisions in this area. It's probably not necessary to go as far as using a linked list or extendable array to store the host API, for the simple fact that it's unlikely to change at runtime. However, a dynamically allocated array might be a nice touch if the host application can choose the size at the time it initializes the XVM.

Adding Host API Functions

With the array decided upon, the host application needs an easy way to add functions to it. This process is called *registering* a host API function, and is handled with the function `XS_RegisterHostAPIFunc ()`:

```
void XS_RegisterHostAPIFunc ( int iThreadIndex, char * pstrName,
    HostAPIFuncPntr fnFunc )
{
    // Loop through each function in the host API until a free index
    // is found
    for ( int iCurrHostAPIFunc = 0;
          iCurrHostAPIFunc < MAX_HOST_API_SIZE;
          ++ iCurrHostAPIFunc )
    {
        // If the current index is free, use it
        if ( ! g_HostAPI [ iCurrHostAPIFunc ].iIsActive )
        {
            // Set the function's parameters
            g_HostAPI [ iCurrHostAPIFunc ].iThreadIndex =
                iThreadIndex;
            g_HostAPI [ iCurrHostAPIFunc ].pstrName = ( char * )
                malloc ( strlen ( pstrName ) + 1 );
            strcpy ( g_HostAPI [ iCurrHostAPIFunc ].pstrName,
                pstrName );
            strupr ( g_HostAPI [ iCurrHostAPIFunc ].pstrName );
            g_HostAPI [ iCurrHostAPIFunc ].fnFunc = fnFunc;
            // Set the function to active
            g_HostAPI [ iCurrHostAPIFunc ].iIsActive = TRUE;
        }
    }
}
```

This function makes use of the usual technique of looping through an array until the first free element is found. Once an inactive structure is located, it's populated with the function's data, which pretty much comes directly from the XS_RegisterHostAPIFunc ()'s parameters, and the structure's iIsActive flag is set.

Of course, because this is a public function, it's prefixed with XS_ and is declared in the XVM header file:

```
void XS_RegisterHostAPIFunc ( int iThreadIndex,
                              char * pstrName,
                              HostAPIFuncPntr fnFunc );
```

In addition, a special constant is declared for use by the host to make the registration of global functions (functions that aren't intended for any specific thread) cleaner and more readable:

```
#define XS_GLOBAL_FUNC              -1
```

This can be passed as the iThreadIndex parameter instead of directly using -1. This also allows you to change the flag later if necessary.

The XVM Host API Function Prototype

The last detail to mention when dealing with the host API is how a host API function is defined. For the most part, when maintaining a list of similar functions like your host API, it's either helpful or downright necessary that all of the functions are of the same prototype—meaning they accept the same parameters and return the same value (if any). For reasons that will become clear in the following sections, every function added to the host API must follow this form:

```
void HostAPIFunc ( int iThreadIndex );
```

Making the Call with CallHost (The Script Side)

It's important to remember that calling a host API function is virtually identical to calling a script-defined function. Like I mentioned, the only difference is using CallHost instead of Call. This instruction has remained unimplemented until now, so let's change that. The instruction's implementation is pretty straightforward, so let's start with the code:

```
case INSTR_CALLHOST:
{
    // Use operand zero to index into the host API call table and
    // get the host API function name
    Value HostAPICall = ResolveOpValue ( 0 );
    int iHostAPICallIndex = HostAPICall.iHostAPICallIndex;

    // Get the name of the host API function
    char * pstrFuncName = char * pstrFuncName = GetHostAPICall (
        iHostAPICallIndex );

    // Search through the host API until the
    // matching function is found
    int iMatchFound = FALSE;
    for ( int iHostAPIFuncIndex = 0;
          iHostAPIFuncIndex < MAX_HOST_API_SIZE;
          ++ iHostAPIFuncIndex )
    {
        // Get a pointer to the name of the current host API function
        char * pstrCurrHostAPIFunc =
            g_HostAPI [ iHostAPIFuncIndex ].pstrName;

        // If it equals the requested name, it might be a match
        if ( strcmp ( pstrFuncName, pstrCurrHostAPIFunc ) == 0 )
        {
            // Make sure the function is visible to the current thread
            int iThreadIndex =
                g_HostAPI [ iHostAPIFuncIndex ].iThreadIndex;
            if ( iThreadIndex == g_iCurrThread || iThreadIndex ==
                XS_GLOBAL_FUNC )
            {
                iMatchFound = TRUE;
                break;
            }
        }
    }

    // If a match was found, call the host API function
    // and pass the current
    // thread index
    if ( iMatchFound )
        g_HostAPI [ iHostAPIFuncIndex ].fnFunc ( g_iCurrThread );

    break;
}
```

The first task is reading the value of operand zero, which is an index into the script's host API call table where the function name string can be found. This index is passed to `GetHostAPICall ()` to retrieve the name of the function the instruction is trying to call. This string is then used as a search key in the host API array to find the function's pointer and other relevant information. The actual search is simple; the specified function name is compared to each in the host API. If a match is found, the function's intended thread index is then compared to the thread that's making the call; if they match, or if the function is global, the `iMatchFound` flag is set. Outside the search loop, this flag is used to determine whether the call should be made.

TIP

Notice that calling a host API function that either isn't defined or isn't intended for the script that's calling it has no effect. You may, however, decide that it's better to flag some sort of error at runtime for debugging purposes. The only reason I haven't implemented that here is that a running game engine will most likely have control of the screen when such an invalid call is made, so the presentation of the error message can change significantly from game to game and platform to platform.

Defining Host API Functions (The Host Side)

Of course, the real driving force behind the host API is the functions themselves. These are created in almost the same way a typical C function would be created, except for two primary differences:

- They must adhere to the prototype mentioned earlier.
- The function's input and output—in other words, its parameters and return value—must be implemented using special helper functions and macros provided by the XVM, because they must interface specifically with the script.

The actual code and logic of the function is written normally. Let's start the discussion of host API functions by looking at a complete example of how a function is created, registered, and called from a script.

The first step, of course, is writing the function. This example function will accept two parameters—a string value and an integer count that tells the function how many times to print the string to the console. For further illustrative purposes, the function will return a string value as well. Here's the function:

```
void HAPI_PrintString ( int iThreadIndex )
{
  char * pstrString = XS_GetParamAsString ( iThreadIndex, 0 );
  int iCount = XS_GetParamAsInt ( iThreadIndex, 1 );

  for ( int iCurrString = 0; iCurrString < iCount; ++ iCurrString )
      printf ( "%s\n", pstrString );

  XS_ReturnString ( iThreadIndex, 2, "This is a return value." );
}
```

Notice first that I prefixed the function name with `HAPI_`; this, like the `XS_` prefix used with the XVM's public functions and constants, is used to prevent name clashes with other functions defined in the program. Of course, because you're most likely going to be writing both the host application's internal functions, as well as the ones it'll expose to scripts via the host API, you really won't have to worry about clashing because you'll be in charge of all of the identifiers. The `HAPI_` prefix adds a bit more readability, however, and helps you out in cases where you have two versions of the same function—one for internal use by the host, and one for use by scripts.

You will notice a few oddities beyond the name, however. Primarily, parameters are retrieved using a set of functions called `XS_GetParamAs* ()`, and the return value is handled with a macro called `XS_Return* ()`. I use the asterisks to show that these functions and macros come in forms for supporting all of the XVM's primitive data types—integers, floats and strings. Figure 11.29 demonstrates the usage of separate functions for reading parameters and returning values.

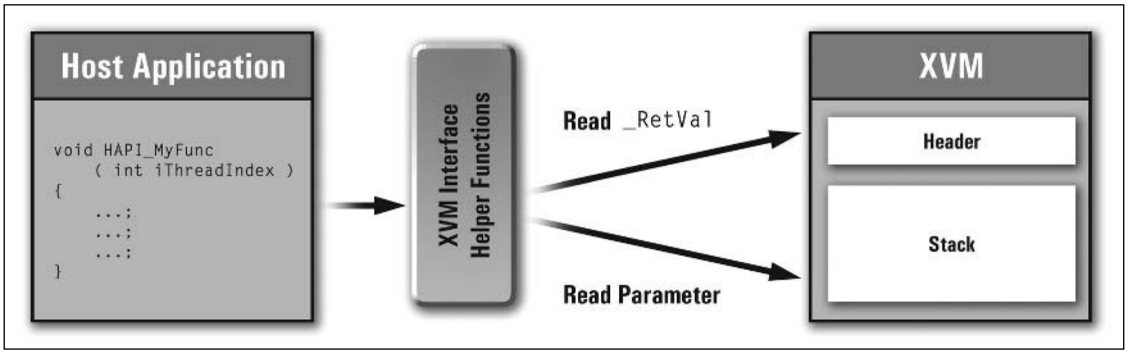


Figure 11.29

Because parameters and return values are kept inside the XVM's `Script` structure, a host API function needs special functions for dealing with them.

READING PARAMETERS

Remember, even from the perspective of a C-defined function, the parameters passed from a script always reside on the thread's runtime stack and are thus inaccessible as formally defined C parameters. For this reason, a number of functions exist to extract parameters and cast them to a specific data type. Remember also that although the XVM is typeless, C is far from it. Because of this, parameters ultimately have to be resolved in the form of a specific C data type. Let's look at the prototypes of the functions you'll have to work with:

```
int XS_GetParamAsInt ( int iThreadIndex, int iParamIndex );
float XS_GetParamAsFloat( int iThreadIndex, int iParamIndex );
char * XS_GetParamAsString ( int iThreadIndex, int iParamIndex );
```

Pretty straightforward, right? Just pass it the index of the thread that called the function and the index of the parameter you want (starting from zero, left to right), and it will cast the `Value` structure residing at the proper stack index to the specified data type, effectively returning the parameter. This also explains why the host API function prototype includes the thread index as its parameter.

The implementation of these functions is also pretty easy. Let's look at `XS_GetParamAsInt ()`:

```
int XS_GetParamAsInt ( int iThreadIndex, int iParamIndex )
{
    // Get the current top element
    int iTopIndex = g_Scripts [ g_iCurrThread ].Stack.iTopIndex;
    Value Param = g_Scripts [ iThreadIndex ].Stack.pElmnts
        [ iTopIndex - ( iParamIndex + 1 ) ];

    // Coerce the top element of the stack to an integer
    int iInt = CoerceValueToInt ( Param );

    // Return the value
    return iInt;
}
```

CAUTION

`XS_GetParamAsString ()` will return a *pointer* to the string value residing on the stack. Because this value may change frequently as the script executes, it's best to use `strcpy ()` to make a copy of the string if you plan on storing the value for later use by the host. Of course, because the script won't have a chance to execute in any way during the host API function's execution (unless you call a script function from it), you can safely use the pointer alone in the short term.

The function first extracts the parameter's `Value` structure from the stack by subtracting the index of the parameter from the top of the script. It then coerces the value structure to the desired type (an integer in this case) with a call to

CoerceValueToInt () and returns it. This pattern is followed by the other two functions, but you can see for yourself by checking out the included XVM source code on the accompanying CD.

Returning Values

Returning values is almost criminally easy. Because the Value structure behind the _RetVal register is freely available in the thread's corresponding Script structure, all you need to do is assign this a new value when the host API function exits. This is done with the XS_Return*FromHost () functions:

```
void XS_ReturnIntFromHost ( int iThreadIndex,
                            int iParamCount, int iInt );
void XS_ReturnFloatFromHost ( int iThreadIndex, int iParamCount, float
    iFloat );
void XS_ReturnStringFromHost ( int iThreadIndex,
                               int iParamCount, char *
    pstrString );
```

The implementation of these functions is even simpler than the parameter retrieving functions described in the last section, but here's the definition of XS_ReturnIntFromHost () anyway:

```
void XS_ReturnIntFromHost ( int iThreadIndex,
                            int iParamCount, int iInt )
{
    // Clear the parameters off the stack
    g_Scripts [ iThreadIndex ].Stack.iTopIndex -= iParamCount;

    // Put the return value and type in _RetVal
    g_Scripts [ iThreadIndex ]._RetVal.iType = OP_TYPE_INT;
    g_Scripts [ iThreadIndex ]._RetVal.iIntLiteral = iInt;
}
```

The function first clears the parameters the caller pushed onto the stack and then stores the return value in _RetVal. So in actuality, this function does more than just return a value; it cleans up the function as well (as it should). The other functions of course follow this pattern as well, as you'd expect, but there is one detail in the process of returning a string that should be mentioned. First, let's look at the code:

```
void XS_ReturnStringFromHost ( int iThreadIndex,
                               int iParamCount, char *
                               pstrString )
```

```
{
    // Clear the parameters off the stack
    g_Scripts [ iThreadIndex ].Stack.iTopIndex -= iParamCount;

    // Put the return value and type in _RetVal
    Value ReturnValue;
    ReturnValue.iType = OP_TYPE_STRING;
    ReturnValue.pstrStringLiteral = pstrString;
    CopyValue ( & g_Scripts [ iThreadIndex ]._RetVal,
        ReturnValue );
}
```

Instead of simply assigning the string pointer to _RetVal, it's first encapsulated by a Value structure and then physically copied into _RetVal with the CopyValue () you saw in the last chapter when implementing the Mov instruction. This is done to prevent any mix-ups from occurring if the host application later makes changes to the string it returned. Because the script and the host would be sharing the same string pointer, this would inadvertently result in unpredictable values and even more unpredictable behavior, possibly even resulting in a crashing. So, CopyValue () is used to make sure that a copy of the string is made for the script.

The only real shortcoming with these functions is that regardless of their intent, they won't actually cause the host API function to return. This is somewhat inconvenient, because it'd be nice to simply end the function with one of the XS_Return*FromHost () functions in the same way C's return statement would. To solve this problem, I've wrapped each of these functions in simple macros that bundle the function call with a return statement. Here's an example:

```
#define XS_ReturnInt( iThreadIndex, iParamCount, iInt )       \
{                                                             \
    XS_ReturnIntFromHost ( iThreadIndex, iParamCount, iInt ); \
    return;                                                   \
}
```

Now, in a single line, you can return from host API functions and automatically return values to the calling script with ease. Because these macros need to be directly available to the host, they're defined in the XVM header file. You can find them there if you'd like to study them further.

Lastly, there's the issue of returning from a host API function without a return value. This doesn't seem like anything worth mentioning at first, until you remember that the XS_Return*FromHost () functions clear the function's parameters from the stack *as well* as return a value; therefore, whether a return value is involved or not, all host API functions must do this somehow. To address this issue, I added a new function and accompanying macro, XS_ReturnFromHost () and XS_Return ():

```
void XS_ReturnFromHost ( int iThreadIndex, int iParamCount )
{
    // Clear the parameters off the stack
    g_Scripts [ iThreadIndex ].Stack.iTopIndex -= iParamCount;
}

#define XS_Return( iThreadIndex, iParamCount )           \
{                                                        \
    XS_ReturnFromHost ( iThreadIndex, iParamCount );     \
    return;                                              \
}
```

With these functions and the macros that wrap them, defining host API functions is easy and well structured. Once you get used to their use, it becomes just as natural as writing any other C function.

Calling the Function from a Script

Let's wrap up the host API with an example of calling one of its functions from a script. Before the function can be called, however, it needs to be registered of course:

```
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC,
                         "PrintString", HAPI_PrintString );
```

This line of code defines a global function called `PrintString ()` that will invoke the host's own `HAPI_PrintString ()` function when called from a script. Remember, because the host application provides the name by which the function will be known to the script, it's free to use a different one than it's defined with in C. Here I've chosen to omit the `HAPI_` prefix, but this was purely arbitrary. Here's a script fragment of the call:

```
Var             MyString
Push            4
Push            "This is a string!"
CallHost        PrintString
Mov             MyString, _RetVal
```

TIP

Aside from enhanced readability, there's no particularly significant reason to use the `HAPI_` prefix when defining the name by which a host API function will be known to the script. This is because there's no possibility for name clashing, because the host API call table is entirely separate from the function table. Of course, having a script-defined function with the same name as a host API function can be confusing, so it's best to either avoid this practice or use a prefix of some sort.

Remember, parameters are always pushed in the reverse order in which they're read, so you push the count before the string in this case (because `HAPI_PrintString ()` read the string first). `CallHost` then calls the function, and the XVM takes over from there. After the function returns, any return value it may have issued will be available in `_RetVal`. In this example, this value was placed in the variable `MyString`.

Script Function Calls

Calling the host API is a reasonably straightforward procedure from beginning to end, but calling script functions from the host is considerably more complicated. The applications of such a feature are far reaching, however, and important to keep in mind while attempting to implement it. The most obvious of these is event handling; by calling a script's function due to a specific condition detected by the game engine, scripts can be fitted to the game's behavior even more closely—on a function level as opposed to a script level.

Exporting Function Names for Late Binding

Before you can do anything, however, the XVM needs to know the name of each function the script defines so the host can call them easily. Calling a function by name is much easier than using an index, so it's important that your system supports this capability. In order to do this, however, XASM needs to be rewritten slightly so that it writes the name of each function in its internal function table to the final .XSE file so the XVM can read them back out (this will require you to update the .XSE format as well). The process of saving a function or variable's identifier beyond the compilation and assembly phases so that it can be referenced at runtime is known as *late binding*.

Like I said, in order to achieve late binding, you need to update XASM's `BuildXSE ()` function and update the .XSE format just slightly so that function names can persist beyond the assembler. On the XVM side of things, the function table structure will need to be expanded to store a name string, and `XS_LoadScript ()` will need to be updated as well. Let's start with the changes to the assembler.

Updating the .XSE Format

The .XSE format currently doesn't have room for a function's name, so you need to make some alterations to the function table section of its structure. Table 11.2 contains the new specification for a member of the .XSE function table.

These changes are small, but any change to a file's format is a significant move. Because of this, it'd be a good idea to update the format version as well. All .XSE's created with the new function table specification will identify themselves as version 0.8 scripts.

Table 11.2 The Function Structure

Name	Size (in Bytes)	Description
Entry Point	4	The index of the first instruction of the function
Parameter Count	1	The number of parameters the function accepts
Local Data Size	4	The total size of the function's local data (the sum of all local variables and arrays)
Function Name Length	1	The length of the following function name, in bytes
Function Name	N	The function name string

Updating XASM

The changes that must be made to XASM are minimal to say the least—it's just a matter of writing the name string along with each function record that's written to the .XSE's function table. Here's the code responsible for emitting the assembled function table in the assembler's `BuildXSE ()` function, with the new code in bold:

```
// Write out the function count (four bytes)
fwrite ( & g_FuncTable.iNodeCount, 4, 1, pExecFile );

// Set the pointer to the head of the list
pNode = g_FuncTable.pHead;

// Loop through each node in the list and
// write out its function info
for ( iCurrNode = 0; iCurrNode < g_FuncTable.iNodeCount;
      ++ iCurrNode )
{
    // Create a local copy of the function
    FuncNode * pFunc = ( FuncNode * ) pNode->pData;
    // Write the entry point (4 bytes)
    fwrite ( & pFunc->iEntryPoint, sizeof ( int ),
             1, pExecFile );

    // Write the parameter count (1 byte)
    cParamCount = pFunc->iParamCount;
    fwrite ( & cParamCount, 1, 1, pExecFile );

    // Write the local data size (four bytes)
    fwrite ( & pFunc->iLocalDataSize, sizeof ( int ),
             1, pExecFile );

    // Write the function name length (1 byte)
    char cFuncNameLength = strlen ( pFunc->pstrName );
    fwrite ( & cFuncNameLength, 1, 1, pExecFile );

    // Write the function name (N bytes)
    fwrite ( & pFunc->pstrName,
             strlen ( pFunc->pstrName ), 1, pExecFile );

    // Move to the next node
    pNode = pNode->pNext;
}
```

The .XSE files generated by this updated version of XASM will now contain the name of each function, much like the names referenced by each host API call are stored. To reflect this new change, the XASM program's version will also be updated to 0.8. This is done by changing the `VERSION_*` constants:

```
#define VERSION_MAJOR       0    // Major version number
#define VERSION_MINOR       8    // Minor version number
```

Invoking a Script Function: Synchronous Calls

If you recall from an earlier section, I defined *synchronous calls* as calls that do not interrupt the concurrent flow of execution within the script. Such calls do not immediately execute, as would the call to a typical C function; rather, they begin with the next XVM time slice, and (usually) execute over the course of multiple time slices thereafter. Because of this, the only difference between a synchronous call from the host and one made directly by the script is where the call came from; once the function is invoked, everything runs like a typical function called with the `Call` instruction. Figure 11.30 demonstrates a synchronous call.

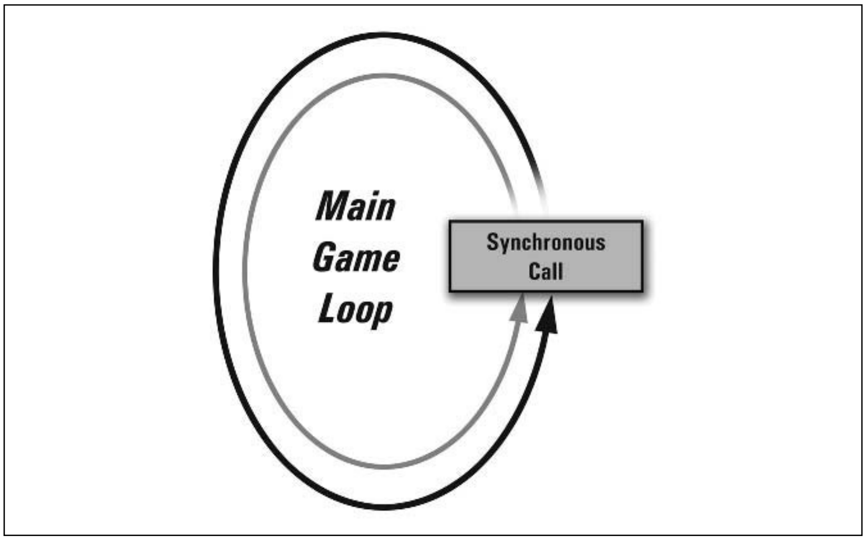


Figure 11.30

Synchronous function calls follow the existing flow of the scripts and game engine, and therefore execute over time as opposed to immediately.

I also like to refer to synchronous calls as *invoking* a script function, and asynchronous calls as *calling* a script function. For this reason, synchronous calls are made with the `XS_InvokeScriptFunc ()` function:

```
void XS_InvokeScriptFunc ( int iThreadIndex, char * pstrName );
```

Simple, huh? Pass it the thread index in which the function resides, as well as the function's name, and it'll begin executing as soon as the next call to `XS_RunScripts ()` is made. The implementation of this function is decidedly simple, because it directly leverages the code you wrote for calling functions when implementing the `Call` instruction in the last chapter. However, in order to do this, `Call`'s code will have to be taken out of its `case` block in `XS_RunScripts ()`'s, and placed in a separate function called `CallFunc ()`. Here's the code for this new function:

```
void CallFunc ( int iThreadIndex, int iIndex )
{
    Func DestFunc = GetFunc ( iThreadIndex, iIndex );

    // Save the current stack frame index
    int iFrameIndex = g_Scripts [ iThreadIndex ].Stack.iFrameIndex;

    // Push the return address, which is the current instruction
    Value ReturnAddr;
    ReturnAddr.iInstrIndex = g_Scripts
                             [ iThreadIndex ].InstrStream.iCurrInstr;
    Push ( iThreadIndex, ReturnAddr );
    // Push the stack frame + 1 (the extra space is
    // for the function index we'll put on the stack after it)
    PushFrame ( iThreadIndex, DestFunc.iLocalDataSize + 1 );

    // Write the function index and old stack frame
    // to the top of the stack
    Value FuncIndex;
    FuncIndex.iFuncIndex = iIndex;
    FuncIndex.iOffsetIndex = iFrameIndex;
    SetStackValue ( iThreadIndex, g_Scripts
                    [ iThreadIndex ].Stack.iTopIndex
                    - 1, FuncIndex );

    // Let the caller make the jump to the entry point
    g_Scripts [ iThreadIndex ].InstrStream.iCurrInstr=
        DestFunc.iEntryPoint;
}
```

Nothing's changed, the function call logic is just embedded in a function now. Of course, this has a serious effect on the `Call` instruction handler, so let's look at its new incarnation:

```
case INSTR_CALL:
{
    // Get a local copy of the function index
    int iFuncIndex = ResolveOpAsFuncIndex ( 0 );

    // Advance the instruction pointer so it points
    // to the instruction immediately following the call
    ++ g_Scripts [ g_iCurrThread ].InstrStream.iCurrInstr;

    // Call the function
    CallFunc ( g_iCurrThread, iFuncIndex );

    break;
}
```

Now it's just a matter of passing some parameters to the `CallFunc ()` function. The instruction pointer is also incremented outside of the function, because, as you'll see shortly, a synchronous call from the script should *not* advance the instruction.

With the function call logic of the XVM embodied in a more modular way, you can implement XS_InvokeScriptFunc () easily. Here's the code:

```
void XS_InvokeScriptFunc ( int iThreadIndex, char * pstrName )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // Get the function's index based on its name
    int iFuncIndex = GetFuncIndexByName ( iThreadIndex, pstrName );

    // Make sure the function name was valid
    if ( iFuncIndex == -1 )
        return;

    // Call the function
    CallFunc ( iThreadIndex, iFuncIndex );
}
```

The function begins with the IsThreadActive () macro used in some of the previous sections to ensure that the specified thread is active and running. A call is then made to a new helper function, GetFuncIndexByName (), which accepts the name of a function, as well as the index of the thread in which the function is thought to reside, and attempts to find its index in the script's function table. The function returns -1 if the index isn't found. Lastly, CallFunc () is called with the newly found function index, and the process is complete.

As I mentioned previously, this is why you don't want to increment the instruction pointer within CallFunc () itself—synchronous function calls have no effect on the current execution path of the script, so IP should be left untouched.

By the way, here's the source to GetFuncIndexByName (); it should speak for itself:

```
int GetFuncIndexByName ( int iThreadIndex, char * pstrName )
{
    // Loop through each function and look for a matching name
    for ( int iFuncIndex = 0; iFuncIndex < g_Scripts
        [ iThreadIndex ].FuncTable.iSize; ++ iFuncIndex )
    {
        // If the names match, return the index
        if ( stricmp ( pstrName, g_Scripts
            [ iThreadIndex ].FuncTable.pFuncs
            [ iFuncIndex ].pstrName )
            == 0 )
            return iFuncIndex;
    }
    // A match wasn't found, so return -1
    return -1;
}
```

Nothing to it—just scan through the array until the specified function name matches something, and return the corresponding index. Return -1 if a match isn't found.

Passing Parameters

Calling functions without parameters is a decent capability, and is more than useful for a number of situations. It won't be long, however, before you want more specific control and need to pass parameters from the host application to the script's function.

As you might expect, this is done in the same way it's done in a script; by pushing them onto the stack before making the call. Of course, the host application has no direct interface to a specific thread's stack within the XVM, so you'll need to create yet another batch of helper functions to provide one:

```
void XS_PassIntParam ( int iThreadIndex, int iInt );
void XS_PassFloatParam ( int iThreadIndex, float fFloat );
void XS_PassStringParam ( int iThreadIndex, char * pstrString );
```

These shouldn't need much explanation—by passing them either an integer, floating point value, or string, these functions will push them onto the stack of the specified thread. Because this is exactly what the script does when it calls one of its own functions, this will solve your parameter passing problem nicely. Here's the code to `XS_PassIntParam ()`:

```
void XS_PassIntParam ( int iThreadIndex, int iInt )
{
    // Create a Value structure to encapsulate the parameter
    Value Param;
    Param.iType = OP_TYPE_INT;
    Param.iIntLiteral = iInt;
    // Push the parameter onto the stack
    Push ( iThreadIndex, Param );
}
```

Nothing tricky going on here. The parameter comes in, it's stuffed into a `Value` structure called `Param`, and is pushed onto the stack, as shown in Figure 11.31. Done deal. Of course, like always, strings have to ruin the fun and require a bit of special attention:

```
void XS_PassStringParam ( int iThreadIndex, char * pstrString )
{
    // Create a Value structure to encapsulate the parameter
    Value Param;
    Param.iType = OP_TYPE_STRING;
    Param.pstrStringLiteral =
        ( char * ) malloc ( strlen ( pstrString ) + 1 );
    strcpy ( Param.pstrStringLiteral, pstrString );
    // Push the parameter onto the stack
    Push ( iThreadIndex, Param );
}
```

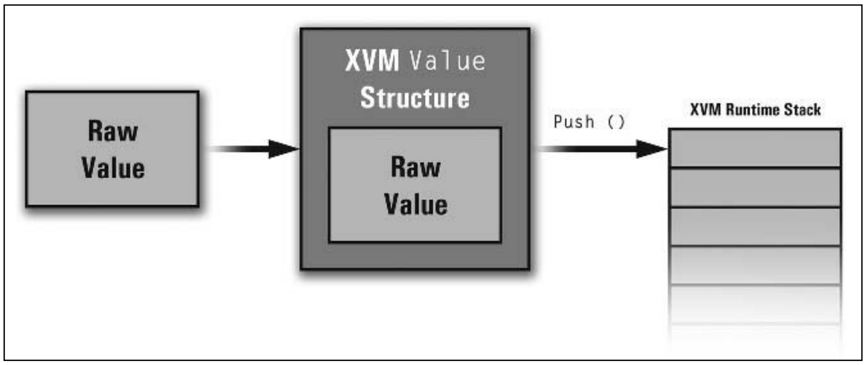


Figure 11.31

Passing parameters from the host by encapsulating them in a `Value` structure and pushing them onto the thread's runtime stack.

The difference here is that a copy of the supplied string is made before the `Value` structure is pushed onto the stack. Remember, just like always, *whenever* a string is passed from the host to the script or vice versa, it's important to make a physical copy of the string data to ensure that changes made to the original pointer on either side won't affect the other.

NOTE

Remember, parameters need to be passed in the proper order from the host as well—either right to left order, or left to right, depending on the function's implementation.

Return Values

Return values aren't really possible when making synchronous calls, because there's no obvious point at which the function ends from the perspective of the host. Because of this, it never receives a concrete signal to extract the value of `_RetVal`, which is where the return value would be.

Fortunately, this really isn't a problem. Synchronous calls aren't meant to be used to calculate values or return information about the script; rather, they're meant for long-term behavior and actions. For example, if an enemy's AI was implemented in functions that corresponded to each of its major behavioral states, each of which contained an infinite loop that would run until it was preempted by another function, synchronous calls could be made by the game engine to branch to another state in reaction to any number of stimuli.

TIP

If you *really* want to be able to receive return values from synchronous calls, there's at least one way to go about doing it. All that's required is to flag the function so that the XVM knows to pass the value of `_RetVal` back to the host when a `Ret` instruction is encountered. This may be a decent amount of extra work to get fully operational, but it's a perfectly good solution if you really need such a capability for whatever reason.

CAUTION

It's *extremely* important to remember that synchronous calls can have a disastrous effect on the running script if they're used without caution. Remember, any global variables that the function modifies may be in use when the call is made, which will wreak havoc when the function returns and execution within the script resumes where it originally was. Imagine writing a C program if you had to worry about the possibility of random global variables suddenly changing their values without warning. So, any function you know will be called from the host should be designed to play nice with *whatever* code may already be running when it's called—this means that at the very least, it should save the value of any global variable it modifies and restore it before returning to ensure that the originally executing code isn't hosed by its sudden invocation. Even better, you may want to alter the VM so that it automatically preserves the value of a script's globals, as well as `_RetVal`, before executing a synchronous call. These values would then be restored when the function's `Ret` instruction is executed, providing more reliable protection.

Calling a Scripting Function: Asynchronous Calls

Asynchronous calls are just the opposite of synchronous calls. Rather than executing over time, within the script's time slices like a function called internally with `Call`, asynchronous calls use script code to simulate C functions defined within the host. They begin executing immediately, halt the program until it's finished, and optionally return a value. Asynchronous calls are good for making quick or immediate changes to the script, reading the value of a script variable wrapped in a "getter" function, or any other task that must be executed within the script, but immediately. Figure 11.32 demonstrates asynchronous calls.

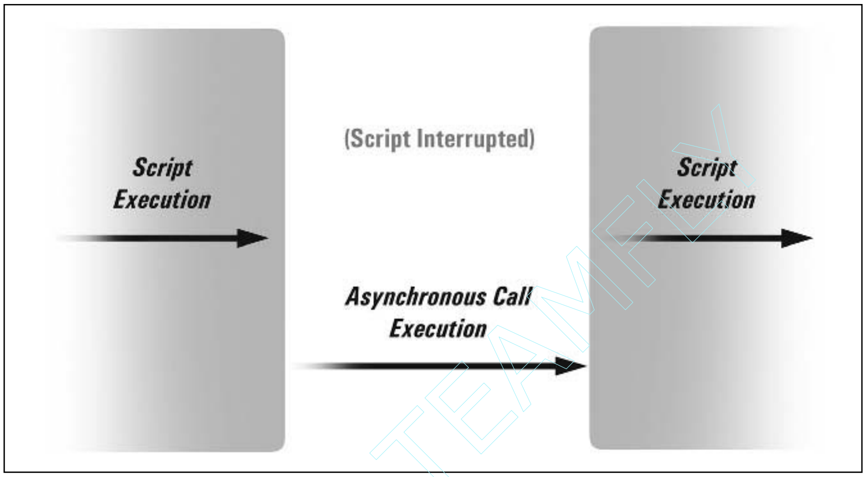


Figure 11.32

Asynchronous function calls interrupt the flow of execution for both the game engine and the script.

On the surface, this almost makes asynchronous calls seem the simpler of the two; after all, they don't seem to disrupt the flow of the script's execution or even require `XS_RunScripts ()` to be directly called by the user. Ironically, this is exactly what makes them so tricky. Remember, whether or not the script executes over time, script code is always executed in the same way—by sequentially handling instructions, incrementing the instruction pointer, and so on.

Because of this, the XVM needs to execute in a dramatically different way when handling asynchronous function calls. The following is a list of major changes that must take place when such calls are executed:

- **The XVM must suppress the multithreading scheduler.** When an asynchronous function call is made, it takes place *outside* of the normal execution of the XVM. Even though the XVM still physically handles the execution, time slicing and multithreading must be ignored. After all, the host is calling a single function in a single script—if other threads were allowed to execute concurrently with this call, it'd result in countless serious side effects.
- **The function should not be limited by the length of its script's time slice.** Remember, an asynchronous call is completed in full immediately. Because of this, in addition to suppressing the context switches that usually take place on a regular basis, the script within which the asynchronous call is executing must be given as much time to execute as it needs.

- **The function must return upon execution of the proper `Ret` instruction.** When the asynchronously called function returns, control must be returned to the host application, not the script. On the surface the solution to this problem may seem as easy as halting execution of the script when the first `Ret` is encountered, but this won't work if the function ends up calling functions of its own, because the second function's `Ret` would end up terminating everything.

This should help you understand why asynchronous script calls are nontrivial to be sure. You can attack the problem systematically, however, so let's just knock out each of the issues raised by this list one by one.

Threading Modes

The first and most obvious problem with using the XVM as-is to execute an asynchronous function call is that the other threads in the system will end up executing during the function's lifespan as well. Because this can easily result in undesirable side effects, the multithreading scheduler must be suppressed when an asynchronous call is in progress.

To do this, you need to introduce the concept of *threading modes* to the XVM. As the name implies, a threading mode is simply a mode of operation for the scheduler; in this case, all you need is the existing multithreading mode and a new single-threading mode. In single-threading mode, the scheduler is bypassed entirely at each iteration of the execution cycle, effectively suppressing context switches. You start by creating two constants to represent these modes:

```
#define THREAD_MODE_MULTI       0    // Multithreaded execution
#define THREAD_MODE_SINGLE      1    // Single-threaded execution
```

A new global variable is also introduced, to represent the current mode:

```
int g_iCurrThreadMode;                    // The current threading mode
```

This variable is checked at each iteration of the execution cycle just before the scheduler checks for a context switch. The following changes are made to the main `while` loop of XS_`RunScripts ()`, and are displayed in bold:

```
// Check for a context switch if the threading
// mode is set for multithreading
if ( g_iCurrThreadMode == THREAD_MODE_MULTI )
{
    // If the current thread's time slice has elapsed, or
    // if it's terminated, switch to the next valid thread
```

```
        if ( iCurrTime > g_iCurrThreadActiveTime + g_Scripts
            [ g_iCurrThread ].iTimesliceDur ||
            ! g_Scripts [ g_iCurrThread ].iIsRunning )
        {
            // Loop until the next thread is found
            while ( TRUE )
            {
                // Move to the next thread in the array
                ++ g_iCurrThread;
                // If we're past the end of the array, loop back around
                if ( g_iCurrThread >= MAX_THREAD_COUNT )
                    g_iCurrThread = 0;
                // If the thread we've chosen is active and running,
                // break the loop
              if ( g_Scripts [ g_iCurrThread ].iIsActive && g_Scripts
                    [ g_iCurrThread ].iIsRunning )
                break;
            }
            // Reset the time slice
            g_iCurrThreadActiveTime = iCurrTime;
        }
    }
```

Now, by setting `g_iCurrThreadMode` to `THREAD_MODE_SINGLE`, the scheduler will be disabled and the first piece of the puzzle falls into place. Figure 11.33 demonstrates the switch from multithreading to single-threading and back again.

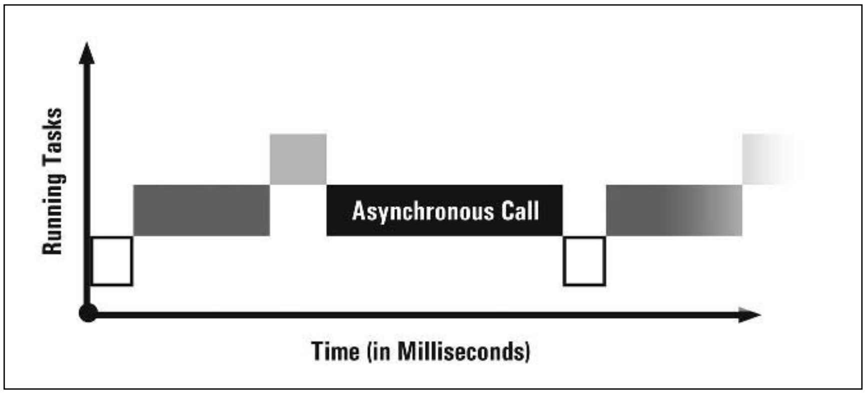


Figure 11.33

The XVM scheduler can switch between the multi- to single-threaded modes at will.

The Stack Base

The next issue is a bit more subtle, but vitally important nonetheless. As I said, it's important that the asynchronous function return control to the host as soon as it finishes executing, rather than returning control to the originally running part of the script. Like I also said, it's tempting to simply solve this problem by creating a flag that tells `XS_RunScripts ()` to return as soon as a `Ret` instruction is executed. Asynchronous calls could then set this flag before entering the script code and ensure that only the desired function would execute.

The problem with this solution is that it robs the function of the capability to call functions of its own, which is obviously a common operation in any type of programming. There is one way to determine when the proper function has ended, however, and that's by monitoring its particular area of the stack. When the stack frame of the asynchronously called function is cleared, you know for certain that call is complete and control can be returned to the host.

In order to determine when the function's frame is cleared, you can set what I like to call a *stack base marker*. A stack base marker is a modification that can be made to any `Value` structure on the stack in order to flag it as the base of the current asynchronous call. Specifically, however, it can be set on the element you're currently using to mark the top of a stack frame—the function index. Whenever a `Ret` instruction is executed, the function index is the first thing it pops off the top of the stack. In addition to using this to determine the size of the frame and other information, it can also be used to determine whether the XVM should halt. Check out Figure 11.34.

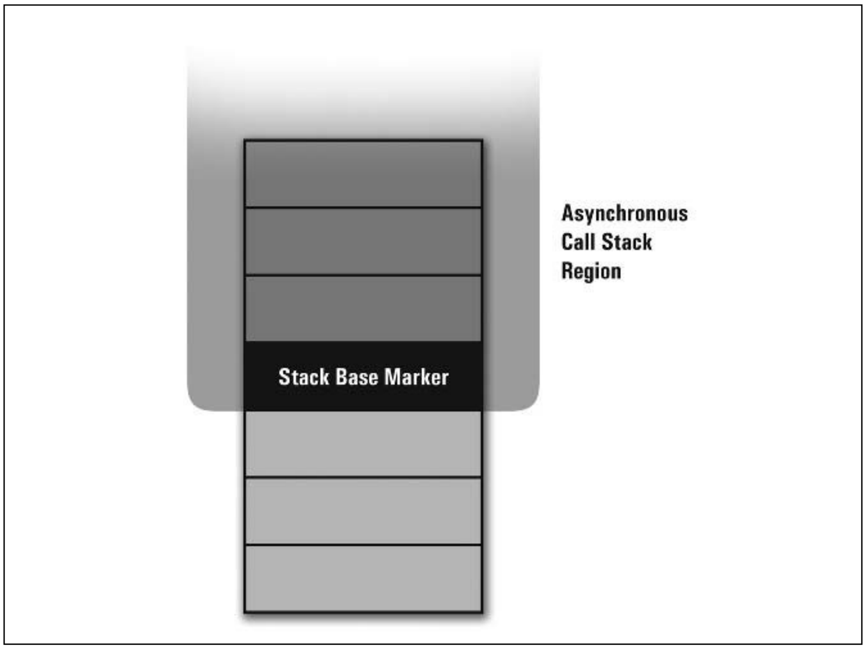


Figure 11.34

Using a stack base marker to alert the XVM when the asynchronously called function returns.

You can implement the stack base marker as a simple `Value` type constant. In addition to `OP_TYPE_INT` and `OP_TYPE_REG`, you now have `OP_TYPE_STACK_BASE_MARKER`:

```
#define OP_TYPE_STACK_BASE_MARKER   9      // Marks a stack base
```

Creating the marker is a simple matter of setting the `iType` field of the function index's `Value` structure at the top of the stack to this constant. Lastly, a small addition is made to `Ret` in order to check for the marker's presence:

```
case INSTR_RET:
{
    // Get the current function index off the top of the stack
    // and use it to get the corresponding function structure
    Value FuncIndex = Pop ( g_iCurrThread );

    // Check for the presence of a stack base marker
    if ( FuncIndex.iType = OP_TYPE_STACK_BASE_MARKER )
        iExitExecLoop = TRUE;

    // Get the previous function index
    Func CurrFunc = GetFunc ( g_iCurrThread, FuncIndex.iFuncIndex );
    int iFrameIndex = FuncIndex.iOffsetIndex;

    // Read the return address structure from the stack, which is
    // stored one index below the local data
    Value ReturnAddr = GetStackValue ( g_iCurrThread, g_Scripts
       [ g_iCurrThread ].Stack.iTopIndex
            - ( CurrFunc.iLocalDataSize + 1 ) );

    // Pop the stack frame along with the return address
    PopFrame ( CurrFunc.iStackFrameSize );

    // Restore the previous frame index
    g_Scripts [ g_iCurrThread ].Stack.iFrameIndex = iFrameIndex;

    // Make the jump to the return address
    g_Scripts [ g_iCurrThread ].InstrStream.iCurrInstr
        = ReturnAddr.iInstrIndex;

    break;
}
```

The instruction works just like it always did, except that any function index element whose iType field has been modified to mark the base of the stack will cause the execution loop to terminate. This, in combination with the capability to run in a single-threaded mode, is almost everything you need to safely execute an asynchronous function call.

An Infinite Time Slice

At the bottom of the execution cycle loop in RunScripts (), the time slice allotted to the XVM by the caller is checked to determine whether the scripts should stop executing. This is normally a must for concurrent execution with a game loop, but asynchronous calls need to run for as long as they need to finish what they're doing. This problem can be solved by creating a new constant that represents an infinite time slice. This value can then be passed to RunScripts () in place of a normal time slice value, telling it to run forever. Here's the constant:

```
#define XS_INFINITE_TIMESLICE  -1  // Allows a thread to run indefinitely
```

Once inside RunScripts (), the usual time slice test needs to be altered to take the constant into account:

```
// If we aren't running indefinitely, check
// to see if the main time slice
// has ended
if ( iTimesliceDur != XS_INFINITE_TIMESLICE )
    if ( iCurrTime > iMainTimesliceStartTime + iTimesliceDur )
        break;
```

NOTE

The XS_INFINITE_TIMESLICE constant is public because it's sometimes useful for the host to run the XVM entirely on its own. Fortunately, even when an infinite time slice is requested, the XVM will still stop executing when all scripts in memory stop running due to an Exit instruction. Of course, if a script with an infinite loop of its own is loaded and run with an infinite time slice, the program will ultimately hang.

The Final XS_CallScriptFunc () Function

With the requirements met, you can combine everything into a single function for making asynchronous function calls. Here's the definition for XS_CallScriptFunc ():

```
void XS_CallScriptFunc ( int iThreadIndex, char * pstrName )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return;

    // ---- Calling the function

    // Preserve the current state of the VM
    int iPrevThreadMode = g_iCurrThreadMode;
    int iPrevThread = g_iCurrThread;

    // Set the threading mode for single-threaded execution
    g_iCurrThreadMode = THREAD_MODE_SINGLE;

    // Set the active thread to the one specified
    g_iCurrThread = iThreadIndex;

    // Get the function's index based on its name
    int iFuncIndex = GetFuncIndexByName ( iThreadIndex, pstrName );

    // Make sure the function name was valid
    if ( iFuncIndex == -1 )
        return;

    // Call the function
    CallFunc ( iThreadIndex, iFuncIndex );

    // Set the stack base
    Value StackBase = GetStackValue ( g_iCurrThread, g_Scripts
        [ g_iCurrThread ].Stack.iTopIndex - 1 );
    StackBase.iType = OP_TYPE_STACK_BASE_MARKER;
    SetStackValue ( g_iCurrThread, g_Scripts
        [ g_iCurrThread ].Stack.iTopIndex - 1, StackBase );

    // Allow the script code to execute uninterrupted until the
    // function returns
    XS_RunScripts ( XS_INFINITE_TIMESLICE );

    // ---- Handling the function return
    // Restore the VM state
    g_iCurrThreadMode = iPrevThreadMode;
    g_iCurrThread = iPrevThread;
}
```

The function begins with the usual check to determine whether the specified thread index is valid and active. It then saves the current threading mode and thread index. This is done to restore the XVM to the *exact* state it was in before the call was made. As for exactly why the threading mode needs to be restored, remember that asynchronous calls can end up interrupting other asynchronous calls, in which case the threading mode can't automatically be set back to `THREAD_MODE_MULTI`.

But how can one asynchronous call interrupt another? It's rare, but imagine the following scenario: an asynchronous call is made to a function that calls a host API function. If, for whatever reason, the host API function makes an asynchronous call to *another* script function, that function will end up being pushed onto the stack above the previous one. Even though they're two separate function calls, they're both asynchronous. Because of this, if the threading mode was always blindly restored to multithreading whenever an asynchronous call returned, the first call wouldn't behave properly. This is a pretty visual concept, so check out Figure 11.35.

Getting back to the function, the threading mode is then set to `THREAD_MODE_SINGLE`. The current thread is then changed to the index of the function. The index of the desired function is then retrieved based on the specified name, and the function is called with `CallFunc ()`. The stack base

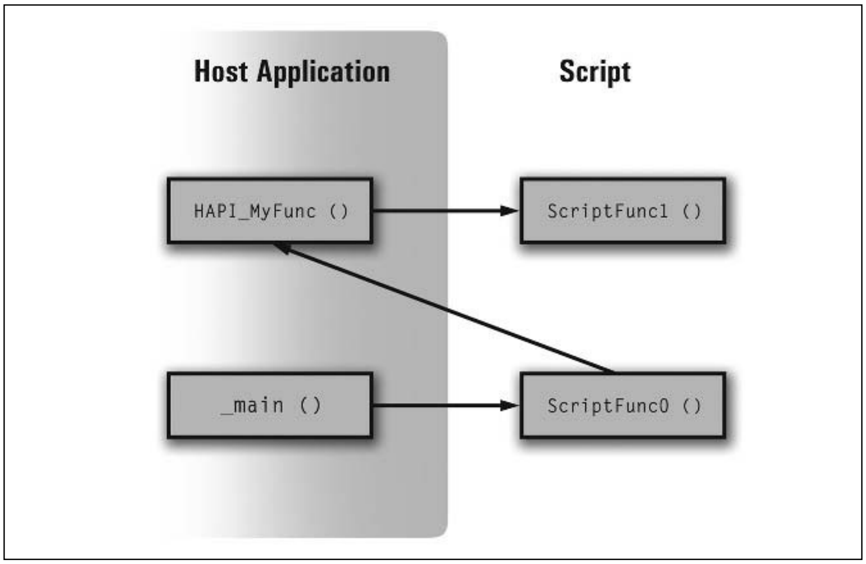


Figure 11.35

Asynchronous calls can interrupt each other if the script function calls the host, which in turn calls the script back.

marker is then set; the top stack element, containing the function index, is read from the stack and changed to OP_TYPE_STACK_BASE_MARKER. The modified function index is then written back to the stack in the same position, and XS_RunScripts () is called with an infinite time slice. This will execute the function in isolation until it returns, at which point the state of the VM (the previous threading mode and the executing thread) is restored. Presto!

Reading Return Values

Parameters are passed to asynchronous calls in the same way they're passed to synchronous ones—with the parameter passing helper functions defined earlier. Unlike synchronous calls, however, asynchronous calls can return values due to their immediate nature. Once again, however, because this means accessing an XVM runtime stack, you need to create some helper functions to assist the host application. Here are the prototypes:

```
int XS_GetReturnValueAsInt ( int iThreadIndex );
float XS_GetReturnValueAsFloat ( int iThreadIndex );
char * XS_GetReturnValueAsString ( int iThreadIndex );
```

The function definitions are almost obscenely simple; all they do is return the value of the specified thread's _RetVal register. Check out this example:

```
int XS_GetReturnValueAsInt ( int iThreadIndex )
{
    // Make sure the thread index is valid and active
    if ( ! IsThreadActive ( iThreadIndex ) )
        return 0;
    // Return _RetVal's integer field
    return g_Scripts [ iThreadIndex ]._RetVal.iIntLiteral;
}
```

The only other detail to mention is yet another string issue; when accepting a string return value, make sure to save a physical copy on the host side if you plan on saving it for prolonged periods or making changes to it. This just helps avoid stepping on anyone else's toes unexpectedly.

Adding Thread Priorities

Your current multithreading scheduler gives each thread in the system an equal time slice. There's nothing horrendously wrong with this, but it's important to recognize that certain scripts are more intensive or vital than others, and should be given the maximum amount of time to do their jobs smoothly.

For example, consider a scenario in a game involving four computer-controlled enemy characters and a floating power-up item. The enemies, power-up, and the level itself are all scripted separately, meaning there are currently six threads executing within the XVM. It's most likely that the player is directly interacting with the enemies—he or she may be engaged in a particularly heated battle that requires at least 90 percent of his or her focus. The remaining sliver of the player's attention is spent on the surrounding environment and the power-up (especially if he or she needs it). Figure 11.36 illustrates this situation.

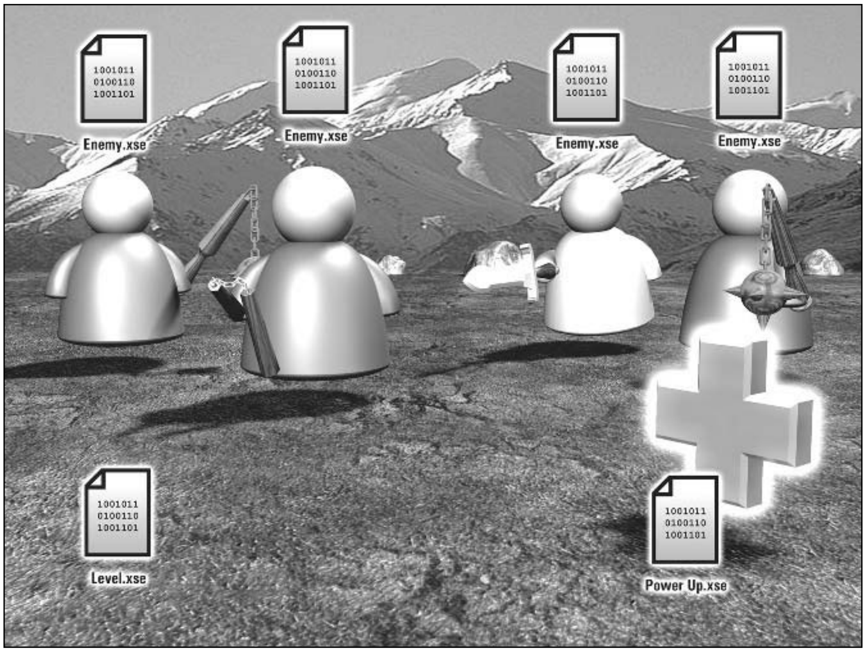


Figure 11.36

A scene in which the level, enemies, and power-ups of a game are scripted.

The level's script is concerned with keeping the ambient entities in motion; it causes leaves in the trees to rustle, tiny waves to move through the water, and rocks and tumbleweed to slide around as if a gust of wind was carrying them. Although this is vital to the overall immersive quality of the game, it doesn't have any direct impact on the player's battle with the enemies. The power-up's script is even simpler; all it does is keep the item bobbing up and down in the air and slowly rotating to get the player's attention.

The scripts that power the enemy's logic, however, are much more intense. They're controlled by complex AI algorithms that allow them to intelligently attack the player and avoid the player's countermoves, upon which the entire game play and fun factor of the game rely. Not only are the enemy scripts more computationally intensive than those of the level and power-up, but they're a lot more important—if the rustling of the trees or the spinning of the power-up were to suddenly become choppy or slow, it really wouldn't make much difference. If the reaction time of the enemies suddenly began to falter, however, the game play experience would be severely jarred. The time graph of Figure 11.37 shows how priority-based threading helps distribute the virtual CPU's load more effectively.

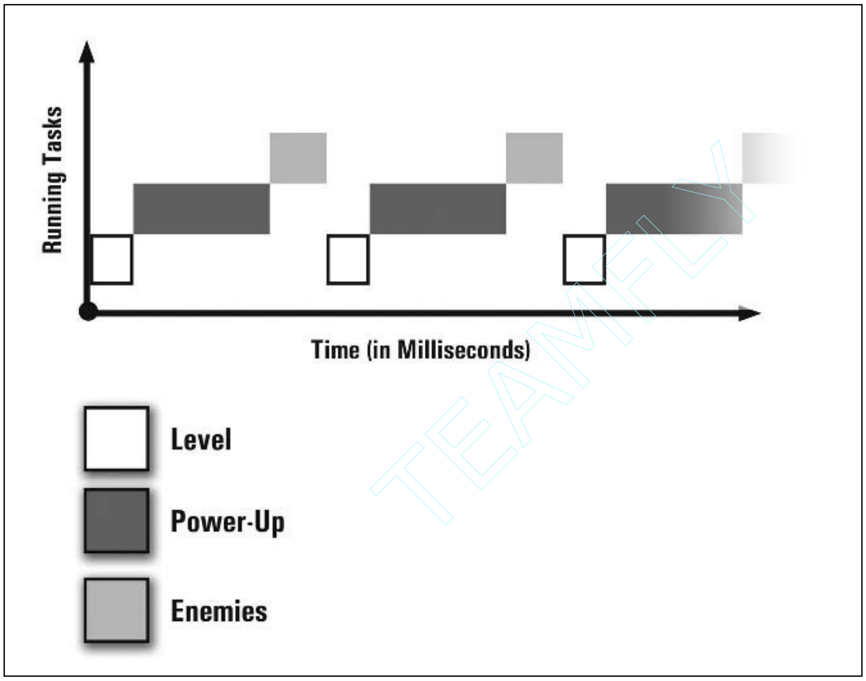


Figure 11.37

Priority-based multi-threading over time.

Because of this, it's important that the scheduler recognize the relative importance of the scripts it distributes the XVM's processing power to. So, as a final touch to your nearly completed XVM module, it'd be nice to go ahead and add the thread priorities I discussed earlier and help the scheduler make more intelligent time slice allocations.

Priority Ranks vs. Time Slice Durations

As mentioned earlier, I like to define priority in terms of a time slice's duration, as opposed to the frequency over time with which it's executed. Higher priority threads are given longer time slices than those with a lower priority, effectively giving them more time overall to do their job. This helps ensure that particularly vital or processor-intensive threads will run smoothly at all times, even if it's at the expense of the lower-priority scripts. As you learned in the last section, however, this is generally a fair compromise.

There are two ways to define a script's priority. On the simplest level, each thread would simply request a specific time slice duration, expressed in milliseconds. For example, a medium-priority might ask for 20 milliseconds, whereas a high priority thread would ask for 50. A lower-priority thread might be content with just 10.

This approach can become tedious, however, if you plan on writing a large number of scripts. Imagine if, after writing 74 scripts that you deemed medium priority and therefore allocated 20 milliseconds, you decided they'd run better with 30. Rooting through all 74 of these scripts and changing their priority request would be a huge pain.

Because of this, I like to also give threads the capability to request a specific *priority rank*, which is a symbolic term or constant that maps to a specific number of milliseconds. If those 74 scripts instead all requested as medium-priority threads, and the exact duration of a medium-priority thread's time slice was defined by the runtime environment, this problem could be averted by simply tweaking the XVM's definition.

Your scripts will therefore be capable of requesting either a specific time slice duration in milliseconds, or one of three priority ranks—low, medium, or high.

Updating the .XSE Format

None of this will be possible without making a minor upgrade to the version 0.8 .XSE format, allowing it to describe a script's priority rank or time slice duration. All this requires is the addition of two new fields to the header—one to define the priority rank, and the other to define the user-defined priority, in milliseconds, if applicable. The priority rank is expressed as a 1-byte code; each of the four valid codes are defined in Table 11.3.

The updated .XSE header is displayed in Table 11.4.

TIP

You may be wondering why user-defined priorities are represented by 0, whereas the ranks themselves are between 1 and 3. I did this to allow the possibility of new priority ranks to be added later on; if the existing priorities were represented by 0 through 2, and user-defined priorities were represented by 3, it'd force any future priorities to begin at 4. I find this numeric discontinuity a bit messy and think this approach ensures a much cleaner way to expand later.

In cases where the user requests a predefined priority rank, the 1-byte code in the Priority Rank field will be a value between 1 and 3, and the user-defined time slice duration field will be a garbage value that should be disregarded. If a specific time slice duration was requested, however, the priority rank field will be zero.

Table 11.3 Updated 0.8 .XSE Main Header

Code	Definition
`0`	User-defined time slice duration (no priority rank)
`1`	Low priority
`2`	Medium priority
`3`	High priority

Table 11.4 Updated 0.8 .XSE Main Header

Name	Size (in Bytes)	Description
ID String	4	Four-character string containing the .XSE ID, "XSE0"
Version	2	Version number; (first byte is major, second byte is minor)
Stack Size	4	Requested stack size (set by `SetStackSize` directive; 0 means use default)
Global Data Size	4	The total size of all global data
Is `_Main ()` Present?	1	Set to 1 if the script implemented a `_Main ()` function, 0 otherwise
`_Main ()` Index	4	Index into the function table at which `_Main ()` resides
Priority Rank	1	The requested priority rank, or user-defined time slice flag
User-Defined Time slice	4	Requested time slice duration, in milliseconds

Updating XASM

Of course, in order to generate this updated version of the 0.8 .XSE format, XASM will need a bit of an upgrade too. Specifically, it needs to produce executables using the new format, and interpret a new directive—SetPriority. The SetPriority directive accepts a single parameter—either an integer literal value corresponding to the desired time slice duration, or one of the following three keywords: Low, Med, or High.

A number of new constants will be introduced into the assembler to support this directive and the new executable format. The first four correspond to the 1-byte codes that are used in the .XSE header to specify the priority type:

```
#define PRIORITY_USER          0      // User-defined priority
#define PRIORITY_LOW           1      // Low priority
#define PRIORITY_MED           2      // Medium priority
#define PRIORITY_HIGH          3      // High priority
```

Next up are string constants that correspond to the three priority-type keywords the SetPriority directive accepts:

```
#define PRIORITY_LOW_KEYWORD    "Low"   // Low priority keyword
#define PRIORITY_MED_KEYWORD    "Med"   // Medium priority keyword
#define PRIORITY_HIGH_KEYWORD   "High"   // High priority keyword
```

The assembler will track a script's priority by also adding two new fields to its internal header structure, ScriptHeader:

```
typedef struct _ScriptHeader            // Script header data
{
    int iStackSize;                     // Requested stack size
    int iGlobalDataSize;                // The size of the script's
                                        // global data

    int iIsMainFuncPresent;             // Is _Main () present?
    int iMainFuncIndex;                 // _Main ()'s function index

    int iPriorityType;                  // The thread priority type
    int iUserPriority;                  // The user-defined priority
                                        // (if any)
}
    ScriptHeader;
```

With the new constants and data structures ready to go, let's look at the code responsible for parsing the new directive.

Parsing the SetPriority Directive

The SetPriority directive is parsed in a manner similar to SetStackSize, so the code should look rather familiar. Let's take an initial look:

```
case TOKEN_TYPE_SETPRIORITY:

    // SetPriority can only be found in the global scope, so make
    // sure you aren't in a function.
    if ( iIsFuncActive )
        ExitOnCodeError ( ERROR_MSSG_LOCAL_SETPRIORITY );

    // It can only be found once, so make sure you
    // haven't already found it
    if ( g_iIsSetPriorityFound )
        ExitOnCodeError ( ERROR_MSSG_MULTIPLE_SETPRIORITIES );

    // Determine the parameter type
    GetNextToken ();
    switch ( g_Lexer.CurrToken )
    {
        // An integer lexeme means the user is
        // defining a specific priority
        case TOKEN_TYPE_INT:
            // Convert the lexeme to an integer value from its string
            // representation and store it in the script header
            g_ScriptHeader.iUserPriority = atoi ( GetCurrLexeme () );
            // Set the user priority flag
            g_ScriptHeader.iStackSize = PRIORITY_USER;
            break;

            // An identifier means it must be one
            // of the predefined priority ranks
            case TOKEN_TYPE_IDENT:
                // Determine which rank was specified
                if ( stricmp ( g_Lexer.pstrCurrLexeme,
                    PRIORITY_LOW_KEYWORD ) == 0 )
                    g_ScriptHeader.iPriorityType = PRIORITY_LOW;
                else if ( stricmp ( g_Lexer.pstrCurrLexeme,
                    PRIORITY_MED_KEYWORD ) == 0 )
                    g_ScriptHeader.iPriorityType = PRIORITY_MED;
                else if ( stricmp ( g_Lexer.pstrCurrLexeme,
                    PRIORITY_HIGH_KEYWORD ) == 0 )
                    g_ScriptHeader.iPriorityType = PRIORITY_HIGH;
                else
                    ExitOnCodeError ( ERROR_MSSG_INVALID_PRIORITY );
                break;

            // Anything else should cause an error
            default:
                ExitOnCodeError ( ERROR_MSSG_INVALID_PRIORITY );
    }

    // Mark the presence of SetStackSize for future encounters
    g_iIsSetPriorityFound = TRUE;

    break;
```

When SetPriority is the initial token, the parser knows it's encountered the directive. It begins by ensuring it's not currently inside a function, because SetPriority can only appear in the global scope. It then makes sure the priority hasn't already been set, because multiple instances are illegal. This is done by checking a global flag called g_iIsSetPriorityFound, which works just like the flag used to ensure that SetStackSize only appears once.

The next token is then read, which is the priority value itself. This can appear in one of two forms—an integer literal value specifying the desired time slice duration, or a string referring to one of the three predefined priority ranks. In the case of an integer, atoi () is used to determine the actual time slice duration, which is saved to the script header. The priority type is then set to PRIORITY_USER to reflect this.

If the token is a string, it must be one of the three priority rank strings. strcmp () is used to determine this, and the corresponding rank value is written to the script header. A string parameter that is not one of these three strings, as well as a parameter that isn't either an integer or a string, result in an error message reporting an invalid priority.

The parsing logic completes by setting the g_iIsSetPriorityFound flag.

Updating the XVM

Lastly, the XVM needs to be updated to support this new priority ranking functionality. Fortunately, it's a pretty simple upgrade.

The Script Structure

First up, the Script function needs to be augmented with a field specifying the script's time slice duration in milliseconds. Here's the new structure with the added field in bold:

```
typedef struct _Script          // Encapsulates a full script
{
    int iIsActive;              // Is this script structure in use?

    // Header data
    int iGlobalDataSize;        // The size of the script's global data
    int iIsMainFuncPresent;     // Is _Main () present?
    int iMainFuncIndex;         // _Main ()'s function index

    // Runtime tracking
    int iIsRunning;             // Is the script running?
    int iIsPaused;              // Is the script currently paused?
    int iPauseEndTime;          // If so, when should it resume?

    // Threading
    int iTimesliceDur;          // The thread's time slice duration

    // Register file
    Value _RetVal;              // The _RetVal register

    // Script data
    InstrStream InstrStream;    // The instruction stream
    RuntimeStack Stack;         // The runtime stack
    FuncTable FuncTable;        // The function table
    HostAPICallTable HostAPICallTable;    // The host API call table
}
    Script;
```

Notice that the Script structure doesn't have a separate field for priority rank. Although you could store this as well, it's actually not necessary. When the script is loaded, its priority rank is immediately checked to determine whether its requested priority was a predefined rank or a time slice duration. If it was the latter, this value is written directly to the iTimesliceDur field. Otherwise, the time slice duration associated with the specified rank is immediately substituted and written to iTimesliceDur.

Here are the constants used to map priority ranks to time slice durations, in milliseconds:

```
#define THREAD_PRIORITY_DUR_LOW  20  // Low-priority thread time slice
#define THREAD_PRIORITY_DUR_MED  40  // Medium-priority thread time slice
#define THREAD_PRIORITY_DUR_HIGH 80  // High-priority thread time slice
```

Loading Version 0.8 Scripts

`XS_LoadScript ()` is then updated to recognize the .XSE format modification. There is one twist though; as an added bonus, I thought it'd be cool to give the host application the capability to force a thread into a certain priority or time slice duration by passing that as a new parameter in `XS_LoadScript ()`. Here's the new prototype:

```
int XS_LoadScript ( char * pstrFilename,
                           int & iScriptIndex,
                           int iThreadTimeslice );
```

Here's how it reads in the thread's priority settings:

```
// Read the priority type (1 byte)
int iPriorityType = 0;
fread ( & iPriorityType, 1, 1, pScriptFile );

// Read the user-defined priority (4 bytes)
fread ( & g_Scripts [ iThreadIndex ].iTimesliceDur,
    4, 1, pScriptFile );

// Override the script-specified priority if necessary
if ( iThreadTimeslice != XS_THREAD_PRIORITY_USER )
    iPriorityType = iThreadTimeslice;

// If the priority type is not set to user-defined,
// fill in the appropriate time slice duration
switch ( iPriorityType )
{
    case XS_THREAD_PRIORITY_LOW:
        g_Scripts [ iThreadIndex ].iTimesliceDur =
            THREAD_PRIORITY_DUR_LOW;
        break;
    case XS_THREAD_PRIORITY_MED:
        g_Scripts [ iThreadIndex ].iTimesliceDur =
            THREAD_PRIORITY_DUR_MED;
        break;
    case XS_THREAD_PRIORITY_HIGH:
        g_Scripts [ iThreadIndex ].iTimesliceDur =
            THREAD_PRIORITY_DUR_HIGH;
        break;
}
```

The priority-type code is read in first. If it specifies a user-defined thread, that value is immediately stuffed into `iTimesliceDur`. Otherwise, a `switch` block is entered to assign the proper `THREAD_PRIORITY_*` constant. Either way, by the time all scripts are loaded, their priority-type codes have been discarded and they all rely on a raw time slice duration.

The Multithreading Scheduler

The final piece of the puzzle is an upgrade to the multithreading scheduler to take each script's time slice duration into account. Fortunately, because you disregarded the scripts' rank when loading them, all you have to deal with is a single integer value. Here's the code:

```
// Check for a context switch if the threading mode
// is set for multithreading
if ( g_iCurrThreadMode == THREAD_MODE_MULTI )
{
    // If the current thread's time slice
    // has elapsed, or if it's terminated
    // switch to the next valid thread
    if ( iCurrTime > g_iCurrThreadActiveTime + g_Scripts
        [ g_iCurrThread ].iTimesliceDur ||
        ! g_Scripts [ g_iCurrThread ].iIsRunning )
    {
        // Loop until the next thread is found
        while ( TRUE )
        {
            // Move to the next thread in the array
            ++ g_iCurrThread;

            // If we're past the end of the array, loop back around
           if ( g_iCurrThread >= MAX_THREAD_COUNT )
               g_iCurrThread = 0;

            // If the thread we've chosen is active and
            // running, break the loop
            if ( g_Scripts [ g_iCurrThread ].iIsActive &&
                 g_Scripts [ g_iCurrThread ].iIsRunning )
                break;
        }

        // Reset the time slice
        g_iCurrThreadActiveTime = iCurrTime;
    }
}
```

As you can see, the only major change is the fact that the former generic time slice duration constant has been replaced with the script's own `iTimesliceDur` field.

Demonstrating the Final XVM

To wrap things up, I've written a very simple program that integrates with the XVM and demonstrates its host API functions. To keep things as neutral as possible, the "integration" of the XVM with a host application really just means linking in `xvm.cpp|h`—I chose this over static or dynamic libraries to help minimize the amount of drama that can arise from anything other than simply including the right files with your project.

The Host Application

The host application in this demo is very simple. All it does is load a single script, define a single host API function for printing string sequences, and then demonstrates some actual functionality by calling and invoking the script's functions. In order to print to the console, the script must call the host's string printing function, thereby demonstrating all of the major integration functions the XVM provides.

The Demo Script

Before getting into the details of the host application's implementation, here's a little demo script I whipped up to test it out. Assume that the function will provide a function called `PrintString ()` that can be used to print a sequence of strings, given a string and an integer counter:

```
; Project.
;   XVM Final
; Abstract.
;   Simple test script.
; Date Created.
;   8.28.2002
; Author.
;   Alex Varanese

; ---- Directives --------

   SetStackSize 512
   SetPriority Low

; ---- Functions -----

    ; ---- Simple function for doing random stuff
    Func DoStuff
    {
         ; Print a string sequence on the host side
         Push     1
         Push     "The following string sequence
             was printed by the host app:"
         CallHost PrintString

         Push     4
         Push     " - Host app string"
         CallHost PrintString

         ; Print a string sequence on  the script side (with added delay)
         Push     1
         Push     "These, on the other hand, were printed individually by the \
                   script:"
         CallHost PrintString

         Var      Counter
         Mov      Counter, 8
         LoopStart:

                 Push     1
                 Push     " - Script string"
                 CallHost PrintString
                 Pause    200
                 Dec      Counter

         JGE      Counter, 0, LoopStart
```

```
        ; Return a value to the host
        Push      1
        Push     "Returning Pi to the host..."
        CallHost PrintString

        Mov      _RetVal, 3.14159
    }

    ; ---- Function to be invoked and run alongside a host application loop

    Func InvokeLoop
    {
        ; Print a string infinitely
        LoopStart:

                 Push     1
                 Push     "Looping..."
                 CallHost PrintString
                 Pause    200

        Jmp       LoopStart
    }
```

The script defines two functions, one to be called synchronously, the other to be called asynchronously. This will all make a bit more sense in the next section, which dissects the host application side of the demo.

Embedding the XVM

The first thing to do in the host application is embed the XVM. As I said, this is a painfully simple matter—just include the `xvm.h` header file and make sure to link `xvm.cpp` with your project. Here's the inclusion of the header along with the other `include` files the demo uses:

```
#include <stdio.h>
#include <conio.h>

// Include the XVM's header
#include "xvm.h"
```

NOTE

To be specific, you can link `xvm.cpp` with your host application in Microsoft Visual C++ simply by loading the file into your project. `xvm.cpp` should then appear in the `Source/` folder along with the main source files of the host app. The included project file on the accompanying CD already does this, so just check it out if you're confused by anything here. You can find it in `Programs/Chapter 11/XVM Final/Source/`.

Defining the Host API

The demo's "host API" is really just one function, called `HAPI_PrintString ()`, which will allow the script to print output to the console. Here's its definition:

```
void HAPI_PrintString ( int iThreadIndex )
{
    // Read in the parameters
    char * pstrString = XS_GetParamAsString ( iThreadIndex, 0 );
    int iCount = XS_GetParamAsInt ( iThreadIndex, 1 );

    // Print the specified string the specified number of
    // times (print everything with a leading tab to separate it from
    // the text printed by the host)

    for ( int iCurrString = 0; iCurrString < iCount; ++ iCurrString )
        printf ( "\t%s\n", pstrString );

    // Return a value
    XS_ReturnString ( iThreadIndex, 2, "This is a return value." );
}
```

Remember, parameters are read in with the `XS_GetParamAs* ()` functions, and return values are returned with `XS_Return* ()`. Once it has read the `pstrString` and `iCount` parameters, it prints the string out the specified number of times in a `for` loop. It prints a single leading tab before the string too, just to help the script's output separate itself from that of the host.

Notice also that the function is reading the string first as parameter 0, and then the integer as parameter 1. This is why the demo script pushed the integer before the string, like this:

```
Push     4
Push     " - Host app string"
CallHost PrintString
```

Remember! You always pass parameters in the opposite order that the function will read them.

The Main Program

The main host application program begins with a call to `XS_Init ()` to initialize the XVM:

```
// Initialize the runtime environment
XS_Init ();
```

It then declares integer variables to hold an error code and thread index, and calls `XS_LoadScript ()` to load the assembled .XSE demo script:

```
// Declare the thread indexes
int iThreadIndex;

// An error code
int iErrorCode;

// Load the demo script
iErrorCode = XS_LoadScript ( "script.xse", iThreadIndex,
    XS_THREAD_PRIORITY_USER );
```

Multithreading won't play a role in this demo, but I load it with `XS_THREAD_PRIORITY_USER` to allow the script to define its own priority anyway. Once the script has been loaded, it's good to check for an error and print out a description in the event that anything went wrong. Otherwise, a success message is printed:

```
// Check for an error
if ( iErrorCode != XS_LOAD_OK )
{
    // Print the error based on the code
    printf ( "Error: " );

    switch ( iErrorCode )
    {
        case XS_LOAD_ERROR_FILE_IO:
            printf ( "File I/O error" );
            break;
        case XS_LOAD_ERROR_INVALID_XSE:
            printf ( "Invalid .XSE file" );
            break;
        case XS_LOAD_ERROR_UNSUPPORTED_VERS:
            printf ( "Unsupported .XSE version" );
            break;
        case XS_LOAD_ERROR_OUT_OF_MEMORY:
            printf ( "Out of memory" );
            break;
        case XS_LOAD_ERROR_OUT_OF_THREADS:
            printf ( "Out of threads" );
            break;
    }
    printf ( ".\n" );
    return 0;
}
else
{
    // Print a success message
    printf ( "Script loaded successfully.\n" );
}
printf ( "\n" );
```

To get things going, the `HAPI_PrintString ()` function is registered with the XVM under the simpler name `PrintString ()`, and the script is started to let the XVM know that its code is executable:

```
// Start up the script
XS_StartScript ( iThreadIndex );
```

Next, the script's `DoStuff ()` function is called asynchronously. You do this to allow the function to run entirely on its own, uninterrupted. You also want to let it return a value, which isn't possible with synchronous calls. After running `DoStuff ()`, the value it returns is printed to the screen and the script's other function, `InvokeLoop ()`, is invoked and run within a loop. This demonstrates an invoked function's capability to run concurrently with the host:

```
// Call a script function
printf ( "Calling DoStuff () asynchronously:\n" );
printf ( "\n" );

XS_CallScriptFunc ( iThreadIndex, "DoStuff" );

// Get the return value and print it
float fPi = XS_GetReturnValueAsFloat ( iThreadIndex );
printf ( "\nReturn value received from script: %f\n", fPi );
printf ( "\n" );

// Invoke a function and run the host alongside it
printf ( "Invoking InvokeLoop () (Press any key to stop):\n" );
printf ( "\n" );

XS_InvokeScriptFunc ( iThreadIndex, "InvokeLoop" );

while ( ! kbhit () )
    XS_RunScripts ( 50 );
```

At this point, the script's functionality has been demonstrated, so you can shut everything down with a simple call to `XS_ShutDown ()`.

```
// Free resources and perform general cleanup
XS_ShutDown ();
```

The Output

What fun would all this be if you couldn't see the output, huh? Upon running the host application demo, you'll see this (of course, it's more interesting to watch it run):

```
XVM Final
XtremeScript Virtual Machine
Written by Alex Varanese

Script loaded successfully.

Calling DoStuff () asynchronously:

        The following string sequence was printed by the host app:
         - Host app string
         - Host app string
         - Host app string
         - Host app string
        These, on the other hand, were printed individually by the script:
         - Script string
         - Script string
         - Script string
         - Script string
         - Script string
         - Script string
         - Script string
         - Script string
         - Script string
        Returning Pi to the host...

Return value received from script: 3.141590

Invoking InvokeLoop () (Press any key to stop):
```

```
Looping...
Looping...
Looping...
Looping...
Looping...
Looping...
Looping...
Looping...
```

Cool, huh? It may be simple, but this output represents a totally finished and fully integrated virtual machine. Game scripting ahoy!

Summary

Whew! With priority-based multithreading and a feature-rich integration interface, your now-embeddable XVM has become quite a slick little piece of software. You now have two of the three major components of the XtremeScript system ready to go, giving you the ability to write assembly-language scripts, assemble them to bytecode, run them concurrently, and allow them to easily communicate with the game engine.

All that's left now is to write a high-level compiler capable of translating the XtremeScript language developed in Chapter 7 to its XVM assembly equivalent. This would give you everything you need to achieve your original goal of scripting games with a high-level language, and nearly complete your quest to attain scripting mastery. In short, if you made it this far, you're doing great! Don't give up now!

On the CD

This chapter covered a lot of ground, and the CD reflects it. The Chapter 11 folder contains two new versions of the XVM (the last of which is the final, embeddable version you'll be using throughout the rest of the book) and the new version 0.8 XASM assembler. Everything can be found in `Programs/Chapter 11/`.

The multithreaded XVM demo and the final, embeddable XVM are in `XVM Demo/` and `XVM Final/`, respectively. Version 0.8 of XASM is in `XASM 0.8/`. As has been the case with most of the programs lately, everything is a console application, which means you shouldn't have much to worry about with regards to compiling and running everything.

The basic multithreading XVM demo in `XVM Demo/` will run as many scripts as you want it to; just specify them all on the command line. The Final XVM demo in `XVM Final/` is a bit different; it's designed to run a specific set of scripts to fully demonstrate its functionality.

Challenges

- *Intermediate:* Implement a mutex and/or semaphore system to protect shared resources within the game engine, and create a set of host API functions for locking and unlocking them.
- *Intermediate:* Implement a thread priority system in which all threads are given the same time slice, but are invoked more or less *frequently* depending on their rank.
- *Difficult:* Add the capability to track global variables exposed by the host from the script, the capability to track globals exposed by the script from the host, or both.

Part Six

Compiling High-Level Code

This page intentionally left blank

CHAPTER 12

Compiler Theory Overview

"I didn't say it would be easy, Neo.
I just said it would be the truth."
——*Morpheus,* The Matrix

At last. After working your way through page after page of prerequisite information and concepts, after enduring an 11-chapter build-up, and after completing two thirds of the XtremeScript system, you're finally on the brink of what will undoubtedly be both the most complex and most rewarding aspect of designing a custom scripting language.

Compiling high-level code is one of those things that, over time, has built up a reputation of being insurmountably difficult to understand and even harder to implement. After all, as I've mentioned on numerous occasions, any time the precisely calculated world of software meets the fuzzy and ambiguous world of human expression, there almost invariably exists a barrier of complexity and error-prone translation that few ever dare to attack head-on.

(Un)fortunately for you, your interest in scripting has lead you down a path that will inevitably end in the belly of this particular beast, and unless you want to spend the rest of your life trying to write scripts in XVM assembly language, you're going to have to face it sooner or later. Fortunately, however, the subject of compiler theory is almost as old as computing itself, which means the algorithms and concepts upon which it's based have been richly developed and documented. Besides, completing a compiler project is a badge of honor you'll be able to proudly wear throughout your coding career, and will help solidify a skill set that will prove useful, if not invaluable, in countless other fields and applications.

This chapter covers

- An overview of compiler theory.
- How the XtremeScript compiler works with XASM.
- Advanced compiler issues.

This chapter may be primarily introductory, but it's required reading for the chapters that follow. The majority of terms and concepts I'll be using over the course of the next few chapters will be introduced here, so don't be surprised if you find yourself lost after skipping it.

An Overview of Compiler Theory

As has been stated a few times already, the subject of translating programming languages from one form to another is encapsulated by a broad field of study known as *compiler theory*. Everything from the assemblers to C compilers to SQL query interpreters draw on the teachings of this subject, and you should already have a pretty good idea of why and how it applies to you.

Just to make sure you're up to speed on a few things, let's review some of the terms and concepts I've attempted to drill into your head over the course of the chapters that have led up to now:

NOTE

Yes, even SQL interpreters are based on the same principals that were used to build high-end compilers like Microsoft Visual C++ and GCC. Large or small, almost any form of language processing and interpretation can benefit from the teachings of compiler theory.

- *High-Level Languages*, or *HLLs*, are languages that are designed to mimic human-readable languages like English for the purpose of clearly describing algorithms, expressions, and procedures. Examples of HLLs include C, C++, Java, and Pascal. High-level languages get their name from the fact that they're strongly abstracted and are separated from the processor (the lowest level) by numerous layers.
- *Low-Level Languages*, or *LLLs*, are among the lowest layers separating HLLs from the processor itself. LLLs include assembly languages and other specialty languages. Low-level languages get their name from the fact that they're separated from the processor by little-to-no abstraction. When comparing equivalent programs written in high- and low-level languages, the low-level versions are invariably faster and smaller (assuming both are written to the fullest extent of their respective languages).
- *Machine Code* is a purely binary language understood directly by processors, consisting entirely of simple instructions that are represented by integer values called opcodes. Machine code is more or less equivalent to assembly language, but is specifically designed for fast and efficient execution. As a result, it's virtually unreadable to humans in a practical context.
- *Bytecode* is another name for the machine code of a virtual machine like the XVM (XtremeScript Virtual Machine) or JVM (Java Virtual Machine).
- *Compiling* is the process of reducing a high-level language to a low-level one.
- *Assembling* is the process of translating the human-readable version of a low-level program to its machine-readable equivalent.

Okay! Good to get that out of the way (and yes, I promise that's the last time I'll go through all that). Now that I'm reasonably sure we're all on the same page, let's take a deeper look at how this translation of high-level languages to low-level languages really works.

Phases of Compilation

You know compilers are used to turn high-level code into assembly language and/or machine code, but how exactly is this done? To answer this question, think back to Chapter 9 when you implemented the XASM assembler. If you recall, the program worked in a number of *phases* (not to be confused with *passes*, which I'll cover separately). The first phase involved a basic processing of the incoming source code; whitespace was removed, comments were stripped away, and so on. The next phase was known as *lexical analysis*, in which the source code stream was broken into streams of tokens and lexemes. This stream was then fed into a *parser*, which was ultimately responsible for the final assembly of the source code.

At first, writing software capable of intelligently translating human-readable code like the following seems nearly impossible:

```
int X = 120;
float Y, Z;
Y = 3.14159;
Z = sin ( X ) * cos ( Y / X );
for ( X = 0; X < 359; ++ X )
      Y *= Z / X * tan ( Y );
MyFunc ( X * Z, Y );
```

NOTE

In reality, `sin ()` and `cos ()` from the standard math library take radian parameters and not degrees; however, this is just an example to show you some basic math code.

And indeed, it is difficult to translate such code. However, when the compiler is split into multiple phases, each of which is responsible for a separate, specific task, the process becomes infinitely easier, at least on a conceptual level. Understanding how source code is reduced to tokens isn't hard, understanding how tokens are parsed isn't all *that* difficult, and if you put them together, you've got a basic compiler laid out already. It's like studying the human brain—the nearly endless versatility and flexibility of human intelligence seems impossible to describe or reproduce at first, but once you learn that the brain is really just a massive collection of interconnected and

NOTE

Think about it—of all the things you do during the day, how many of them are actually approached *without* first breaking them down into their constituent parts? Walking across the street would be pretty difficult without the ability to take intermediate steps—you'd need pretty long legs otherwise. So, instead of thinking about what sort of godlike program could turn a C file into an executable, think instead about the multitude of small, simple programs that perform each intermediate step. When you get to the last one, you'll be able to look back and see that the initially huge challenge was really just a bunch of smaller ones. And that, kids, is enough eastern philosophy for one day.

simplistic components, the magic is demystified. If there's one thing that all software engineers should understand—incredible complexity can be attained simply by combining the right pieces in the right way. This is exactly how the construction of a compiler is approached.

Chapter 5 saw your first real introduction to the phases of a compiler. In Chapter 9, you even implemented a few of them, albeit in an admittedly watered-down way. You've learned that on a basic level, virtually all compilers consist of the same fundamental phases:

- Lexical analysis
- Parsing
- Semantic analysis
- I-code generation
- Target code emission

None of these phases is particularly hard to understand if they're explained properly, and once you can implement them all, you're capable of building a compiler. Figure 12.1 presents them in sequence.

NOTE

You'll notice a lot of references to files with an .XSS extension throughout this chapter. Just to give you a heads-up, this stands for *XtremeScript Source*, and is the extension you'll be using for all of your high-level XtremeScript scripts.

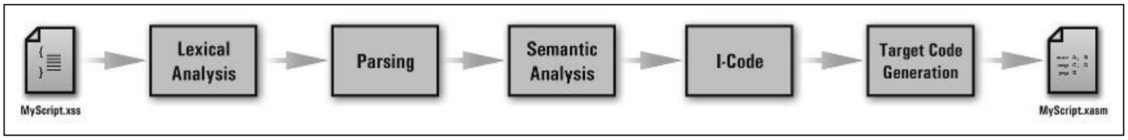


Figure 12.1

Phases of a compiler.

Lexical Analysis/Tokenization

Lexical analysis, or *lexing* for short, has made numerous appearances in the book so far. From the simple command based language of Chapters 3 and 4 to the XASM assembler of Chapter 9, the process of converting a raw stream of characters to distinct "words" or "chunks" makes your life considerably easier when attempting to translate and understand various forms of scripting languages. Figure 12.2 illustrates the concept of lexical analysis.

To recap the process, a lexical analyzer takes as its input an incoming stream of source code, such as the following:

```
X = MyVar * 2;
```

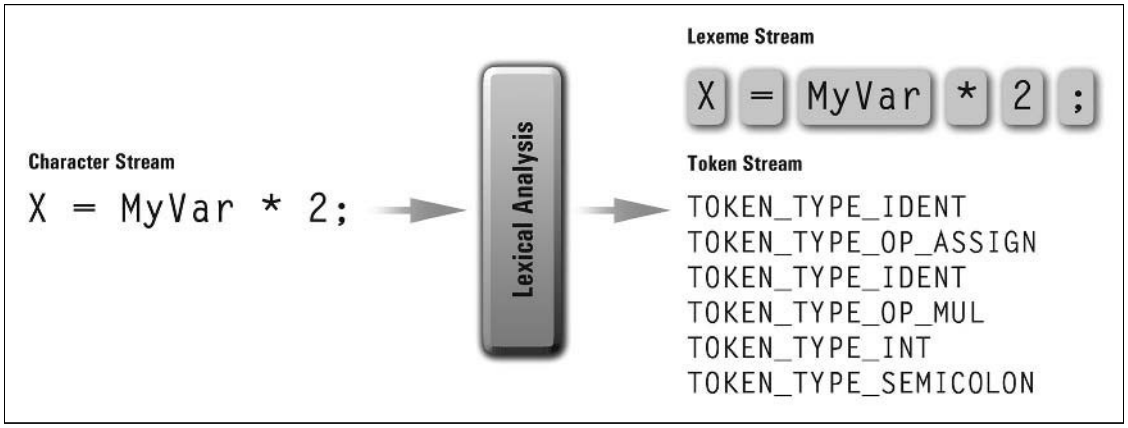


Figure 12.2

Lexical analysis breaks up the incoming character stream into lexeme and token streams.

and produces two forms of output; a stream of *lexemes* and a stream of *tokens*. The lexeme stream is rather similar to the original source code, except that each unique "word" or "component" has been isolated. The previous line would be returned from the lexer in this order:

```
X
=
MyVar
*
2
;
```

This is definitely an improvement, because it's a lot easier to analyze each individual lexeme than it is to deal with the entire line (or source file) as a whole. What's really important, however, is what each lexeme *represents*. In other words, what the compiler really wants to know is that `X` is a variable, `=` is the assignment operator, `MyVar` is another variable, `*` is the multiplication operator, and `2` is an integer literal value. The token stream provides exactly this:

```
TOKEN_TYPE_IDENT
TOKEN_TYPE_OP_ASSIGN
TOKEN_TYPE_IDENT
TOKEN_TYPE_OP_MUL
TOKEN_TYPE_INT
TOKEN_TYPE_SEMICOLON
```

The lexer allows you to think of the source code in much higher-level, abstracted terms. No longer is it necessary to hunt and peck your way through a raw chunk of character data; instead, you now have a simple but significant glimpse of what the source code *means*.

It doesn't take a PhD to understand that anything becomes simpler if you can isolate and group common elements. For example, cleaning a house in which every floor is covered with dirty clothes and garbage can be a long and tedious job, but it would be exponentially easier if every 5 or 10 pieces of clothing and garbage were wrapped up into a small bag together. Picking up even a large number of bagged items is a lot easier, because the grouping cuts down the complexity and depth considerably.

Lexer Implementation

The implementation of a lexer really just boils down to a decent amount of string processing; because its only job is to determine which characters belong to the same lexeme, there's naturally going to be a lot of substring isolation and analysis. There are a number of ways to approach the problem, however.

The Brute Force Approach

The first and perhaps most obvious approach is just to use brute force, which served you well in Chapter 9 during the development of the XASM assembler. What I mean by "brute force" isn't that the solution is crude or unintelligent, but rather that the lexer is written with a rather narrow focus, with a number of hard-coded elements. A brute force lexer operates in a number of phases:

- Leading whitespace before the lexeme is consumed.
- The lexeme is slowly built up from the character stream until a delimiter character of some form, such as a comma, bracket, or more whitespace is encountered.
- The lexeme is isolated and analyzed by comparing it to a number of strings and string classifications.

As you can see, this approach to lexing is quite logical and natural; in fact, it's the first solution to the problem I came up with when I was initially getting into this stuff. Let's look at an example; consider the following line of text (not including the surrounding quotes):

```
"     MyVar A , 32768 $  "
```

The lexer, in an attempt to extract the first lexeme and token from the string, would begin by consuming the leading whitespace before `MyVar`. The string would now conceptually look like this:

```
"MyVar A , 32768 $  "
```

Of course, the lexer doesn't physically delete anything; but this is how it will perceive the string from now on. With the whitespace out of the way, the first character of the lexeme itself will be read and the lexeme extraction process will begin. It will start with the character `M` and work its way through `yVar` until the first whitespace character is encountered. This lets the lexer know that the end of the lexeme has been found. A substring is extracted between the lexeme's start and end points, which results in the following:

```
"MyVar"
```

You now have the lexeme, so half of the lexer's job is over (although technically, the lexer's entire job is over because the rest of this phase belongs to the tokenizer). The next task is to determine its token type, which is done by comparing it to a number of string classifications. To keep this simple, let's just say you have three token types to work with: integer literal values, floating-point literal values, and identifiers. The lexer would first determine whether `MyVar` was an integer. It would do this by determining whether each character was a digit between `0` and `9`, and that the first character was optionally a minus sign to represent a negative value. Because `MyVar` hardly passes this test, one of three possible token types has been eliminated. It would then attempt to classify it as a floating-point value, which would fail even more miserably, because a float is just an integer with an optional radix point. Lastly, it would compare it to the definition of an identifier token, which is a string or characters that can either be letters, digits, or identifiers such that the first character is not a digit. Because `MyVar` consists solely of letters, it passes this test and the token type is set to `TOKEN_TYPE_IDENT`. Figure 12.3 provides a graphical view of this lexing method.

NOTE

Don't forget, *lexing* (extracting the current lexeme from the character stream) and *tokenization* (determining the lexeme's type) are two different processes. However, due to their closely related nature, they're usually just lumped together as "the lexer". Unless otherwise stated, I'll always mean both lexical analysis and tokenization when I refer to the lexer's role in the compiler.

As you can see, this method definitely works and is easy to understand. It's not the most flexible or compact method, however. As is hopefully clear, a number of loops are executed to fully extract the lexeme, followed by a possibly huge number of comparisons to determine what the lexeme's token type is. A full-scale compiler will have to understand far too many token types to hard-code them all directly into a brute force lexer.

The State Machine Approach

Fortunately, a far more elegant approach exists in the form of *finite state machines*, also known as *FSMs*. A finite state machine can be described most simply as a basic loop, each iteration of which

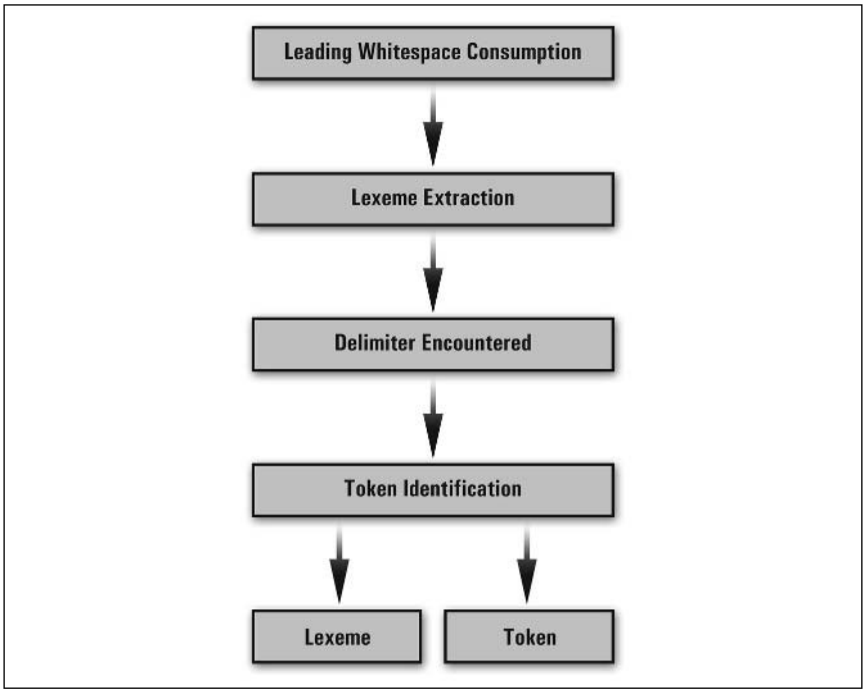


Figure 12.3

Brute force lexical analysis.

is in one of a finite number of *states*, and contains code that allows it to *transition* to other states based on certain circumstances.

State machines are great because they're written in such a generic manner that any number of tokens can be processed by a single character-processing loop. At each iteration of the loop, a new character is read from the input stream and used as criteria for a possible state transition. As states transition from one to another, the loop slowly builds a stronger and stronger idea of what the overall lexeme is. For example, the loop may start in the state `STATE_INIT`. If the first character read from the stream is a letter, the loop may switch to `STATE_IDENT`, because it assumes it's processing an identifier. As long as letters, numbers, and underscores keep coming in, the state will remain `STATE_IDENT` because each of these character types satisfies the rules of a valid definition. If a dollar sign was suddenly read, however, the state machine would find that no rule exists that allows that particular character to transition from `STATE_IDENT` to anything meaningful, so an error would occur.

As you can imagine, state machines are very powerful and highly expandable. All you need to do to introduce a whole new array of token types is just add more states and state transition rules. This is a stark contrast to the brute force method, in which huge chunks of code must be added, removed, or modified to achieve the same results. Figure 12.4 presents a state-diagram for a number lexing state machine.

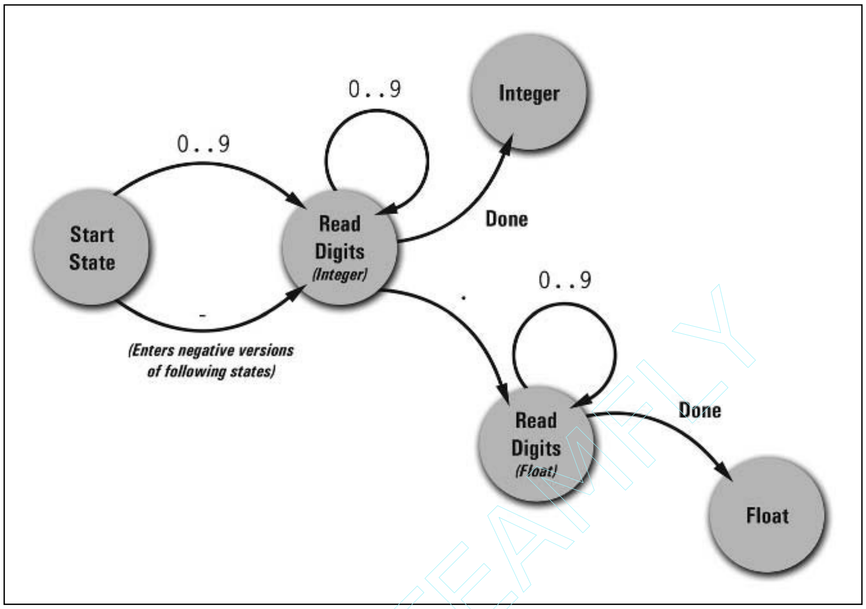


Figure 12.4

A state machine for lexing integer and floating-point values.

Parsing

The stream of tokens and lexemes generated by the lexer in the lexical analysis phase is fed directly to the parser for the parsing phase. *Parsing* is the process of analyzing incoming tokens and determining how they fit into the language. Parsing is quite possibly the most complex part of a basic compiler, and there are numerous ways to go about doing it.

Regardless of how it's done, however, the goal of the parser is to create what is known as a *parse tree*, which is a hierarchical representation of the source code. For example, the parse tree for the following line of code is displayed in Figure 12.5:

```
MyFunc ( X = Y, Z );
```

The actual creation of this tree, however, is usually the defining characteristic of a parsing method. The most general way to categorize these methods is *top-down* parsing versus *bottom-up* parsing.

Top-Down Parsing

Top-down parsing can probably be considered the more intuitive of the two methods. It's the unofficial basis for the parsing strategy I chose during the development of the XASM assembler, and is most often mentioned in reference to *recursive descent* parsing.

Recursive descent can best be explained with an example. Take the following line of code:

```
while ( X < Y * 2 )
```

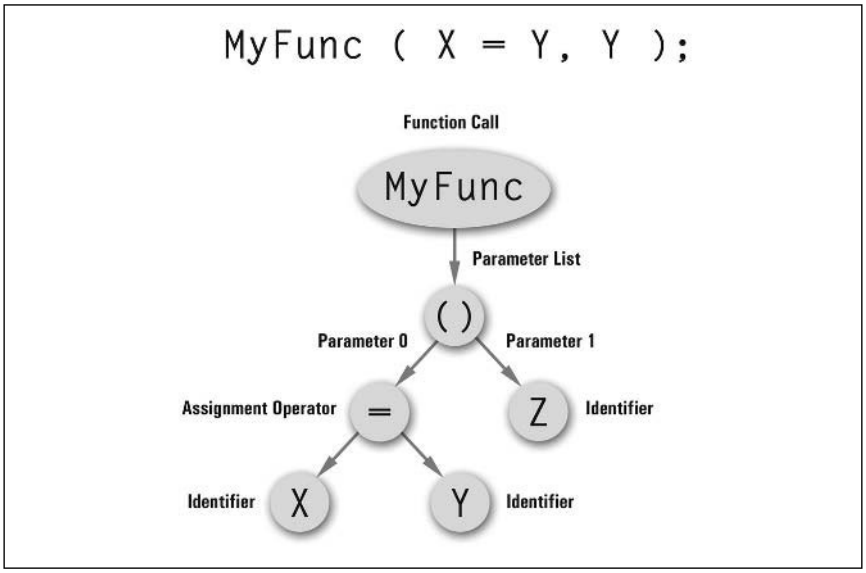


Figure 12.5

The parse tree provides a hierarchical view of the source code.

As humans, we know upon first glance that we're dealing with a `while` loop. We know this because the first word we saw on the line was the `while` keyword itself. If that keyword had been anything else, we wouldn't have come to the conclusion that we were looking at such a loop. Beyond this, we know that the criteria of the loop is a Boolean expression. This could have been any number of things—it could've been a simple constant reference, like `while ( TRUE )` or it might've been a single function call, like `while ( MyFunc () )`. But instead, we knew it was a Boolean expression because we saw the variable `X` immediately followed by a binary operator. Based on this, we knew it was an expression, and given the context of the `while` loop, we knew it was specifically a Boolean expression.

Recursive descent parsing works in a similar manner, which right off the bat should help you understand why it's often considered one of the easier or more natural methods. A recursive descent parser would first read the `while` token and come to the same conclusion we did, using the neural parser in our brains—that a `while` loop is in the works. It would then unconditionally *expect* an opening parenthesis, because they invariably follow the `while` token according to the rules of the language. Once the token has been parsed, the parser knows an expression is coming.

Based on what I've explained so far, the "descent" part of the name should make some sense. According to the parse tree diagram presented in Figure 12.6, you've moved progressively downward from the top node of the tree. But where does the recursion come in?

Once the parser reaches the parenthesized expression, its `while` loop parsing logic will no longer suffice. It will instead have to switch to another parsing mechanism, one geared towards parsing expressions. The expression parser will then run until the expression is complete, at which point

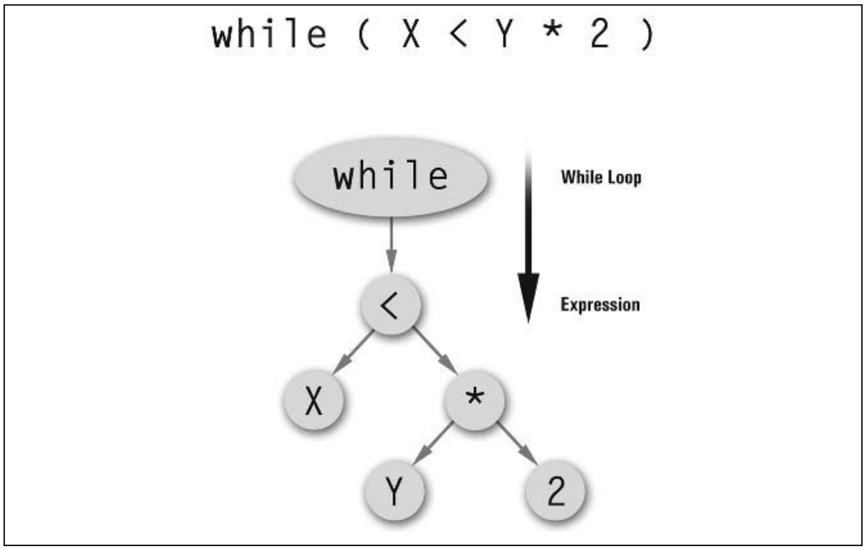


Figure 12.6

The recursive descent parsing of a `while` loop.

it will hit the closing parenthesis, which is back within the jurisdiction of the `while` loop parsing logic. This is all very visual, so check out Figure 12.7.

As you may be starting to suspect, a recursive descent parser has a separate parsing mechanism for every major language feature. For example, when a `while` token is read, the `while` loop parser is activated. When the first token of what appears to be an expression is read, an expression parser is activated. With all of these different mechanisms to deal with, it'd make sense to wrap them all in functions, right? Then, when I read a `while` token, I make a call to `ParseWhileLoop ()` and forget about it. Once that function reaches the inside of the opening parenthesis, it'll call `ParseExpression ()` and the process will continue.

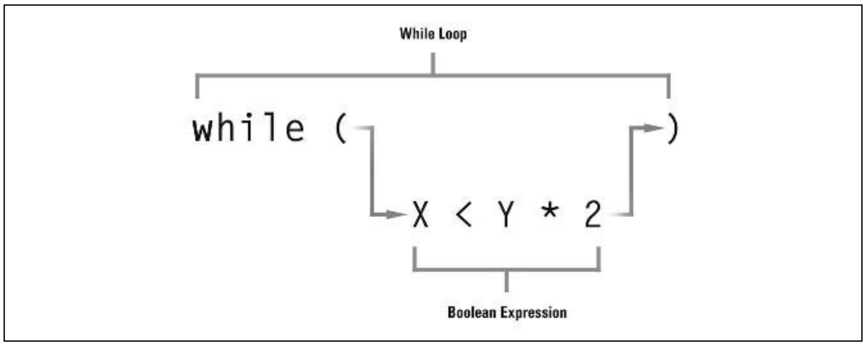


Figure 12.7

Two levels of parsing taking place—one for the `while` loop's core syntax, the other for the expression.

So, recursive descent parsing is heavily defined by its repetitious use of nested, and often times, *recursive* function calls. For a specific example of recursion, consider the following expression:

```
X = 8 * ( 16 + Y ) / Z;
```

You have one overall expression, but there are definitely "sub-expressions" within it. `8 * ( 16 + Y ) / Z` is a large expression, with smaller expressions like `16 + Y` and `8 * ( 16 + Y )` within it. Rather than attempting to write a single, convoluted expression parser to directly parse the previous statement, it's easier and cleaner to write a very basic parser that calls itself repeatedly as each sub-expression is encountered.

Recursive descent parsing suffers from the main drawback of being inefficient. Parsing even a simple expression or statement will involve multiple nested function calls and possibly a considerable amount of recursion. When this is applied to every line of code in a large program, performance suffers and stack space is threatened. Furthermore, because each "parsing mechanism" resides in its own function, recursive descent parsers are primarily hard-coded and therefore more difficult to modify than other, more flexible methods.

NOTE

Recursive descent parsers, unlike many other types of parsers, can be written by hand. In the case of parsers that are too complex to be written manually, special utilities are used that generate the code for a parser based on a file that specifies the rules of the language. You'll learn a little more about these utilities later in the chapter.

Regardless, it's also quite easy to understand and much simpler to implement than some of the alternatives. Because of this, the XtremeScript compiler will be built around a recursive descent parser.

Bottom-Up Parsing

Bottom-up parsing is a significantly different approach, and one that requires a bit more thought to grasp than its top-down counterpart. When working your way up from the bottom of the parse tree to the top, you're only seeing the tree's terminal nodes. Because of this, you don't get an immediate big-picture view of things like you would with the recursive descent. So, instead of using an initial token to predict what's ahead and branching off to a specific parsing mechanism, the parser must instead use inductive reasoning to piece together a progressively more refined idea of what larger structure the tokens are attempting to describe. Figure 12.8 illustrates the basic concept behind bottom-up parsing.

Despite the obvious increase in complexity, bottom-up parsing tends to be significantly more efficient than top-down due to its reliance on a single compact loop that, rather than branching to multiple functions to handle specific cases, refers to a large, procedurally-generated lookup table that helps it detect patterns in the token stream. In this regard, the difference between top-down

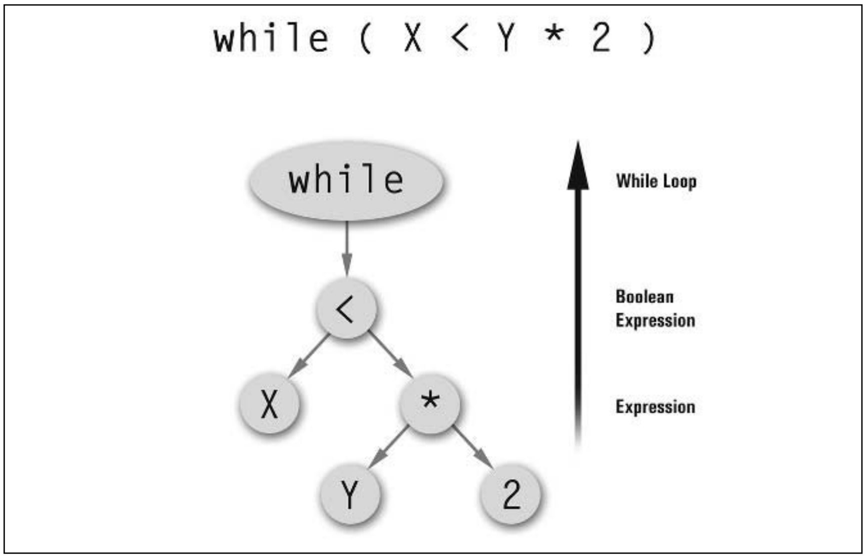


Figure 12.8

Bottom-up parsing requires the parser to determine the overall pattern of a set of tokens.

and bottom-up parsing is analogous to that of brute force and state machine based tokenization. Both brute force tokenization and top-down parsing require the compiler to be written in a specific, nearly hard-coded fashion that gets a very specific job done with no fuss. State machine tokenizers and bottom-up parsers, on the other hand, are based around simplistic and generic loops that can be easily altered to accept and produce completely different input and output. In fact, bottom-up parsers are, more or less, simply state machines.

Semantic Analysis

At each phase in the compiler, you've seen the source code go from a raw stream of pure character data to an almost fully understood script or program. The lexical analyzer made sure that the character stream was in the form of valid lexemes, whereas the parser made sure that the lexeme stream was in the form of valid syntax. Syntax checking is not enough, however, because beyond the syntax of a language lies the more nebulous *semantics* of the language.

Semantics operate on code that has already proven its syntactical validity. For example, the following line of code is perfectly correct according to the rules of C:

```
X = Y;
```

Or is it? What if `X` is a constant? In this case, the syntax of the expression is correct (because it's basically saying that this identifier is assigned that identifier), but the semantics of a constant being assigned a value are nonsensical and invalid. This is an example of semantic analysis. Other examples of semantic errors that would make it past the parsing phase unnoticed include identifier re-declaration and using identifiers outside of their scope.

I-Code

The result of the parsing and semantic analysis phases is a version of the script's source code, represented entirely in *I-code*. I-code stands for *Intermediate Code*, and is a clean and structured way to store the script internally without worrying about the details of the source language. I-code is very similar to assembly language or machine code, because it's based around a set of fine-grained instructions that express the logic of the original source code in a much more compact and easily modifiable form.

NOTE

The term "P-code" is often used instead of I-code. *P-code* was the name for the byte-code of an old virtual machine designed to run Pascal programs, and in that regard, was much like I-code in that it was a simple, instruction-based internal format for representing Pascal programs. Despite the fact that the term was initially only related to Pascal, it eventually slipped into the general compiler theory vernacular and is now a common synonym for I-code.

The key feature of I-code is that at least theoretically, it is *entirely* unrelated to any specific source or target language. The I-code of Microsoft Visual Studio, for example, especially with the emergence of the .NET runtime system, is entirely unrelated to C, C++, or a specific machine code or assembly language. Intermediate code sits in between all of these languages, able to freely translate to and from any of them. Figure 12.9 demonstrates this concept graphically.

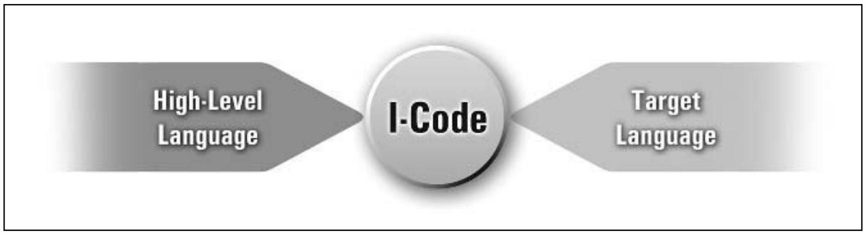


Figure 12.9

I-code sits in between the source and target languages, and is therefore independent of them.

This will not entirely be the case with XtremeScript, however, because you really only need to make room for one source language (XtremeScript) and one target language (XVM assembly). Because of this, there will most likely end up being a strong similarity between the compiler's I-code instructions and the XVM's instruction set. Regardless of how similar these particular languages are, however, it won't change the fact that even XtremeScript I-code will be capable of supporting a multitude of source and target languages.

Single-Pass versus Multi-Pass Compilers

As initially explained in Chapter 9, compilers and assemblers can be categorized based on the number of passes they make over the source code. A *pass* is defined as any complete scan of the entire source code, regardless of what information it's used to collect. Single-pass compilers are capable of fully understanding and translating the entire source code script without backtracking, whereas multi-pass compilers need to make at least two trips.

The difference between single- and multi-pass compilation isn't entirely a matter of how the compiler is written, however. The deciding factors in how many passes are required to compile a program are far more related to the design of the language itself. Take, for instance, C++ function prototypes. Before using a function, it's usually a good idea to precede all of your code with a function prototype that defines it, like so:

```
void MyFunc ( int iX, int iY );
```

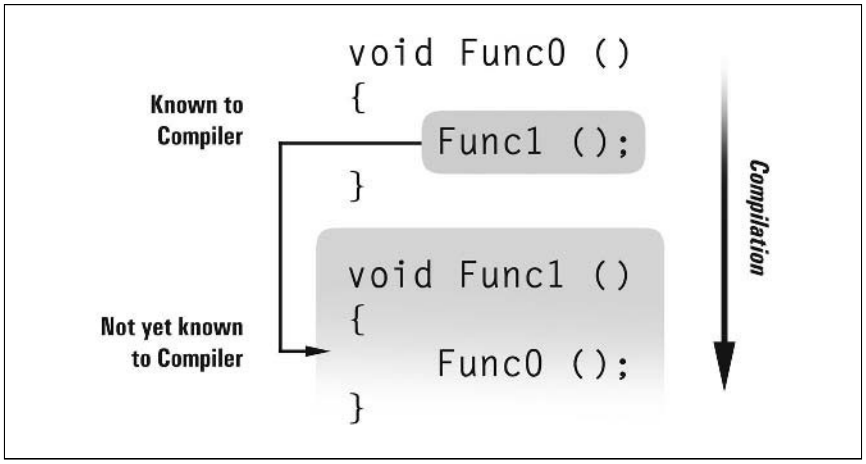
Now, regardless of where I am in the source, I can freely reference `MyFunc ()` and be sure that the compiler won't be confused. This is because C++ is compiled in a single pass. In order to properly parse function definitions and references without backtracking requires a list of prototypes that, before any code is analyzed, make the compiler aware of all of the program's functions.

C, on the other hand, doesn't support function prototypes but is still compiled in a single pass. This results in slight limitations on the coder in regards to what can and can't be done when making nested function calls. Take the following code, for example:

```
void Func0 ()
{
    Func1 ();
}

void Func1 ()
{
    Func0 ();
}
```

Here we have two function definitions, wherein each function calls the other. This will present a problem for a single-pass compiler, because when parsing the definition of `Func0 ()`, which calls `Func1 ()`, the compiler doesn't yet know `Func1 ()` exists. Because compilers are rarely designed to give coders the benefit of the doubt, it will assume `Func1 ()` is an invalid call and report a compile-time error, as shown in Figure 12.10.

This problem could be resolved in the same way it was resolved in XASM; by simply making multiple passes over the script. If the first pass collects information about all of the script's functions,



```
void Func0 ()
{
    Func1 ();
}

void Func1 ()
{
    Func0 ();
}
```

Figure 12.10

The problems with single-pass compiling and function references.

future passes will have this information readily available in full no matter where they are, thereby allowing `Func0 ()` to call `Func1 ()`. Once again, referring to an identifier before its declaration is called a *forward reference*, and is very important in the context of functions.

Of course, especially in the case of particularly huge programs (which high-end compilers deal with regularly), multiple passes over the source may be costly. For this reason, C++ compilers have opted to go with function prototypes that precede all function references to save the compiler from having to scan through the entire source file multiple times. In C++, the previous example could be written like this:

```
void Func0 ();       // These prototypes let the compiler know
void Func1 ();       // that Func0 () and Func1 () exist no matter
                     // where it is in the source code.

void Func0 ()
{
    Func1 ();
}

void Func1 ()
{
    Func0 ();
}
```

NOTE

I personally prefer multi-pass compiling, because I've never liked the idea of function prototypes (or other such cues and hints that are forced on the coder). I would much rather my compiler take the time to familiarize itself with my source code automatically, rather than relying on me to enter redundant information just so it can have more information at arbitrary places in the source file without backtracking.

Target Code Emission

Implementing the last phase of a compiler requires a solid understanding of the target platform, because it revolves around the translation of I-code to executable machine code or assembly language. In either case, because the I-code is often such a simplified representation of the program, considerable work is involved with this conversion. The 80x86, for example, has literally *thousands* of opcodes, compared to the 33 of the XVM. Within this huge set of data, large groups of opcodes are often dedicated to the machine-code equivalent of all of the different forms of a single assembly-language instruction. The code emitter must fully understand all of these details in order to generate a valid and efficient executable.

Fortunately for you, code emission will be among the easiest phases of the compiler. Because the I-code will have an almost one-to-one mapping with XVM assembly language, this will be a trivial matter of translating each I-code instruction to an XVM assembly instruction mnemonic.

The Front and Back Ends

Each of the phases of the compiler discussed so far is part of a larger whole, but you can further classify them by introducing the concept of the front end and the back end. A compiler's *front end* consists of the lexical analyzer, parser, semantic analyzer, and I-code generator. The *back end* consists of the target code emitter and other optional phases like optimization. The difference between the two ends is simple; the front end's goal is to generate I-code based on the source file, whereas the back end's goal is to translate that I-code to the compiler's target language (either executable machine code or assembly language). Figure 12.11 illustrates this concept.

NOTE

Because the front end is primarily concerned with interpreting and validating the source code, and the back end is primarily concerned with translating I-code into the target language, the front and back ends are often called the *analysis phase* and *synthesis phase*, respectively.

As you'll see later in this chapter, grouping the phases of the compiler into front and back ends helps open the door to a number of possibilities, such as optimization and retargeting. For now, however, you can merely think of them conceptually.

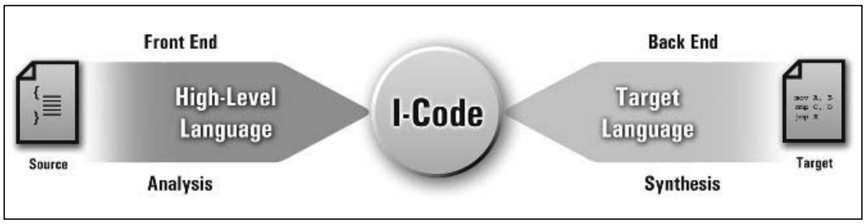


Figure 12.11

The front and back ends of a compiler.

Compiler Compilers

Compilers are all over the place; they exist in huge numbers and have limitless applications in all sorts of language and data translation fields. Because of this, it was inevitable that someone would finally sit down and create a set of tools to help automate the process of creating a new compiler. These tools usually consist of programs that can generate entire chunks of a compiler's code base, based on a specification or definition file of some sort that helps it understand how the compiler should work and what sort of language it operates on. These types of utilities are known as *compiler compilers*.

The two most popular examples of compiler compilers by far are the UNIX/Linux programs *lex* and *yacc*. The first of these, lex, is used for generating FSM lexical analyzers based on an input file that defines each of the lexemes and token types the lexer should understand. yacc stands for *Yet Another Compiler Compiler*, and is similar to lex except that it generates entire shift/reduce parsers based on a file that describes the syntax of the language. Check out Figure 12.12.

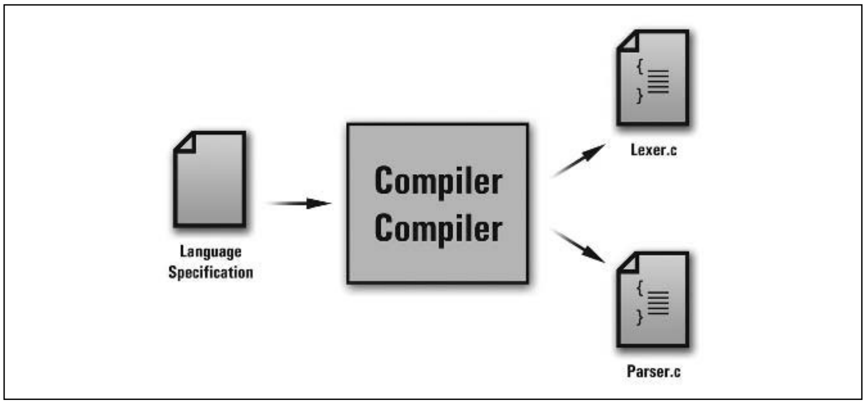


Figure 12.12

Compiler compilers generate large portions of finished compilers based simply on language specification files.

lex and yacc have been in heavy use for years, and have been ported to the Win32 platform under the names Flex and Bison (yacc, Bison, get it?). These programs are infamous among compiler writers and, when used properly, can fractionalize the development time of a compiler project.

How XtremeScript Works with XASM

Most compilers accept a source code file as their input and directly produce an executable as their output. Although the XtremeScript compiler could certainly work that way (and, from the perspective of the end user, *will* work that way), it's actually only the first of a two-step process that takes high-level code and turns it into an .XSE.

Because XASM is such a high-level assembler, with built-in support for variables, arrays, and even functions, it'd be silly not to leverage all that power. So, rather than directly produce a finished .XSE, the XtremeScript compiler instead generates an ASCII-based .XASM file containing XVM assembly code that will be automatically fed to XASM to get the final executable file. This allows you to take advantage of the preexisting capabilities of the assembler, which means the compiler will be much easier and faster to write. In essence, a large portion of the compiler's general functionality is already done. Figure 12.13 presents a graphical view of how XtremeScript and XASM fit together.

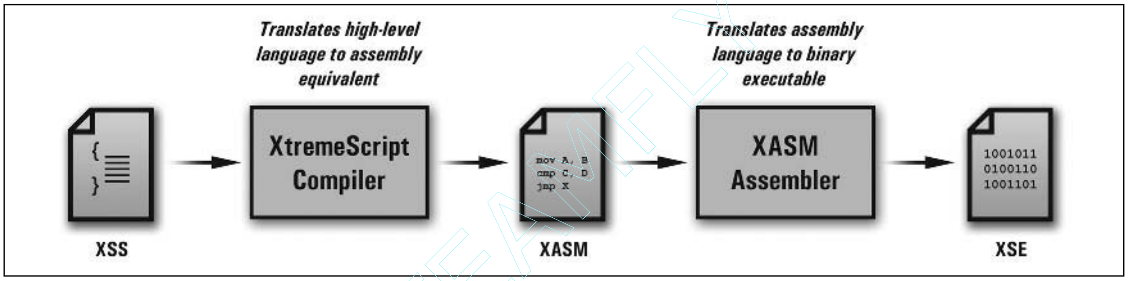

Translates high-level language to assembly equivalent
Translates assembly language to binary executable
XtremeScript Compiler
XASM Assembler
mov A, B
cmp C, D
jmp X
1001011
0100110
1001101
XSS
XASM
XSE

Figure 12.13

XtremeScript and XASM working together.

For example, XASM has its work cut out for it when assembling variables. In addition to the obvious stuff like keeping track of a variable's scope, size, and so on, it's also in charge of building its function's stack frame and assigning it a relative stack index. The XtremeScript compiler, however, will simply be able to generate the proper `Var` variable declaration and be done with it.

The same goes for functions. Because XASM already has direct support for functions with its `Func` directive, XtremeScript can literally translate its own functions directly to XASM functions using `Func` and `Param`.

NOTE

Don't get the wrong idea—the XtremeScript compiler will still be a true compiler, of course. Many already existing compilers have opted to generat ASCII-based assembly language files rather than straight machine code, so you aren't alone in this decision. Furthermore, everything you've learned through the development of XASM will be directly applicable to a higher-level compiler, so if you'd like to rewrite XtremeScript to directly produce .XSEs, you should be more than capable of doing so.

In a nutshell, XASM will do anywhere from 30-50 percent of the job when compiling high-level scripts. XtremeScript will undoubtedly be a complex piece of software, but taking advantage of the preexisting XASM code base will make things a lot easier.

Advanced Compiler Theory Topics

You should understand how the basics work, at least conceptually, and what specific topics will apply most significantly to the XtremeScript compiler and how. But compiler theory is a huge subject—one that I couldn't hope to do justice in the context of a book like this, so don't forget that even at their most complex, the things you've learned so far and the things you'll learn in the following pages are only scratching the surface.

If you're anything like me, though, you'd still at least like to learn a thing or two about these alleged "advanced subjects". After all, you'll need to know where to go if you choose to continue your studies in the field after this book, right? So, before I get back to the matters at hand, let's take a brief detour and learn about some of the more advanced topics and issues that compiler writers deal with. I always encourage further study, and you may very well find that some of these issues can be productively applied to your own scripting system.

Optimization

Game programmers rarely agree, but perhaps the strongest thread that binds and unites them is the never-ending quest for more speed. Games are among the most performance-critical forms of software in existence, which means too much speed is never enough. Scripting, unfortunately, introduces a number of bottlenecks due to its high-level, virtual nature. Because of this, the script code should be as tight as possible to help minimize its overall impact on frame rates.

However, as deeply as programmers may be wrapped up in the idea of being able to write scripts in a C-style language, they can't forget that high-level code brings with it an inherent overhead because compilers tend to produce more code than is technically necessary when producing a high-level script's low-level equivalent. Because of this, a script that may have been slow in pure XVM assembly will be even slower when written in XtremeScript and compiled down. After all, a compiler has a very hard time looking at the "big picture" of a script, which is something humans tend to take for granted. With such a narrow focus, it's hard for a compiler

TIP

Even though XtremeScript will not be an optimizing compiler, there is still one way to enjoy the benefits of high-level coding while retaining the ability to tighten up certain portions of the code. Because XtremeScript directly outputs XVM assembly, you're always free to stop the compiler from automatically passing it to XASM, and tighten up any blatantly un-optimized code yourself. Once it's been assembled, the script engine won't know the difference and you won't have to bother with it again. Of course, because any future changes to the high-level source would overwrite your low-level optimizations, be sure to make such changes only when you're sure the high-level code is done.

to notice the large-scale patterns and relationships that ultimately lead to the optimizations that you might notice at first glance alone.

Of course, real compilers like Microsoft Visual C++ have been in a constant state of evolution, the brunt of which has been focused specifically on optimizing the code they generate. Scores of math-heavy algorithms and techniques have spilled out of colleges and R&D labs over the last few decades, all aimed at helping compilers understand when and how the code they generate can be reworked and tightened to achieve higher performance and lower overhead. These days, the state of the art has reached a point where cutting edge compilers often produce code that nearly rivals hand-written assembly. Unfortunately, significant optimizations tend to be extremely complex to implement, often to the point of dwarfing the rest of the compiler.

Optimization is usually implemented in the back end, after the I-code has been generated but before the final target code is emitted. Back-end optimizations can be one of two classifications: what I call "logic optimizations", and target machine optimizations. Logic optimizations are independent of the final platform for which the code will be generated, whether it's the XVM or an 80x86. These optimizations focus primarily on rewriting portions of the I-code to perform the same task faster or in a smaller space. Target machine optimizations are highly platform-independent, however, and take advantage of the specific characteristics of the target environment to determine where optimizations can be applied. For example, if a script written for the XVM was recompiled for the 80x86, an optimizing back end might realize that many of the memory references that are acceptable on the XVM could be replaced by the 80x86's high-speed registers.

TIP

It's also worth noting that even when XtremeScript is done, it's not like you'll *have* to write every script you ever use with it. You'll always be free to bounce between XtremeScript and XASM, writing high- and low-level scripts when appropriate. Many small, constantly executing background scripts might work out better when written directly in assembly, whereas larger, single-use scripts can stay in XtremeScript.

As an example of a logic optimization, consider the following code:

```
X = 20;
Y = ( X - 2 ) * 4 + ( X - 2 ) * 8;
```

If this code were translated to assembly as-is, the `X - 2` sub-expression would be evaluated twice, even though its value doesn't change from one instance to the next. An optimizing compiler would notice this and possibly save the value of `X - 2` once in a temporary variable or register before evaluating the larger expression.

Preprocessing

Anyone who's used a C compiler before will be familiar with the concept of preprocessing. A *preprocessor* is a special layer of software that sits between the source code and the lexical analyzer, adding an additional early phase to the compilation process. The preprocessor filters and transforms the incoming source code according to special *directives* written directly into the code itself by the users. These directives tell the preprocessor to perform specific tasks and help create an enhanced, clarified version of the source code just before the compiler itself sees it.

Preprocessing, whose name literally means "processing that occurs before compilation," is generally most useful for allowing the user and the compiler to see the source code in two different ways. As an example of this, let's look at two of the most useful functions of a preprocessor—file inclusion and macro expansion. Figure 12.14 demonstrates the role of the preprocessor in the compilation process.

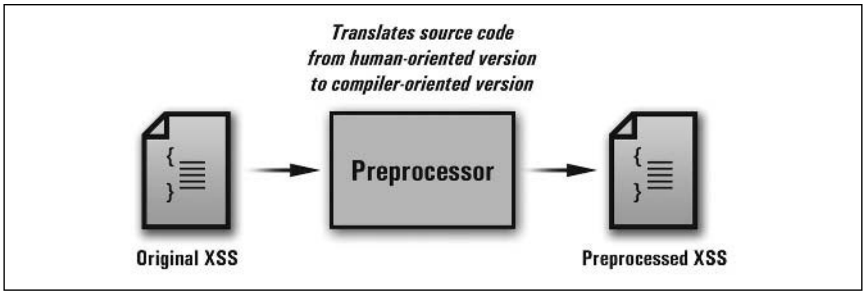


Figure 12.14

The preprocessor's role in the compilation process.

File Inclusion

File inclusion directives allow the user to write code in multiple files for the purpose of physically separating various components of the source, which are collapsed to a single file just before being fed to the compiler. For example, let's look at three different files, each of which contain C code:

file0.c

```
void Func0 ()
{
    printf ( "This is function zero." );
}
```

file1.c

```
    void Func1 ()
    {
        printf ( "This is function one." );
    }
```

file2.c

```
    #include "file0.c"
    #include "file1.c"

    void Func2 ()
    {
        printf ( "This is function two." );
    }

    main ()
    {
        Func0 ();
        Func1 ();
        Func2 ();

        return 0;
    }
```

Without the help of the preprocessor and its `#include` directive, `file2.c` would not compile. Even if the functions `Func0 ()` and `Func1 ()` were defined in their respective files, the compiler wouldn't have any idea they existed and would consider the calls to them invalid. In addition, the `#include` lines themselves would cause an error simply because the compiler wouldn't understand what `#include` is. With file inclusion, however, the contents of `file0.c` and `file1.c` are merged into `file2.c` in the preprocessing phase, replacing the directives that referenced them. The compiler ultimately ends up seeing the following, without ever knowing more than one file was involved:

```
void Func0 ()
{
    printf ( "This is function zero." );
}
```

```
void Func1 ()
{
    printf ( "This is function one." );
}

void Func2 ()
{
    printf ( "This is function two." );
}

main ()
{
    Func0 ();
    Func1 ();
    Func2 ();

    return 0;
}
```

Check out Figure 12.15 to see a more visual take on this process. Because the `#include` lines were physically replaced with the contents of the files they specified, the compiler never knew they were there. This is a good thing, because the compiler doesn't even necessarily know the preprocessor exists and certainly wouldn't understand its directives.

Using file inclusion directives allows you to logically group your functions and variables in separate files, and even build libraries of reusable code. Ultimately, even game scripting projects will benefit greatly from the ability of one source file to reference another when the project's complexity reaches a certain point.

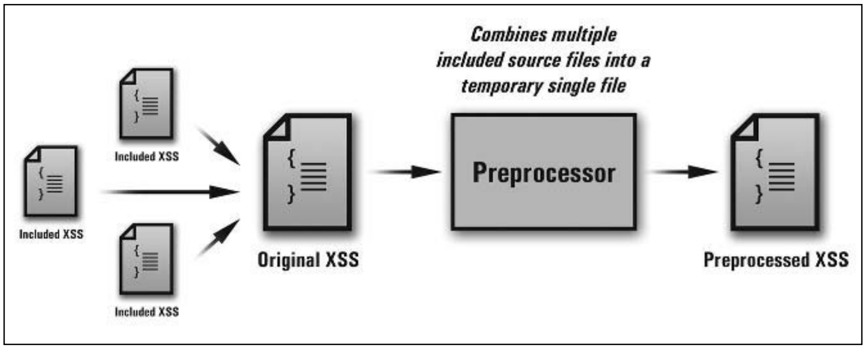


Figure 12.15

File inclusion enables the user and compiler to see two different versions of the source.

Macro Expansion

Macros are another popular feature of the C preprocessor, and are a great way to define symbolic constants or encapsulate logic without using a function.

Macros in C are defined with the `#define` statement, which simply replaces all instances of the macro's name with its value. For instance, consider the following constants, each of which are defined with macros:

```
#define X 32
#define Y 8192
#define Z 32768
```

In this example, the macro *names* are `X`, `Y`, and `Z`, whereas the *values* are `32`, `8192`, and `32768`. Consider these constants in a simple expression:

```
int MyVal = X + ( Y * Z );
```

If the compiler were to attempt to process this, it'd recognize `MyVal` as a valid identifier but consider `X`, `Y`, and `Z` to be undeclared and report an error. Fortunately, the preprocessor *expands* each macro by replacing the name with its value, which means the compiler will end up seeing this:

```
int MyVal = 32 + ( 8192 * 32768 );
```

which the compiler will of course consider perfectly acceptable. The beauty of macros, however, is that there's no space or performance issue associated with their use whatsoever. There's no need to allocate space for macros on the stack, because they're used directly in the source as literal values. For this same reason, they're even faster than using variables or traditional constants, because the runtime environment doesn't have to look up their values at runtime. Despite this, however, the programmer still gets the advantages of dealing with a symbolic constant instead of a raw value.

The important point to realize about macros is that they aren't restricted by the same limitations as a constant defined with the compiler. For example, using the C++ keyword `const`, you can create the same constants created in the previous example, with the added benefit of strong type checking:

```
const int X = 32;
const int Y = 8129;
const int Z = 32768;
```

`#define`, however, can be used for quite a bit more than just single values. For example, imagine the following:

```
#define string char *
```

The previous line of code associates `char *` with the name `String`, so you could declare a string-returning function like this:

```
string MyFunc ();
```

The preprocessor will automatically expand this out to the following before the compiler gets its hands on it:

```
char * MyFunc ();
```

Notice that here, the macro name was replaced with an entire string of text, containing spaces and everything. `#define` is not simply limited to numeric values.

Once again, the common thread is that macros let the user see the source in one form, which is often more convenient or easier to work with, whereas the compiler sees it in a different form.

Parameterized Macros

In addition to simply defining symbolic constants and arbitrary strings, another form of macros, known as *parameterized macros*, can accept parameters and actually "behave" differently based on their values. For example, imagine the following macro:

```
#define Square( X ) X * X
```

What this macro is saying is that `Square` is replaced with `X * X`, such that `X` is replaced with whatever the user provides. For example, the following line of code:

```
int Expr = Square ( 4 ) + Square ( 8 );
```

would be expanded by the preprocessor to:

```
int Expr = 4 * 4 + 8 * 8;
```

`Square` is an example of using a macro to encapsulate an entire expression, making the code easier to read and more flexible. And with the capability to pass parameters to take the place of variables within the expression, parameterized macros are almost as versatile as actual functions. Unlike functions, however, a call is never made, a stack frame is never produced, and the flow of execution never changes at runtime. It's all resolved at compile-time, meaning there's literally zero speed overhead when using a macro instead of a hard-coded expression. From the compiler's perspective, it *is* hard-coded.

NOTE

Remember, hard coding is only bad when it's performed manually by a human. If a preprocessor translates a source file containing directives into another version that appears as if it were hard-coded to the compiler, it has no negative effect on the coder and is therefore acceptable. In other words, hard-coding is fine as long as it's *transparent* from the coder's perspective.

Retargeting

You learned earlier that a compiler can be split into two distinct halves: the *front end* and the *back end*. The front end is in charge of turning the source language into I-code, whereas the back end is in charge of translating that I-code to a specific assembly language like XVM assembly. What you may notice here, however, is that the front and back ends work almost entirely independently of each other, as shown in Figure 12.16. Think back to the discussion of integration and abstraction layers in Chapter 6. It's the same idea.

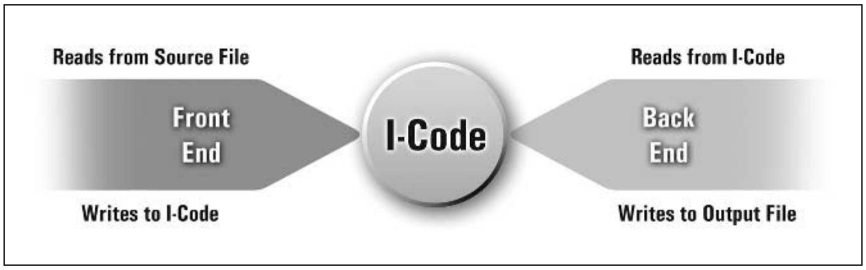


Figure 12.16

The front and back ends are entirely oblivious to each other's actions, thanks to the intermediate I-code layer.

The back end, for example, doesn't care or even know where the I-code it's working with came from. The original source language may have been XtremeScript, C, Pascal, Sanskrit, or whatever, but as long as it's reduced to valid I-code, the back end won't know the difference or have any reason to care. This means that the source language of the compiler can change without affecting the back end. If you suddenly decide that you'd rather implement Pascal in your scripting system instead of XtremeScript, the back end would never have to know the difference. Or, you may just want to open up the possibility of using both languages. You could leverage the common back end to make the construction of the second compiler much easier.

The same goes for the front end. Once the source code has been compiled down to I-code, the front end doesn't care what the back end does with it. It may translate it to XVM assembly, or even directly covert it to XVM bytecode. It'd even be possible to go as far as writing a back end that takes XtremeScript I-code and translates it to 80x86 machine code, allowing your scripts to run directly on the hardware. No matter what it does, the front end never has to change to accommodate it. This means that one front end can be used with multiple back ends, a process known as *retargeting* (because the *target platform* of the compiler can be changed or swapped).

NOTE

Many compilers are designed specifically *for* the purpose of retargeting. Writers of such compilers can then focus all their attention on the front end and logic optimization, leaving the details of the back end and target code generation up to specific users.

Retargeting has become a ubiquitous practice with the emergence of so many new platforms. Specifically in the case of console gaming, C and C++ compilers are needed for multitudes of hardware, ranging from the Gameboy Advance to the Playstation II, to the Xbox. Many of the compilers used to write code for these systems are simply retargeted versions of typical 80x86 compilers. Check out Figure 12.17.

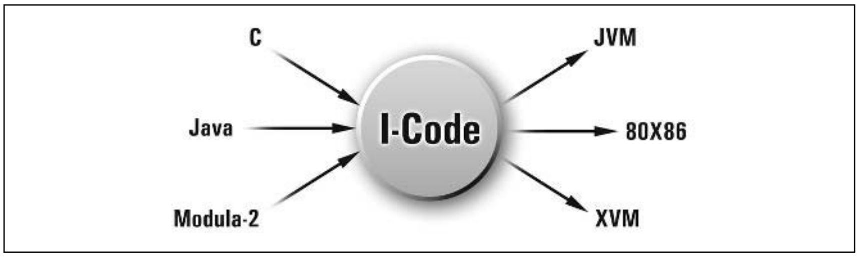


Figure 12.17

Source and target code can be swapped in and out, using the I-code as the "pivot point", as I like to call it.

Linking, Loading, and Relocatable Code

The XVM makes it pretty easy to load and execute scripts because each thread is given a separate address space. This means that no matter how many scripts are loaded at once, they all start from instruction zero. The same goes for stack indexes; globals always start at the bottom of the stack and work upward, with the stack frames of the script's functions being piled directly on top.

In the real world, things aren't often so pretty. Even though a Windows application is fully capable of multithreading, for example, each thread is loaded into the same overall address space, which means that only the first thread will have the luxury of beginning at index zero. If that thread's code consists of 1297 instructions, it'll occupy indexes 0 through 1296 of the code segment, which means that the second thread will start at index 1297 and move outward from there, as Figure 12.18 demonstrates.

This may not seem like a huge deal, but think about how jump instructions operate; at assemble-time, their labels are replaced with raw numeric indexes into the instruction stream, like `Jmp 3482`, for example. Because these indexes are calculated relative to zero, this means that in a shared address space, only the first thread would function properly. All other threads would inadvertently reference different blocks of code and end up making misguided jumps that would lead to an inevitable crash very quickly.

This problem is solved with what is known as *relocatable machine code*. When code is loaded into memory, the loader makes changes to the machine code on the fly to allow it to run properly relative to its *base address*. The base address is wherever the code is loaded from; in the case of the example mentioned previously, the first thread's base address was 0, whereas the second thread's was 1297.

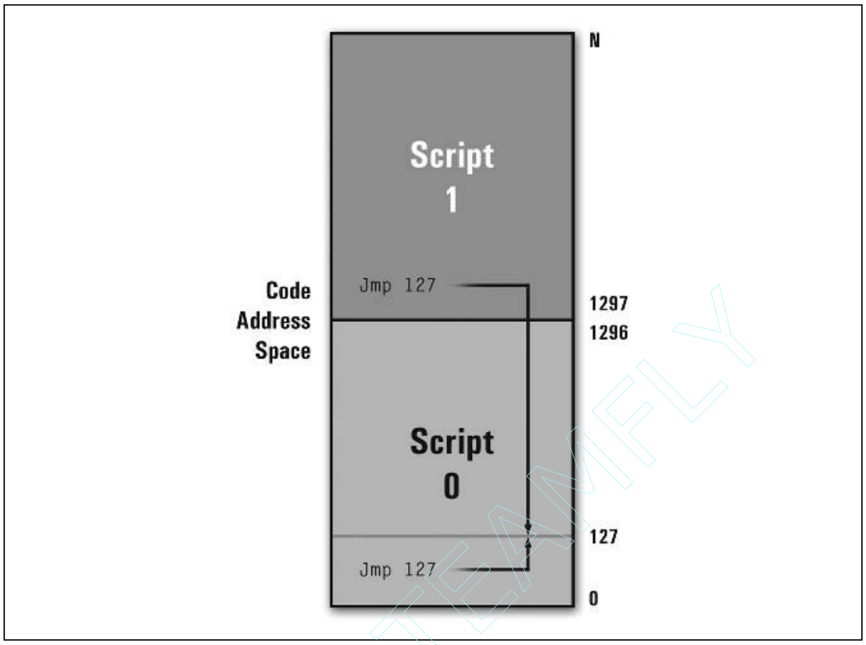


Figure 12.18

Loading code at an arbitrary address.

Imagine that the second thread contained three jumps, to addresses `22`, `481`, and `1906`. Because these addresses are relative to a base address of zero, which the second thread doesn't have, the real base address will need to be added to each jump target address so that the jumps will once again point to the proper instructions. The new jump targets will therefore be `22 + 1297 = 1319`, `481 + 1297 = 1778`, and `1906 + 1297 = 3203`.

Issues like relocation are handled by two pieces of software—the *linker* and the *loader*. The linker operates just after the compiler, and is used to translate the compiler's output (usually a machine code format called *object code*) to a ready-to-run executable. The loader is usually part of the operating system or runtime environment and is in charge of reading the executable's contents from the disk and properly placing it in memory, taking relocatable addresses into account. Fortunately for you, object code, linking, and relocation aren't among your concerns. They are helpful concepts to understand, however, and play a major role in other applications of compiler theory. Figure 12.19 illustrates this.

Targeting Hardware Architectures

Once the virtual compiler is done, you can *really* up the ante by attempting to retarget it for a hardware platform like the 80x86. Advantages of this might be to directly output .DLLs instead of .XSEs, allowing compiled scripts to run at hardware-level speeds while still being written in a sim-

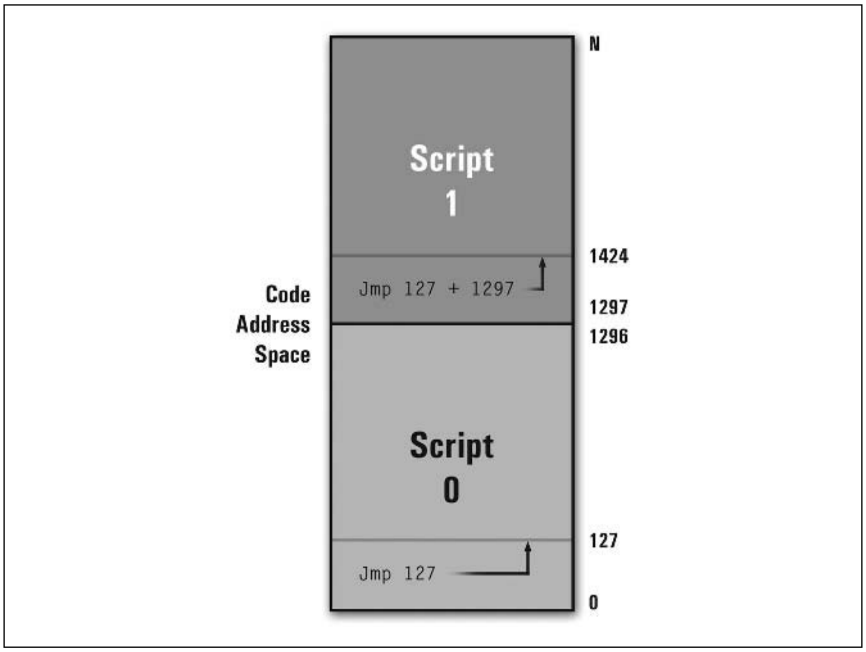


Figure 12.19

Relocatable code can be loaded anywhere in memory.

plistic and custom-designed language like XtremeScript. The script editor for Quake 3, for example, is capable of producing both hardware machine code DLLs and virtual machine-compatible executable scripts.

Targeting a hardware platform is hardly a trivial matter, however. The virtual machine in this book is designed with the utmost of simplicity and ease of use in mind; chief among examples of this design strategy is its typeless nature. Platforms like the 80x86, however, are strongly typed; many members of this particular family only deal directly with integer data. Strings must be manually managed by the programmer, and floating-point numbers can only be manipulated by accessing special external hardware like the 80X87 family of FPUs. To make matters worse, the issue of relocation will rear its ugly head, as well as the countless other complications of running code on real hardware. Memory protection, I/O permissions, precompiled runtime libraries—the list goes on and on.

Regardless of the complexity, however, there's a definite advantage to be had if you can pull it off. Dynamically loading compiled script code at runtime allows the programmer to maintain the same flexibility and ease of use scripting systems like XtremeScript are known for, but without the huge speed overhead associated with code running in a virtual machine.

Summary

If anything, this chapter has served as a much-deserved break after pressing through the workload of Chapters 9 through 11. Unfortunately, it's more like the calm before the storm, however, because there won't be a moment's rest in the upcoming chapters. Now that you can talk the talk of compiler writers, it's time to see whether you can handle the reality of translating high-level code to assembly language. Remember, this is what it's all been leading up to—you're finally in the home stretch.

Starting in the next chapter, you're actually going to start writing the XtremeScript compiler. When it's complete, the XtremeScript system will be finished, and you'll have everything you need for custom game scripting. This chapter has introduced you to almost everything you'll learn in order to do it, so you should have a good idea of what lies ahead. You've made it this far, so hold that chin up and keep moving!

CHAPTER 13

Lexical Analysis

"I'm a geneticist—I write code.
A, G, T, P, in different combinations."
——*Burchenal,* Red Planet

After all the build-up and preparation, it's time to really get your hands dirty by building the first major component of the XtremeScript compiler—the lexical analyzer. As you learned in Chapter 9, the lexer is one of the most pivotal phases of a compiler's pipeline; despite it's semi-trivial implementation, it provides one of the most straightforward and effective ways to break down and analyze human-readable source, by converting a raw stream of characters into two, far more structured streams of lexemes and tokens, as shown in Figure 13.1.

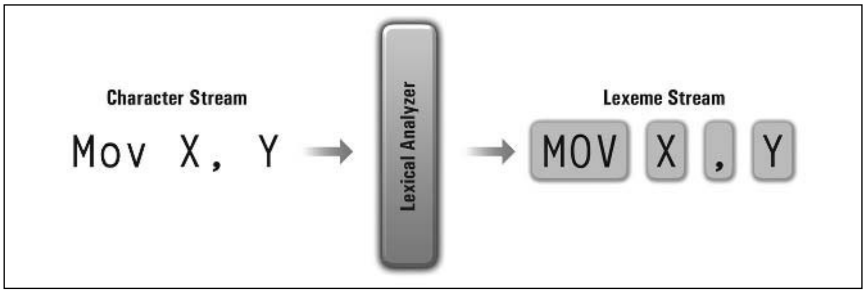


Figure 13.1

A raw character stream being converted to a stream of lexemes.

As I said, lexical analyzers are definitely among the easier components of a compiler to build. They only require a basic knowledge of string processing, and once your lexer can identify only a handful of major token types, you're capable of understanding a huge portion of the code out there. Lexing also provides the very foundation for *parsing* (the subject of the next chapter), the phase in which a basic compiler does most of its work to ascertain the meaning of the incoming source code.

So without further ado, let's get started. This chapter will be a reasonably simple and straightforward one, but will be highly productive in your quest to compile high-level code. It will cover:

- The basics of lexical analysis and the many ways in which it can be approached.
- The construction of a basic, state machine-based lexer capable of lexing integer and floating-point values.
- A second lexer that builds on the first by adding support for identifiers and reserved words.
- A third, complete lexer that understands the full XtremeScript language syntax by adding support for operators, delimiter characters and string literals.

By the end of this chapter, you'll have a finished lexical analyzer, and the XtremeScript compiler will already be partially complete. So let's get started!

The Basics

You've already learned about the theory and concepts behind lexical analysis fairly thoroughly (in Chapters 9 and 12). The construction of the XASM assembler in Chapter 9 required a structured and robust lexical analyzer, so you should already have a reasonable grasp of what's going on here. For the sake of completeness, however, and to make these chapters a bit more self-contained, I'm going to gloss over it all, very quickly, one last time.

From Characters to Lexemes

Lexical analysis is all about the conversion of a stream of raw character data, which a script's source code is initially presented as, into a more structured format. The first step in this process is isolating patterns in this character data that represent larger, more coarsely-grained structures known as *lexemes*. Lexemes are to characters like words are to letters, and by isolating them, the lexical analyzer has created a more coarse-grained, easy-to-use data set. For example, consider the following stream of characters:

```
if ( GetPlayerLocation ( X, Y ) == CASTLE_DRAWBRIDGE )
```

As humans, we can easily read it and identify its language and format (a C-style script fragment), as well as its meaning (a test to see whether the player is standing in front of a castle). To a compiler, however, it's just a meaningless string of characters. The *reason* we can read it, however, stems from our ability to break it up into logical groups and patterns. We know that the spaces, commas, and parentheses are there to help separate entities, and that certain character sequences can be combined to form reserved words, identifiers, and operators. Armed with this information, it's considerably easier to determine what's going on. Fortunately, this is exactly what a lexer's job is. After making a pass over the source code, it'll output this:

```
IF
(
GETPLAYERLOCATION
(
X
,
Y
)
==
CASTLE_DRAWBRIDGE
)
```

In one fell swoop, it's isolated the statement's major components and separated them so they can be parsed sequentially. It's also done a bit of clean-up by discarding whitespace and converting everything to uppercase. If you can imagine reading a book one character at a time, perhaps by having a friend look at the pages and verbally tell you each character individually, you can imagine how hard it'd be to detect words, sentences, and inflection on the fly. Without the help of the lexer, this is what a compiler would have to do. With the lexer, however, the compiler can look at the source code in (almost) the same way you could if you had the book right in front of you and could read it like you normally would. Check out Figure 13.2.

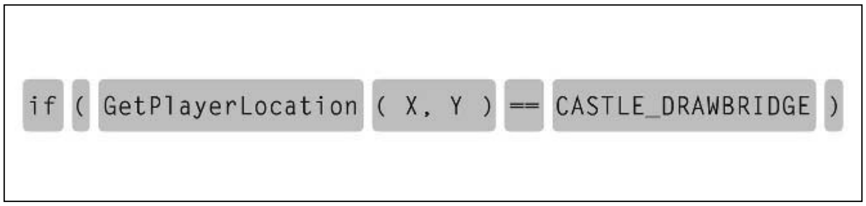


Figure 13.2

Reducing a raw string of characters to more coarse-grained elements.

NOTE

I've been using the terms *fine-grained* and *course-grained* fairly often throughout this book. In case you aren't familiar with what I mean, think of it like this—a *fine-grained* stream of data is like sand; it's composed of hundreds, thousands, or millions of very tiny pieces that have no direct relationship to one another. Like a handful of sand, a stream of characters is hard to sift through because it doesn't contain any big, easily usable chunks. Now imagine that sand being densely packed together to form pebbles; the material has now gone from being *fine* to slightly *coarser*, which is why I call it coarse-grained. The pebbles are analogous to lexemes; small groups of the sand particles are lumped together, resulting in a smaller overall set of larger individual parts. If these pebbles were further mashed together to form larger rocks, the overall size of the set would decrease even more, whereas the size of its constituents would increase proportionally. At this point, the set is becoming even more coarsely grained, like lexemes being grouped into statements, blocks, and functions. As you can imagine, coarse data sets are usually easier to work with than fine-grained ones because they're simpler and more self-evident.

Tokenization

Of course, even with the character stream grouped into lexemes, there's still a lot the compiler has to do in order to determine what each word means. At the very least, it'll have to constantly perform string comparisons with `strcpy ()` to determine the difference between `3.14159`, `IF`, and +=. It'd be nice if the lexer would not only produce the lexemes, but also perform its own internal analysis to determine what exactly the lexemes are. This is handled in an additional phase called *tokenization.*

The tokenizer aspect of a lexical analyzer is responsible for determining exactly what type of character sequence was extracted from the source code. The result of this analysis is a piece of information known as a *token,* which is basically a simple code that refers to a specific *type* of lexeme. This way, the rest of the compiler not only has a well-defined stream of lexemes, it also has a stream of tokens that can be used to more clearly identify those lexemes' types.

As I mentioned in Chapter 9, lexical analysis and tokenization are lumped together into a single phase. In the XASM implementation, however, they still occurred serially; the lexer would first find the next lexeme by reading all characters until a proper delimiter was reached (like a comma or whitespace). The tokenizer would then perform a number of comparisons and other forms of analysis to determine exactly what the lexer found. Of course, this is all how it was done using the "brute force" method; as I mentioned, there are more sophisticated ways to lexically analyze input, and as you'll see later in this chapter, this solution generally performs tokenization and lexical analysis in parallel. In fact, the XtremeScript lexer will actually *have* to perform these two tasks simultaneously to complete its job.

Lexing Methods

Generally speaking, there are two ways to classify lexical analyzers—those that are written by hand and those that are generated using a utility of some sort. In the former case, the compiler writer manually codes the functionality of the lexer and tokenizer and uses any method. In the latter case, the compiler writer prepares a file describing the different lexemes and tokens that the source language uses (most commonly through a series of regular expressions to literally define the character sequences in which they'll appear), which the utility uses to output actual C or C++ code implementing the lexer that users can copy and paste into the compiler's

NOTE

In case you aren't familiar, *regular expressions* are a way to describe intricate character sequences and patterns for use in heavy string processing. They're commonly used to describe the exact forms in which lexemes will appear in the source file, and are commonly used by lexical analyzer generators.

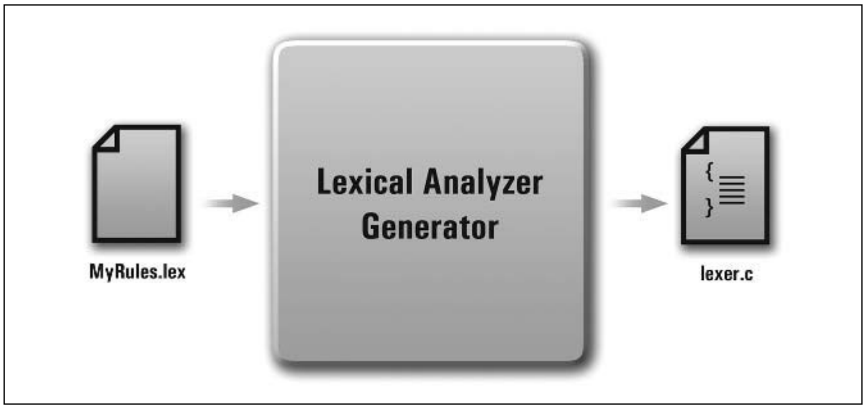


Figure 13.3

Lexical analyzer generators use files containing rules and descriptions to produce a lexer in actual C/C++ code.

framework (see Figure 13.3). Examples of such utilities are *lex*, a common UNIX and Linux utility, and *Flex*, lex's Win32 port.

Lexer Generation Utilities

I won't be covering the use of programs like lex and Flex to generate lexers; they're invaluable when creating real-world compilers, but they obviously don't shed much light on a lexer's inner workings. From the perspective of a book, it makes more sense to do things by hand and learn what's actually going on than to have something do it for you. You'll still probably opt to go with a lexer generating utility in your future projects, but you'll do so with far more insight and understanding as to what's going on under the hood.

Hand-Written Lexers

Hand-written lexers are still commonly used in small projects where minimal language translation must be performed. Of course, there's no law telling you that you can't manually write the lexer for a full-scale compiler, and because you'll learn so much more that way, that's what you're going to do in this book.

Overall, compiler theory is a strongly refined, highly structured field with countless time honored practices. Ironically, however, with the immense proliferation of lexer-generating utilities, hand-written lexical analyzers have been more or less left behind, giving you free reign to approach them in any way you see fit. Of course, this doesn't mean you can't take a few cues from lexer generators, and in fact, you'll use their approach as the basis for your own. You'll still have the luxury, however, of simplifying things here and there and taking a somewhat unorthodox approach to certain aspects for the sake of keeping things simple.

Let's start by discussing some of the ways in which a hand-written lexer can be written.

Brute Force

The lexer you built for the XASM assembler in Chapter 9 was what I like to call a brute-force lexer. It got the job done in a simple and straightforward manner by grouping every character in the stream up to the next delimiter or instance of whitespace into a lexeme, and then performed some basic string analysis to determine exactly what it was. The advantage to this approach is that once you understand what's going on, the code is very readable and completely serial, as shown in Figure 13.4. Major events happen in a strongly defined sequence, making it easy to follow. Here are the specific steps that were followed:

- The next lexeme was extracted by reading all characters up until the next delimiter (like a comma or brace) or instance of whitespace. This substring of the character stream was considered the lexeme.
- The lexeme was physically copied, character-by-character, into a separate string buffer for further analysis.
- The isolated lexeme string was processed in a number of ways to determine exactly what it contained: an integer, a floating-point value, an instruction, or whatever.
- The token was returned to the caller, whereas the lexeme string itself was available via a separate function that could be called afterwards.

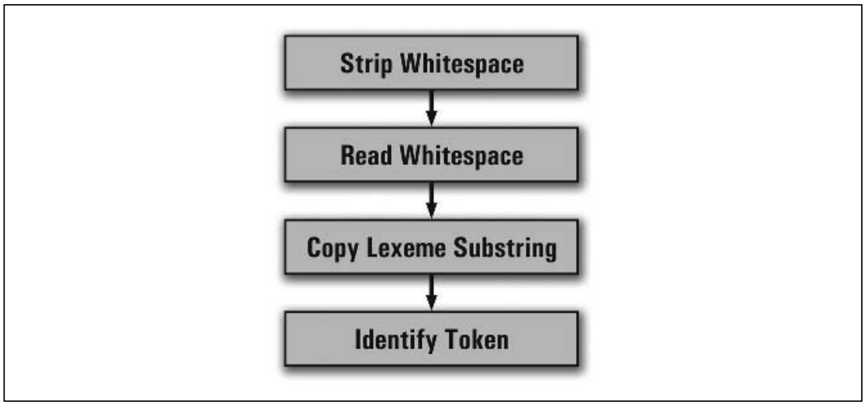


Figure 13.4

The major steps of the XASM lexer occurred serially.

This approach served you well by being simple, accessible, and clean enough to get the job done without resulting in spaghetti code or instability. Of course, there are other ways to go about it, most of which offer more structure and/or flexibility.

Semi-State Machines

I've seen this next class of hand-written state machines in a number of books involving compiler theory or related subjects. In these cases, the approach is closer to a state machine than the all-out brute force approach, but still not completely there. For this reason, I call them "semi-state machine" lexical analyzers.

The basic idea is to start by reading the first character from the next lexeme. Based on this initial character, a number of paths can be taken; if a digit or radix point is detected, a numeric token is probably being read. If a letter or underscore is detected, it's probably an identifier. If it's a delimiter or operator character, it's probably a delimiter or operator. In short, these lexers work by writing specialized functions or local blocks of code (usually organized in a `switch` block) for handling each token type.

Once the initial character is identified, a specific block of code can be invoked for reading the rest of the lexeme, because the lexer already has a good idea of what to expect. This is roughly the opposite of the XASM lexer's approach, which reads the lexeme first and tries to identify it afterwards. Because of a semi-state machine lexer's initial comparison, it has to perform minimal analysis only after extracting the lexeme, if then, because it usually knows what it's getting beforehand. Figure 13.5 demonstrates how this works.

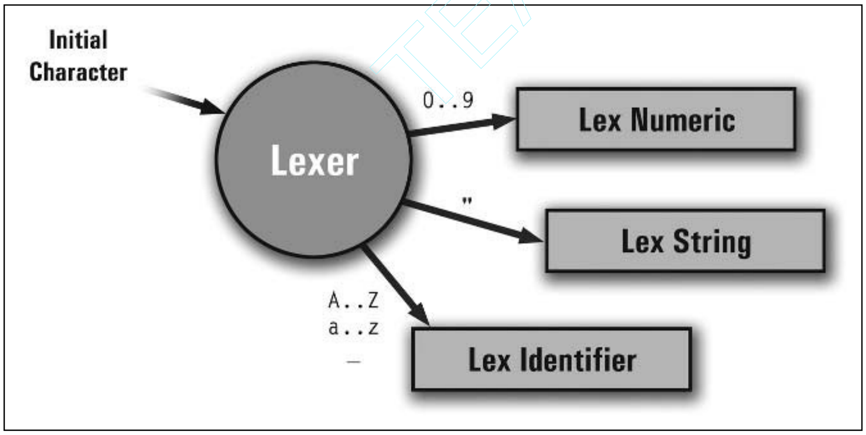


Figure 13.5

The "semi-state machine" approach to lexing.

Overall, this is certainly a clean and simple way to approach lexical analysis. It works in a straightforward manner, and gets the job done in a more compact manner by coupling lexeme extraction with token identification. It shares some of the behavior of a state machine by using an initial condition (the value of the first character) to alter its behavior later. This is similar to how a pure state machine lexer gets started. However, once inside a specific lexeme-extraction function, there generally isn't a whole lot of leeway to switch from one token type to another. This is where true state machines come in.

State Machines

State machines work on a simple principal—perform a task only once at each iteration of a loop, but do it differently depending on the situation. State machines can be applied effectively to string processing, because strings have to be iteratively analyzed—in other words, they must be dealt with on a sequential character-by-character basis. During this iteration, however, the capability to suddenly switch gears depending on the value of each newly read character allows the string processor to flexibly handle a wide range of input.

The layout of a state machine-based lexer is pretty simple; the entire thing takes place in one large loop, rather than a number of sequential loops like a brute force lexer. By the time this single loop is done iterating, the lexer has been completely isolated and the token has been identified. Programs like lex and Flex generate state machine-based lexers.

So how does a single loop do so much without being a huge, bulky mess? By alternating between a set of strongly defined *states*. Each iteration of the loop does three major things—reads in the next character, *transitions* to the next state if necessary, and performs whatever action the active state demands. This simple three-step process is enough to handle the entire set of lexemes and tokens of a high-level language. See Figure 13.6.

For example, when the lexer begins, it's in the "start state". The first character is read, and the loop determines whether the character's value is a sign to switch to another state. In the case of the start state, it almost always is. Let's say the initial character is `8`. The lexer immediately switches to the integer state, assuming that the character is the first digit in an integer numeric value.

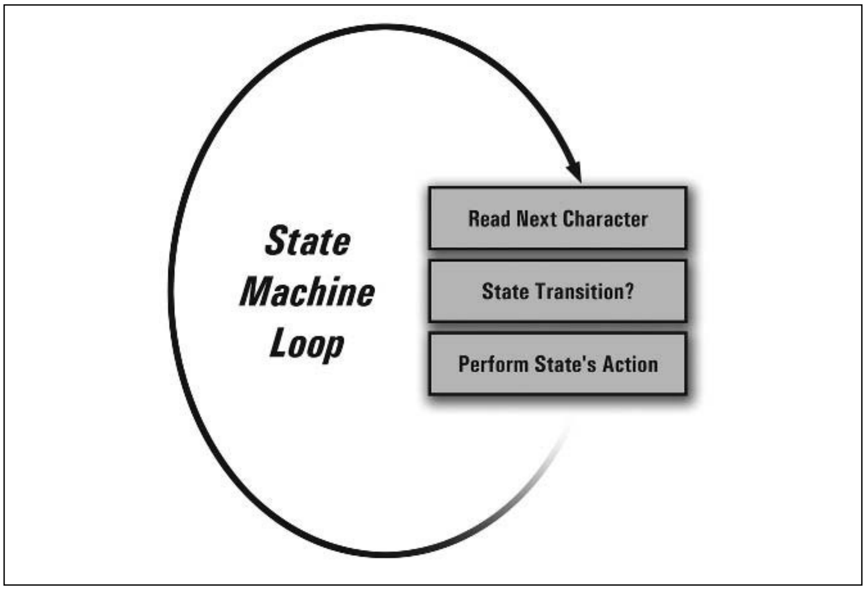


Figure 13.6

The basics of a state machine.

As the loop progresses, more and more characters are read in. Each time, their values are used to make a possible state transition. However, as long as digits are read in, the integer state is maintained. Furthermore, each newly read character is added to the end of an accumulating lexeme buffer. Finally, a character is read. This invokes another state transition—the loop now considers the lexeme a floating-point value. The remaining characters are digits, and as each is read in, it's added to the growing lexeme string. Furthermore, because all of them are valid digits, they don't disrupt the state—it remains a floating-point value in the eyes of the lexer. Figure 13.7 depicts a basic numeric lexer's state machine.

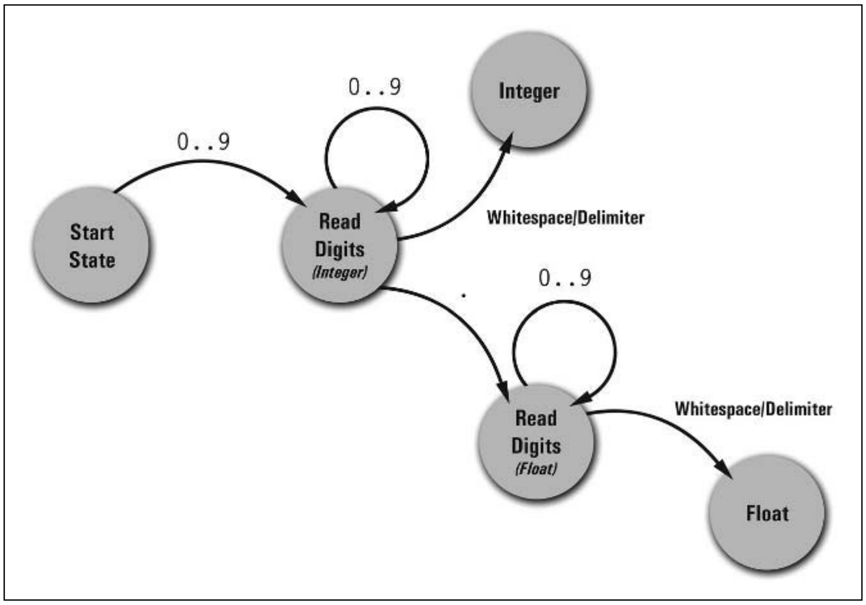


Figure 13.7

A numeric lexer's state machine.

When this loop completes, the lexeme buffer will contain the completed floating-point value, and the lexer's floating-point state will be equivalent to the token type. The beauty of this approach is that everything is done implicitly and in parallel. The tokenization aspect of the lexer's job is implemented via states and their transitions among one another. The lexeme extraction is done by adding each character as it's read to a growing string buffer. By the time the loop is done, everything is finished.

This chapter focuses on the construction of a state machine-based lexical analyzer for the XtremeScript compiler. You'll see how states and state transitions can be used to manage the formidable complexity of high-level code, and you'll finish with a complete lexer module that's almost totally ready to be dropped into the compiler framework you'll complete in the following chapters.

THE LEXER'S FRAMEWORK

You're going to begin by setting up a basic framework within which you can build the lexer. Specifically, you need a way to:

- Read a text file from the hard drive, line by line.
- Store the contents of the text file in a single, contiguous region of memory for easy processing.
- Display the output of the lexer's processing—both the current lexeme and token.

By getting this out of the way first, you can focus solely on the lexer's core logic for the rest of the chapter.

Specifically, the following lexical analyzers will be implemented as console application demos that load a text file and attempt to lex it. Each lexeme and token found in the file will be printed in a vertical list. The finished lexer will be capable of listing the lexemes and tokens for an entire XtremeScript source file.

Reading and Storing the Text File

Unlike XASM, a free-form, high-level language like XtremeScript is almost entirely unconcerned with line breaks, and considers them just another form of whitespace. Because of this, the lexer will accept its input as a single, null-terminated string. This way, from the first line of code to the last, the lexer can steadily read characters until it hits the null terminator that marks the end of the file.

The demo's `main ()` function will start by reading a single command-line argument that specifies which file should be loaded. If a file isn't specified, usage info is printed and the program exits. Otherwise, the file is opened for binary input (you'll see why in a moment):

```
main ( int argc, char * argv [] )
{
    // Print the logo
    printf ( "Lexical Analyzer Demo\n" );
    printf ( "\n" );

    // Validate the command line argument count
    if ( argc < 2 )
    {
        // If at least one filename isn't present, print
        // the usage info and exit
        printf ( "Usage:\tLEXER Source.TXT\n" );
        return 0;
    }

    // Create a file pointer for the script
    FILE * pSourceFile;

    // Open the script and print an error if it's not found
    if ( ! ( pSourceFile = fopen ( argv [ 1 ], "rb" ) ) )
    {
        printf ( "File I/O error.\n" );
        return 0;
    }
```

With the file open in binary mode, you can use the `fseek ()` command to determine its exact size and allocate a buffer accordingly. Remember, you're no longer concerned with individual source lines. In most free-form, C-style languages, the entire program can be thought of as one big character stream, and ultimately one contiguous stream of lexemes and tokens.

```
fseek ( pSourceFile, 0, SEEK_END );
int iSourceSize = ftell ( pSourceFile );
fseek ( pSourceFile, 0, SEEK_SET );
g_pstrSource = ( char * ) malloc ( iSourceSize + 1 );
```

`g_pstrSource` is a global string buffer containing the source file. Here's its declaration:

```
char * g_pstrSource;
```

You now have a character buffer large enough to hold the entire source file, so you're ready to read it in. There is one issue to note, however, and that's the highly system-dependent nature of line break codes within a text file. On a Windows or MS-DOS system, a newline is represented with a two-character sequence—the character values `13` and `10`. On a UNIX system, on the other hand, it's simply represented by a single byte of the value `10`. Other systems have even more exotic methods of marking the end of a line.

The upshot is that the compiler should store the source file in a platform-neutral format, so any unorthodox newline issues can be taken care of as the file is loaded. By converting the native platform's format to a consistent, neutral format, you can eliminate this issue early on. I've chosen to represent line breaks internally simply as typical C `\n` newlines, and because I developed XtremeScript on the Win32 platform, this means I have to detect and convert its native two-character line break codes. Here's the source file loader, with the line break issue taken into account:

```
char cCurrChar;
for ( int iCurrCharIndex = 0;
      iCurrCharIndex < iSourceSize; ++ iCurrCharIndex )
{
    // Analyze the current character
    cCurrChar = fgetc ( pSourceFile );
    if ( cCurrChar == 13 )
    {
        // If a two-character line break is found, replace
        // it with a single newline
        fgetc ( pSourceFile );
        -- iSourceSize;
        g_pstrSource [ iCurrCharIndex ] = '\n';
    }
    else
    {
        // Otheriwse use it as-is
        g_pstrSource [ iCurrCharIndex ] = cCurrChar;
    }
}
g_pstrSource [ iSourceSize ] = '\0';

// Close the script
fclose ( pSourceFile );
```

In the final compiler, the lexer itself won't be responsible for loading the source file, but this chapter's demos will, so it's good to iron this issue out now. You now have the entire source file represented internally as a contiguous, null-terminated string.

Displaying the Results

As the lexer runs, the program will print out its results line-by-line. Afterwards, a small summary will be printed, consisting of the number of lexemes detected. Here's a slightly gutted version of the loop that will generate this output:

```
// The current token
Token CurrToken;

// The token count
int iTokenCount = 0;

// String to hold the token type
char pstrToken [ 128 ];
```

```
// Tokenize the entire source file
while ( TRUE )
{
    // Get the next token
    CurrToken = GetNextToken ();

    // Make sure the token stream hasn't ended
    if ( CurrToken == TOKEN_TYPE_END_OF_STREAM )
        break;

    // Convert the token code to a descriptive string
    switch ( CurrToken )
    {
        // Create a string to represent the token
    }

    // Print the token and the lexeme
    printf ( "%d: Token: %s, Lexeme: \"%s\"\n",
        iTokenCount, pstrToken, GetCurrLexeme () );

    // Increment the token count
    ++ iTokenCount;
}

// Print the token count
printf ( "\n" );
printf ( "\tToken count: %d\n", iTokenCount );
```

Some of this won't make much sense at this point, because you haven't actually covered the lexer itself yet, but most of it should be self-explanatory based on what you learned in Chapter 9. Like before, you're going to create a simple `Token` data type that represents a token (it's really just an integer wrapped with a `typedef`). A `Token` is then declared to hold the current token retrieved by the lexer. A token counter is declared and set to zero, and a string that will contain the token's description is statically allocated.

The loop itself runs until the `GetNextToken ()` function returns a flag indicating the end of the token stream. Based on the current token, a `switch` block is used to fill the token description string with some small piece of information that can be printed along with the lexeme to describe what's going on. This information is printed, and the token count is incremented. Finally, outside of the loop, the total token count is printed and the program exits.

Error Handling

Error handling won't be a particularly huge concern of these small demos, but just to keep things clean, unexpected character input will be flagged with the following function:

```
void ExitOnInvalidInputError ( char cInput )
{
    printf ( "Error: '%c' unexpected.\n", cInput );
    exit ( 0 );
}
```

Whenever the lexer reads something it doesn't understand, it'll use this function to alert the users and exit the program. Simple and to the point.

A Numeric Lexer

With the framework in place, you're ready to get started with the first version of your culminating XtremeScript lexer. To start off with a simple but effective example, you're going to lex a text file containing randomly spaced, nonnegative numeric values in either integer or floating-point format. As an example, here's the text file I created to test it:

```
293048 24 895523
3.14159
235
        253
          52435 345

           459245

 22 .5 .35 2.0

1
 0.0
  1.0
   0

    02345

    63246 0.2346
    34.0
```

As you can see, it has an intentionally extreme amount of whitespace irregularity to make sure the lexer's robustness is really put through its paces. There are few things in the world more irritating than a compiler whose acceptance of whitespace can't be trusted; we should go to great lengths to ensure that using XtremeScript is just as easy and natural as using a C/C++ compiler from Microsoft or Borland.

A Lexing Strategy

It's time to code the lexer itself, so let's review the strategy. Of course, it's all about the state machine. To lex integers and floats, your lexer needs to support a small number of states that can transition from one to another easily. As I mentioned, here's the basic process of a state machine-based lexer at work:

1. Just as in the case of the lexer developed in Chapter 9, two indexes into the character stream are initialized. They both point to the start of the current lexeme. The second of these two indexes will move forward as the lexeme is read, so that it points to the end when the loop finishes.
2. A variable used to track the loop's current state is declared and initialized to the start state.
3. The next character is read in from the stream. If this character is a null terminator, the end of the source file has been reached, and the loop breaks.
4. The current character is analyzed depending on the certain state. Each state has a set of characters that it accepts as valid input, a set of characters that indicate it should transition to another state, and a set of characters that are entirely invalid and thus erroneous (which is usually any character not in the first two sets). If a state transition is not warranted, the state remains the same. Otherwise, the loop's state tracking variable is set to another value to facilitate the transition.
5. With the current state handled, as well as any possible state transitions, the current character is added to a string buffer containing the culminating lexeme. The index to the end of the lexeme is incremented as well.
6. If the character warranted a state transition to what is known as a *terminal state*, the lexeme is complete and the loop is terminated.
7. Outside the loop, a null terminator is applied to the lexeme buffer to make it a complete string.
8. The index pointing to the end of the lexeme is decremented by one, because whichever character transitioned to the loop to a terminal state is not part of the lexeme itself, but rather the first character of the next lexeme (or the whitespace that precedes it).
9. The final lexer state is used to determine the token type, which is often a one-to-one mapping.
10. The token type is returned to the caller.

Seems like a pretty straightforward process, huh? Now that you have a conceptual overview of what the lexer will do, let's jump into the code. The lexer is primarily implemented with the `GetNextToken ()` function, which performs the previous steps and returns a `Token` value to the user, indicating the type of the lexeme it read. Just like in XASM, the lexeme is not returned by this function, but rather available through another function, `GetCurrLexeme ()`. This function just returns a pointer to a global string buffer containing the lexeme extracted by `GetNextToken ()` (again, like in XASM).

State Diagrams

You've already seen a few, but I'd like to take a quick moment to introduce *state diagrams*. State diagrams are used to express state machines in a visual manner, and consist of two major elements—*states* and *edges*. States are usually represented within the diagram as circles with a caption inside that describes what the state does. Edges connect states, and therefore represent state transitions. Each edge has a label that defines which criteria are required to invoke the transition. Figure 13.8 demonstrates an example of a state diagram.

Notice that sometimes, an edge will transition into the state from which it originated. This is a commonly seen notation, as it's often helpful to explicitly define which criteria cause the active state to remain where it is.

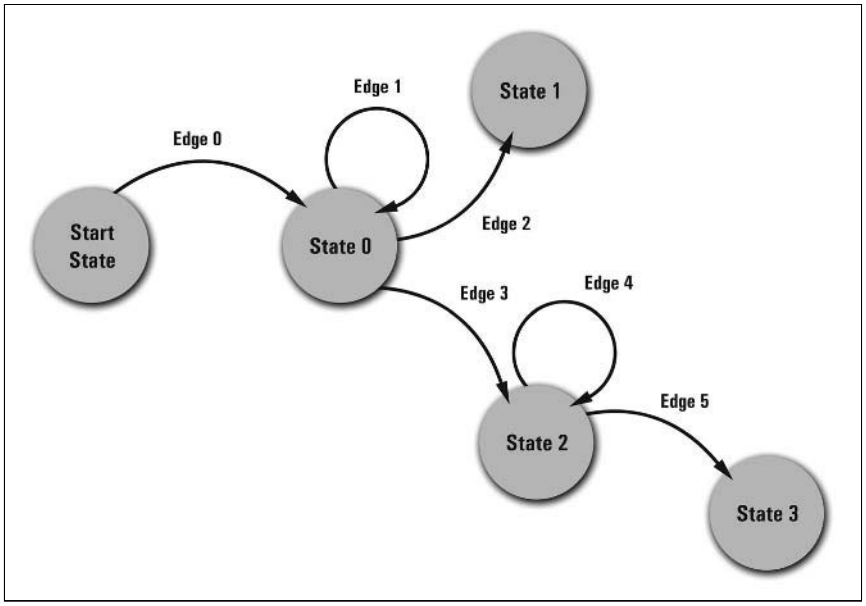


Figure 13.8

An example of a state diagram.

States and Token Types

As the lexer executes, it will frequently transition from one state to the other to follow the format of the input. Rather than just refer to these states as arbitrary numbers, it helps to use symbolic constants to make everything easier to read. The same goes for token types—as you already saw in Chapter 9, tokens can be represented well using constants.

Let's start with the lexer states:

```
#define LEX_STATE_START          0       // Start state
#define LEX_STATE_INT            1       // Integer
#define LEX_STATE_FLOAT          2       // Float
```

The lexer begins in a start state, and can transition to an integer and floating-point state. These three constants give you everything you need to track such transitions. Now let's look at the token types:

```
#define TOKEN_TYPE_END_OF_STREAM    0       // End of the token stream
#define TOKEN_TYPE_INT              1       // Integer
#define TOKEN_TYPE_FLOAT            2       // Float
```

When the lexing process is finished, the caller will be left with an integer token, a floating-point token, or a flag representing the end of the token stream.

Lastly, even though tokens are just numeric values, I like to wrap them in the `Token` type to make things more readable:

```
typedef int Token;
```

Initializing the Lexer

Before anything can happen, the lexer needs some basic initialization. Currently, in the case of the simple numeric lexer, all this means is setting the lexer's indexes to zero, so that it knows to start from the beginning of the file. Even though this is all you need for now, it's a good idea to wrap this process in a small function so you can add to it as the lexer grows more complicated if necessary. Here's the code:

```
void InitLexer ()
{
    // Reset the start and end of the current lexeme to the
    // beginning of the source
    g_iCurrLexemeStart = 0;
    g_iCurrLexemeEnd = 0;
}
```

These indexes are global so that functions like this and others, as well as `GetNextToken ()`, can access them easily. Here's their declaration:

```
int g_iCurrLexemeStart;
int g_iCurrLexemeEnd;
```

With the initialization out of the way, let's get back to the lexer itself. First, however, let's quickly cover the string buffer that will be filled with the lexeme by `GetNextToken ()`. Here's its declaration:

```
char g_pstrCurrLexeme [ MAX_LEXEME_SIZE ];
```

The `MAX_LEXEME_SIZE` constant dictates the maximum size a given lexeme can be. I like to set it to 1024, but any reasonably large number should do. I wouldn't set it any lower than 512 or 256, however, because string literals are treated like typical lexemes. Because game scripting often involves heavy use of dialogue, you want to have all the legroom you need for strings:

```
#define MAX_LEXEME_SIZE             1024
```

Beginning the Lexing Process

Whenever the lexer is called, its first task is to initialize its own internals. The first step is to set the index pointing to the beginning of the current lexeme to the one pointing to the end. The reason for this is simple—after the last call to `GetNextToken ()`, the second index points to the character just after the end of the last lexeme, which is where the current one begins. By setting the first index to this value, the two indexes will both point to the start, where they should. This index is then compared to the length of the string—if it's beyond the last character, `TOKEN_TYPE_END_OF_STREAM` is returned. Let's take a look at the code for starting up the lexing process. I'll discuss the rest of what it does afterwards:

```
Token GetNextToken ()
{
    // ---- Start the new lexeme at the end of the last one
    g_iCurrLexemeStart = g_iCurrLexemeEnd;

    // If we're past the end of the file, return an end of stream token
    if ( g_iCurrLexemeStart >= ( int ) strlen ( g_pstrSource ) )
        return TOKEN_TYPE_END_OF_STREAM;

    // ---- Set the initial state to the start state
    int iCurrLexState = LEX_STATE_START;
    // ---- Flag to determine when the lexeme is done
    int iLexemeDone = FALSE;

    // ---- Loop until a token is completed
    // Current character

    char cCurrChar;

    // Current position in the lexeme string buffer
    int iNextLexemeCharIndex = 0;

    // Should the current character be included in the lexeme?
    int iAddCurrChar;
```

Once the lexeme indexes have been synchronized, `iCurrLexState` is set to `LEX_STATE_START`. As you'd imagine, this is the variable you'll be using to track the current state as the loop executes. You then create a flag called `iLexemeDone`, which is set to `FALSE`. As the loop executes, this flag is continually checked to determine whether the lexeme is done and the loop can terminate. A character called `cCurrChar` is then declared—it will hold the current character as the loop executes. As each character is read, you'll also be adding them to a string buffer that will ultimately contain the entire lexeme. To track the current index in this buffer, you declare `iNextLexemeCharIndex` and set it to zero.

Lastly, a flag is declared called `iAddCurrChar`. Although it's true that characters read from the character stream are appended to the current lexeme, not all of these characters should be included. For example, you intentionally want to omit whitespace characters, as well as the delimiter or whitespace that will directly follow the lexeme. Because of this, each state in the loop that doesn't want its current character added to the lexeme can set this flag to `FALSE` to suppress it.

The lexer is primed at this point, so it's time for the state machine loop to begin.

The Lexing Loop

The lexing loop revolves around the currently read character, so the first order of business is reading it from the stream. You must also set the `iAddCurrChar` to `TRUE` by default, because most characters are added to the lexeme:

```
while ( TRUE )
{
    // Read the next character and exit if the end of the source
    // has been reached
    cCurrChar = GetNextChar ();
    if ( cCurrChar == '\0' )
        break;

    // Assume the character will be added to the lexeme
    iAddCurrChar = TRUE;
```

Next, the current state is used to determine what should be done with the character. Naturally, to determine what the current state is, you use a `switch` block. The first state to consider is the start state, represented by the `LEX_STATE_START` constant. From this state, anything other than whitespace will transition to another state, or to an error. The actual process of reading the next character is handled by a function called `GetNextChar ()`:

```
char GetNextChar ()
{
    // Return the current character and increment the lexeme end pointer
    return g_pstrSource [ g_iCurrLexemeEnd ++ ];
}
```

You'll notice that the lexeme end index is incremented automatically as the character is read. This is why I made it global. Now, simply by calling the function to read the next character, one of our two lexeme indexes is updated transparently.

Currently, you just need to worry about transitions to the integer and float states:

```
switch ( iCurrLexState )
{
    // The start state
    case LEX_STATE_START:

        // Just loop past whitespace, and don't add it to the lexeme
        if ( IsCharWhitespace ( cCurrChar ) )
        {
            ++ g_iCurrLexemeStart;
            iAddCurrChar = FALSE;
        }

        // An integer is starting
        else if ( IsCharNumeric ( cCurrChar ) )
        {
            iCurrLexState = LEX_STATE_INT;
        }
        // A float is starting
        else if ( cCurrChar == '.' )
        {
            iCurrLexState = LEX_STATE_FLOAT;
        }

        // It's invalid
        else
            ExitOnInvalidInputError ( cCurrChar );

        break;
```

The first thing the `LEX_STATE_START` state handler does is look for whitespace. Remember, the beginning of the lexeme is the only place whitespace is valid (because what you call "trailing whitespace" is actually the leading whitespace of the next lexeme). If the character is whitespace, the index to the start of the lexeme is incremented and the state doesn't change. Furthermore, you set `iAddCurChar` to `FALSE` because the lexeme itself should not contain its surrounding white-space. The `IsCharWhitespace ()` function is virtually identical to the one used in XASM, but of course, line breaks are now valid:

```
int IsCharWhitespace ( char cChar )
{
    // Return true if the character is a space or tab.
    if ( cChar == ' ' || cChar == '\t' || cChar == '\n' )
        return TRUE;
    else
        return FALSE;
}
```

Here's the `IsCharNumeric ()` function as well, just for reference:

```
int IsCharNumeric ( char cChar )
{
    // Return true if the character is between 0 and 9 inclusive.
    if ( cChar >= '0' && cChar <= '9' )
        return TRUE;
    else
        return FALSE;
}
```

After the check for whitespace, the state handler looks for a numeric digit. No matter what the lexeme turns out to ultimately be (integer or float), the occurrence of a digit in the start state is always interpreted as an integer lexeme, so the LEX_STATE_INT state is transitioned to. Of course, certain floating-point values can still be detected here, if they begin with a leading radix point, like .8 and .0123. If a radix point is found, the state transitions to LEX_STATE_FLOAT. Because the lexer currently only accepts integers and floats (as well as the whitespace between them), anything else is invalid and causes an error. The offending character is passed to ExitOnInvalidInputError (), and the program exits.

If the start state is not active, the machine then checks the integer state:

```
case LEX_STATE_INT:

    // If a numeric is read, keep the state as-is
    if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_INT;
    }

    // If a radix point is read, the numeric is really a float
    else if ( cCurrChar == '.' )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // If whitespace is read, the lexeme is done
    else if ( IsCharWhitespace ( cCurrChar ) )
    {
        iAddCurrChar = FALSE;
        iLexemeDone = TRUE;
    }

    // Anything else is invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

The first thing the state handler does is look for a valid numeric digit. If it finds one, the state can remain LEX_STATE_INT. For illustrative purposes, I've actually added code that explicitly assigns the state tracker the integer state, even though it's already set. This is obviously a bit redundant, but it helps readability. If the character isn't a digit, the handler determines whether it's a radix point. This isn't a valid integer character, but it indicates a state transition should be made to LEX_STATE_FLOAT. This should be a good indication of the elegance of the state machine approach—with only a few lines of code, you've got a lexer capable of seamlessly transitioning from the interpretation of an integer to that of a floating-point value. The next character comparison is against whitespace, because the occurrence of such characters marks the end of the lexeme. If this is the case, iLexemeDone is set to TRUE to break the loop. iAddCurrChar is also set to FALSE, because you don't want this extra whitespace character appended to the otherwise purely numeric lexeme. Any other character is invalid and is flagged as erroneous. This process is illustrated in the state diagram in Figure 13.9. Note that I use the * (asterisk) symbol to represent any character that isn't included in the other edges.

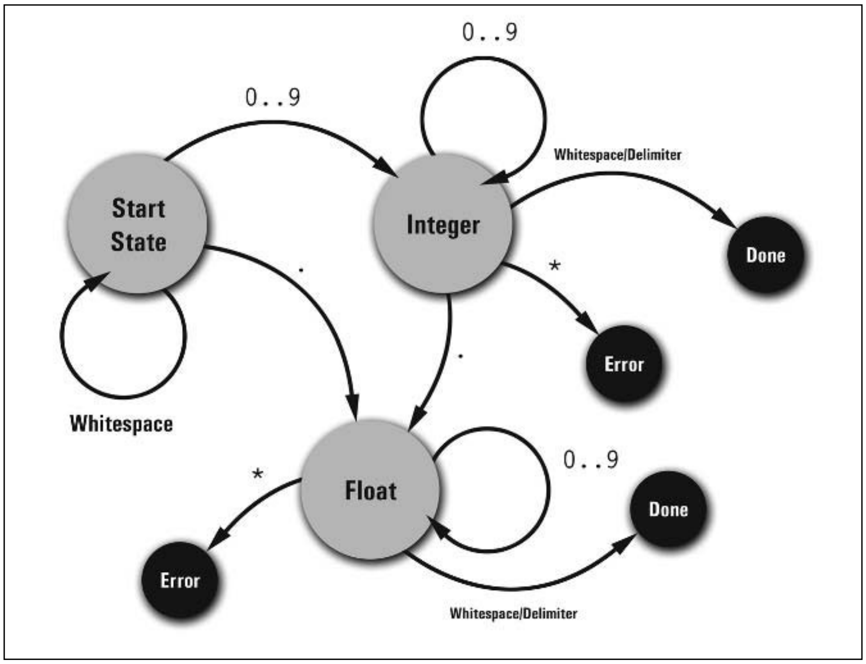


Figure 13.9

The numeric lexing state machine.

The only state left to check is LEX_STATE_FLOAT:

```
case LEX_STATE_FLOAT:

    // If a numeric is read, keep the state as-is
    if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // If whitespace is read, the lexeme is done
    else if ( IsCharWhitespace ( cCurrChar ) )
    {
        iLexemeDone = TRUE;
        iAddCurrChar = FALSE;
    }

    // Anything else is invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

This state is even simpler than the integer state. Any valid integer digit is added to the lexeme buffer, whitespace terminates the lexeme, and anything else is invalid. Once again, this demonstrates how a state machine's simplicity goes hand in hand with its power—although the last lexer had to perform convoluted string analysis on the lexeme to determine whether it was a float, it's all done implicitly with the new lexer. For example, there's no need to manually make sure the users inputted only one radix point per float value. The first instance of the point will simply transition the integer state to a float (or directly from the start state to the float), whereas any further encounters will be automatically sent to the error-handling function by the `LEX_STATE_FLOAT` state handler.

This finishes up the states, so the last order of business is rounding out the loop:

```
    // Add the next character to the lexeme and increment the index
    if ( iAddCurrChar )
    {
        g_pstrCurrLexeme [ iNextLexemeCharIndex ] = cCurrChar;
        ++ iNextLexemeCharIndex;
    }

    // If the lexeme is complete, exit the loop
    if ( iLexemeDone )
        break;
}
```

All you're doing here is appending the current character to the lexeme buffer, assuming the current state didn't suppress it, and ending the loop if the lexeme has been flagged as complete. Once the loop ends, there's a tiny bit of extra housekeeping to do as well:

```
// Complete the lexeme string
g_pstrCurrLexeme [ iNextLexemeCharIndex ] = '\0';

// Retract the lexeme end index by one
-- g_iCurrLexemeEnd;
```

Of course, it's all quite simple. A null terminator is slapped onto the end of the lexeme so it can be treated like a valid string, and the index that points to the end of the lexeme is retracted by one. Remember, whichever character ultimately ends the lexing process is actually part of the next lexeme. Because you don't want to skip over this character when the next lexeme is being processed, you need to back the index up by one.

All that's left to do in `GetNextToken ()` is map the terminating lexing state to a specific token type:

```
Token TokenType;
switch ( iCurrLexState )
{
    // Integer
    case LEX_STATE_INT:
        TokenType = TOKEN_TYPE_INT;
        break;

    // Float
    case LEX_STATE_FLOAT:
        TokenType = TOKEN_TYPE_FLOAT;
        break;

    // All that's left is whitespace, which means the end of the stream
    default:
        TokenType = TOKEN_TYPE_END_OF_STREAM;
}

// Return the token type
return TokenType;
```

A Token variable is declared, and a switch is used to determine which state the lexer was in when it finished. It's pretty self-explanatory. If it ended in LEX_STATE_INT, the token type is TOKEN_TYPE_INT. If it ended in LEX_STATE_FLOAT, the token type is TOKEN_TYPE_FLOAT. If anything else was returned, it must be a pure whitespace string (because if it wasn't pure whitespace, it'd either already have been identified as a numeric or be invalid). The only time whitespace can exist on its own without being stripped is when it trails the *last* lexeme in the file. You can therefore use this as a flag that the stream has ended, and return TOKEN_TYPE_END_OF_STREAM.

That wraps up GetNextToken (). Remember, once this function has been called, the lexeme is available in the global g_pstrCurrLexeme string buffer, a pointer to which can be received from GetCurrLexeme ():

```
char * GetCurrLexeme ()
{
    return g_pstrCurrLexeme;
}
```

Completing the Demo

To wrap things up, let's flesh out the code for displaying the results of the lexer's work. This will be done in a simple loop that calls GetNextToken () to get the next token and lexeme, checks for the end of the token stream, and prints out a reasonably verbose description of what was read. It finishes by printing the total number of tokens found. Here's the code:

```
while ( TRUE )
{
    // Get the next token
    CurrToken = GetNextToken ();

    // Make sure the token stream hasn't ended
    if ( CurrToken == TOKEN_TYPE_END_OF_STREAM )
        break;

    // Convert the token code to a descriptive string
    switch ( CurrToken )
    {
        // Integer
        case TOKEN_TYPE_INT:
            strcpy ( pstrToken, "Integer" );
            break;
        // Float
        case TOKEN_TYPE_FLOAT:
            strcpy ( pstrToken, "Float" );
            break;
    }

    // Print the token and the lexeme
    printf ( "%d: Token: %s, Lexeme: \"%s\"\n",
        iTokenCount, pstrToken,
        GetCurrLexeme () );

    // Increment the token count
    ++ iTokenCount;
}

// Print the token count
printf ( "\n" );
printf ( "\tToken count: %d\n", iTokenCount );
```

The token is used to fill the `pstrToken` string with a small description of the lexeme. In the case of the simple numeric lexer, it'll either say `"Integer"` or `"Float"`. The token string and lexeme are then written out, and the token count is incremented. Here's the demo's output when fed the source file I listed earlier:

```
Lexical Analyzer Demo

0: Token: Integer, Lexeme: "293048"
1: Token: Integer, Lexeme: "24"
2: Token: Integer, Lexeme: "895523"
3: Token: Float, Lexeme: "3.14159"
4: Token: Integer, Lexeme: "235"
5: Token: Integer, Lexeme: "253"
6: Token: Integer, Lexeme: "52435"
7: Token: Integer, Lexeme: "345"
8: Token: Integer, Lexeme: "459245"
9: Token: Integer, Lexeme: "22"
10: Token: Float, Lexeme: ".5"
11: Token: Float, Lexeme: ".35"
12: Token: Float, Lexeme: "2.0"
13: Token: Integer, Lexeme: "1"
14: Token: Float, Lexeme: "0.0"
```

```
15: Token: Float, Lexeme: "1.0"
16: Token: Integer, Lexeme: "0"
17: Token: Integer, Lexeme: "02345"
18: Token: Integer, Lexeme: "63246"
19: Token: Float, Lexeme: "0.2346"
20: Token: Float, Lexeme: "34.0"

        Token count: 21
```

Cool, huh? Using state machines, you've lexed a highly free-form source file containing a number of different numeric values. The whitespace was gracefully handled, and state transitions allowed the two different numeric formats to be interpreted easily and robustly. More importantly, you've taken a large step towards completing the actual lexer you'll use when building the XtremeScript compiler.

Let's move on by adding new lexeme and token types.

Lexing Identifiers and Reserved Words

The next step is adding identifiers, such as function and variable names, and the XtremeScript reserved word set. With these two additions, you'll have taken your next major step towards implementing the entire XtremeScript lexer.

One interesting point worth noting is that reserved words and identifiers are implemented the same way from the perspective of the lexer. After all, what's an identifier composed of? Alphanumeric digits and underscores. What's a reserved word composed of? The same thing. So, the strategy here is a bit unorthodox when compared to the pure state machine lexing of the last demo. You'll use the machine to lex identifiers only, and then compare the string it produced to a list of reserved words to find out what it really is.

This is where the difference between state machine-based lexers written by hand and those generated by utilities becomes more visible. In order to recognize the reserved words specified in the description of the language, the lexer machine would literally need *hundreds* of new states, because each letter in each word is technically a unique state. Furthermore, because each reserved word in the language stems from the same alphabet, it's entirely possible that the first few letters of one reserved word can actually transition to an entirely different word if the right letter is read, introducing countless additional state transitions from one word to another. Managing that many permutations is not something the human mind was cut out for, so you can take the easier way out here. As an example, however, consider this subset of the reserved words of the Pascal language:

```
AND
ARRAY
DO
DOWNTO
RECORD
REPEAT
```

Because each letter of each of these words is a different state, you can imagine how many transitions are represented here. Right off the bat, `AND` and `ARRAY` both start with `A`. So, when `A` is read, its state has to recognize transitions initiated by both `N` and `R`. `DO` and `DOWNTO` are even worse, because they share two initial letters; the `O` state in `DO` needs to know that it can either represent the last letter of one reserved word, or the second of another. Lastly, `RECORD` and `REPEAT` are the most complex, because both of their `E` states must be ready both for `C` and `P`, possibly allowing them to either stay in their current word or switch to the other.

In short, it's extremely tedious and difficult to hand-write such a state machine-based lexer, and the resulting code would be a nightmare to read and maintain even if it worked beautifully. Lexer generators don't have to worry about this, because any modification you want to make can be done in the much more readable description file and used to generate a new version. Humans are much better off performing a small number of comparisons after the loop terminates.

Figure 13.10 presents a state diagram for lexing identifiers and reserved words.

New States and Tokens

The first addition that must be made to the existing lexer is more states and tokens to represent the new forms of input it will accept. First up is the new lexer state:

```
#define LEX_STATE_IDENT                  5
```

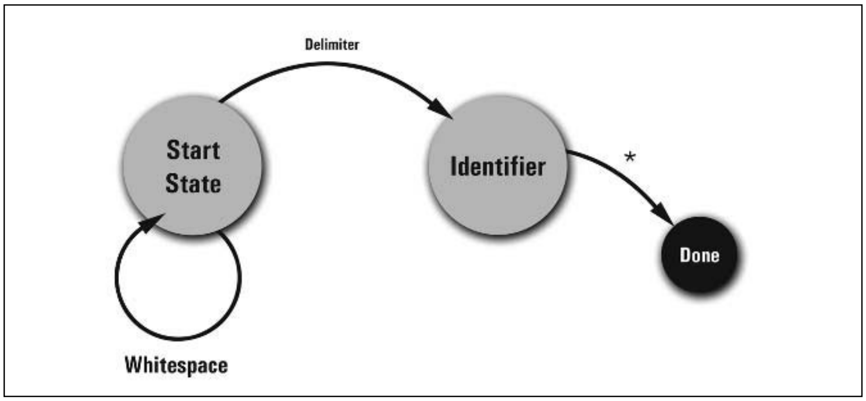


Figure 13.10

Lexing identifiers and reserved words.

That's right, just one state needed. From start to finish, every character of an identifier is classified the same way (an alphanumeric digit or underscore), so state transitions aren't necessary. Furthermore, because reserved words are treated as identifiers until after the lexing phase, they don't need separate states. Next are the new tokens:

```
#define TOKEN_TYPE_IDENT                3
#define TOKEN_TYPE_RSRVD_VAR            4
#define TOKEN_TYPE_RSRVD_TRUE           5
#define TOKEN_TYPE_RSRVD_FALSE          6
#define TOKEN_TYPE_RSRVD_IF             7
#define TOKEN_TYPE_RSRVD_ELSE           8
#define TOKEN_TYPE_RSRVD_BREAK          9
#define TOKEN_TYPE_RSRVD_CONTINUE       10
#define TOKEN_TYPE_RSRVD_FOR            11
#define TOKEN_TYPE_RSRVD_WHILE          12
#define TOKEN_TYPE_RSRVD_FUNC           13
#define TOKEN_TYPE_RSRVD_RETURN         14
```

Notice I've defined a separate token for each reserved word in the language. You could create a single token called `TOKEN_TYPE_RSRVD`, for example, that represents all words in the language. A separate function could then be called, much like `GetCurrLexeme ()` that provides the rest of the information—in this case, it might be called `GetCurrRsrvdWord ()` and return a constant that maps to a specific word.

Assigning a separate token to each word, however, makes things easier on the parser; it's a lot easier to determine whether `TOKEN_TYPE_RSRVD_FOR` was found when parsing a loop, than it is to call two functions to do the same thing.

The Test File

To test the new lexer, I've added identifiers and reserved words to the previous source file. Here it is:

```
293048 24 895523
3.14159
235
        253
          52435 345

MyVar0 MyVar1 MyVar2
           459245
```

```
    rEtUrN

TRUE false

 22 .5 .35 2.0

while

1
 0.0 var
  1.0 var
   0

       This_is_an_identifier

    02345

        _so_is_this___

    63246 0.2346
    34.0
```

Upgrading the Lexer

Adding identifier and reserved word support to the lexer is actually quite simple. All that you really need to do is look for valid identifier characters in the start state, use them to transition to an identifier state, and keep reading them in until the lexeme is terminated by whitespace. The resulting lexeme is either an identifier or a reserved word, a determination that's made outside of the state machine loop.

To determine whether a character can be part of a valid identifier, a new function has been created called `IsCharIdent ()`, and is identical to the one used in XASM. Here it is anyway, just for reference:

```
int IsCharIdent ( char cChar )
{
    // Return true if the character is between 0 or 9 inclusive
    // or is an uppercase or lowercase letter or underscore
    if ( ( cChar >= '0' && cChar <= '9' ) ||
         ( cChar >= 'A' && cChar <= 'Z' ) ||
         ( cChar >= 'a' && cChar <= 'z' ) ||
         cChar == '_' )
        return TRUE;
    else
        return FALSE;
    }
```

Armed with this function, adding identifier support to the lexer will be a snap. The first thing to do is add a check for identifier characters to the start state:

```
case LEX_STATE_START:

    // Just loop past whitespace, and don't add it to the lexeme
    if ( IsCharWhitespace ( cCurrChar ) )
    {
        ++ g_iCurrLexemeStart;
        iAddCurrChar = FALSE;
    }

    // An integer is starting
    else if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_INT;
    }

    // A float is starting
    else if ( cCurrChar == '.' )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // An identifier is starting
    else if ( IsCharIdent ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_IDENT;
    }

    // It's invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

Observant readers may have noticed, however, that making a call to `IsCharIdent ()` in the start state isn't technically correct, because it accepts characters `0-9`, even though identifiers can't start with numbers. Fortunately, if you notice the order in which the start state evaluates the input character, it checks for digits first. This effectively weeds out any possibilities of identifiers starting with numbers; rather, the lexer will simply flag the nonnumeric as an invalid integer character.

Now that you can initiate the `LEX_STATE_IDENT` state, you need to handle it so the next iteration through the loop has somewhere to go. Here's the identifier state handler:

```
case LEX_STATE_IDENT:

    // If an identifier character is read, keep the state as-is
    if ( IsCharIdent ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_IDENT;
    }

    // If whitespace is read, the lexeme is done
    else if ( IsCharWhitespace ( cCurrChar ) )
    {
        iAddCurrChar = FALSE;
        iLexemeDone = TRUE;
    }

    // Anything else is invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

This state handler follows the pattern that originated in the last lexer—it accepts any character that's within its own domain, (if it's an identifier character, the state remains `LEX_STATE_IDENT`) terminates when it encounters whitespace, and reports an error when it reads anything else. The real changes to `GetNextToken ()` come after the state machine loop completes. At this point, you think you have an identifier, but you may actually have a reserved word. To resolve this situation, you need to compare the lexeme produced by the machine to every reserved word in the XtremeScript language. Although there are a number of ways to go about doing this, I decided to keep things simple and just make a number of comparisons with `strcpy ()`:

```
Token TokenType;
switch ( iCurrLexState )
{
    // Integer
    case LEX_STATE_INT:
        TokenType = TOKEN_TYPE_INT;
        break;

    // Float
    case LEX_STATE_FLOAT:
        TokenType = TOKEN_TYPE_FLOAT;
        break;

    // Identifier/Reserved Word
    case LEX_STATE_IDENT:

        // Set the token type to identifier in case none
        // of the reserved words match
        TokenType = TOKEN_TYPE_IDENT;

        // ---- Determine if the "identifier" is actually a reserved word

        // var/var []
        if ( stricmp ( g_pstrCurrLexeme, "var" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_VAR;

        // true
        if ( stricmp ( g_pstrCurrLexeme, "true" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_TRUE;

        // false
        if ( stricmp ( g_pstrCurrLexeme, "false" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_FALSE;

        // if
        if ( stricmp ( g_pstrCurrLexeme, "if" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_IF;

        // else
        if ( stricmp ( g_pstrCurrLexeme, "else" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_ELSE;
        // break
        if ( stricmp ( g_pstrCurrLexeme, "break" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_BREAK;

        // continue
        if ( stricmp ( g_pstrCurrLexeme, "continue" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_CONTINUE;

        // for
        if ( stricmp ( g_pstrCurrLexeme, "for" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_FOR;

        // while
        if ( stricmp ( g_pstrCurrLexeme, "while" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_WHILE;

        // func
        if ( stricmp ( g_pstrCurrLexeme, "func" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_FUNC;

        // return
        if ( stricmp ( g_pstrCurrLexeme, "return" ) == 0 )
            TokenType = TOKEN_TYPE_RSRVD_RETURN;

        break;

    // All that's left is whitespace, which means the end of the stream
    default:
        TokenType = TOKEN_TYPE_END_OF_STREAM;
}
```

The first thing it does is set the token type to TOKEN_TYPE_IDENT, which will only change if one of the reserved word comparisons below it matches. If not, the token type remains an identifier as it should. Otherwise, it's replaced with a specific token representing whichever reserved word was detected.

And that's it—the lexer is now capable of identifiers and reserved words. The only thing left to do is build a new demo around it.

Completing the Demo

To test the new lexer, let's add some code to the `main ()` function that prints out the lexer's results. As you can see, the additions are similar to those made to the end of `GetNextToken ()`—mostly just comparisons to determine which reserved word was found:

```
while ( TRUE )
{
    // Get the next token
    CurrToken = GetNextToken ();

    // Make sure the token stream hasn't ended
    if ( CurrToken == TOKEN_TYPE_END_OF_STREAM )
        break;

    // Convert the token code to a descriptive string
    switch ( CurrToken )
    {
        // Integer
        case TOKEN_TYPE_INT:
            strcpy ( pstrToken, "Integer" );
            break;

        // Float
        case TOKEN_TYPE_FLOAT:
            strcpy ( pstrToken, "Float" );
            break;

        // Identifier
        case TOKEN_TYPE_IDENT:
            strcpy ( pstrToken, "Identifier" );
            break;

        // Reserved words
        case TOKEN_TYPE_RSRVD_VAR:
            strcpy ( pstrToken, "var" );
            break;

        case TOKEN_TYPE_RSRVD_TRUE:
            strcpy ( pstrToken, "true" );
            break;
        case TOKEN_TYPE_RSRVD_FALSE:
            strcpy ( pstrToken, "false" );
            break;

        case TOKEN_TYPE_RSRVD_IF:
            strcpy ( pstrToken, "if" );
            break;

        case TOKEN_TYPE_RSRVD_ELSE:
            strcpy ( pstrToken, "else" );
            break;

        case TOKEN_TYPE_RSRVD_BREAK:
            strcpy ( pstrToken, "break" );
            break;

        case TOKEN_TYPE_RSRVD_CONTINUE:
            strcpy ( pstrToken, "continue" );
            break;

        case TOKEN_TYPE_RSRVD_FOR:
            strcpy ( pstrToken, "for" );
            break;

        case TOKEN_TYPE_RSRVD_WHILE:
            strcpy ( pstrToken, "while" );
            break;

        case TOKEN_TYPE_RSRVD_FUNC:
            strcpy ( pstrToken, "func" );
            break;

        case TOKEN_TYPE_RSRVD_RETURN:
            strcpy ( pstrToken, "return" );
            break;
    }

    // Print the token and the lexeme
    printf ( "%d: Token: %s, Lexeme: \"%s\"\n", iTokenCount, pstrToken,
        GetCurrLexeme () );
    // Increment the token count
    ++ iTokenCount;
}

// Print the token count
printf ( "\n" );
printf ( "\tToken count: %d\n", iTokenCount );
```

With this code in place, the source file listed previously will produce the following results:

```
Lexical Analyzer Demo

0: Token: Integer, Lexeme: "293048"
1: Token: Integer, Lexeme: "24"
2: Token: Integer, Lexeme: "895523"
3: Token: Float, Lexeme: "3.14159"
4: Token: Integer, Lexeme: "235"
5: Token: Integer, Lexeme: "253"
6: Token: Integer, Lexeme: "52435"
7: Token: Integer, Lexeme: "345"
8: Token: Identifier, Lexeme: "MyVar0"
9: Token: Identifier, Lexeme: "MyVar1"
10: Token: Identifier, Lexeme: "MyVar2"
11: Token: Integer, Lexeme: "459245"
12: Token: return, Lexeme: "rEtUrN"
13: Token: true, Lexeme: "TRUE"
14: Token: false, Lexeme: "false"
15: Token: Integer, Lexeme: "22"
16: Token: Float, Lexeme: ".5"
17: Token: Float, Lexeme: ".35"
18: Token: Float, Lexeme: "2.0"
19: Token: while, Lexeme: "while"
20: Token: Integer, Lexeme: "1"
21: Token: Float, Lexeme: "0.0"
22: Token: var, Lexeme: "var"
23: Token: Float, Lexeme: "1.0"
24: Token: var, Lexeme: "var"
25: Token: Integer, Lexeme: "0"
26: Token: Identifier, Lexeme: "This_is_an_identifier"
27: Token: Integer, Lexeme: "02345"
```

```
28: Token: Identifier, Lexeme: "_so_is_this___"
29: Token: Integer, Lexeme: "63246"
30: Token: Float, Lexeme: "0.2346"
31: Token: Float, Lexeme: "34.0"

        Token count: 32
```

How cool is that? It not only lexes the file, but also detects and prints the reserved word associated with each lexeme (if applicable). You're closely approaching a complete lexer that will be ready to form the basis of the XtremeScript compiler. So now, to finish things off, let's add what's missing—delimiter characters, like commas, parentheses and braces, operators, and string literals. Although many lexers also handle comments, you're going to stick to the technique used with XASM and actually take comments out of the source before passing it to the lexer.

The Final Lexer: Delimiters, Operators, and Strings

With two thirds of the lexer finished, all that remains are delimiters, operators, and strings. Of course, the phrase "all that remains" implies that what you have left is easy—in reality, operators specifically will present a great deal of complexity. Fortunately, delimiters and strings are pretty easy, so let's start with those.

What's so great about this lexer is that it really will be finished. With the exception of comments, which will be handled by another part of the compiler, this thing can accept entire scripts and convert them to lexeme and token streams. At the end of this chapter, I demonstrate this with a source file containing valid XtremeScript code.

I like to get the easy stuff out of the way, however, so let's start with delimiters. As you'll see, these are the easiest of the three additions.

Lexing Delimiters

The easy thing about delimiters is that every delimiter in the XtremeScript language is a single character. You can take advantage of this fact to minimize the amount of additional code the lexer will need to handle them. Figure 13.11 contains a state diagram for lexing delimiters.

New States and Tokens

Like identifiers, delimiters can be represented with a single lex state:

```
#define LEX_STATE_DELIM                 7
```

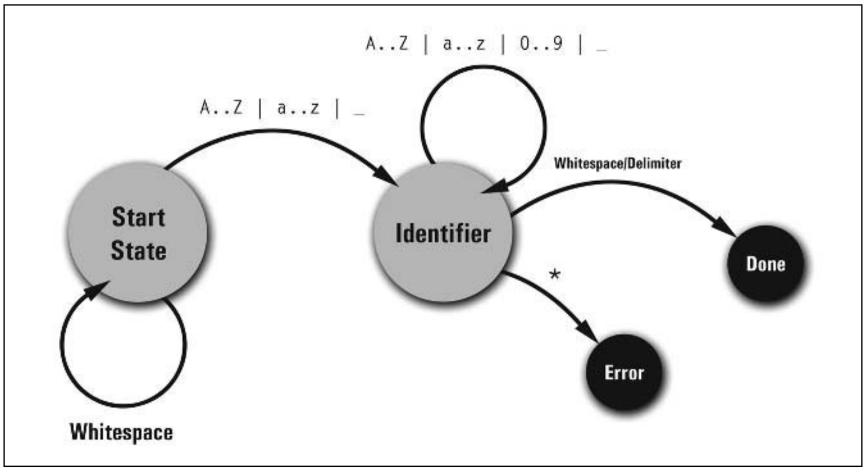


Figure 13.11

A delimiter-lexing state diagram.

Like reserved words, however, each delimiter gets its own token type.

```
#define TOKEN_TYPE_DELIM_COMMA              16
#define TOKEN_TYPE_DELIM_OPEN_PAREN         17
#define TOKEN_TYPE_DELIM_CLOSE_PAREN        18
#define TOKEN_TYPE_DELIM_OPEN_BRACE         19
#define TOKEN_TYPE_DELIM_CLOSE_BRACE        20
#define TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE   21
#define TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE  22
#define TOKEN_TYPE_DELIM_SEMICOLON          23
```

Which, again, makes things easier on the parser. This saves you from having to consult some other global variable or function to find out which specific delimiter was found if a `TOKEN_TYPE_DELIM` token is reported.

Upgrading the Lexer

To lex delimiters, the additions made to the lexer are rather simplistic. By adding an `IsCharDelim ()` function, you can easily add code to the start state that looks for delimiters. If it finds one, it transitions to `LEX_STATE_DELIM`. The state handler for delimiters is perhaps the simplest of all—it just terminates the lexeme. Because delimiters are always one character, the moment you enter the lexeme state you know you're at the first character of the next lexeme and can stop scanning.

The only minor complication is adding an `IsCharDelim ()` function. There's nothing complex about it, it's just that there are barely more than a handful of delimiters, which makes it a bit difficult to rig up a single `if` statement to do it all. So, you can dump them into a static array, like so:

```
#define MAX_DELIM_COUNT                 24
char cDelims [ MAX_DELIM_COUNT ] =

    { ',', '(', ')', '[', ']', '{', '}', ';' };
```

IsCharDelim () can now scan through this array to determine whether the specified character is a delimiter:

```
int IsCharDelim ( char cChar )
{
    // Loop through each delimiter in the array and compare
    // it to the specified character
    for ( int iCurrDelimIndex = 0; iCurrDelimIndex < MAX_DELIM_COUNT;
        ++ iCurrDelimIndex )
    {
        // Return TRUE if a match was found
        if ( cChar == cDelims [ iCurrDelimIndex ] )
            return TRUE;
    }

    // The character is not a delimiter, so return FALSE
    return FALSE;
}
```

Within GetNextToken (), the first change to make is adding the check for a delimiter in the start state:

```
case LEX_STATE_START:

    // Just loop past whitespace, and don't add it to the lexeme
    if ( IsCharWhitespace ( cCurrChar ) )
    {
        ++ g_iCurrLexemeStart;
        iAddCurrChar = FALSE;
    }

    // An integer is starting
    else if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_INT;
    }
    // A float is starting
    else if ( cCurrChar == '.' )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // An identifier is starting
    else if ( IsCharIdent ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_IDENT;
    }

    // A delimiter has been read
    else if ( IsCharDelim ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_DELIM;
    }

    // It's invalid
    else
        ExitOnInvalidInputError ( cCurrChar );
```

This is easy enough, but like I said, all the `LEX_STATE_DELIM` handler does is terminate the lexeme. Let's take look:

```
case LEX_STATE_DELIM:

    // Don't add whatever comes after the delimiter
    // to the lexeme, because it's done
    iAddCurrChar = FALSE;
    iLexemeDone = TRUE;
    break;
```

This wraps up the state machine, but once you're outside the loop you need to check the delimiter that was found and set the proper token type. You can do this automatically within the state machine, but that'd require a separate state for each delimiter, which would be pretty messy and redundant for a hand-written lexer. The following code is an addition to the `switch` block used to convert the final lex state into a token type:

```
case LEX_STATE_DELIM:

    // Determine which delimiter was found

    switch ( g_pstrCurrLexeme [ 0 ] )
    {
        case ',':
            TokenType = TOKEN_TYPE_DELIM_COMMA;
            break;

        case '(':
            TokenType = TOKEN_TYPE_DELIM_OPEN_PAREN;
            break;

        case ')':
            TokenType = TOKEN_TYPE_DELIM_CLOSE_PAREN;
            break;

        case '[':
            TokenType = TOKEN_TYPE_DELIM_OPEN_BRACE;
            break;

        case ']':
            TokenType = TOKEN_TYPE_DELIM_CLOSE_BRACE;
            break;

        case '{':
            TokenType = TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE;
            break;

        case '}':
            TokenType = TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE;
            break;

        case ';':
            TokenType = TOKEN_TYPE_DELIM_SEMICOLON;
            break;
    }

    break;
```

That's all it takes to add delimiters. As I said, it's a very easy addition. Strings are up next, which are incrementally more complex, but still nothing to worry about.

Lexing Strings

Strings represent a subtle departure from the types of lexemes you've been handling in the lexer so far. Integers, floating-point values, identifiers, reserved words, and delimiters are all implemented with a single state—the state is entered in the start state, and continues onwards until the lexeme is done. The only exceptions to this rule are integers and floats, because an integer can transition to a float during the lexing process.

Strings, however, are single entities that are composed of multiple states. The first state represents the opening quote, and only exists implicitly when it's detected by the start state. It then shifts over to a state that reads in the string body as a whole. Along the way, when an escape sequence is read, it switches again to a state that reads in escape sequence characters, and then immediately switches back. Finally, it ends with the closing quote state. If you think back to your development of the XASM lexer in Chapter 9, you'll remember the considerable complexity entailed by string support. You'll be pleasantly surprised to see that a state machine lexer can handle strings in a much more graceful, simplistic manner.

Figure 13.12 presents a state machine for lexing strings.

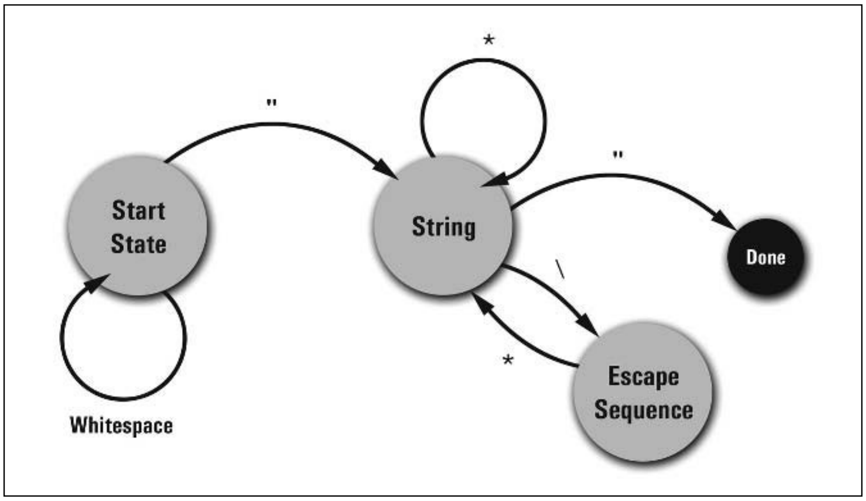


Figure 13.12

Lexing strings.

New States and Tokens

As I said, strings are the first entities that transition through multiple states before completing. Because of this, this will be the first time a lexeme has more lexer states than it does token types. Here are its states:

```
#define LEX_STATE_STRING                8
#define LEX_STATE_STRING_ESCAPE         9
#define LEX_STATE_STRING_CLOSE_QUOTE    10
```

Remember, the opening quote isn't represented by an explicit state. This is because once the quote is detected by the start state, it immediately transitions to `LEX_STATE_STRING`. Here's the new token type strings will be represented by:

```
#define TOKEN_TYPE_STRING               24
```

Upgrading the Lexer

The additions to the lexer's start state are almost as trivial as those made for delimiters. If a quote is read from the character stream, it's treated as a sign to transition to the `LEX_STATE_STRING` state. From here, the body of the string is read into the current lexeme. For this reason, there's no need for a `LEX_STATE_STRING_OPEN_QUOTE` state. Here's the code:

```
case LEX_STATE_START:

    // Just loop past whitespace, and don't add it to the lexeme
    if ( IsCharWhitespace ( cCurrChar ) )
    {
        ++ g_iCurrLexemeStart;
        iAddCurrChar = FALSE;
    }

    // An integer is starting
    else if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_INT;
    }

    // A float is starting
    else if ( cCurrChar == '.' )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // An identifier is starting
    else if ( IsCharIdent ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_IDENT;
    }
    // A delimiter has been read
    else if ( IsCharDelim ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_DELIM;
    }

    // A string is starting, but don't add the
    // opening quote to the lexeme
    else if ( cCurrChar == '"' )
    {
        iAddCurrChar = FALSE;
        iCurrLexState = LEX_STATE_STRING;
    }

    // It's invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

Remember, you have to set `iAddCurrChar` to false to make sure the opening quote isn't part of the final lexeme. Remember the hoops you had to jump through just to get the XASM lexer to avoid the opening quote? Now, you just clear a flag and its history.

The next state to worry about is `LEX_STATE_STRING`, which is directly transitioned to by the start state. This state just consumes each character it reads and dumps it into the lexeme. Whitespace, delimiters, you name it—it's all valid when a string is being lexed. The only characters that get this state's attention are the double quote, which of course terminates the string, and the escape sequence backslash. I'll talk more about escape sequences in a moment, so let's look at the code:

```
case LEX_STATE_STRING:

    // If the current character is a closing quote, finish the lexeme
    if ( cCurrChar == '"' )
    {
        iAddCurrChar = FALSE;
        iCurrLexState = LEX_STATE_STRING_CLOSE_QUOTE;
    }

    // If it's an escape sequence, switch to the escape
    // state and don't add the backslash to the lexeme
    else if ( cCurrChar == '\\' )
    {
        iAddCurrChar = FALSE;
        iCurrLexState = LEX_STATE_STRING_ESCAPE;
    }

    // Anything else gets added to the string

    break;
```

The cool thing about lexing a string is that you literally don't need to do anything—the way the state machine is set up, characters are added to the lexeme automatically, so by literally doing nothing, the string lexeme is populated.

One character of interest, however, is the double quote. When this character is read, you know the string is ending, and the program transitions to the LEX_STATE_STRING_CLOSE_QUOTE state:

```
case LEX_STATE_STRING_CLOSE_QUOTE:

    // Finish the string lexeme
    iAddCurrChar = FALSE;
    iLexemeDone = TRUE;

    break;
```

The primary job of this state is to terminate the lexeme, but it also has to make sure not to let the current character be printed, because it's the closing quote.

The only other detail about string lexing is the escape sequence. Escape sequences were another tricky part of the XASM lexer; you had to jump ahead two characters whenever a double-quote sign was read, the lexeme substring had to be copied in a special way, and overall it was a big mess. As you may have already assumed, however, the iAddCurrChar flag will make escape sequences almost criminally easy to support in the new lexer.

As you have seen, the LEX_STATE_STRING state transitions to the LEX_STATE_STRING_ESCAPE state whenever a backslash character is read (by the way, the \\ notation is used because even single characters recognize the backslash as an escape in C/C++). It also keeps the backslash from being printed, by setting iAddCurChar to FALSE as I mentioned. Let's look at the escape sequence state handler:

```
case LEX_STATE_STRING_ESCAPE:

    // Immediately switch back to the string state,
    // now that the character's been added
    iCurrLexState = LEX_STATE_STRING;

    break;
```

You know something's easy when the comment lines out-number the code. That's right, all the escape sequence state does is transition back to the normal string state. Remember, *all* states automatically append the current character to the lexeme unless they explicitly request otherwise, so all you have to do is let the current character be written (which is the character you want to include, like the " double quote), and switch back to the string.

With escape sequences nailed down, string support in the XtremeScript lexer is finished. That leaves you with the final, and most complex, hurdle—operators.

Operators

Operators are the last addition needed before you can call the XtremeScript lexer complete. Unfortunately, they're also the most difficult. The reason for their relative complexity is that they consist of multiple characters, and each character must be implemented as a separate and unique state. For example, consider the following operators:

```
<    <<    <<=
```

The first operator is the relational less-than operator. The second is a bitwise left shift, and the third is a bitwise left shift assignment shorthand. Each of these operators is built on the one before it, meaning they all share a number of states. Figure 13.13 contains a state machine capable of lexing these three operators.

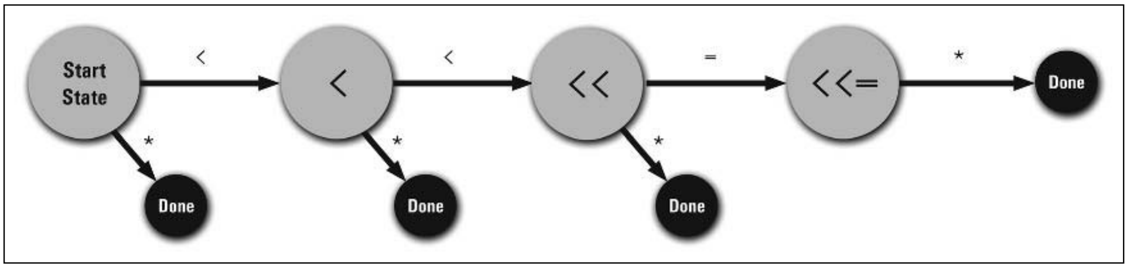


Figure 13.13

Lexing the <, <<, and <<= operators.

You could take the easy way out and simply create an array consisting of the union of all characters found in all operators, and create a single operator state that reads out strings of these characters in the state machine and compares them to predefined operator strings like "++", "*" and "!=" outside of the loop. This would work, but there'd be a lot of strings to compare, as XtremeScript has 34 operators. Besides, it wouldn't be nearly as much of a learning experience. :)

You could also apply brute force to the whole situation and spend a good six hours hard-coding each of the states a set of 34 operators would require. The amount of permutations and transitions between them would boil down to an astronomical number of separate states, but it'd work.

But you can do better than this. It sounds a bit strange to think of it this way, but the actual solution to the problem lies in a realization of how big it is. By understanding the sheer volume of the separate states involved in lexing operators, you can mentally switch gears and learn to apply a more iterative, generic solution.

To put this in other words, think back to when you first started programming. Like a lot of people, there was probably a point in your earlier days when you wanted to represent a large quantity of related data—perhaps for an address book program or something—but didn't know anything about arrays yet. You may have then proceeded to hard-code the declarations for each of the 20, or 30, or 200 items you wanted to represent, and found it extremely difficult to deal with.

Fortunately, you wouldn't have gotten very far with such an approach, and most likely would've given up quickly. It isn't long before a person in this position discovers arrays and other forms of aggregate data structures. Upon making such a discovery, you would've immediately realized how to solve the problem the right way. This is exactly the sort of revelation you need to make when approaching this problem.

Sure, the potentially hundreds of states and transitions can be hard-coded directly into the lexer—and an automatic lexer generation utility would probably do just this—but the key to these operators and the states they're composed of is that they're all strongly related and very similar. Like the names and numbers in an address book, aside from the actual operator character itself, every state in the set of XtremeScript operators would more or less do the same thing—it'd either add itself to the current lexeme or find a reason to switch to the state of another operator (like in the case of the three operators mentioned earlier).

The solution discussed in the following sections is somewhat tricky your first time through. Because of this, I ask that you read everything through before deciding you don't understand it. Furthermore, if you don't get it the first time, try reading it one or two more times—it should be no problem after a few passes.

Breaking Operators Down

So, what you need to do is break down the characters of the operators you're trying to lex and derive a better way to represent their states. The first important observation to make is that the transitions that can be made between states always happen from one character index to the next. By "character index," I mean the index of the character within the string that composes the operator. For example, the following operator has three character indexes:

```
>>=
```

These indexes are numbered 0-2: > is 0, > is 1, and = is 2. When lexing the following operators:

```
>      >>      >>=
```

The state transitions are sequential—the first character, >, transitions into the second, >. This then transitions into the third, =, and the process is complete. It's not possible for the first > to transition to =; in other words, there's no chance that when lexing the > operator, you may suddenly realize you're lexing >>=; you'd have to lex >> first. Check out Figure 13.14 to see this visually.

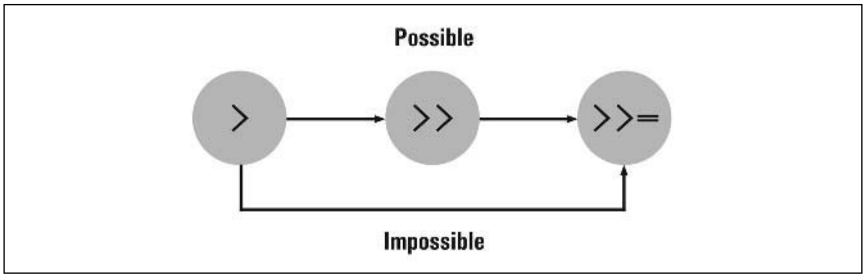


Figure 13.14

Operators that share subsets of each other transition gradually.

The point to all this is that you can first break down the problem based on these character indexes. Because no operator in XtremeScript has more than three characters, you can initially break the new states down to three groups. For example, consider the following subset of operators:

```
+     -     +=     ++     -=     --     <     >     <<     >>     <<=     >>=
```

These twelve operators range from one to three characters in length. Furthermore, there are a number of transitions between these operators, as the current character can either be an operator unto itself, part of a larger operator, or part of a different operator than is currently being lexed. If the + is read, however, there are a number of possibilities to consider:

- It's the binary add operator, and + is the first and last character.
- It's the binary add/assignment operator, and + is the first character of the += string.
- It's the unary increment operator, and + is the first character of the ++ string.

From this list, you can draw a number of conclusions. Before mentioning them, however, let's look at these 12 operators in a slightly different way:

```
+     -     =     +     =     -     <     >     <     >     =     =
```

To understand what I've done here, compare these 12 characters to the 12 operators I've listed previously. Simply put, I've reduced each operator to the extra character it provides among all of the operators that can transition to it. For example, out of the +, ++, and += operators, the + operator is represented simply by the + character. ++, however, is *based* on the original +; it just adds another + to create ++. Therefore, the extra character it adds is +. += also builds on the additional +, so its extra character is =. Figure 13.15 demonstrates this graphically.

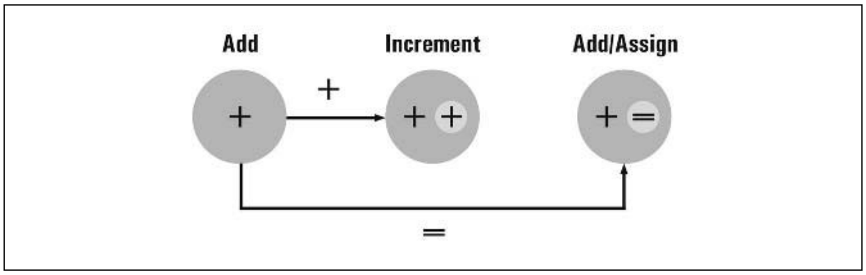


Figure 13.15

The "extra characters" added by each successive operator.

Now back to the conclusions. First of all, each of the 12 single characters has a number of properties. These properties can be used to determine how many states they're capable of transitioning to, as well as what those states are. For example, the + character, if it's the first character of the lexeme, is associated with three states. First, it can be its own state—the addition operator. It can also branch to two *substates* from here as well: ++ and +=. In the case of the ++ operator, the second + character can't branch to any other states and represents a terminal state that always marks the completion of the ++ operator. This is because there is no operator based on ++, like "++=" or something. In the case of the += operator, the = character has the same properties—because no operators are based on +=, it can't transition to any further substates. Lastly, each of these characters ultimately represents a unique operator. The first + represents +, the second + represents ++, and = represents +=. Check out Figure 13.16.

To help drive this point home, there are three separate characters to consider among these three operators: +, +, and = (even though the two +'s are both the same character, they have different properties and are therefore separate). Table 13.1 lists these characters and their relevant properties.

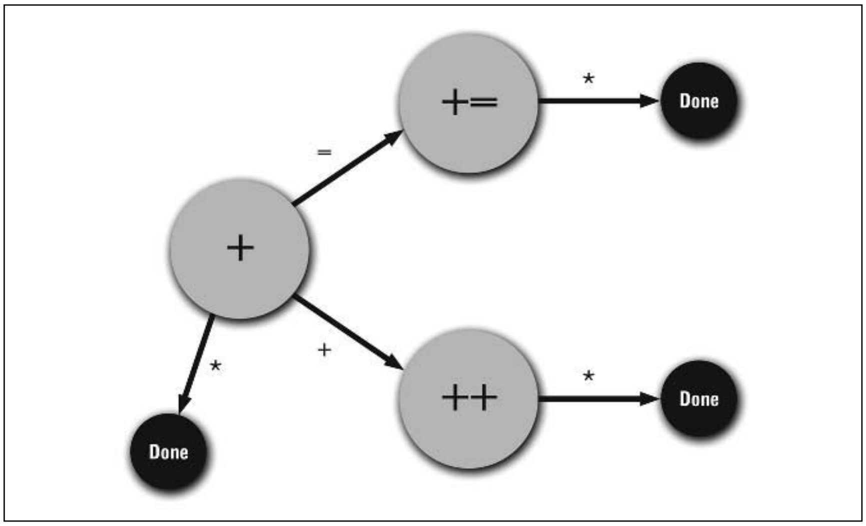


Figure 13.16

The state transitions and terminal states of the +, ++, and += operators.

Table 13.1 Character States

Character	Substate Count	Substates	Operator Represented
+	2	+, =	+
+	0	*None*	++
=	0	*None*	+=

Note that the *Substates* column doesn't list full operators; rather, it lists the characters that can immediately follow to invoke the transition to the substate. The first + row says that its substates are + and =, meaning that if either of these characters are read after the +, they'll invoke a transition to the ++ or += substates. Armed with this table, here's a simple breakdown of how these three operators could be lexed accurately:

- The first character of the new lexeme is read. It's a +.
- The second character is read. If it's any character other than the two substate transitions listed by the + character's properties, meaning any characters other than another + or =, you know that can't combine with the current + to form a valid operator and thus, the + operator is finished.
- If the character is another +, you find it in the first + character's properties, listed as a possible substate. You therefore transition from the + state to the ++ substate. The next character is then read, but you don't care what it is. Because the second + character's properties state that it has no substates, you therefore know the ++ operator can't be the basis for any further operators and must be complete.
- If the character is =, you follow the same process outlined in the last bullet point—it's a valid substate of +, which transitions to a += substate. Again, you don't care what the next character read is after this point, because the = character of the += operator has no substates, and must represent a completed lexeme.

You should now have a pretty good handle on the situation—there are initially three groups you can make, based on the characters at each of the three indexes an operator can occupy. Within these groups, you have a number of single characters, all of which correspond to the character of a specific operator at their index. Lastly, each of these characters has a number of properties that tell the lexer where to go, with regards to the current state, as it's read.

Characters in the first group—the index 0 group—represent both single character operators (such as ~ or bitwise not), as well as the first character of double- and triple-character operators, such as < and +. Characters in the second group—index 1—represent both the final characters of double-character operators, like the = in +=, but also represent the second character in triple-character operators, like the second < in <<=. Characters in the final group, index 2, only represent the final character of triple-character operators. Because there are no operators in XtremeScript with four or more characters, every member of this group must be a terminal character. Figure 13.17 provides a visual example of operands being assembled from these tables.

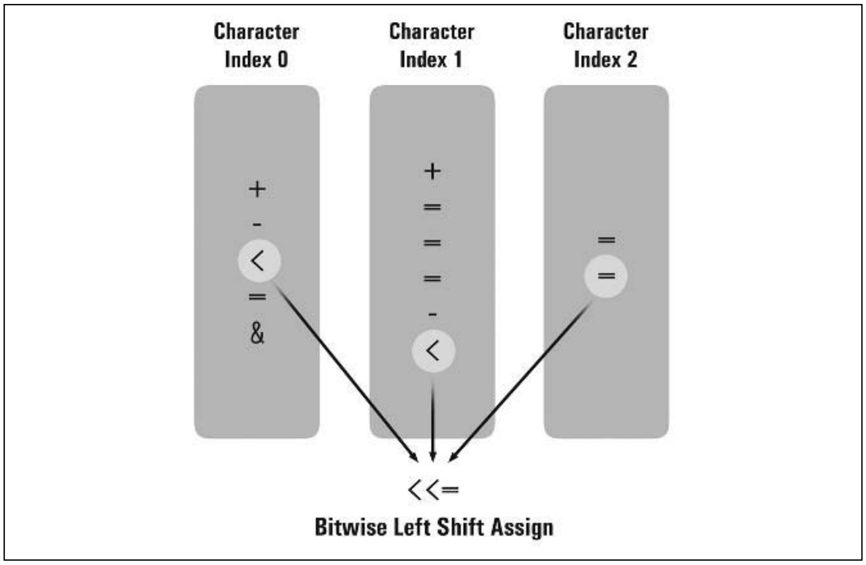


Figure 13.17

Assembling operands using three operand state tables.

Building Operator State Transition Tables

If you followed everything in that last section (which you may want to reread once or twice, because I know it's a bit tricky the first time through), you should be able to understand now how you'll represent the massive amounts of states required to properly lex 34 multi-character operators. Rather than hardcode anything, you'll build a number of tables to represent the states and transitions that each operator character is associated with. There will be three tables in total—one for each of the character index groups mentioned previously. Each member of each table will either represent the terminal character in an operator, or a character capable of transitioning to another operator (although most will be both).

To represent these characters, you need a structure capable of holding everything listed in Table 13.1. Fortunately, this is an easy conversion:

```
typedef struct _OpState          // Operator state
{
    char cChar;                  // State character
    int iSubStateIndex;          // Index into substate array where
                                  // sub states begin
    int iSubStateCount;           // Number of substates
    int iIndex;                   // Operator index
}
    OpState;
```

First and foremost, this structure holds the character to which the remaining properties apply in cChar. The next two members of the structure represent the character's substates. iSubStateCount is of course the number of states it can transition to. iSubStateIndex, simply put, is an index into the next state table (remember, there are three—one for each character index), where the substates begin. I'll cover this more in a second, so don't worry if you don't quite get what I mean. Lastly, iIndex is a special code that represents the operator this character would represent if it either has no substates, or if none of its substates are transitioned to. You'll see more of how this field works shortly.

The OpState structure represents a complete state by associating itself with a specific character, as well as a number of state transition properties. I'm now going to show you the code for declaring and initializing the operator state tables. Again, there will be three of these—one for each character index. Here they are:

```
// ---- First operator characters
OpState g_OpChars0 [ MAX_OP_STATE_COUNT ] = { { '+', 0, 2, 0 },
                                { '-', 2, 2, 1 },
                                { '*', 4, 1, 2 },
                                { '/', 5, 1, 3 },
                                { '%', 6, 1, 4 },
                                { '^', 7, 1, 5 },
                                { '&', 8, 2, 6 },
                                { '|', 10, 2, 7 },
                                { '#', 12, 1, 8 },
                                { '~', 0, 0, 9 },
                                { '!', 13, 1, 10 },
                                { '=', 14, 1, 11 },
                                { '<', 15, 2, 12 },
                                { '>', 17, 2, 13 } };
```

```
// ---- Second operator characters
OpState g_OpChars1 [ MAX_OP_STATE_COUNT ] = { { '=', 0, 0, 14 },
                                { '+', 0, 0, 15 },    // ++
                                { '=', 0, 0, 16 },    // -=
                                { '-', 0, 0, 17 },    // --
                                { '=', 0, 0, 18 },    // *=
                                { '=', 0, 0, 19 },    // /=
                                { '=', 0, 0, 20 },    // %=
                                { '=', 0, 0, 21 },    // ^=
                                { '=', 0, 0, 22 },    // &=
                                { '&', 0, 0, 23 },    // &&
                                { '=', 0, 0, 24 },    // |=
                                { '|', 0, 0, 25 },    // ||
                                { '=', 0, 0, 26 },    // #=
                                { '=', 0, 0, 27 },    // !=
                                { '=', 0, 0, 28 },    // ==
                                { '=', 0, 0, 29 },    // <=
                                { '<', 0, 1, 30 },    // <<
                                { '=', 0, 0, 31 },    // >=
                                { '>', 1, 1, 32 } };  // >>

// ---- Third operator characters
OpState g_OpChars2 [ MAX_OP_STATE_COUNT ] = { { '=', 0, 0, 33 },
                              { '=', 0, 0, 34 } };  // >>=
```

These arrays are dimensioned with a constant called `MAX_OP_STATE_COUNT`. This constant determines how many operator states each group can hold, which I have set for 32. I've used a nested `{}` notation to initialize both each element of the array, as well as each member of the array's structures. For example, in the case of the + element of the `g_OpChars0 []` array, you find this:

```
{ '+', 0, 2, 0 }
```

The first value, `'+'`, is of course the character itself. The second value, `0`, is the index into the second array at which its substates begin. The third value, `2`, is the number of substates it can transition to. In this case, because + can transition to both ++ and +=, there are two substates. The final value, `0`, is the index of the operator that this character would represent if it either had no substates, or none of its substates were transitioned to. Because the addition operator is the first one in the list, it's been assigned index 0. Of course, this is totally arbitrary—as long as it's unique, this index could be anything.

To help you understand this more clearly, let's revisit the previous example, but with direct assistance from these three arrays this time.

- The first character of the new lexeme is read in by the lexer, and it's a `<`. Because you haven't started lexing an operator yet, you're still at character zero. You therefore look for `<` in the `cChar` element of each `OpState` structure in the `g_OpChars0 []` array. It's found, so you know an operator is beginning. You set the current character index of the operator lexeme to 1, because you've already read the index zero character.
- This character could be the entire lexeme; if the next character read is not one of its substates, you know the lexeme is the `<` relational less-than operator. If this were the case, you'd look at the operator index within this character's `OpState` structure, which is 12 (go check it out in the array listing for yourself). Therefore, the relational less-than operator is represented by index 12. The lexeme would be complete, and you could return this information to the parser (or whoever called `GetNextToken ()`).
- The next character is read, and it's another `<`. In order to find out what this means, you need to consult the substate transition information stored in the first `<` character's `OpState` structure. It says that it has two substates, starting at index 15 in the `g_OpChars1 []` array. Therefore, the `OpState` structures at indexes 15 and 16 of this array contain the two possible substates of the `<` character. The first of these structures, the one at index 15, is for the = character, which would represent the `<=` operator. This doesn't match, however, so you check the next one, at index 16. This structure's `cChar` element is `<`, which matches the character you read. You now know to transition to this state, so you save the `OpState` structure and set the current character index to 2 (because we've now read in both 0 and 1). At this point, the lexeme is `<<`, which could be either the bitwise left shift operator, or the `<<=` bitwise left shift assignment operator. Its operator index is 30, though, so you know that `<<` is represented by this value. If the next character is not a valid substate, you can return this information to the caller.
- The next character is read, and it's =. You now consult the `< OpState` structure, and find that it can transition to one substate, starting at index 0 of the `g_OpChars2 []` array. You read out the `OpState` structure found there, and sure enough, its `cChar` element is =. You know the newly read character represents a transition to the `<<=` substate. You once again increment the character index to 3.
- The next character is read, and it's `M`. This could mean any number of things, but it doesn't matter because the `iSubStateCount` field of the = character's `OpState` structure is 0. This alerts you that the character has no substates, and is therefore the terminal character of the `<<=` operator. The operator index is 33, which corresponds to `<<=`. You're finished, so you can return this information to the caller.

Phew! There were quite a lot of details to get from point A to point B, but ultimately you lexed the `<<=` operator and paid close attention to all of the alternate paths it could've branched to. Along with the `g_OpChar* []` arrays, this logic is enough to transition through all 34 operator's states and substates and arrive at a solid conclusion.

New States and Tokens

So, with a firm grasp on the logic behind the state transition tables and the code that will utilize them, let's specify some new lexer states and tokens for `GetNextToken ()` to work with. Here's the new lexer state:

```
#define LEX_STATE_OP                        6
```

You need only one new state because all operators will be lexed in the same way, using the same state transition tables. That's why I call the operator states "substates"—they all take place within the larger, more general `LEX_STATE_OP` state. Here's the new token:

```
#define TOKEN_TYPE_OP                       15
```

I've chosen to use a single token to represent all operators because it keeps the token list a bit cleaner. A separate function called `GetCurrOp ()`, much like `GetCurrLexeme ()`, can be called after `GetNextToken ()` returns `TOKEN_TYPE_OP` to determine which specific operator was lexed. `GetCurrOp ()` will return one of the following constants:

```
// ---- Arithmetic
#define OP_TYPE_ADD                    0       // +
#define OP_TYPE_SUB                    1       // -
#define OP_TYPE_MUL                    2       // *
#define OP_TYPE_DIV                    3       // /
#define OP_TYPE_MOD                    4       // %
#define OP_TYPE_EXP                    5       // ^

#define OP_TYPE_INC                    15      // ++
#define OP_TYPE_DEC                    17      // --

#define OP_TYPE_ASSIGN_ADD             14      // +=
#define OP_TYPE_ASSIGN_SUB             16      // -=
#define OP_TYPE_ASSIGN_MUL             18      // *=
#define OP_TYPE_ASSIGN_DIV             19      // /=
#define OP_TYPE_ASSIGN_MOD             20      // %=
#define OP_TYPE_ASSIGN_EXP             21      // ^=

// ---- Bitwise
#define OP_TYPE_BITWISE_AND                 6            // &
#define OP_TYPE_BITWISE_OR                  7            // |
#define OP_TYPE_BITWISE_XOR                 8            // #
#define OP_TYPE_BITWISE_NOT                 9            // ~
#define OP_TYPE_BITWISE_SHIFT_LEFT          30           // <<
#define OP_TYPE_BITWISE_SHIFT_RIGHT         32           // >>
```

```
#define OP_TYPE_ASSIGN_AND              22          // &=
#define OP_TYPE_ASSIGN_OR               24          // |=
#define OP_TYPE_ASSIGN_XOR              26          // #=
#define OP_TYPE_ASSIGN_SHIFT_LEFT       33          // <<=
#define OP_TYPE_ASSIGN_SHIFT_RIGHT      34          // >>=

// ---- Logical
#define OP_TYPE_LOGICAL_AND             23          // &&
#define OP_TYPE_LOGICAL_OR              25          // ||
#define OP_TYPE_LOGICAL_NOT             10          // !

// ---- Relational
#define OP_TYPE_EQUAL                   28          // ==
#define OP_TYPE_NOT_EQUAL               27          // !=
#define OP_TYPE_LESS                    12          // <
#define OP_TYPE_GREATER                 13          // >
#define OP_TYPE_LESS_EQUAL              29          // <=
#define OP_TYPE_GREATER_EQUAL           31          // >=
```

Upgrading the Lexer

The last step is to apply this all to the state machine. The first stop is just before the state machine loop; the machine will need a few extra local variables for some internal bookkeeping:

```
int iCurrOpCharIndex = 0;
int iCurrOpStateIndex = 0;
OpState CurrOpState;
```

`iCurrOpCharIndex` keeps track of the current character index within the operator—this can be a value between 0 and 2, because XtremeScript operators have at most three characters. Of course, this is set to 0 by default. `iCurrOpStateIndex` stores the index of the current operator state *within* the `g_OpChars* []` array specified by `iCurrOpCharIndex`. Lastly, `CurrOpState` is a local instance of the `OpState` structure, and will contain the current operator state's information.

In addition, you'll also need a global variable to store the current operator index, as found in the `OpState` structure's `iIndex` field. After an operator is fully lexed, you'll arrive at a final index that will correspond to the operator. Because `GetNextToken ()` will only return `TOKEN_TYPE_OP`, you can use this global to store the specific operator index. The caller can then use `GetCurrOp ()` to retrieve this value. Here's the global:

```
int g_iCurrOp;
```

Here's `GetCurrOp ()`, which simply returns it:

```
int GetCurrOp ()
{
    return g_iCurrOp;
}
```

With these variables in place, you can start writing the state handlers. Here are the additions that need to be made to the start state (in bold, as usual):

```
case LEX_STATE_START:

    // Just loop past whitespace, and don't add it to the lexeme
    if ( IsCharWhitespace ( cCurrChar ) )
    {
        ++ g_iCurrLexemeStart;
        iAddCurrChar = FALSE;
    }

    // An integer is starting
    else if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_INT;
    }

    // A float is starting
    else if ( cCurrChar == '.' )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // An identifier is starting
    else if ( IsCharIdent ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_IDENT;
    }

    // A delimiter has been read
    else if ( IsCharDelim ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_DELIM;
    }
    // An operator is starting
    else if ( IsCharOpChar ( cCurrChar, 0 ) )
    {
        // Get the index of the initial operand state
        iCurrOpStateIndex = GetOpStateIndex ( cCurrChar, 0, 0, 0 );
        if ( iCurrOpStateIndex == -1 )
            ExitOnInvalidInputError ( cCurrChar );

        // Get the full state structure
        CurrOpState = GetOpState ( 0, iCurrOpStateIndex );

        // Move to the next character in the operator (1)
        iCurrOpCharIndex = 1;

        // Set the current operator
        g_iCurrOp = CurrOpState.iIndex;

        iCurrLexState = LEX_STATE_OP;
    }

    // A string is starting, but don't
    // add the opening quote to the lexeme
    else if ( cCurrChar == '"' )
    {
        iAddCurrChar = FALSE;
        iCurrLexState = LEX_STATE_STRING;
    }

    // It's invalid
    else
        ExitOnInvalidInputError ( cCurrChar );

    break;
```

As you can see, operators are the most complex addition to the start state. Actually determining whether an operator is starting is actually pretty easy, however—you just call `IsCharOpChar ()` to determine whether the character is a valid operator character. You don't want to check for just any operator character, however—you only want to know if it's a valid character within the first character index group, because at the start state you know you'd be dealing with the operator's first character. `IsCharOpChar ()` therefore accepts two parameters—the character you want to check, for which you pass cCurrChar, as well as the character index group to which the character may belong. For this, you pass zero.

Here's the code to IsCharOpChar ():

```
int IsCharOpChar ( char cChar, int iCharIndex )
{
    // Loop through each state in the specified character
    // index and look for a match
    for ( int iCurrOpStateIndex = 0; iCurrOpStateIndex
          < MAX_OP_STATE_COUNT;
          ++ iCurrOpStateIndex )
    {
       // Get the current state at the specified character index
       char cOpChar;
       switch ( iCharIndex )
       {
           case 0:
               cOpChar = g_OpChars0 [ iCurrOpStateIndex ].cChar;
               break;
           case 1:
               cOpChar = g_OpChars1 [ iCurrOpStateIndex ].cChar;
               break;
           case 2:
               cOpChar = g_OpChars2 [ iCurrOpStateIndex ].cChar;
               break;
       }

       // If the character is a match, return TRUE
       if ( cChar == cOpChar )
           return TRUE;
    }

    // Return FALSE if no match is found
    return FALSE;
}
```

This function scans through each OpState structure in each of the three g_OpChars* [] arrays. It then extracts the character from the desired array and compares it to the specified character. If they match, the character belongs to this group of operator states, and TRUE is returned. Otherwise, FALSE is returned.

Once the start state knows an operator character from the first character index has been found, it knows an operator is starting. It then calls `GetOpStateIndex ()` to find the index into the `g_OpChars0 []` array where the character's `OpState` structure resides (I'll explain what each of those zeroed parameters following `cCurrChar` mean in a moment.) You technically know this index exists, because it was already checked by `IsCharOpChar ()`, but I threw in some code to make sure the returned index wasn't `-1` anyway. You now know where within the array the `g_OpChars0 []` array your character's structure is, which you'll put to use in a second. First, here's the code for `GetOpStateIndex ()`:

```
int GetOpStateIndex ( char cChar,
                      int iCharIndex,
                      int iSubStateIndex,
                      int iSubStateCount )
{
    int iStartStateIndex;
    int iEndStateIndex;

    // Is the character index zero?
    if ( iCharIndex == 0 )
    {
        // Yes, so there are no substates to worry about
        iStartStateIndex = 0;
        iEndStateIndex = MAX_OP_STATE_COUNT;
    }
    else
    {
        //  No, so save the substate information
        iStartStateIndex = iSubStateIndex;
        iEndStateIndex = iStartStateIndex + iSubStateCount;
    }

    // Loop through each possible substate and look for a match
    for ( int iCurrOpStateIndex = iStartStateIndex;
        iCurrOpStateIndex < iEndStateIndex; ++ iCurrOpStateIndex )
    {
        // Get the current state at the specified character index
        char cOpChar;
        switch ( iCharIndex )
        {
            case 0:
                cOpChar = g_OpChars0 [ iCurrOpStateIndex ].cChar;
                break;
            case 1:
                cOpChar = g_OpChars1 [ iCurrOpStateIndex ].cChar;
                break;
            case 2:
                cOpChar = g_OpChars2 [ iCurrOpStateIndex ].cChar;
                break;
        }

        // If the character is a match, return the index
        if ( cChar == cOpChar )
            return iCurrOpStateIndex;
    }

    // Return -1 if no match is found
    return -1;
}
```

This function does almost the same thing `IsCharOpChar ()` does, except it returns the specific index rather than simply `TRUE` or `FALSE`. However, it does some extra stuff, which is why it needs those three parameters following `cChar`. Shortly, you'll also be using this function to search a character's substates. As you saw in the last section, a character's substates always occupy a contiguous region of one of the `g_OpChars* []`, so by passing this function the index to start searching from, as well as the number of substates to search, it will focus its scanning to that specific region. However, because the first character of an operator can be anything, and therefore isn't confined to a specific region, you pass all zeroes to tell the function to scan through everything in the `g_OpChars0 []` array. This is what the following code does:

```
int iStartStateIndex;
int iEndStateIndex;

// Is the character index zero?
if ( iCharIndex == 0 )
{
    // Yes, so there are no substates to worry about
    iStartStateIndex = 0;
    iEndStateIndex = MAX_OP_STATE_COUNT;
}
else
{
    //  No, so save the substate information
    iStartStateIndex = iSubStateIndex;
    iEndStateIndex = iStartStateIndex + iSubStateCount;
}
```

You then call `GetOpState ()` to use the index returned by `GetOpStateIndex ()` to retrieve the actual `OpState` structure associated with the character read. You pass it zero, along with this index, to tell it to return the structure found at the specified index within `g_OpChars0 []`, as opposed to the other two arrays. Here's `GetOpState ()`:

```
OpState GetOpState ( int iCharIndex, int iStateIndex )
{
    OpState State;

    // Save the specified state at the specified character index
    switch ( iCharIndex )
    {
        case 0:
            State = g_OpChars0 [ iStateIndex ];
            break;
        case 1:
            State = g_OpChars1 [ iStateIndex ];
            break;
        case 2:
            State = g_OpChars2 [ iStateIndex ];
            break;
    }

    return State;
}
```

You now have the operator substate structure, so the only thing left to do is set `iCurrOpCharIndex` (the current character index) to 1, `g_iCurrOp` to the index in the current `OpState` structure, and the lexer state to `LEX_STATE_OP`. Remember, you set `g_iCurrOp` now, just in case this happens to be the first and last character of the operator (as it would be in the case of single-character operators). If this turns out to be the case, you'll already have the operator's index saved globally, so `GetNextToken ()` can simply return `TOKEN_TYPE_OP` and rely on `GetCurrOp ()` to provide the caller with the rest of the information.

This takes care of the start state. After the next character is read, the machine will be in the `LEX_STATE_OP` state, so let's check out its handler:

```
case LEX_STATE_OP:

    // If the current character within the operator
    // has no substates, we're done
    if ( CurrOpState.iSubStateCount == 0 )
    {
        iAddCurrChar = FALSE;
        iLexemeDone = TRUE;
        break;
    }

    // Otherwise, find out if the new character is a possible substate
    if ( IsCharOpChar ( cCurrChar, iCurrOpCharIndex ) )
    {
        // Get the index of the next substate
        iCurrOpStateIndex = GetOpStateIndex (
           cCurrChar, iCurrOpCharIndex,
           CurrOpState.iSubStateIndex, CurrOpState.iSubStateCount );
        if ( iCurrOpStateIndex == -1 )
            ExitOnInvalidInputError ( cCurrChar );

        // Get the next operator structure
        CurrOpState = GetOpState ( iCurrOpCharIndex,
            iCurrOpStateIndex );

        // Move to the next character in the operator
        ++ iCurrOpCharIndex;

        // Set the current operator
        g_iCurrOp = CurrOpState.iIndex;
    }

    // If not, the lexeme is done
    else
    {
        iAddCurrChar = FALSE;
        iLexemeDone = TRUE;
    }

    break;
```

The first check this handler makes is for the possibility that the current operator state has no substates. In this case, no matter what the current character is, you know you're done. Next, it compares the current character to the current operator state's substates to determine whether the operator is being further developed. If so, you basically repeat the process from the start state—you use GetOpStateIndex () to get the index into the current g_OpChars* [] array (which you specify with iCurrOpCharIndex). You also make sure to pass it the CurrOpState.iSubStateIndex, as well as CurrOpState.iSubStateCount, so the function knows where in the array to focus its search. Once you get the next character's operator state index, you can use it to get its corresponding OpState structure with GetOpState (). You then increment the character index, and finish up by updating g_iCurrOp to represent whatever operator could potentially be finished by this character. If the current character doesn't match any of the operator state's substate transitions, the lexeme is finished.

Completing the Demo

This last section has been a significant one—you've added support for delimiters, strings and multi-character operators. Because of this, you cannot only lex more complex source files, you can actually lex complete XtremeScript scripts!

Before you can do any of this, you need to make one final change to the program's main () function so that it properly handles the most recently added forms of output from GetNextToken (). The following code is added to the switch block that fills the pstrToken string with a description of the current token code.

```
// Operators
case TOKEN_TYPE_OP:
    sprintf ( pstrToken, "Operator %d", GetCurrOp () );
    break;

// Delimiters
case TOKEN_TYPE_DELIM_COMMA:
    strcpy ( pstrToken, "Comma" );
    break;

case TOKEN_TYPE_DELIM_OPEN_PAREN:
    strcpy ( pstrToken, "Opening Parenthesis" );
    break;

case TOKEN_TYPE_DELIM_CLOSE_PAREN:
    strcpy ( pstrToken, "Closing Parenthesis" );
    break;
```

```
case TOKEN_TYPE_DELIM_OPEN_BRACE:
    strcpy ( pstrToken, "Opening Brace" );
    break;

case TOKEN_TYPE_DELIM_CLOSE_BRACE:
    strcpy ( pstrToken, "Closing Brace" );
    break;

case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
    strcpy ( pstrToken, "Opening Curly Brace" );
    break;

case TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE:
    strcpy ( pstrToken, "Closing Curly Brace" );
    break;

case TOKEN_TYPE_DELIM_SEMICOLON:
    strcpy ( pstrToken, "Semicolon" );
    break;

// Strings
case TOKEN_TYPE_STRING:
    strcpy ( pstrToken, "String" );
    break;
```

This completes the program that fully lexes the entire XtremeScript language. Let's make one more version of the source file you've been adding to throughout this chapter to test it out:

```
293048 24 895523
-3.14159
235
        253
          {} 52435 345 {}

[ MyVar0, MyVar1, MyVar2 ]
           459245;

    rEtUrN

TRUE, false, ();
```

```
3 ++ 2 ( 4 / 2 ) * 2

 -22 .5 -.35 2.0

> >> >>=

While

"Hello, world!"

1
 0.0 var
  1.0 var
   0

       This_is_an_identifier

    02345

        _so_is_this___

    if ( X < Y ) Z;

    63246 -0.2346
    34.0
```

When this file is passed through the final lexer, it produces the following results:

```
Lexical Analyzer Demo

0: Token: Integer, Lexeme: "293048"
1: Token: Integer, Lexeme: "24"
2: Token: Integer, Lexeme: "895523"
3: Token: Operator 1, Lexeme: "-"
4: Token: Float, Lexeme: "3.14159"
5: Token: Integer, Lexeme: "235"
6: Token: Integer, Lexeme: "253"
7: Token: Opening Curly Brace, Lexeme: "{"
8: Token: Closing Curly Brace, Lexeme: "}"
9: Token: Integer, Lexeme: "52435"
10: Token: Integer, Lexeme: "345"
```

```
11: Token: Opening Curly Brace, Lexeme: "{"
12: Token: Closing Curly Brace, Lexeme: "}"
13: Token: Opening Brace, Lexeme: "["
14: Token: Identifier, Lexeme: "MyVar0"
15: Token: Comma, Lexeme: ","
16: Token: Identifier, Lexeme: "MyVar1"
17: Token: Comma, Lexeme: ","
18: Token: Identifier, Lexeme: "MyVar2"
19: Token: Closing Brace, Lexeme: "]"
20: Token: Integer, Lexeme: "459245"
21: Token: Semicolon, Lexeme: ";"
22: Token: return, Lexeme: "rEtUrN"
23: Token: true, Lexeme: "TRUE"
24: Token: Comma, Lexeme: ","
25: Token: false, Lexeme: "false"
26: Token: Comma, Lexeme: ","
27: Token: Opening Parenthesis, Lexeme: "("
28: Token: Closing Parenthesis, Lexeme: ")"
29: Token: Semicolon, Lexeme: ";"
30: Token: Integer, Lexeme: "3"
31: Token: Operator 15, Lexeme: "++"
32: Token: Integer, Lexeme: "2"
33: Token: Opening Parenthesis, Lexeme: "("
34: Token: Integer, Lexeme: "4"
35: Token: Operator 3, Lexeme: "/"
36: Token: Integer, Lexeme: "2"
37: Token: Closing Parenthesis, Lexeme: ")"
38: Token: Operator 2, Lexeme: "*"
39: Token: Integer, Lexeme: "2"
40: Token: Operator 1, Lexeme: "-"
41: Token: Integer, Lexeme: "22"
42: Token: Float, Lexeme: ".5"
43: Token: Operator 1, Lexeme: "-"
44: Token: Float, Lexeme: ".35"
45: Token: Float, Lexeme: "2.0"
46: Token: Operator 13, Lexeme: ">"
47: Token: Operator 32, Lexeme: ">>"
48: Token: Operator 34, Lexeme: ">>="
49: Token: while, Lexeme: "While"
50: Token: String, Lexeme: "Hello, world!"
51: Token: Integer, Lexeme: "1"
```

```
52: Token: Float, Lexeme: "0.0"
53: Token: var, Lexeme: "var"
54: Token: Float, Lexeme: "1.0"
55: Token: var, Lexeme: "var"
56: Token: Integer, Lexeme: "0"
57: Token: Identifier, Lexeme: "This_is_an_identifier"
58: Token: Integer, Lexeme: "02345"
59: Token: Identifier, Lexeme: "_so_is_this___"
60: Token: if, Lexeme: "if"
61: Token: Opening Parenthesis, Lexeme: "("
62: Token: Identifier, Lexeme: "X"
63: Token: Operator 12, Lexeme: "<"
64: Token: Identifier, Lexeme: "Y"
65: Token: Closing Parenthesis, Lexeme: ")"
66: Token: Identifier, Lexeme: "Z"
67: Token: Semicolon, Lexeme: ";"
68: Token: Integer, Lexeme: "63246"
69: Token: Operator 1, Lexeme: "-"
70: Token: Float, Lexeme: "0.2346"
71: Token: Float, Lexeme: "34.0"

        Token count: 72
```

This is certainly nice, but to *really* test it, let's throw a full, basic script at it, written entirely in the XtremeScript language developed in Chapter 7:

```
func MyFunc ( Param0, Param1, Param2 )
{
    return ( Param0 + Param1 ) * Param2;
}

func main ()
{
    var MyString;
    var X;

    MyString = "This is a \"real\" XtremeScript script!";
    X = 256;

    MyFunc ( MyString, 3.14159, X );
}
```

Here are the results:

```
Lexical Analyzer Demo

0: Token: func, Lexeme: "func"
1: Token: Identifier, Lexeme: "MyFunc"
2: Token: Opening Parenthesis, Lexeme: "("
3: Token: Identifier, Lexeme: "Param0"
4: Token: Comma, Lexeme: ","
5: Token: Identifier, Lexeme: "Param1"
6: Token: Comma, Lexeme: ","
7: Token: Identifier, Lexeme: "Param2"
8: Token: Closing Parenthesis, Lexeme: ")"
9: Token: Opening Curly Brace, Lexeme: "{"
10: Token: return, Lexeme: "return"
11: Token: Opening Parenthesis, Lexeme: "("
12: Token: Identifier, Lexeme: "Param0"
13: Token: Operator 0, Lexeme: "+"
14: Token: Identifier, Lexeme: "Param1"
15: Token: Closing Parenthesis, Lexeme: ")"
16: Token: Operator 2, Lexeme: "*"
17: Token: Identifier, Lexeme: "Param2"
18: Token: Semicolon, Lexeme: ";"
19: Token: Closing Curly Brace, Lexeme: "}"
20: Token: func, Lexeme: "func"
21: Token: Identifier, Lexeme: "main"
22: Token: Opening Parenthesis, Lexeme: "("
23: Token: Closing Parenthesis, Lexeme: ")"
24: Token: Opening Curly Brace, Lexeme: "{"
25: Token: var, Lexeme: "var"
26: Token: Identifier, Lexeme: "MyString"
27: Token: Semicolon, Lexeme: ";"
28: Token: var, Lexeme: "var"
29: Token: Identifier, Lexeme: "X"
30: Token: Semicolon, Lexeme: ";"
31: Token: Identifier, Lexeme: "MyString"
32: Token: Operator 11, Lexeme: "="
33: Token: String, Lexeme: "This is a "real" XtremeScript script!"
34: Token: Semicolon, Lexeme: ";"
35: Token: Identifier, Lexeme: "X"
36: Token: Operator 11, Lexeme: "="
37: Token: Integer, Lexeme: "256"
38: Token: Semicolon, Lexeme: ";"
39: Token: Identifier, Lexeme: "MyFunc"
40: Token: Opening Parenthesis, Lexeme: "("
41: Token: Identifier, Lexeme: "MyString"
42: Token: Comma, Lexeme: ","
43: Token: Float, Lexeme: "3.14159"
44: Token: Comma, Lexeme: ","
45: Token: Identifier, Lexeme: "X"
46: Token: Closing Parenthesis, Lexeme: ")"
47: Token: Semicolon, Lexeme: ";"
48: Token: Closing Curly Brace, Lexeme: "}"

        Token count: 49
```

How cool is this? The lexer *completely* understands the language, which means you have a nearly finished foundation upon which to build the parser, and ultimately, the rest of the compiler.

Summary

With the exception of the operator lexing nightmare near the end, this has hopefully been a relatively straightforward chapter. The results were anything but trivial however—you now have a fully featured lexer for your language. You'll have to do a little bit of integrating to get it to work with the rest of the compiler, which you'll begin building in the next chapter, but the real work behind lexical analysis is now behind you. As you know by now, lexing is a very important phase in the pipeline of a basic compiler, so your accomplishments in this chapter are significant.

As usual, you're strongly encouraged to check out the source for the three lexer demos built in this chapter. I specifically aimed to cover virtually every line of code in all three demos in this chapter, which I did, but it still helps a lot to see everything put together in its final configuration.

On the CD

This chapter contains three programs—the three lexer demos you designed and implemented. These demos are found in the `Programs/Chapter 13/` directory. Within this directory you'll find three directories in which the specific lexers reside; `13_01/`, `13_02/`, and `13_03/`. As usual, the demos come in both source and executable form.

This chapter has been solely concerned with text processing, so everything is a simple console application and should compile and run very easily.

Each of the lexer executables accepts a command-line argument to specify which file to lex. Go ahead and write your own source files to test out its robustness.

Challenges

- *Easy:* Add some extra multi-character operators, and see whether you can properly insert them into the operator transition state tables. Remember to add them to the end of the tables so they don't disrupt the preexisting indexes.
- *Intermediate:* Currently, the delimiters are all one character and can thus be supported more easily than operators because there's no possibility of state transitions. However, many languages have multi-character delimiters. In order to support this in your own lexer, you'd need to implement a system similar to what you used for operators in order to handle such delimiters. Try adding such a system, using the existing operator code as a guide.
- *Difficult:* Add comments to the final lexer. Comments like `// ...` and `/* ... */` can be implemented using states, much like strings; each character within the comment syntax is a separate state, along with another state for the comment's body. This isn't as easy as it sounds, however. The problem is, both comments share the `/` character, which is also used for the division operator. The only way to resolve this issue is to implement a look-ahead character, much like the one used in XASM's parser, to determine whether another slash, or an asterisk, appears afterwards. This chapter's lexers didn't need a look-ahead simply because there were no such clashes among characters. As you can see, however, it's a vital feature in such cases.

CHAPTER 14

Building the XtremeScript Compiler Framework

"Telephone, computer, fax machine, fifty-two weekly paychecks and forty-eight flight coupons… we now had corporate sponsorship."
—*Jack,* Fight Club

With 13 chapters behind you, the moment has finally arrived. You're now ready to dive headlong into the real inner-workings of the XtremeScript Compiler—the high-level, human interface to our nearly complete scripting system.

Regardless of the reasonable complexity associated with both the virtual machine and assembler, no scripting system is really worth using without a high-level language to drive it in large projects. Although it's certainly not impossible, scripting an entire game in pure XVM assembly would be an exercise in tedium.

In this chapter, you're going to

- Plan the design and general architecture of the XtremeScript compiler.
- Integrate the lexer built in the last chapter with the compiler's framework.
- Discuss and create many of the compiler's major components, including the I-code module and code emitter.

The construction of the XtremeScript compiler will ultimately be a three-chapter process. The last chapter started with the design and implementation of a full-featured lexical analyzer module that's ready to be dropped into place. This chapter builds a solid foundation upon which to base the rest of the compiler, as well as the lexer, by organizing and encapsulating the compiler's major structures and modules. The final chapter dealing with the compiler, which is up next, focuses entirely on parsing the XtremeScript code and converting it to an intermediate format that the code emitter can output in the form of XVM assembly. Although you won't actually process any high-level code in this chapter directly, you will be able to hard-code some values into the compiler and use them as test data for generating real .XSE executables.

A Strategic Overview

As is the case with all large and complex software projects, you must be careful to ensure that the data and code is encapsulated in a clean and logical manner. Chaos is the result of bad organization, and because the implementation of a high-level compiler is an uphill battle to begin with, you don't want to make things any harder than they already are by being messy.

Fortunately, you don't have to go *too* far. The project certainly isn't so big that it necessarily demands the use of OOP, so using nearly pure C will still be fine (although as always, I'll be using many of C++'s syntactic conveniences). Furthermore, although strongly designed and enforced interfaces are generally a good thing, you don't need to follow this rule too strictly. There might still be a handful of globals floating around, or other such "cheating," but the final result will be more than clean enough for the purposes here.

As I've demonstrated frequently throughout the book so far, compilers are generally built as two separate "ends," separated by what is known as an *I-code module*. I-code is a way to represent a program's source code in a way that's independent of any source or target language, allowing the compiler to be retargeted or supportive of multiple high-level languages. The compiler will loosely follow this format, so let's talk about these major components in more detail. The concept of I-code separating the front and back ends is illustrated in Figure 14.1.

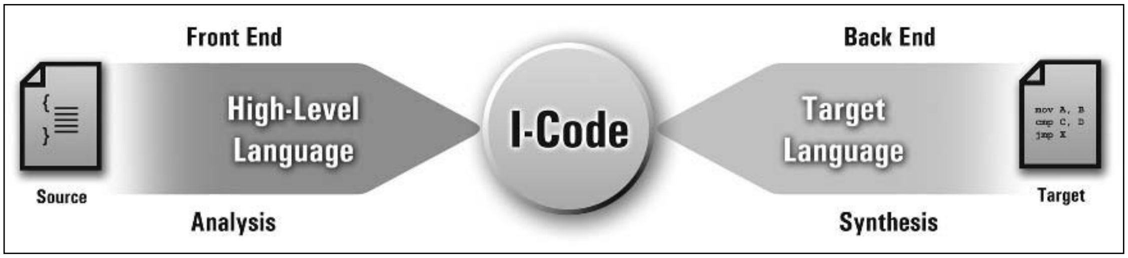


Figure 14.1

The I-code module is used to separate the front and back ends.

The Front End

The front end of the compiler will be responsible for loading the source code, preprocessing it, lexing it into a stream of tokens and lexemes, and parsing it into an equivalent I-code representation. By the time the front end is done with its job, you will have stripped away all traces of human interaction, and have a structured, validated, in-memory version of the source code that can be easily translated to XVM assembly.

The front end will be by far the most complex aspect of the compiler, so let's break it into its constituent modules. A graphical overview of the front end is illustrated in Figure 14.2.

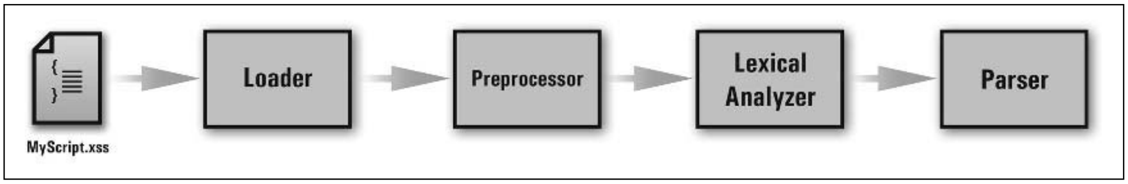


Figure 14.2

The modules of the front end.

The Loader Module

The loader module is responsible for initially loading the source code from an .XSS (XtremeScript Source) file into memory. Although this may seem like a trivial job at first, there are still some important details to consider.

Storing the Source Code

Unlike the simplified examples of the lexer built in the last chapter, the XtremeScript compiler will not store the entire source file in a single string. Although this does have some advantages (because it's certainly easier to lex a contiguous stream of characters than some other, more complex data structure), you are better off with a linked list, wherein each node stores a single source line, for a number of reasons. For example, having each line in a separate node allows you to track both the current line's string and number, allowing you to produce verbose error messages that highlight the exact problem (as you did in XASM). Check out Figure 14.3.

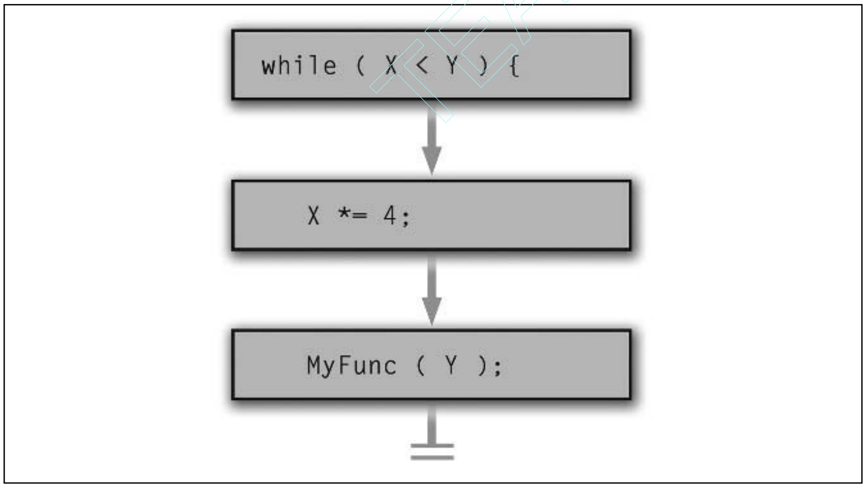


Figure 14.3

Storing source code in a linked list, wherein each node represents a separate line.

Internalization of the Source Format

Although virtually every plain text file in the world is stored in the ASCII format, the specific method for denoting line breaks often changes significantly from one platform to the next. To make things easier to manage internally, and to aid in portability, the loader will be responsible for ensuring that the in-memory version of the source code uses a consistent representation for line breaks and newlines.

The Preprocessor Module

Once the loader has populated the compiler's internal source code linked list, you're almost ready to pass things to the lexer and parser so the compilation process can begin. Before doing so, however, you have the opportunity to filter and convert the source code to a more convenient a format via the preprocessor. By inserting a preprocessor module in between the loader and the lexical analyzer, you can perform any sort of preprocessing operation you want transparently, as shown originally in Figure 14.1 and more closely in Figure 14.4.

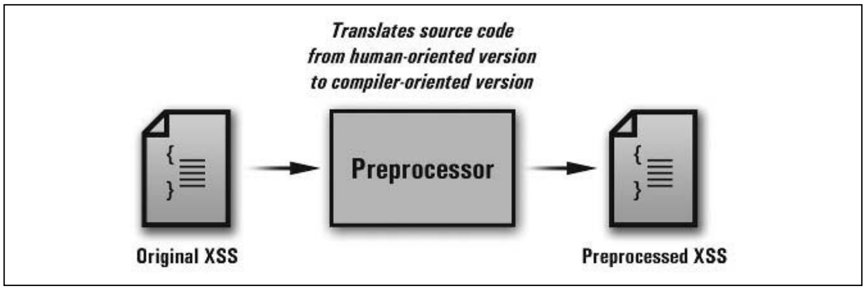


Figure 14.4

The preprocessor translates the original form of the source file to a different form.

I actually prefer to treat comments as preprocessor "directives," unlike many compiler writers, simply because it makes the implementation of the lexer a bit cleaner. For this reason, the preprocessor will need to do this at the very least. Of course, the XtremeScript language specification from Chapter 7 also calls for two basic directives: `#include` for including files and `#define` for defining simple symbolic constants in the form of expandable macros. I'll talk more about these later.

The Lexical Analyzer Module

As you know well by now, the lexer is responsible for converting the raw source code into a more usable format for the parser. This particular module doesn't do anything on its own, however; although its convention to treat the lexer as its own conceptual step that takes place independently, before the parser, it actually operates in parallel with it. The parser is responsible for invoking the lexer on a regular basis to return the next token in the stream, so the lexer doesn't actually execute until the parser explicitly calls it.

The lexer you wrote in the last chapter was specifically designed for use in XtremeScript, so your only job now is to integrate it with the rest of the framework. You'll see how this is done later in the chapter.

The Parser Module

In addition to being the most complex aspect of the compiler, the parser also takes center stage among the various modules of the front end, and is its final phase. The parser is responsible for converting the stream of tokens and lexemes produced by the lexical analyzer into I-code, which is then converted to XVM assembly by the back end. The relationship between the parser and lexer is depicted in Figure 14.5.

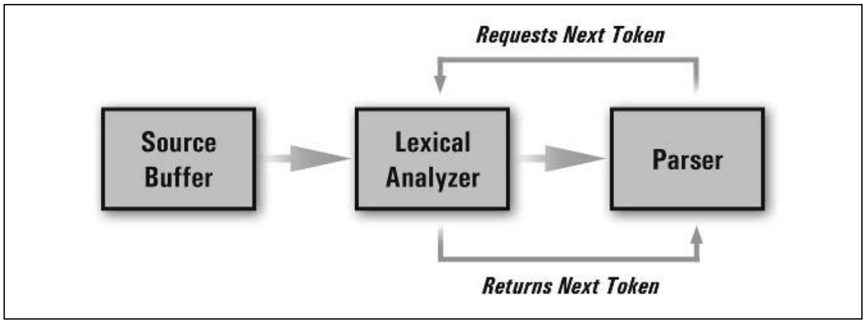


Figure 14.5

The relationship between the lexer and parser.

The I-Code Module

The front and back ends never communicate with each other directly, but rather do so indirectly by interfacing with a common I-code module. Once the front end has produced the I-code, it's entirely removed from the picture (conceptually, at least). The focus then shifts exclusively to the back end, which is responsible for translating the I-code into the target format (which, in this case, is XVM assembly).

As you'll see in more detail later in this chapter, the I-code module will really just be a stream of instructions, very similar in nature to the assembled instruction stream maintained by XASM. The parser will use a number of I-code interface functions to generate instructions within this stream and define their operands, which will make the code emitter's job very easy. Check out Figure 14.6.

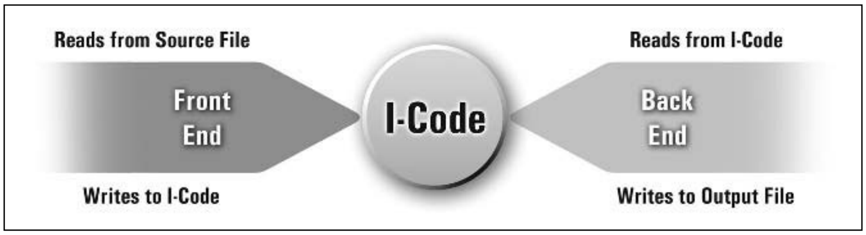


Figure 14.6

Separating the code emitter from the parser via an I-code module simplifies and abstracts both tasks.

The Back End

The back end is responsible for converting the contents of the I-code module to XVM assembly and invoking the XASM assembler to create a ready-to-use .XSE executable from it.

The Code Emitter Module

The XtremeScript compiler doesn't generate actual .XSE executables; rather, it generates an ASCII-formatted XVM assembly file and relies on the XASM assembler built in Chapter 9 to finish the job. Although these two tasks could certainly be combined into a single program (which you could do easily, using only what you've learned from this book), this approach is both easier to grasp from an educational standpoint and also gives you the option to hand-tune the compiler's assembly output before passing it to the assembler. Figure 14.7 depicts the back end and its modules.

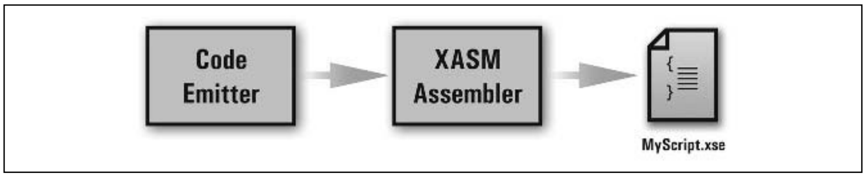


Figure 14.7

The compiler's back end and its modules.

The XASM Assembler

The second "module" of the compiler's back end is actually an entirely separate program. Once the code emitter has done its job, a text file containing an XVM assembly script will be ready to feed into the assembler to produce the final executable. Therefore, the last step in the compiler's lifespan is to briefly invoke the XASM assembler to carry out this final task. The assembly file is then deleted, leaving the user with the original .XSS (source) file, and a newly created executable script (.XSE) file. To the end user, this process is transparent.

Major Structures

In addition to the phases of the compiler, there will also be a number of structures that play a vital role in the conversion of high-level code to its low-level equivalent. As you'll see, most of these will strongly mirror the structures used in XASM. Let's take a look.

The Source Code

As mentioned earlier, the source code will be stored internally as a linked list wherein each node contains a single line of code. Don't confuse lines of code with statements, however. For example, the following line of code is represented as a single node within the list, even though it contains multiple statements:

```
X = 256; Y = 512; MyString = "Hello, world!";
```

Furthermore, single statements can often span multiple lines, such as the following:

```
X
=
256;
```

This would be stored internally as three nodes.

The Script Header

Much like the XASM assembler and the XVM, the compiler will maintain a script's header data and other miscellaneous properties in a global structure known as the script header. As usual, you can use this space to store the script's requested stack size, thread priority, the presence of `_Main ()`, and other such information.

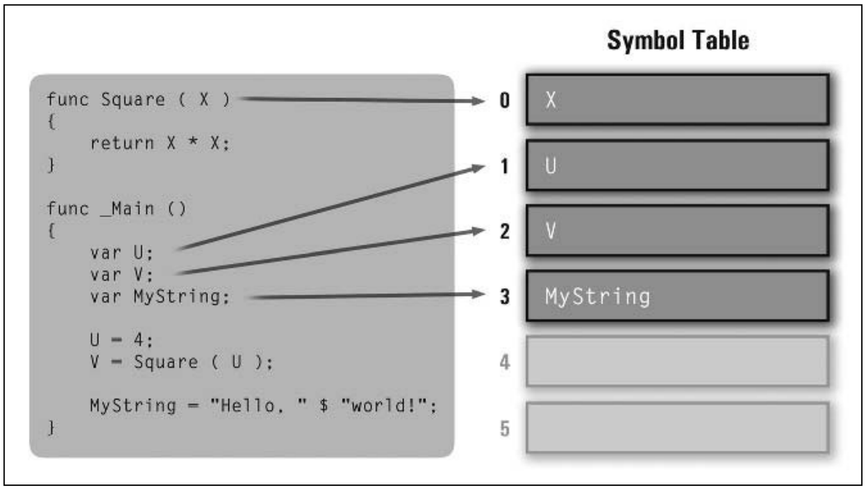
The Symbol Table

As the source file is parsed, variable declarations are interpreted as signs to add data to the symbol table. By the end of the parsing process, the symbol table is a complete and detailed reflection of the script's variables and arrays. Each entry in the table corresponds to a specific variable or array, and contains pertinent information such as its identifier, size, scope, and so on.

In addition to writing data to the symbol table to record a variable, the table will be frequently read to validate the use of a variable based on the context in which its found——its scope, whether an array subscript was accessed, and so on. Check out Figure 14.8 for a visual explanation of the symbol table.

NOTE

Just as was the case with XASM, I use the term *symbol table* in a somewhat ad-hoc sense. Although most compilers tend to use one giant table to store all of a program's identifiers, whether they're functions, variables, structures, classes, or labels, I chose to break it into multiple tables. Although this particular table, because it mainly stores variables and arrays, should probably be called the "variable table," I like to hold on to the original term for posterity.



```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

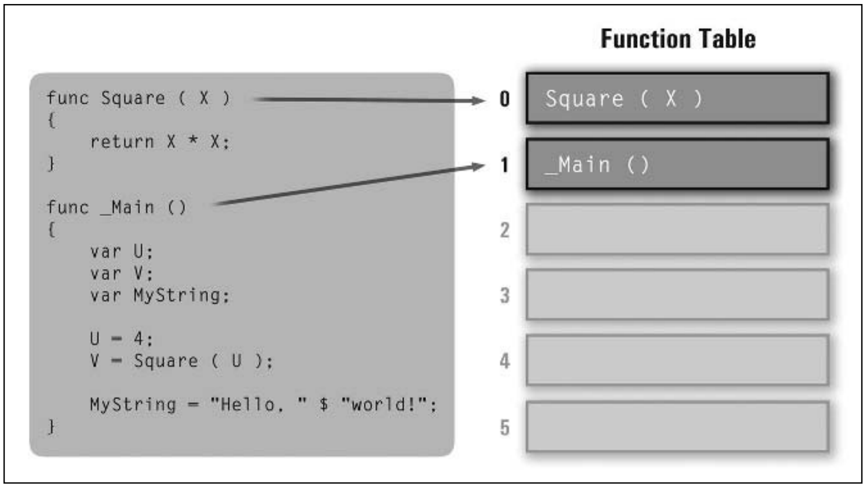
Figure 14.8

The symbol table stores the variables and arrays of a source script.

The Function Table

The function table is similar to the symbol table, but maintains a record of the script's functions, rather than its variables and arrays. The function table stores each function's name, parameter count, and other such information. Like the symbol table, it's written to as functions are initially parsed, and read from as they're called.

One major difference between the XtremeScript compiler and XASM is that there's not a separate table for storing host API calls. You'll find out how this is done later in the chapter, but for now, just make a mental note of the fact that a single table is used to store all functions—whether they're defined by the script or the host. Figure 14.9 depicts the function table.



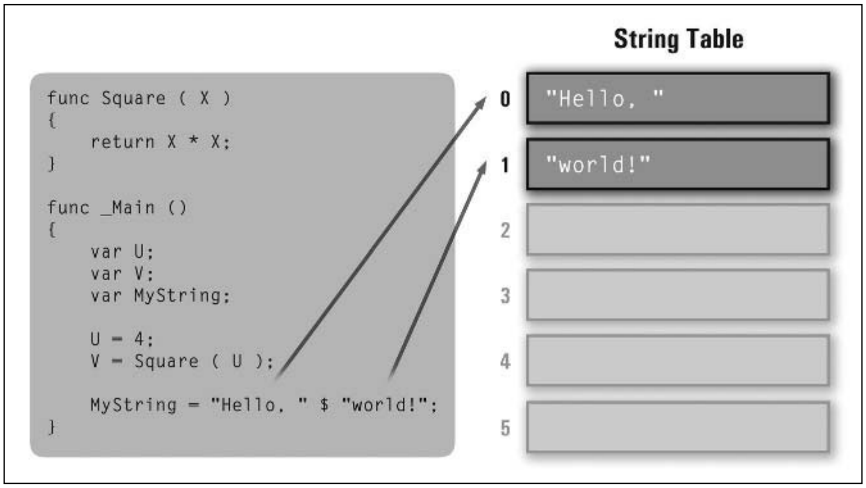
```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

Figure 14.9

The function table stores the functions of a source script.

The String Table

Like an XVM assembly script, a script written in XtremeScript is also likely to contain a number of string literals. In addition to converting the script's statements and declarations to valid XVM assembly, the parser will also be responsible for collecting these strings and storing them in a table (as well as filter out any duplicates it may come across). Take a look at Figure 14.10 for a better idea of what the string table does.



```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

Figure 14.10

The string table stores the string literal values of a source script.

The I-Code Stream

On one end of the spectrum, you have the raw source code stored in a linked list. On the other end, there's a stream of I-code that represents the program in an assembly-like form. To be specific, however, there won't be a single, global I-code structure. Remember, XVM only allows code to appear inside functions; because code in the global scope is illegal, it's better to associate a single block of I-code with each entry in the function table.

Interfaces and Encapsulation

As important as the compiler's structures are, it's equally important that these structures are accessible to other aspects of the program in a clean and consistent way. Instead of thinking of these structures as huge blocks of unwieldy data, it's far easier to conceptualize them as small groups of functions. These functions may be used to read from and write to the structure, sort or organize the structure's data, or any number of things. By not having to deal with the structure's implementation itself, the rest of the compiler can focus on how to use it, rather than how it works. When every module and structure in the compiler looks at every other module and structure in this same way, the entire process of compiling a source file can be broken into a rather straightforward hierarchy of function calls.

Although object-oriented programming is generally the best foundation for the interfaces and implementation of a large program's structures, this compiler still certainly falls within the boundaries of what C is capable of. Because of this, you'll not only pass on classes in favor of `struct`s and functions, but will also break the rules here and there. You could bend over backwards to respect and uphold every last convention for truly hiding and encapsulating the compiler's data, but in the interest of getting things done quickly and easily, you can let it slide here and there.

The Compiler's Lifespan

With all of that out of the way, let's look at a brief and simplified rundown of the major points in the compiler's lifespan.

Reading the Command Line

As soon as the compiler starts, the command line is read to interpret the specified filenames and parameters. The compiler's internal flags, preferences, and structures are initialized with the parameters entered by the users, or with defaults for any parameter that was omitted. If any vital parameters are left out or malformed, an error is displayed and the program exits.

Loading the Source Code

The loader module is then invoked, which uses the source filename specified on the command line. If the file is found, it's opened and loaded into a linked list, and its newline conventions are converted to the compiler's internal format. If the file isn't found, an error is displayed and the program exits.

Preprocessing

The source code then undergoes a preprocessing phase, whose primary job is to strip away comments, both the single-line `//` variety and the `/* */` block style. Other preprocessor directives can be handled here as well, for tasks such as file inclusion and macro expansion.

Parsing

With the source code loaded and preprocessed, the parser is ready to run. It begins its scan at the top of the source file and works its way to the bottom. As each statement and declaration of the script is parsed, information is read from and written to almost all of the compiler's structure; the symbol, function, and string tables are accessed to add new entries and verify existing ones, for example. It's also important to mention that the XtremeScript compiler will work strictly in a single pass; rather than scanning through the file multiple times like XASM, it will work its way from top to bottom in a straight line. This brings with it some restrictions—for example, forward references of functions will be illegal. It will help you understand the concept of single-pass versus multi-pass compilers in a first-hand sense, however.

Of course, the real job of the parser is to generate an I-code representation of the program. By the time the parser is done with its job, this process is complete and the original source code is no longer of any use.

Code Emission

At this point, the script's I-code equivalent has been fully generated, and the compiler is ready to let the code emitter produce a complete XVM assembly file based on it. The emitter's job is really quite simple in the case of this compiler—all it does is scan through each I-code instruction and convert it to its assembly equivalent (although I'll discuss this in far more detail later). When this process is complete, a new file exists in the working directory of the compiler—an .XASM file containing the equivalent of the original .XSS source file.

Invoking XASM

Finally, XASM is transparently invoked from within the compiler to finish the job and convert the resulting .XASM file into a ready-to-run .XSE executable. After XASM runs, the compiler-generated .XASM file is deleted.

The Compiler's `main ()` Function

To wrap this section up, let's look at the finished compiler's `main ()` section:

```
main ( int argc, char * argv [] )
{
    // Print the logo
    PrintLogo ();

    // Validate the command line argument count
    if ( argc < 2 )
    {
    // If at least one filename isn't present, print the usage info and
    // exit
        PrintUsage ();
        return 0;
    }
    // Verify the filenames
    VerifyFilenames ( argc, argv );

    // Initialize the compiler
    Init ();

    // Read in the command line parameters
    ReadCmmndLineParams ( argc, argv );

    // ---- Begin the compilation process (front end)

    // Load the source file into memory
    LoadSourceFile ();

    // Preprocess the source file
    PreprocessSourceFile ();

    // ---- Compile the source code to I-code

    printf ( "Compiling %s...\n\n", g_pstrSourceFilename );
    CompileSourceFile ();

    // ---- Emit XVM assembly from the I-code representation (back end)

    EmitCode ();

    // Print out compilation statistics
    PrintCompileStats ();

    // Free resources and perform general cleanup
    ShutDown ();

    // Invoke XASM to assemble the output file to create the .XSE, unless the
    // user requests otherwise
    if ( g_iGenerateXSE )
        AssmblOutputFile ();

    // Delete the output (assembly) file unless the user requested it to be
    // preserved
    if ( ! g_iPreserveOutputFile )
        remove ( g_pstrOutputFilename );

    return 0;
}
```

Even without an understanding of the rest of the program, this should make reasonable sense. You start by printing the program's "logo," which is really just its title and version information. The number of command-line arguments is then checked; if it's less than 2 (meaning only the name of the program was passed), the user hasn't specified any action to be taken. In response to this, usage information that explains the command-line interface is printed and the program exits. The filenames are then verified, the same basic initialization is performed, and the remaining arguments are read.

At this point, the initial interface with the user is over and the source code is loaded and preprocessed. A message is then printed, alerting the user that the file is compiling, and the compilation process begins. Once an I-code representation has been generated, the code is emitted and various statistics gathered during and after the compilation process are presented (just like in XASM). The compiler then shuts down by freeing its internal structures.

The process isn't over yet, however. To finish the job, XASM is invoked to convert the assembly file to an executable and the temporary XVM assembly file is deleted. Notice also that global variables are checked before both of these tasks are executed; this is done to allow the user to suppress either the generation of the executable or the deletion of the assembly file.

What's really important to notice here is that the entire operation of the compiler boiled down to a handful of function calls. This is what I mean by clean interfaces; by reducing the compiler's lifespan to a series of discrete and coarse-grained steps, everything becomes much easier to implement.

The Command-Line Interface

Because the XtremeScript compiler is a console application, its primary interface is the command line. In addition to specifying input and output files, a number of parameters can be interpreted as well that afford the user more precise control of the compiler's output. Here's the general format of the compiler's command-line interface (notice first that the program's name is `XSC`, which stands for **X**treme**S**cript **C**ompiler):

```
XSC Source.XSS [Output.XASM] [Options]
```

The Logo and Usage Info

What I call the "logo" really just boils down to the program's name and version information. It's always a good idea to print one at the top of any program you intend for general use. Naturally, `PrintLogo ()` is a pretty simple function:

```
void PrintLogo ()
{
    printf ( "XSC\n" );
    printf ( "XtremeScript Compiler Version %d.%d\n", VERSION_MAJOR,
        VERSION_MINOR );
    printf ( "Written by Alex Varanese\n" );
    printf ( "\n" );
}
```

After printing the logo, `main ()` then prints the program's usage info and exits if the user didn't supply any command-line arguments:

```
void PrintUsage ()
{
    printf ( "Usage:\tXSC Source.XSS [Output.XASM] [Options]\n" );
    printf ( "\n" );
    printf ( "\t-S:Size      Sets the stack size (must be decimal integer
        value)\n" );
    printf ( "\t-P:Priority  Sets the thread priority: Low, Med, High or
        timeslice\n" );
    printf ( "\t             duration (must be decimal integer value)\n" );
    printf ( "\t-A           Preserve assembly output file\n" );
    printf ( "\t-N           Don't generate .XSE (preserves assembly
        output file)\n" );
    printf ( "\n" );
    printf ( "Notes:\n" );
    printf ( "\t- File extensions are not required.\n" );
    printf ( "\t- Executable name is optional; source name is used by
        default.\n" );
    printf ( "\n" );
}
```

Reading Filenames

The source filename is a mandatory parameter and must come first. Without this, the program will not operate properly, if at all. Beyond the source filename, a number of optional arguments may follow, starting with the output filename. Because the compiler technically generates only .XASM files on its own (it relies on the assembler to create the executable), the filename specified here relates to the .XASM file it will generate. Of course, the compiler passes this same name as the output filename for the assembler, so it technically counts for both purposes. If an output filename is not specified, the input filename is used in its place. Furthermore, neither filename is required to have an extension; entering `MySource` as the input name is no different than `MySource.XSS`. The compiler will automatically append one in its absence.

Implementation

According to the compiler's `main ()` function, the compiler calls a function called `VerifyFilenames ()` to read the filenames from the command line, append file extensions if necessary, and store them for subsequent use by other modules. Regardless of how many filenames were initially specified by the user, `VerifyFilenames ()` produces two separate strings and stores them globally in these string variables:

```
char g_pstrSourceFilename [ MAX_FILENAME_SIZE ],
     g_pstrOutputFilename [ MAX_FILENAME_SIZE ];
```

`g_pstrSourceFilename` stores the .XSS filename, whereas `g_pstrOutputFilename` stores the filename that will be used for both the .XASM assembly file and the .XSE executable. Although global strings certainly aren't the cleanest or most encapsulated way to pass filenames around, they do make it a lot easier for any given module to access them when necessary. Both functions make use of the `MAX_FILENAME_SIZE` constant, which I have to 2048:

```
#define MAX_FILENAME_SIZE           2048
```

Sure, 2048 is complete and utter overkill, but I like to be safe. *Really* safe. With this kind of padding, this compiler will still be running happily in the *year* 2048. :)

Here's the code for `VerifyFilenames ()`:

```
void VerifyFilenames ( int argc, char * argv [] )
{
    // First make a global copy of the source filename and convert it to
    // uppercase
    strcpy ( g_pstrSourceFilename, argv [ 1 ] );
    strupr ( g_pstrSourceFilename );

    // Check for the presence of the .XASM extension and add it if it's not
    // there
    if ( ! strstr ( g_pstrSourceFilename, SOURCE_FILE_EXT ) )
    {
    // The extension was not found, so add it to string
        strcat ( g_pstrSourceFilename, SOURCE_FILE_EXT );
    }

    // Was an executable filename specified?
    if ( argv [ 2 ] && argv [ 2 ][ 0 ] != '-' )
    {
    // Yes, so repeat the validation process
        strcpy ( g_pstrOutputFilename, argv [ 2 ] );
        strupr ( g_pstrOutputFilename );
    // Check for the presence of the .XSE extension and add it if it's not
    // there
        if ( ! strstr ( g_pstrOutputFilename, OUTPUT_FILE_EXT ) )
        {
    // The extension was not found, so add it to string
            strcat ( g_pstrOutputFilename, OUTPUT_FILE_EXT );
        }
    }
    else
    {
    // No, so base it on the source filename

    // First locate the start of the extension, and use pointer subtraction
    // to find the index

        int ExtOffset = strrchr ( g_pstrSourceFilename, '.' ) -
            g_pstrSourceFilename;
        strncpy ( g_pstrOutputFilename, g_pstrSourceFilename, ExtOffset );

    // Append null terminator
        g_pstrOutputFilename [ ExtOffset ] = '\0';

    // Append executable extension
        strcat ( g_pstrOutputFilename, OUTPUT_FILE_EXT );
    }
}
```

The function accepts two parameters—the argument count and argument array passed to the `main ()` function from the command line. The argument at array index 1 should contain the filename, so you can start there. The first task is to copy the string into the `g_pstrSourceFilename` global, and then to convert it to uppercase to keep things uniform and consistent. The `strstr ()` function is then used to determine the presence of the "." character. If it's not found, it's taken as a sign that the extension was omitted and `strcat ()` is used to append one using the `SOURCE_FILE_EXT` constant:

```
#define SOURCE_FILE_EXT          ".XSS"
```

NOTE

Simply checking for the presence of the dot character isn't enough to truly verify whether the extension was supplied, but it's close enough. Chances are, the extension will either be there or it won't; anything else will be more or less considered malformed and ultimately cause a fatal error down the line. Either way, the user will be alerted to the mistake sooner or later.

This takes care of the first filename, so the second one is read next. This part is a twofold job; in addition to checking for the presence of the extension, it must be determined whether a second filename was specified. If not, the filename is based on the first.

The second filename should be located at index 2 of the `argv []` array. To find out if the filename was provided, it's first determined if the string is null. If not, the string's first character is then read—if it's a dash (`-`) character, you know it's not a filename. This is because, as you'll see soon, command-line options are always preceded by a dash. As was the case with the first filename, the string is copied into its respective global variable (`g_pstrOutputFilename` in this case) and the `OUTPUT_FILE_EXT` constant is appended:

```
#define OUTPUT_FILE_EXT             ".XASM"
```

If, however, an output filename wasn't specified, it must be based on the source filename. The process here is easy; because you know for sure that the filename has an extension at this point, you can simply copy `g_pstrSourceFilename` into `g_pstrExecFilename` and replace the first instance of the `.` character with a `'\0'` null terminator. This effectively "cuts the string off" at that point, allowing you to append the proper extension easily.

Reading Options

Following the filename(s), a number of options may be passed as well. Table 14.1 summarizes them.

Table 14.1 Compiler Command-Line Options

Name	Valid Values	Description
`S`	Decimal Integer	Sets the script's requested stack size
`P`	Decimal Integer, `Low`, `Med`, or `High`	Sets the script's thread priority
`A`	None	Prevents the compiler-generated assembly file from being deleted
`N`	None	Prevents XASM from being invoked, thereby suppressing the generation of an .XSE executable. Also forces the preservation of the assembly file

All command-line options must be preceded with a dash (-) to differentiate them from filenames. Each of them are optional, and although A and N are technically mutually exclusive, this isn't enforced. Lastly, the options can appear in any order (unlike the filenames, which must always be either the first, or first and second in the list). As is shown in the table, the -S and -P options accept values. Options with values are written in the form of -Option:Value, so a stack size of 8192 could be requested like this:

```
-S:8192
```

A thread time slice of 120 could be set with the priority option like this:

```
-P:120
```

However, -P also accepts the three keyword strings listed in Table 14.1, so a medium-level priority could be requested like this:

```
-P:Med
```

Of course, it's all case-insensitive.

Implementation

Reading these options in is handled by the ReadCmmndLineParams () function. The function begins by entering a loop that reads each command from the argv [] array and converts it to uppercase. The loop then determines whether the current argument is a valid option by checking for the presence of a dash in the first character:

```
void ReadCmmndLineParams ( int argc, char * argv [] )
{
    char pstrCurrOption [ 32 ];
    char pstrCurrValue [ 32 ];
    char pstrErrorMssg [ 256 ];

    for ( int iCurrOptionIndex = 0; iCurrOptionIndex < argc;
        ++ iCurrOptionIndex )
    {
    // Convert the argument to uppercase to keep things neat and tidy
        strupr ( argv [ iCurrOptionIndex ] );

    // Is this command line argument an option?
        if ( argv [ iCurrOptionIndex ][ 0 ] == '-' )
        {
```

`pstrCurrOption`, `psrtCurrValue`, and `pstrErrorMssg` are just local copies of various strings read from the arrays as the loop executes. Once it's determined that the current argument is a valid option, its actual data is extracted. This can be either a one- or two-step process, depending on whether the option accepts a value. Both the `-S` and `-P` options do, but `-A` and `-N` don't:

```
// Parse the option and value from the string
int iCurrCharIndex;
int iOptionSize;
char cCurrChar;

// Read the option up till the colon or the end of the string
iCurrCharIndex = 1;
while ( TRUE )
{
    cCurrChar = argv [ iCurrOptionIndex ][ iCurrCharIndex ];
    if ( cCurrChar == ':' || cCurrChar == '\0' )
        break;
    else
        pstrCurrOption [ iCurrCharIndex - 1 ] = cCurrChar;
    ++ iCurrCharIndex;
}
pstrCurrOption [ iCurrCharIndex - 1 ] = '\0';

// Read the value till the end of the string, if it has one
if ( strstr ( argv [ iCurrOptionIndex ], ":" ) )
{
    ++ iCurrCharIndex;
    iOptionSize = iCurrCharIndex;

    pstrCurrValue [ 0 ] = '\0';
    while ( TRUE )
    {
        if ( iCurrCharIndex > ( int ) strlen ( argv [ iCurrOptionIndex ] ) )
            break;
        else
        {
            cCurrChar = argv [ iCurrOptionIndex ][ iCurrCharIndex ];
            pstrCurrValue [ iCurrCharIndex - iOptionSize ] = cCurrChar;
        }
        ++ iCurrCharIndex;
    }
    pstrCurrValue [ iCurrCharIndex - iOptionSize ] = '\0';
    // Make sure the value is valid
    if ( ! strlen ( pstrCurrValue ) )
    {
        sprintf ( pstrErrorMssg, "Invalid value for -%s option",
            pstrCurrOption );
        ExitOnError ( pstrErrorMssg );
    }
}
```

NOTE

As you can see, command-line options can be more than one character, even if none of the existing options has taken advantage of this.

This is handled in two loops. The first reads all characters until the end of the string or the first instance of the : character and adds them to the `pstrCurrOption` string. When this loop is complete, `pstrCurrOption` will contain the complete option string.

The second loop starts where the first left off, but only if the option string contains a : character. If it doesn't, the loop is skipped entirely because it's clear that the option doesn't contain a value. Otherwise, the character index is incremented to move it past the : read in the last loop, and every subsequent character read from the string is added to `pstrCurrValue`. When these two loops have completed, `pstrCurrOption` and `pstrCurrValue` will be populated with separate strings containing the option's name and value. Lastly, the resulting `pstrCurrValue` string is analyzed to make sure it's a valid value (in other words, its length has to be greater than zero).

With the option's name and value isolated, the option is carried out. First up is the `-S` directive, which sets the stack size:

```
// Set the stack size
if ( stricmp ( pstrCurrOption, "S" ) == 0 )
{
    // Convert the value to an integer stack size
    g_ScriptHeader.iStackSize = atoi ( pstrCurrValue );
}
```

As you can see, it's pretty simple; the `pstrCurrValue` string is converted to an integer with the `atoi ()` function, and placed in the `g_ScriptHeader.iStackSize` field. I haven't covered the `g_ScriptHeader` structure yet, but this should all be pretty self-explanatory.

Next up is `-P`, which sets the script's thread priority:

```
// Set the priority
else if ( stricmp ( pstrCurrOption, "P" ) == 0 )
{
    // ---- Determine what type of priority was specified

    // Low rank
    if ( stricmp ( pstrCurrValue, PRIORITY_LOW_KEYWORD ) == 0 )
    {
        g_ScriptHeader.iPriorityType = PRIORITY_LOW;
    }

    // Medium rank
    else if ( stricmp ( pstrCurrValue, PRIORITY_MED_KEYWORD ) == 0 )
    {
        g_ScriptHeader.iPriorityType = PRIORITY_MED;
    }

    // High rank
    else if ( stricmp ( pstrCurrValue, PRIORITY_HIGH_KEYWORD ) == 0 )
    {
        g_ScriptHeader.iPriorityType = PRIORITY_HIGH;
    }

    // User-defined time slice
    else
    {
        g_ScriptHeader.iPriorityType = PRIORITY_USER;
        g_ScriptHeader.iUserPriority = atoi ( pstrCurrValue );
    }
}
```

This one's a bit more work because it not only has to interpret integer values, but the `Low`, `Med`, and `High` strings as well. `pstrCurrValue` is compared to the `PRIORITY_*_KEYWORD` constants to determine whether it's one of them, and the `g_ScriptHeader.iPriorityType` field is set accordingly with one of three `PRIORITY_*` constants. Here they are:

```
#define PRIORITY_NONE               0
#define PRIORITY_USER               1
#define PRIORITY_LOW                2
#define PRIORITY_MED                3
#define PRIORITY_HIGH               4
```

```
#define PRIORITY_LOW_KEYWORD        "Low"
#define PRIORITY_MED_KEYWORD        "Med"
#define PRIORITY_HIGH_KEYWORD       "High"
```

If the option's value doesn't match any of the keywords, `pstrCurrValue` is unconditionally converted to an integer and assigned to `g_ScriptHeader.iUserPriority`. The `iPriorityType` field is then set to `PRIORITY_USER` to reflect this.

The last two command-line options are `-A` and `-N`, which preserve the assembly file and suppress the generation of the executable, respectively:

```
// Preserve the assembly file
else if ( stricmp ( pstrCurrOption, "A" ) == 0 )
{
    g_iPreserveOutputFile = TRUE;
}

// Don't generate an .XSE executable
else if ( stricmp ( pstrCurrOption, "N" ) == 0 )
{
    g_iGenerateXSE = FALSE;
    g_iPreserveOutputFile = TRUE;
}
```

Because these options don't relate specifically to the script itself, I kept them in separate global variables. `iPreserveOutputFile` is `TRUE` if the compiler-generated .XASM file should be saved, and `g_iGenerateXSE` is `FALSE` if XASM should not be invoked to create an .XSE executable. Notice that the `-N` option automatically preserves the assembly file, whether or not the `-A` option was present—without this, the `-A` and `-N` option would result in the compiler doing nothing at all if they were both passed by the user.

Any option other than these is invalid:

```
// Anything else is invalid
else
{
    sprintf ( pstrErrorMssg, "Unrecognized option: \"%s\"", pstrCurrOption );
    ExitOnError ( pstrErrorMssg );
}
```

Throughout this section, you've been making use of the `ExitOnError ()` function. This behaves just as the function of the same name did in XASM, but I'll cover it in a moment anyway. It shouldn't take much effort to figure out what it does until then, however.

Elementary Data Structures

Before proceeding, I'd like to get one thing out of the way. This chapter, as well as the next, will make heavy use of both the stack and linked list data types. Although these are obviously both simple to understand and implement, I think it's a good idea to briefly cover their specific implementation in the XtremeScript compiler, so you'll fully understand their usage later.

TIP

Naturally, in a real project I'd suggest taking an object-oriented approach to these structures, and would probably just recommend leveraging the existing STL versions. After all, the STL has been in steady development for years, is feature rich and easy to use, and is highly robust. For the purpose of a book, however, it's often easier to simply go with traditional, C-style custom solutions that readers can be walked through in entirety.

Linked Lists

The compiler makes heavy use of linked lists, which are implemented in a quick and simple way using C structures and functions. I've implemented the lists with two structures: one to represent nodes and one to represent a list's base structure that keeps track of everything. Let's start with the node structure, `LinkedListNode`:

```
typedef struct _LinkedListNode   // A linked list node
{
    void * pData;    // Pointer to the node's data
    _LinkedListNode * pNext;     // Pointer to the next node in
                                // the list
}
    LinkedListNode;
```

As you can see, this is a singly linked list, so it can only be traversed in a single direction. Each node needs only two fields—the void data pointer, `pData`, and the pointer to the next node in the chain, `pNext`.

The list itself is maintained with the `LinkedList` structure:

```
typedef struct _LinkedList      // A linked list
{
    LinkedListNode * pHead,     // Pointer to head node
                   * pTail;     // Pointer to tail nail node
    int iNodeCount;             // The number of nodes in the
                                // list
}
    LinkedList;
```

This simple structure consists of three fields. The two pointers, `pHead` and `pTail`, point to the head and tail nodes of the list. `iNodeCount` keeps track of how many nodes the list contains. Check out Figure 14.11.

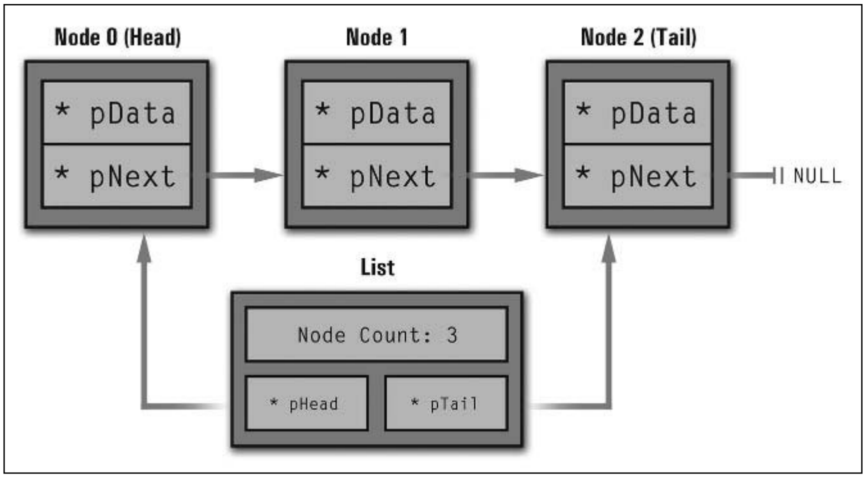


Figure 14.11

The linked list structure.

The Interface

The linked list interface is rather simple; it has a handful of functions for initializing and freeing lists, adding nodes, deleting nodes, and managing string nodes.

Initializing Lists

Let's start with `InitLinkedList ()`, which initializes a linked list:

```
void InitLinkedList ( LinkedList * pList )
{
    // Set both the head and tail pointers to null
    pList->pHead = NULL;
    pList->pTail = NULL;

    // Set the node count to zero, since the list is currently empty
    pList->iNodeCount = 0;
}
```

The function does its job simply by pointing the head and tail nodes at nothing and resetting the node count, based on the specified `LinkedList` structure pointer.

Freeing Lists

Initializing a list is easy, but freeing one is a bit more complex:

```
void FreeLinkedList ( LinkedList * pList )
{
    // If the list is empty, exit
    if ( ! pList )
        return;

    // If the list is not empty, free each node
    if ( pList->iNodeCount )
    {
    // Create a pointer to hold each current node and the next node
        LinkedListNode * pCurrNode,
                       * pNextNode;

    // Set the current node to the head of the list
        pCurrNode = pList->pHead;

    // Traverse the list
        while ( TRUE )
        {
    // Save the pointer to the next node before freeing the current one
            pNextNode = pCurrNode->pNext;

    // Clear the current node's data
            if ( pCurrNode->pData )
                free ( pCurrNode->pData );

    // Clear the node itself
            if ( pCurrNode )
                free ( pCurrNode );

    // Move to the next node if it exists; otherwise, exit the loop
            if ( pNextNode )
                pCurrNode = pNextNode;
            else
                break;
        }
    }
}
```

The function takes a single LinkedList structure pointer. The list is traversed with two node pointers, pCurrNode and pNextNode. pCurrNode is set to the head of the list, and the traversal begins. At each iteration of the loop, the pointer to the next node is saved in pNextNode. The current node's data is then freed, as well as the structure representing the node itself, and the saved pCurrNext pointer is used to advance to the next node in the list. If the next node is null, it's taken as a sign that the tail has been reached and the loop exits.

Adding Nodes

Adding a node to the list requires two cases to be considered; that either the new node is the head (because the list was empty before the addition), or the node is being added to a non-empty list and is therefore the new tail. Let's have a look:

```
int AddNode ( LinkedList * pList, void * pData )
{
    // Create a new node
    LinkedListNode * pNewNode = ( LinkedListNode * )
        malloc ( sizeof ( LinkedListNode ) );

    // Set the node's data to the specified pointer
    pNewNode->pData = pData;

    // Set the next pointer to NULL, since nothing will lie beyond it
    pNewNode->pNext = NULL;

    // If the list is currently empty, set both the head and
    // tail pointers to the new node
    if ( ! pList->iNodeCount )
    {
    // Point the head and tail of the list at the node
        pList->pHead = pNewNode;
        pList->pTail = pNewNode;
    }

    // Otherwise append it to the list and update the tail pointer
    else
    {
    // Alter the tail's next pointer to point to the new node
        pList->pTail->pNext = pNewNode;
    // Update the list's tail pointer
        pList->pTail = pNewNode;
    }

    // Increment the node count
    ++ pList->iNodeCount;

    // Return the new size of the linked list - 1,
    // which is the node's index
    return pList->iNodeCount - 1;
}
```

The first thing the function does is allocates space for the new node's `LinkedListNode` structure. It then sets the node's data pointer to the `pData` parameter, and the next node pointer to `NULL`. The specified list, `pList`, is then analyzed to determine whether it's already populated with at least one node. If not, both its head and tail pointers are set to the new node. Otherwise, the tail node is updated to point to the new node (which becomes the new tail), and the base `LinkedList` structure's tail is updated to point to the new node as well. The function wraps up by incrementing the node count and returning the node count minus one as an index. You subtract one because otherwise, the index would always be one higher than it needs to be; when the node count is one, the first index is zero, and so on.

Deleting Nodes

Deleting a node also requires that attention be paid to specific cases. Care must be taken to patch up the hole left by the deleted node, so that its immediate neighbors will link with one another and keep the list contiguous. This matter is complicated by the fact that the head node does not require any patching. Here's the function:

```
void DelNode ( LinkedList * pList, LinkedListNode * pNode )
{
    // If the list is empty, return
    if ( pList->iNodeCount == 0 )
        return;

    // Determine if the head node is to be deleted
    if ( pNode == pList->pHead )
    {
    // If so, point the list head pointer to the node just after the
    // current head
        pList->pHead = pNode->pNext;
    }
    else
    {
    // Otherwise, traverse the list until the specified node's previous
    // node is found
        LinkedListNode * pTravNode = pList->pHead;
        for ( int iCurrNode = 0; iCurrNode < pList->iNodeCount; ++ iCurrNode )
        {
    // Determine if the current node's next node is the specified one
            if ( pTravNode->pNext == pNode )
            {
    // Determine if the specified node is the tail
                if ( pList->pTail == pNode )
                {
    // If so, point this node's next node to NULL and set it as
    // the new tail
                    pTravNode->pNext = NULL;
                    pList->pTail = pTravNode;
                }
                else
                {
    // If not, patch this node to the specified one's next node
                    pTravNode->pNext = pNode->pNext;
                }
                break;
            }

    // Move to the next node
            pTravNode = pTravNode->pNext;
        }
    }

    // Decrement the node count
    -- pList->iNodeCount;

    // Free the data
    if ( pNode->pData )
        free ( pNode->pData );

    // Free the node structure
    free ( pNode );
}
```

The function accepts a linked list pointer, `pList`, and a node pointer, `pNode`. It starts by determining whether the node to be deleted is the head node. If so, it sets the base structure's head pointer to the node just after the current head pointer.

If the node to be deleted isn't the head, it creates a new node pointer called `pTravNode` to traverse the node and find it. At each iteration, `pTravNode`'s next node is compared to `pNode` to determine whether they match. If so, the function then finds out if the node to be deleted is the tail. The tail node is handled by setting `pTravNode`'s next pointer to `NULL`, and setting the base structure's tail pointer to `pTravNode`. This effectively separates the old tail from the list and allows you to safely delete it. If the node isn't the tail, it simply sets `pTravNode`'s next pointer to the node immediately following its current next node. The function ends by decrementing the node count and freeing both the node's data and `LinkedListNode` structure.

Adding String Nodes

The string table is an example of a table in which every node's `pData` field simply points to a string. In XASM, because both the string and host API call tables were implemented in the same way, I created a generic function called `AddString ()` that would add any string pointer to any linked list. This allowed both tables to leverage the same function and minimize the redundant code that would've otherwise resulted. Even though the XtremeScript compiler will only use one pure string linked list, I left `AddString ()` unchanged:

```
int AddString ( LinkedList * pList, char * pstrString )
{
    // ---- First check to see if the string is already in the list

    // Create a node to traverse the list
    LinkedListNode * pNode = pList->pHead;

    // Loop through each node in the list
    for ( int iCurrNode = 0; iCurrNode < pList->iNodeCount; ++ iCurrNode )
    {
    // If the current node's string equals the specified string, return its
    // index
        if ( strcmp ( ( char * ) pNode->pData, pstrString ) == 0 )
            return iCurrNode;

    // Otherwise move along to the next node
        pNode = pNode->pNext;
    }
    // ---- Add the new string, since it wasn't added

    // Create space on the heap for the specified string
    char * pstrStringNode = ( char * ) malloc ( strlen ( pstrString ) + 1 );
    strcpy ( pstrStringNode, pstrString );

    // Add the string to the list and return its index
    return AddNode ( pList, pstrStringNode );
}
```

This function accepts two parameters—a linked list pointer called `pList` and a string pointer called `pstrString`—and gets most of its functionality from `AddNode ()`, which is ultimately called to add the string to the list. Before doing so, however, it iterates through each string in the table and compares it to `pstrString`. If they match, the function returns the index of the current node; otherwise, the string is added and the index returned by `AddNode ()` is returned to the caller.

Retrieving String Nodes

The last basic linked list function I've included is called `GetStringByIndex ()`, and returns the string at the node corresponding to the specified index:

```
char * GetStringByIndex ( LinkedList * pList, int iIndex )
{
    // Create a node to traverse the list
    LinkedListNode * pNode = pList->pHead;

    // Loop through each node in the list
    for ( int iCurrNode = 0; iCurrNode < pList->iNodeCount; ++ iCurrNode )
    {
    // If the current node's string equals the
    // specified string, return its index
        if ( iIndex == iCurrNode )
            return ( char * ) pNode->pData;

    // Otherwise move along to the next node
        pNode = pNode->pNext;
    }

    // Return a null string if the index wasn't found
    return NULL;
}
```

This function accepts a linked list pointer, pList, as well as an integer index, iIndex, which it uses to find the desired string. It does so by iterating through each node in the list and comparing the current node counter, iCurrNode, to the specified index. If a match is found, the node's pData pointer is cast to a string pointer and returned to the caller. Otherwise, NULL is returned.

Stacks

Although stacks are unique data structures unto themselves, I've based their implementation almost entirely on the previous linked list. You can see this quite clearly in the implementation of the Stack structure:

```
typedef struct _Stack           // A stack
{
    LinkedList ElmntList;       // An internal linked list to
                                // hold the elements
}
    Stack;
```

By basing the stack on a linked list, it always takes up exactly as much memory as it needs, and can grow and shrink indefinitely. Stacks are illustrated in Figure 14.12.

NOTE

Notice I continue to use a Stack structure, even though it consists solely of a nested LinkedList structure and could very well be omitted. I did this to help abstract the underlying implementation of the stack so it can be changed later without breaking any code.

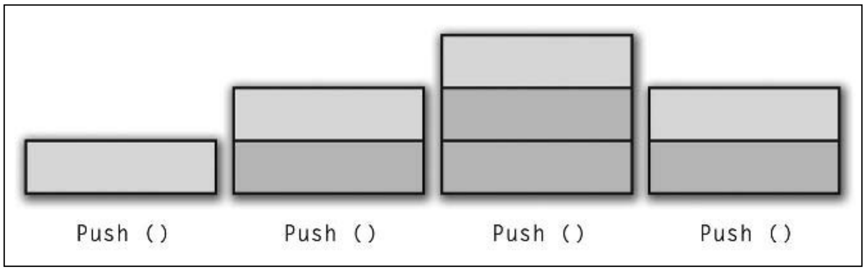


Figure 14.12

The stack structure.

The Interface

As you probably imagine, the stack structure's interface is pretty simple. And because it's based entirely on the pre-existing linked list interface, the functions are extremely short. Let's have a quick look.

Initializing Stacks

Because the initialization of a stack really just means the initialization of its underlying linked list, all this function boils down to is a call to `InitLinkedList ()`:

```
void InitStack ( Stack * pStack )
{
    // Initialize the stack's internal list
    InitLinkedList ( & pStack->ElmntList );
}
```

Freeing Stacks

The same goes for freeing a stack; all that's required is to free its internal linked list:

```
void FreeStack ( Stack * pStack )
{
    // Free the stack's internal list
    FreeLinkedList ( & pStack->ElmntList );
}
```

Determining Whether a Stack is Empty

As you'll see later, it will be important down the line to quickly determine whether a stack is empty. This can be done by evaluating the linked list's `iNodeCount` field. If it's greater than zero, `TRUE` can be returned. Otherwise, `FALSE` is returned:

```
int IsStackEmpty ( Stack * pStack )
{
   if ( pStack->ElmntList.iNodeCount > 0 )
        return FALSE;
    else
        return TRUE;
}
```

Pushing Elements onto a Stack

This brings you to the first of part of the classic stack interface, pushing an element. Because a push operation always puts the new element on top of the stack, you could use the linked list's `pTail` pointer to determine where the current "top" is, and simply add the node after that. In fact, `AddNode ()` already does this for you, which means `Push ()` really just wraps it:

```
void Push ( Stack * pStack, void * pData )
{
    // Add a node to the end of the stack's internal list
    AddNode ( & pStack->ElmntList, pData );
}
```

Popping Elements off a Stack

The opposite of pushing, of course, is popping. Unlike traditional stacks, however, the stack will not return the data member it removes from the top of the stack; rather, it will simply delete it. It does this by passing `DelNode ()` a pointer to the list's tail node:

```
void Pop ( Stack * pStack )
{
    // Free the tail node of the list and its data
    DelNode ( & pStack->ElmntList, pStack->ElmntList.pTail );
}
```

Peeking at the Top Element

Because `Pop ()` doesn't return the actual data it removes, you need another way to access it. This can be done with `Peek ()`, which returns a pointer to the topmost element's data:

```
void * Peek ( Stack * pStack )
{
    // Return the data at the tail node of the list
    return pStack->ElmntList.pTail->pData;
}
```

Initialization and Shutdown

To steer the discussion back to reality, let's shift the focus to the basic initialization and shutdown process taken by the compiler. In order for this to make sense, however, I have to cover some of the compiler's basic global variables and structures first.

Global Variables and Structures

The major global variables and structures used by the program consist of the script header, the script's source code, and the symbol, function, and string tables. All but the first are implemented as linked lists:

```
LinkedList g_SourceCode;
LinkedList g_FuncTable;
LinkedList g_SymbolTable;
LinkedList g_StringTable;
```

The script header, however, is an instance of the `ScriptHeader` structure. Here's its definition:

```
typedef struct _ScriptHeader    // Script header data
{
    int iStackSize;             // Requested stack size

    int iIsMainFuncPresent;     // Is _Main () present?
    int iMainFuncIndex;         // _Main ()'s function index

    int iPriorityType;          // The thread priority type
    int iUserPriority;          // The user-defined priority
                                // (if any)
}
    ScriptHeader;
```

If you recall from Chapter 9, notice that it's almost identical to the XASM script header structure. It keeps track of the stack size, the whereabouts of the `_Main ()` function, and information on the script's thread priority (with space for both a rank and a user-defined time slice duration).

As mentioned, `g_ScriptHeader` is simply an instance of the structure:

```
ScriptHeader g_ScriptHeader;
```

Initialization

When the compiler starts, it calls the `Init ()` function to perform some basic setup:

```
void Init ()
{
    // ---- Initialize the script header

    g_ScriptHeader.iIsMainFuncPresent = FALSE;
    g_ScriptHeader.iStackSize = 0;
    g_ScriptHeader.iPriorityType = PRIORITY_NONE;

    // ---- Initialize the main settings
    // Mark the assembly file for deletion
    g_iPreserveOutputFile = FALSE;

    // Generate an .XSE executable
    g_iGenerateXSE = TRUE;

    // Initialize the source code list
    InitLinkedList ( & g_SourceCode );

    // Initialize the tables
    InitLinkedList ( & g_FuncTable );
    InitLinkedList ( & g_SymbolTable );
    InitLinkedList ( & g_StringTable );
}
```

The function should be pretty clear. It starts by initializing `g_ScriptHeader` some default values. It initially assumes that `_Main ()` isn't present, and that a stack size and thread priority were not requested (hence the `PRIORITY_NONE` constant). The `g_iPreserveOutputFile` and `g_iGenerateXSE` global flags are set to their defaults as well, which might not be overwritten by the command-line arguments passed by the user. Lastly, the compiler's linked lists are initialized.

Shutting Down

The shutdown sequence is even easier than initialization. All that's necessary is the freeing of the compiler's linked lists:

```
void ShutDown ()
{
    // Free the source code
    FreeLinkedList ( & g_SourceCode );

    // Free the tables
    FreeLinkedList ( & g_FuncTable );
    FreeLinkedList ( & g_SymbolTable );
    FreeLinkedList ( & g_StringTable );
}
```

Another function is provided for causing the compiler to exit at any time, called `Exit ()`:

```
void Exit ()
{
    // Give allocated resources a chance to be freed
    ShutDown ();

    // Exit the program
    exit ( 0 );
}
```

This decidedly trivial function simply allows the caller to run the compiler's shutdown sequence and exit the program in a single call.

The Compiler's Modules

Because the compiler is a decidedly more complex project than the XASM assembler or the XVM, it's broken into a number of source and header files to further abstract and encapsulate its various modules. These files are listed in Table 14.2.

Table 14.2 Compiler Module Files

Filename	Description
code_emit.cpp|h	The code emission module
error.cpp|h	Error handling
func_table.cpp|h	The function table
i_code.cpp|h	The I-code module
lexer.cpp|h	The lexical analyzer module
linked_list.cpp|h	Linked list implementation
parser.cpp|h	The parser module
preprocessor.cpp|h	The preprocessor module
stack.cpp|h	Stack implementation
symbol_table.cpp|h	The symbol table
xsc.cpp|h	The main module, in charge of running everything else
globals.h	Basic global data that all modules share

By breaking the project down like this, it's simply a matter of knocking out each module, one by one, until they're all finished. You've already seen some of this; much of `xsc.cpp|h` has been explained in the earlier sections (although the rest will be revisited), I just finished a thorough discussion of both `linked_list.cpp|h` and `stack.cpp|h`, and `lexer.cpp|h` will be a slightly modified version of the lexer implemented in the last chapter. The rest of this chapter will be concerned with the implementation of each of these remaining modules, with the exception of `parser.cpp|h`—it's the focus of the next chapter. Figure 14.13 depicts the layout of the compiler's modules.

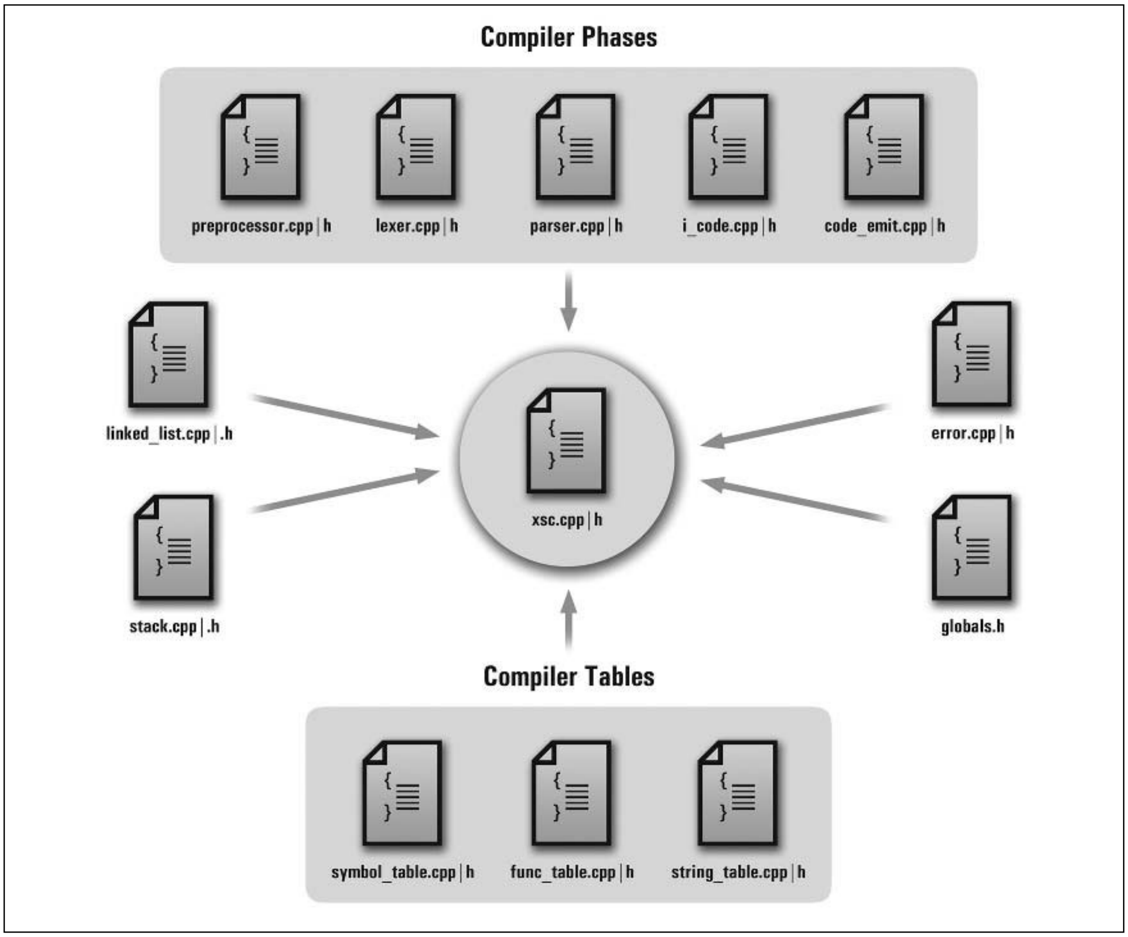


Figure 14.13

The layout of the compiler's modules.

While you're at, you might as well get `globals.h` taken out too. As Table 14.2 mentions, this just contains some basic global data that everyone needs, which really just boils down to the `TRUE` and `FALSE` macros, as well as some useful `#includes`:

```
#ifndef XSC_GLOBALS
#define XSC_GLOBALS

// ---- Include Files ---------------------------------------

    #include <stdlib.h>
    #include <stdio.h>
    #include <stddef.h>
    #include <string.h>
    #include <time.h>
    #include <process.h>

// ---- Constants -------------------------------------------

    // ---- General -------------------------------------

        #ifndef TRUE
            #define TRUE                    1   // True
        #endif

        #ifndef FALSE
            #define FALSE                   0   // False
        #endif

#endif
```

This module listing gives you something of a road map to follow in the discussion of the rest of the compiler, so let's knock them out one by one.

The Loader Module

If you recall the initial overview at the beginning of this chapter, you'll remember that the XtremeScript compiler is broken up into a front end, I-code module, and the back end. The front end contains a number of modules, the first of which is the *loader module*.

The loader module isn't explicitly defined in the file structure, because it really just boils down to a single in `xsc.cpp|h`. It's also rather simple, as you probably expect:

```
void LoadSourceFile ()
{
    // ---- Open the input file
    FILE * pSourceFile;

    if ( ! ( pSourceFile = fopen ( g_pstrSourceFilename, "r" ) ) )
        ExitOnError ( "Could not open source file for input" );

    // ---- Load the source code

    // Loop through each line of code in the file
    while ( ! feof ( pSourceFile ) )
    {
        // Allocate space for the next line
        char * pstrCurrLine = ( char * ) malloc ( MAX_SOURCE_LINE_SIZE + 1 );

        // Read the line from the file
        fgets ( pstrCurrLine, MAX_SOURCE_LINE_SIZE, pSourceFile );

        // Add it to the source code linked list
        AddNode ( & g_SourceCode, pstrCurrLine );
    }

    // ---- Close the file
    fclose ( pSourceFile );
}
```

The file is opened using the filename stored in `g_pstrSourceFilename` by `VerifyFilenames ()`. Each line of the file is then read with `fgets ()` into a locally allocated string buffer, which is then added to the `g_SourceCode` linked list with a call to `AddNode ()` (you can't use `AddString ()` here because you need to preserve duplicate lines of code—how many times does something like "++ `X;`" appear in your code?). Once the EOF is reached, the file is closed and the function exits. The result is a linked list containing each line of the source code, as shown in Figure 14.14.

Because of the nature of `fgets ()`, which returns everything from the beginning of the line until the first instance of a newline, including the newline itself, you more or less get implicit internalization in regards to a consistent newline format. If you had been reading the file character by character in a binary-safe mode, however, you have to be careful to watch for line break/newline sequences and convert them as an internal character buffer was filled. Fortunately, between the use of `fgets ()` and the line-by-line separation of the linked list, you can safely assume the handling of the newline situation is adequate.

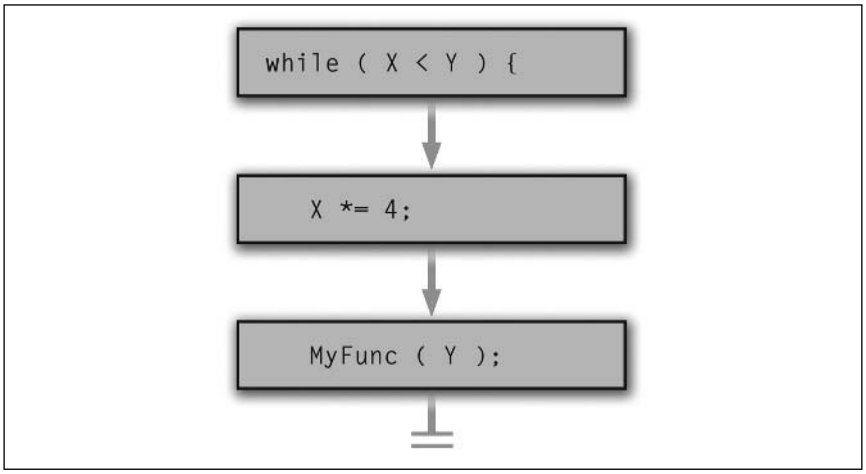


Figure 14.14

The loader populates a linked list with the source code.

At this point, you've loaded a source file into memory and are ready to go. Let's move on to see how the file will be transformed and converted as it passes through the compiler's remaining modules.

THE PREPROCESSOR MODULE

The preprocessor is the source code's first stop on its trip through the system. The preprocessor is implemented as a single function called `PreprocessSourceFile ()`, found in `preprocessor.cpp|h`. Its main job is to rid the source code linked list of both single-line and block comments.

The function begins by declaring a few flags, as well as a local `LinkedListNode` structure pointer that will be used to traverse the `g_SourceCode` linked list. It then proceeds to loop through each line of the source file, and makes a local copy of the node's string pointer. Take a look:

```
void PreprocessSourceFile ()
{
    // Are we inside a block comment?
    int iInBlockComment = FALSE;

    // Are we inside a string?
    int iInString = FALSE;

    // Node to traverse list
    LinkedListNode * pNode;
    pNode = g_SourceCode.pHead;
    // Traverse the source code
    while ( TRUE )
    {
    // Create local copy of the current line
        char * pstrCurrLine = ( char * ) pNode->pData;
```

The iInBlockComment and iInString flags are there so the preprocessor can tell at all times whether it's currently inside a string or block comment. You'll see why the former of these two flags is important in a moment, but you should already recognize the iInString flag from the StripComments () function in XASM. Remember, it's valid for a string literal to contain // or /*, so the preprocessor needs to know when it's inside a string literal in order to intelligently determine what is and isn't a comment.

At each iteration of the while loop, a for loop is started that scans through each character in the current line, looking for comments. The first order of business within this loop is updating the iIsInString flag:

```
for ( int iCurrCharIndex = 0; iCurrCharIndex < ( int ) strlen ( pstrCurrLine );
    ++ iCurrCharIndex )
{
    // If the current character is a quote, toggle the in string flag
    if ( pstrCurrLine [ iCurrCharIndex ] == '"' )
    {
        if ( iInString )
            iInString = FALSE;
        else
            iInString = TRUE;
    }
```

At this point, you have the current character of the current line of code and know whether you're inside a string. You're all set to nuke some comments.

Single-Line Comments

The first catch of the day will be single-line comments, denoted with the //. The basic strategy here is this: whenever a / character is found, read the character immediately following it to find out if it's a / as well. If so, the // token has been found, which denotes the beginning of a multi-line comment. Before proceeding, make sure the iInString and iInBlockComment flags are both FALSE. If so, replace the first / with a null terminator, thus terminating the string at the start of the comment. This, for example, will convert the following line:

```
ScreenX = X / Z;    // Get the screen space coordinate of X
```

to this:

```
ScreenX = X / Z;
```

Of course, there's still the whitespace in between the semicolon and the start of the former comment, but that obviously doesn't matter. Here's the code:

```
// Check for a single-line comment, and terminate the rest
// of the line if one is found
if ( pstrCurrLine [ iCurrCharIndex ] == '/' &&
     pstrCurrLine [ iCurrCharIndex + 1 ] == '/' &&
     ! iInString && ! iInBlockComment )
{
    pstrCurrLine [ iCurrCharIndex ] = '\n';
    pstrCurrLine [ iCurrCharIndex + 1 ] = '\0';
    break;
}
```

Block Comments

Block comments allow both multi-line blocks of code to be commented out, as well as individual character strings within a given line. They start with an opening /* token and end with */. The strategy behind removing them from the source code is a bit more brute-force oriented than were single-line comments, but it's very easy.

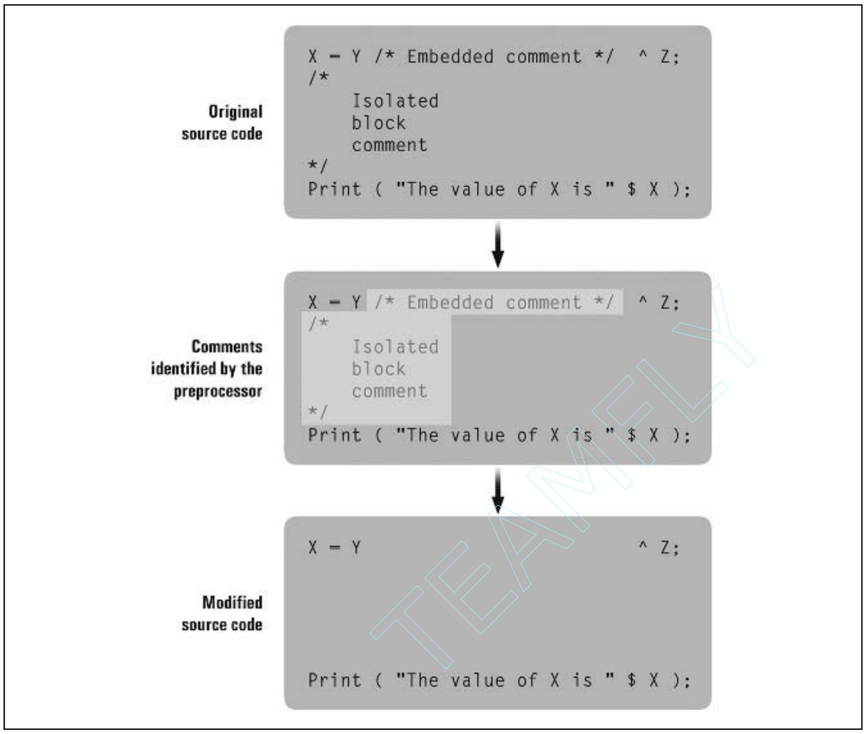
You could start by replacing the /* with a null terminator, just like you did with //, but that would only take out the first line in a potentially multi-line block. Furthermore, not all block comments are meant to comment out the entire remainder of the line. For example, this is a valid comment:

```
U = V /* Comment */ + W;
```

Although this certainly calls the coder's style into question, it's still valid according to syntax. Replacing the /* with a line break would result in this:

```
U = V
```

This is not only different than what the coder wanted, but is syntactically illegal. So, a better solution is to set a flag when the opening /* is reached, and replace each character starting from that index with a space until the closing */ is found. Although this doesn't actually remove the space taken up by the comment, it replaces it with harmless whitespace, so the effect is the same overall. Figure 14.15 demonstrates the identification and deletion of block comments by the preprocessor.



```
X = Y /* Embedded comment */  ^ Z;
/*
    Isolated
    block
    comment
*/
Print ( "The value of X is " $ X );
```

```
X = Y                         ^ Z;



Print ( "The value of X is " $ X );
```

Figure 14.15

The preprocessor identifying and deleting block comments.

TIP

If you really want to physically remove comments, the algorithm is conceptually simple but might be a bit tricky to implement. The key is understanding that block comments can result in a number of different "line types". The first line type is just like the single-line comment; a /* that opens up somewhere within the line extends all the way to the end. This is handled just like //—by replacing it with a null terminator. The next case is a line that is entirely contained with a larger block comment. In this case, `DelNode ()` can be used to dispose of it entirely. The next type is a comment that ends on the current line but starts on a previous one; in this case, the first character *after* the closing */ is considered the new first character of the line, which means the string has to be shifted to the left until the space taken up by the comment is entirely overwritten. A null terminator is then placed after the last character of the original string to free the garbage characters left over on the right side. Lastly is a line type wherein the block comment starts and ends on the same line. In this case, the process is similar to the last case—starting at the closing */, shift every character over to the left until it reaches the opening /*, and insert a null terminator after the last non-garbage character of the new string to clear off the now unused right side.

Here's the code for replacing a block comment with whitespace:

```
// Check for a block comment
if ( pstrCurrLine [ iCurrCharIndex ] == '/' &&
     pstrCurrLine [ iCurrCharIndex + 1 ] == '*' &&
     ! iInString && ! iInBlockComment )
{
    iInBlockComment = TRUE;
}

// Check for the end of a block comment
if ( pstrCurrLine [ iCurrCharIndex ] == '*' &&
     pstrCurrLine [ iCurrCharIndex + 1 ] == '/' &&
     iInBlockComment )
{
    pstrCurrLine [ iCurrCharIndex ] = ' ';
    pstrCurrLine [ iCurrCharIndex + 1 ] = ' ';
    iInBlockComment = FALSE;
}

// If we're inside a block comment, replace the
// current character with whitespace
if ( iInBlockComment )
{
    if ( pstrCurrLine [ iCurrCharIndex ] != '\n' )
        pstrCurrLine [ iCurrCharIndex ] = ' ';
}
```

Whenever a / is read, the next character is read to determine whether it's a *. If it is, and the block comment and string flags are both `FALSE`, the `iInBlockComment` flag is set. If a * is read, and the character immediately following it is /, and the `iInBlockComment` flag *is* set, the flag is cleared and the two characters composing the */ are replaced with whitespace. Otherwise, the `iInBlockComment` is checked for any other character; if it's set, the character is replaced with whitespace.

NOTE

Notice that the character at `iCurrCharIndex + 1` is read without any checks to make sure the index isn't beyond the end of the string. You can do this safely, because you're only looping through the string from index zero to the length of the string minus one, as returned by `strlen ()`. Because of this, even if you were to read a / on the very last character in the string, `iCurrCharIndex + 1` would point to the \0 character immediately following it, and therefore still be a safe operation.

Preprocessor Directives

The language specification from Chapter 7 included two preprocessor directives, `#include` and `#define`. `#include` replaces itself with the contents of the file it specifies, whereas `#define` defines a symbolic constant and assigns it a value. The preprocessor then scans over the entire source code and replaces all instances of the symbol's name with the specified value.

I've decided to leave the implementation of the preprocessor directives to you, as an intermediate-level challenge. Handling these directives is a lot simpler than it may sound, and requires only the skills you've already learned. To get you started though, let's take a quick look at some implementation ideas.

Implementing `#include`

The primary principal behind `#include` is that the contents of whatever file it specifies is used to physically replace the directive. Once the preprocessor is done, there shouldn't be any trace that an `#include` directive was ever there.

The fact that the source code is stored as a linked list makes `#include` remarkably easy in a lot of ways. For example, removing the `#include` directive from the source code is as easy as deleting its node, whereas adding the newlines of the file is as easy as inserting new nodes just before the node containing the line that immediately follows the `#include` directive's line.

In order to make this work, a new function must be added to the linked list implementation, perhaps called `InsertNode ()`. `InsertNode ()` is a lot like `AddNode ()`, except that it accepts a `LinkedListNode` structure pointer in addition to the data pointer. The node pointed to by the node pointer is found, and the new node is *inserted* into the list either directly in front of or directly behind it. Inserting a node is similar to deleting a node in the sense that you have to patch up the pointers that bind a node to its next node, and have to be on the lookout for the special cases of the head and tail nodes.

Once you can insert a node, the next challenge is parsing the `#include` line. Fortunately, you can use the lexer designed in the last chapter for this. By passing the source code through the lexer in a preprocessing stage, and adding a new token type, `TOKEN_TYPE_PREP_INCLUDE`, perhaps, you can scan the source file for include directives. Then, as long as a `TOKEN_TYPE_STRING` token immediately follows, you've got a valid directive and can use the string lexeme as the filename.

Open the specified file and read each line of code. As each line is read, use `InsertNode ()` to insert them just *after* the `#include` line. Once the file is fully loaded, use the pointer to the `#include` line to delete it with `DelNode ()`. Figure 14.16 summarizes the job of the `#include` directive.

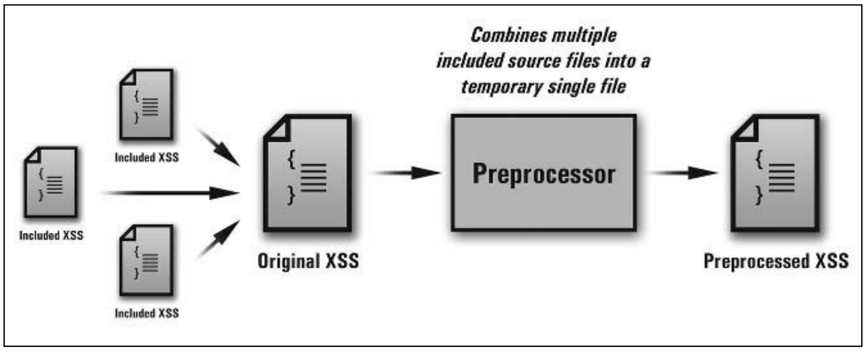


Figure 14.16

The `#include` directive in action.

Nested `#include` Directives

The only caveat left is the issue of nested `#include` directives, wherein the file you're including includes files of its own. Because this is a vital feature of file inclusion directives, it's important to support this feature.

I personally think the best way to solve this issue is to make the `#include` directive's handler function recursive. As it's reading the file, allow it to scan each line for `#include` as well, and call itself in the event that it finds one. The recursively called function will then open the next include file and begin inserting its lines as well.

There is the issue of which files are nested, however. The same file should never be included more than once, for example—this can lead to both wasted memory and compile time errors if variables and functions are declared multiple times as a result. There are far more dire consequences of improper use of the directive as well; imagine if a file attempts to include itself, or if two files include each other. In these cases, the compiler will hang until it either runs out of stack space from too many recursive calls, or runs out of heap space from the source code growing too large. To prevent these situations from happening, I suggest keeping a record of the filenames (including paths) of all included files. This way, whenever a new `#include` is encountered, its specified file can be checked against those already loaded, and the entire directive can be ignored if a match is found.

Implementing `#define`

`#define` is somewhat similar to `#include`, in the sense that it involves replacing instances of a directive with other data. This process is known as *macro expansion*, and was used in C to define symbolic constants until C++ introduced the `const` keyword (although anyone still programming in pure C uses traditional macros, of course).

The implementation of `#define` is a bit more in-depth than `#include`. Let's first review its syntax. Although C's `#define` is capable of macros that span multiple lines and even accept parameters, this version of `#define` is relegated to symbolic constants that map a single-line string to an identifier, like these:

```
#define MY_NAME    "Alex"
#define PI         3.14159
#define BEGIN      {
#define END        }
```

The first step in implementing this directive is making sure to record each macro in a table as it's encountered. A hash table structure would come in handy here, as it's really just a matter of storing them in *key:value* form. The macros' identifiers, like `MY_NAME` and `PI` are the keys, whereas the values, like `"Alex"` and `3.14159`, are the values.

Parsing the `#define` line itself is easy; just as you added `TOKEN_TYPE_PREP_INCLUDE` to the lexer in the last section, you can add `TOKEN_TYPE_PREP_DEFINE` so the lexer will automatically notice `#define` and return it as such. Once a `#define` token is found, the next lexeme, no matter what it is, is the macro's value. Simply read it, add it to the table, and move on.

Now, as each line is read, the macro's identifier (the key) needs to be specifically searched for. To do this, feed each lexeme returned by the lexer to a function that uses it as a search key in the macro table. If a match is found, the value associated with that key needs to replace it on the line.

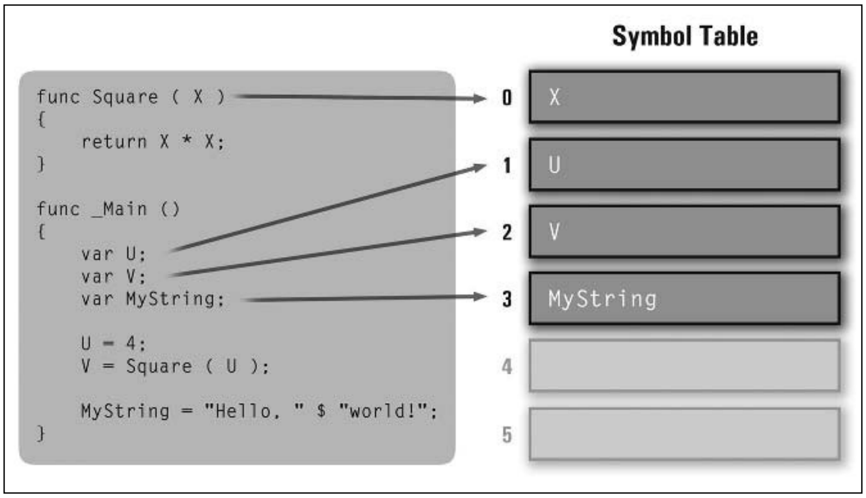
This is the tricky part. One simple approach is to simply allocate a new string, `MAX_SOURCE_LINE_SIZE` in length, and use it to piece together a new line of code based on the old line and macro's value. First, read every character up until the macro lexeme, and add it to the newly allocated line. Now, dump the macro's value directly into the source code immediately following the characters you just added. Finally, resynchronize your pointer within the old source line so that it lies just *after* the macro identifier, and append the remainder of the old line to the new one. You can then delete the old node and replace it with the new one. The only problem here is that it quickly becomes an inefficient solution when a single line contains more than one or two macros, because you're constantly freeing and allocating large character blocks, as well as performing costly string copy operations. It works, though, and because I can assure you that a script compiler will rarely need to worry about performance, there's nothing wrong with it.

The Compiler's Tables

In order to properly parse and understand the script's source code, the compiler maintains a number of tables. As you've seen, these tables are all based on the linked list structure developed earlier, and they are further enhanced by a specific interface of functions that allows them to be accessed and manipulated easily.

The Symbol Table

As the source code is parsed, perhaps the most obvious collection of data that needs to be organized, maintained, and tracked is the script's variables and arrays (see Figure 14.17). As you can imagine, high-level programming wouldn't get very far without them, so it's a logical place to start.



```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

Figure 14.17

The symbol table tracking the script's variables.

The symbol table is implemented in `symbol_table.cpp|h`, and provides a number of functions to make the otherwise pure linked-list implementation easier to work with.

The `SymbolNode` Structure

You've already declared the `g_SymbolTable` linked list, but each node in that list needs a data member. Each symbol table node will be embodied by the `SymbolNode` structure:

```
typedef struct _SymbolNode          // A symbol table node
{
    int iIndex;                     // Index
    char pstrIdent [ MAX_IDENT_SIZE ];      // Identifier
    int iSize;                      // Size (1 for variables, N
                                    // for arrays)
    int iScope;                     // Scope (0 for globals, N
                                    // for locals' function index)
    int iType;                      // Symbol type (parameter
                                    // or variable)
}
    SymbolNode;
```

As will be the case with most node structures, the first field is an integer index called `iIndex`. The reason you need an explicit field for this, as opposed to simply basing a node's index on its physical position within the list, is to prepare for the possibility of the lists order changing arbitrarily. If this were to happen for whatever reason, it would be helpful if its existing nodes were able to retain their indexes, because that's what they are known by.

Next up are the obvious fields: the identifier and size. The symbol's identifier is stored in the statically allocated `pstrIdent` string, whose length is stored in the `MAX_IDENT_SIZE` constant:

```
#define MAX_IDENT_SIZE          256
```

As usual, I've chosen overkill over sensibility because it really doesn't matter either way and I always like to err on the side of too much. In this specific case, however, I got the 256-character figure from Java; the `javac` compiler imposes the same limit on its identifiers.

A symbol's size is measured in XVM stack indexes, and because the XtremeScript language is so strongly typeless, this means that all non-array variables occupy a single stack index in all cases, and therefore have a size of 1. Arrays, because they're simply an aggregate of single-index variables, are measured by the same scale and range from 1 to *N*.

The `iScope` field tracks a variable's scope; in other words, where it can be referenced. Because XtremeScript doesn't support classes, structures, or nested functions, a symbol can have only one of two scopes—the global scope, or the local scope of a particular function. In the first case, `iScope` is set to zero, which is a special flag that marks the symbol as a global. In the case of local variables and arrays, the `iScope` field is set to the function's index in the function table. If you recall Chapter 9, you'll recognize this as the same scheme used to track a variable's scope in XASM.

Last up is `iType`, which is used to track the *type* of the symbol. In the case of XtremeScript, this boils down to one of two things—variables (which include arrays as well, and are independent of scope) or parameters (which can only be single variables, and are highly dependent on scope). Because a parameter can often be thought of simply as just another local variable within a function, the same symbol table is used to store them. The only difference is that their `iType` flag is set to reflect their status as parameters.

To make things easier to work with, `symbol_table.h` defines a few constants for making a variable's settings more symbolic. For example, the `iType` field of all globals is zero, so I provided the `SCOPE_GLOBAL` constant, which is of course set to zero, to add a bit of readability to the process of dealing with globals. Second, I could've simply used `TRUE` and `FALSE` to represent whether a variable is a parameter, but this is not only less readable, but also more or less cuts off the possibility of adding additional symbol types later. To remedy this, I defined the `SYMBOL_TYPE_VAR` and `SYMBOL_TYPE_PARAM` constants.

```
#define SCOPE_GLOBAL                    0

#define SYMBOL_TYPE_VAR                 0
#define SYMBOL_TYPE_PARAM               1
```

The Interface

The symbol table interface is more or less what you expect—it provides a function for adding a new symbol, retrieving symbols based on their indexes and identifiers, and so on.

Adding Symbols

Because the first and most vital operation as the compiler slowly lifts off the ground will be adding a symbol to the table, let's look at the `AddSymbol ()` function, which does just that:

```
int AddSymbol ( char * pstrIdent, int iSize, int iScope, int iType )
{
    // If a label already exists
    if ( GetSymbolByIdent ( pstrIdent, iScope ) )
        return -1;

    // Create a new symbol node
    SymbolNode * pNewSymbol = ( SymbolNode * )
        malloc ( sizeof ( SymbolNode ) );

    // Initialize the new label
    strcpy ( pNewSymbol->pstrIdent, pstrIdent );
    pNewSymbol->iSize = iSize;
    pNewSymbol->iScope = iScope;
    pNewSymbol->iType = iType;

    // Add the symbol to the list and get its index
    int iIndex = AddNode ( & g_SymbolTable, pNewSymbol );

    // Set the symbol node's index
    pNewSymbol->iIndex = iIndex;

    // Return the new symbol's index
    return iIndex;
}
```

The first thing the function does is call GetSymbolByIdent () to find out if the symbol already exists. I haven't covered this function yet, so rest assured that it does just what it says—returns a pointer to the symbol matching the specified identifier if one was found, and returns NULL otherwise. If this function returns a valid pointer, it means the symbol already resides in the table and -1 is returned to the caller of AddSymbol () to alert them.

If this first test passes, the symbol's SymbolNode structure is allocated and initialized. The identifier is copied into the string, the size, scope and type is set, and AddNode () is called to add the completed symbol node to the list. The returned index is then used to set the symbol node's iIndex field, and is also returned to the caller.

Retrieving Symbols

You just witnessed the necessity of a function that returns the pointer to a symbol's SymbolNode structure based on its identifier, so let's define it next:

```
SymbolNode * GetSymbolByIdent ( char * pstrIdent, int iScope )
{
    // Local symbol node pointer
    SymbolNode * pCurrSymbol;

    // Loop through each symbol in the table to find the match
    for ( int iCurrSymbolIndex = 0;
          iCurrSymbolIndex < g_SymbolTable.iNodeCount;
          ++ iCurrSymbolIndex )
    {
    // Get the current symbol structure
        pCurrSymbol = GetSymbolByIndex ( iCurrSymbolIndex );

    // Return the symbol if the identifier and scope matches
        if ( pCurrSymbol && stricmp ( pCurrSymbol->pstrIdent, pstrIdent )
            == 0 && pCurrSymbol->iScope == iScope )
            return pCurrSymbol;
    }

    // The symbol was not found, so return a NULL pointer
    return NULL;
}
```

The function's main purpose is traversing the symbol table. Once again, however, you find a call to an as-of-yet undefined function, this time called GetSymbolByIndex (). This function does the same thing as GetSymbolByIdent (), except it returns the symbol corresponding to the specified index (obviously). Once the symbol has been read from the table, its identifier is compared to the specified one, as well as its scope. If a match is found, the structure is returned; otherwise, NULL is returned.

Moving on, the next function is GetSymbolByIdent (), which does almost the same job, and was referenced by the last function:

```
SymbolNode * GetSymbolByIndex ( int iIndex )
{
    // If the table is empty, return a NULL pointer
    if ( ! g_SymbolTable.iNodeCount )
        return NULL;

    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = g_SymbolTable.pHead;

    // Traverse the list until the matching structure is found
    for ( int iCurrNode = 0; iCurrNode < g_SymbolTable.iNodeCount; ++ iCurrNode )
    {
    // Create a pointer to the current symbol structure
        SymbolNode * pCurrSymbol = ( SymbolNode * ) pCurrNode->pData;

    // If the indexes match, return the symbol
        if ( iIndex == pCurrSymbol->iIndex )
            return pCurrSymbol;

    // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The symbol was not found, so return a NULL pointer
    return NULL;
}
```

This function works in a familiar manner. Using a symbol node, it traverses the list by jumping from pointer to pointer until the matching index is found. Upon the discovery of a match, the corresponding pointer is returned. If a match is not found, NULL is returned.

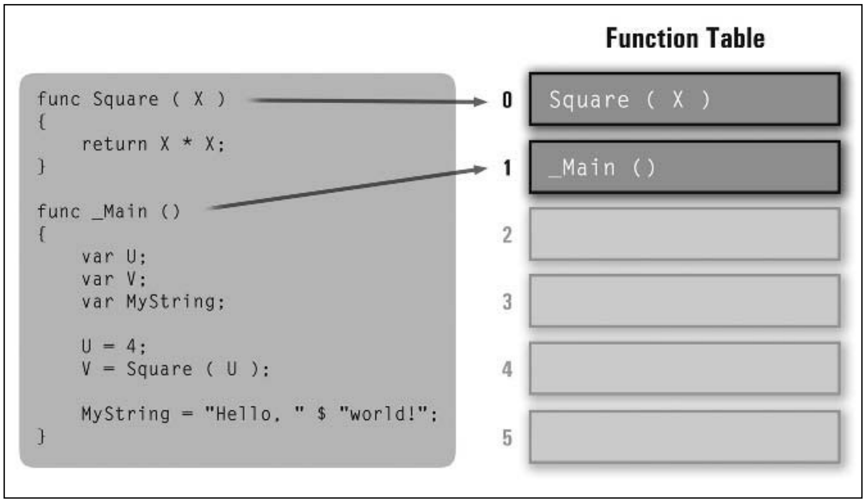
Lastly, there's one more function worth mentioning. As you'll see when you write the parser, it can be useful to get a variable's size quickly and easily (for example, when it needs to be verified that the specified identifier is indeed an array). In these cases, GetSizeByIdent () is called—pass it the variable's identifier, and it returns its size:

```
int GetSizeByIdent ( char * pstrIdent, int iScope )
{
    // Get the symbol's information
    SymbolNode * pSymbol = GetSymbolByIdent ( pstrIdent, iScope );

    // Return its size
    return pSymbol->iSize;
}
```

Pretty simple, huh? With one call to GetSymbolByIdent (), it has the symbol. It returns its iSize field and calls it a day.

The Function Table

The function table is very similar to the symbol table in most respects, so this section should be pretty easy if you understood how symbols were dealt with. As was shown in Table 14.2, the function table is implemented in function_table.cpp|h and tracks the script's functions (see Figure 14.18).



```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

Figure 14.18

The function table tracking the script's functions.

The FuncNode Structure

Just as symbols needed a separate structure to store each of their nodes, so do functions:

```
typedef struct _FuncNode            // A function table node
{
    int iIndex;                     // Index
    char pstrName [ MAX_IDENT_SIZE ];   // Name
    int iIsHostAPI;                 // Is this a host API
                                    // function?
    int iParamCount;                // The number of accepted
                                    // parameters
    LinkedList ICodeStream;         // Local I-code stream
}
    FuncNode;
```

In a lot of ways it's similar to the SymbolNode structure; its first field is an explicit index, and its second is an identifier string the size of MAX_IDENT_SIZE. Up next is iIsHostAPI. As I mentioned earlier, the XtremeScript compiler doesn't maintain a separate function table for host API calls; rather, both host and script functions are stored in the same table and differentiated based on this flag. You'll learn more about how host API calls work in the high-level XtremeScript language in the next chapter.

The next parameter is iParamCount, which of course stores the number of parameters the function accepts. Unlike XASM, which had no way to determine how many parameters a function was being passed (because they were all handled with separate Push instructions), the XtremeScript compiler is explicitly told which parameters are being passed to each function. iParamCount helps the compiler validate them.

Lastly, there's a nested linked list called ICodeStream. I'll talk about this in far greater depth later in the chapter, but for now, all you need to know is that this is where the function's I-code is stored. Remember, because a valid XVM assembly script has *no* code outside of functions, there's no need for a global I-code stream. Rather, each function has its own "local" block of I-code.

The Interface

Continuing with the parallels, the interface to the function table will of course bear a striking resemblance to symbol table. Right off the bat you'll have functions for adding functions to the table, reading them based on their names, indexes, and so on.

Adding Functions

Let's start at the beginning, with the predictably titled AddFunc ():

```
int AddFunc ( char * pstrName, int iIsHostAPI )
{
    // If a function already exists with the specified name,
    // exit and return an invalid index

    if ( GetFuncByName ( pstrName ) )
        return -1;

    // Create a new function node
    FuncNode * pNewFunc = ( FuncNode * ) malloc ( sizeof ( FuncNode ) );

    // Set the function's name
    strcpy ( pNewFunc->pstrName, pstrName );

    // Add the function to the list and get its index, but add
    // one since the zero index is reserved for the global scope
    int iIndex = AddNode ( & g_FuncTable, pNewFunc ) + 1;

    // Set the function node's index
    pNewFunc->iIndex = iIndex;

    // Set the host API flag
    pNewFunc->iIsHostAPI = iIsHostAPI;

    // Set the parameter count to zero
    pNewFunc->iParamCount = 0;

    // Clear the function's I-code block
    pNewFunc->ICodeStream.iNodeCount = 0;

    // If the function was _Main (), set its flag and index in the header
    if ( stricmp ( pstrName, MAIN_FUNC_NAME ) == 0 )
    {
        g_ScriptHeader.iIsMainFuncPresent = TRUE;
        g_ScriptHeader.iMainFuncIndex = iIndex;
    }

    // Return the new function's index
    return iIndex;
}
```

The basic strategy here is just the same as it was in `AddSymbol ()`:

- Determine whether the function being added is already in the table. If so, return the existing node's index to the caller (note I haven't covered `GetFuncByName ()` yet).
- Allocate a `FuncNode` structure and initialize it based on the parameters passed.
- Add the node to the table.
- Return the index to the caller.

And, as you can see, this is more or less what happens. The only extra detail worth covering is the issue of the `_Main ()` function. Just as it was in XASM, it's important to track both the presence and index of `_Main ()`, because it has exceptional properties that require special treatment on behalf of the compiler. To this end, the function closes with a comparison of the specified function name and a constant called `MAIN_FUNC_NAME`, which looks like this:

```
#define MAIN_FUNC_NAME        "_Main"
```

If the comparison results in a match, the `_Main ()` function has been found, so the script header's `iIsMainFuncPresent` field is set to `TRUE`, and the `iMainFuncIndex` field is set to whatever index `AddNode ()` returned.

Speaking of `AddNode ()`'s index, it's *very* important to note that you add one to it. Why? Because, if you remember, the zero index of the function table is reserved for the global scope. For example, the `SymbolNode` structure uses a single field to determine both the scope of a symbol, as well as its index into the function table in the event that it's global. In order for this to work, the zero index can't be associated with any specific function.

Retrieving Functions

Because the last function made a call to `GetFuncByName ()` to determine whether the new function was already in the table, I should cover this one next:

```
FuncNode * GetFuncByName ( char * pstrName )
{
    // Local function node pointer
    FuncNode * pCurrFunc;

    // Loop through each function in the table to find the match
    for ( int iCurrFuncIndex = 1; iCurrFuncIndex <= g_FuncTable.iNodeCount;
        ++ iCurrFuncIndex )
    {
    // Get the current function structure
        pCurrFunc = GetFuncByIndex ( iCurrFuncIndex );
    // Return the function if the name matches
        if ( pCurrFunc && stricmp ( pCurrFunc->pstrName, pstrName ) == 0 )
           return pCurrFunc;
    }

    // The function was not found, so return a NULL pointer
    return NULL;
}
```

Again, just as was the case with the symbol table interface, this function is making repeated calls to `GetFuncByIndex ()` as it iterates through the function table. As each node is read, its `pstrName` field is compared to the specified name to determine a match.

Continuing along the food chain, `GetFuncByName ()` called `GetFuncByIndex ()`. Let's have a look:

```
FuncNode * GetFuncByIndex ( int iIndex )
{
    // If the table is empty, return a NULL pointer
    if ( ! g_FuncTable.iNodeCount )
        return NULL;

    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = g_FuncTable.pHead;

    // Traverse the list until the matching structure is found
    for ( int iCurrNode = 1; iCurrNode <= g_FuncTable.iNodeCount;
        ++ iCurrNode )
    {
    // Create a pointer to the current function structure
        FuncNode * pCurrFunc = ( FuncNode * ) pCurrNode->pData;

    // If the indexes match, return the current pointer
        if ( iIndex == pCurrFunc->iIndex )
            return pCurrFunc;

    // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The function was not found, so return a NULL pointer
    return NULL;
}
```

At this point, the function should be entirely self-explanatory. A local node pointer is used to traverse the list, node by node, until a match is found by comparing the specified index to each node's `iIndex` field. In the event of a match, the pointer is returned; otherwise, `NULL` is returned when the end of the loop is reached.

Updating a Function's Parameter Count

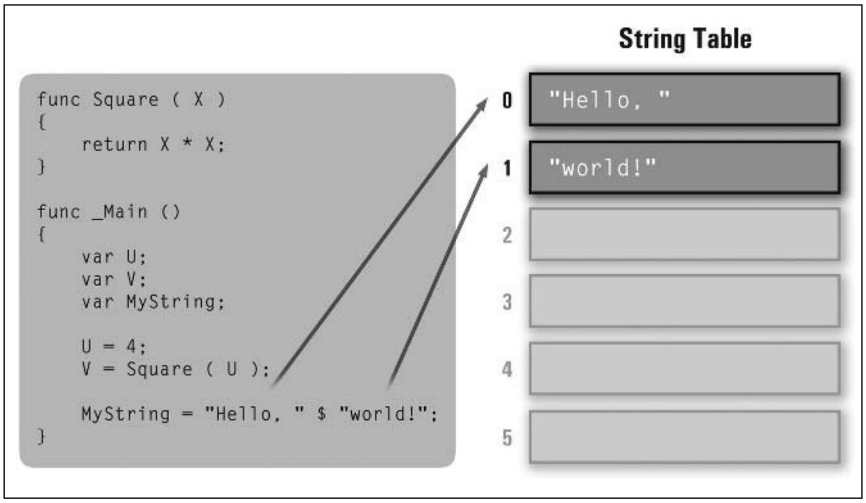
There's one last function to describe to complete the function table's interface, so let's knock it out. It's called `SetFuncParamCount ()`, and is used to set the parameter count of a function table that already exists in the table:

```
void SetFuncParamCount ( int iIndex, int iParamCount )
{
    // Get the function
    FuncNode * pFunc = GetFuncByIndex ( iIndex );

    // Set the parameter count
    pFunc->iParamCount = iParamCount;
}
```

Using `GetFuncByIndex ()`, this function grabs the function from the table based on its index and sets its parameter count to the specified value. Although this function will make the most sense once you reach the parser, it should be easy enough to understand after having done virtually the same thing in XASM with the `SetFuncInfo ()` function. A function is immediately added to the table when its found—even before its parameter list is parsed—so you need a separate function for adding the parameter count retroactively.

The String Table

The string table is barely a table onto itself; its implementation is solely based on the vanilla linked list covered earlier. There's no need to create any extra structures for maintaining its nodes, because each node's data member is simply a raw string. And you've already got `AddString ()`, which not only adds strings to the specified list, but also automatically filters out duplicates. Because you've seen the implementation of both the linked list and the `AddString ()` and `GetStringByIndex ()` functions already, there's nothing left to cover here. Figure 14.19 once again demonstrates the string table.



```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;
    var MyString;

    U = 4;
    V = Square ( U );

    MyString = "Hello, " $ "world!";
}
```

Figure 14.19

The string table tracks the script's string literal values.

Integrating the Lexical Analyzer Module

The last chapter saw you through the design and implementation of a lexical analyzer capable of lexing the entire XtremeScript language. Although the lexer you built was complete, the issue of integrating it smoothly with the compiler framework you're building in this chapter is still significant.

Rewinding the Token Stream

In the last chapter, the lexer's only job was to read the next token from the character stream and spit it out. Things aren't so cut and dried in the XtremeScript compiler, however—for example, you may want to read the look-ahead character like you did in XASM in Chapter 9 (which Ill come back to in a moment). You may need to take even more drastic action, by reading an entire token from the stream and "putting it back" if we decide it's not what we thought it was going to be. As you can imagine, reading a token and putting it back is almost like using the look-ahead character; it allows you to find out what lies beyond the current token without permanently disturbing the stream.

As you'll see during the implementation of the parser in the next chapter, this capability to read a token and later restore the stream to the status it held before the token was read is invaluable in certain situations. This process is called "rewinding the token stream," and is illustrated in Figure 14.20.

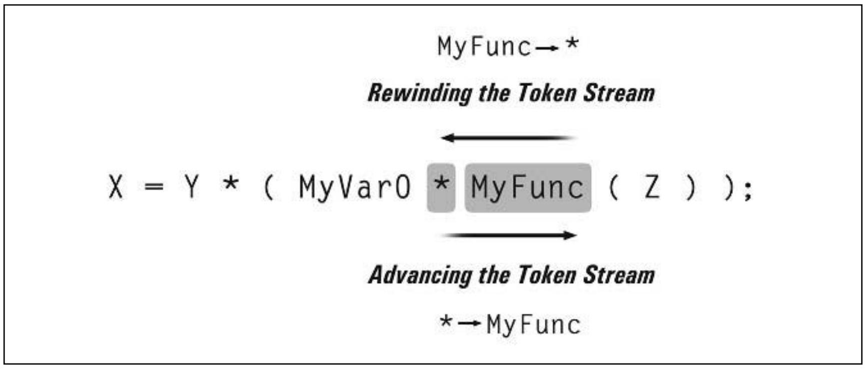

MyFunc→*
Rewinding the Token Stream
X = Y * (MyVar0 * MyFunc (Z));
Advancing the Token Stream
*→MyFunc

Figure 14.20

Rewinding the token stream versus advancing it.

Lexer States

So how is the token stream rewound? The key to understanding how it works is to simply realize that at any given time, the lexer is in a particular "state" (not to be confused with the states of the state machine in `GetNextToken ()`). By "state", I mean the lexeme stream contains a specific lexeme, the current token contains a specific token code, and the lexeme pointers within the current line of code are pointing to specific locations (among many other things).

Whenever a new token is read, these values are disturbed; the lexeme string is overwritten with a new one, the token code is updated, and the lexeme pointers advance through the current line by a certain amount. It seems, then, that an easy way to "rewind" the token stream is simply to save the state of each of these variables *before* `GetNextToken ()` is called. This way, if it's later decided that reading the token was a mistake, the state of the lexer before the read occurred can be restored simply by reading the saved variables. Of course, unless an array or other aggregate structure is used, this means the stream can only be rewound once per token read. Fortunately, this won't pose a problem.

In order to save the lexer's state, your first reaction might simply be to duplicate each of the lexer's globals, like this, for example:

```
// ---- Main
char g_pstrCurrLexeme [ MAX_LEXEME_SIZE ];   // Current lexeme
char g_pstrPrevLexeme [ MAX_LEXEME_SIZE ];

// ---- Current Lexeme
int g_iCurrLexemeStart;      // Current lexeme's starting index
int g_iCurrLexemeEnd;        // Current lexeme's ending index
int g_iPrevLexemeStart;
int g_iPrevLexemeEnd;
```

```
// ---- Operators
int g_iCurrOp;                // Current operator
int g_iPrevOp;
```

As you can see by the bold code, each variable has been duplicated and prefixed with `Prev`. Now, `GetNextToken ()` can save each of the `Curr` versions to the `Prev` versions of the variable, like `g_iCurrLexemeStart` to `g_iPrevLexemeStart`, for example. Once this is done, the caller then has the option of rewinding the stream by moving `g_iPrevLexemeStart` back into `g_iCurrLexemeStart`, along with the rest of them.

Although this solution works, there's a lot of redundancy going on. Although it's true that each variable does need a duplicate, or backup, in order to preserve the state long enough to facilitate a rewinding of the stream, there's a better way to do this than by applying brute force and just duplicating everything. Specifically, it would be better to just wrap everything in a single `struct`, and then make both a "current" and "previous" instance of that structure. All of the original `Curr` variables can be wrapped in the `LexerState` structure, like this:

```
typedef struct _LexerState          // The lexer's state
{
    char pstrCurrLexeme [ MAX_LEXEME_SIZE ];    // Current lexeme
    int iCurrLexemeStart;           // Current lexeme's
                                    // starting index
    int iCurrLexemeEnd;             // Current lexeme's ending
                                    // index
    int iCurrOp;                    // Current operator
}
    LexerState;
```

Now, you can simply instantiate this structure as many times as you want and be done with it. The unwieldy collection of globals from the original example can now be replaced with just two:

```
LexerState g_CurrLexerState;
LexerState g_PrevLexerState;
```

Furthermore, by writing a function that will copy each of the fields from one `LexerState` structure to another, you can save and restore the lexer's state easily. Of course, this means that any reference to a global variable in the lexer's code from now on will be prefixed by one of the lexer state structures. For example, this:

```
++ g_iCurrLexemeStart;
```

becomes this:

```
++ g_CurrLexerState.iCurrLexemeStart;
```

Now that you can arbitrarily instantiate lexer states at will, an important operation will be copying the contents of one state to another. This is facilitated with the CopyLexerState () function, which accepts two LexerState pointers and copies one into the other:

```
void CopyLexerState ( LexerState & pDestState, LexerState & pSourceState )
{
    // Copy each field individually to ensure a safe copy
    strcpy ( pDestState.pstrCurrLexeme, pSourceState.pstrCurrLexeme );
    pDestState.iCurrLexemeStart = pSourceState.iCurrLexemeStart;
    pDestState.iCurrLexemeEnd = pSourceState.iCurrLexemeEnd;
    pDestState.iCurrOp = pSourceState.iCurrOp;
}
```

Naturally, there's not much to explain. Each field is copied from pSourceState to pDestState. The best part is, with this function finished, you can rewind the token stream in a single line. Here's the RewindTokenStream () function:

```
void RewindTokenStream ()
{
    CopyLexerState ( g_CurrLexerState, g_PrevLexerState );
}
```

Pretty simple, huh? This can be called at any time after calling GetNextToken () to restore the lexer to the state it was in before the call. Remember, though, that because you have only one previous state instance, the token stream can only be rewound once per token read. Of course, none of this matters if GetNextToken () doesn't take advantage of it, so let's add a call to CopyLexerState () to the top of the function:

```
Token GetNextToken ()
{
    // Save the current lexer state for future rewinding
    CopyLexerState ( g_PrevLexerState, g_CurrLexerState );
```

Locked and loaded.

A New Source Code Format

One of the most significant differences between the lexer in the last chapter and the version you're adapting to work with the XtremeScript compiler is the format of the source code. The original demo was so simplistic that all it really needed was a large character string to work with. XtremeScript, because of its greater complexity, functions better with a linked list wherein each node represents a separate line from the original source file. Getting the lexer to work with this new format will be the next challenge to face.

If you recall, the lexer in the last chapter relied heavily on a function called `GetNextChar ()`. At any time, this function could be called to both read and return the next character from the source buffer, but would automatically increment the lexeme end pointer as well so that the next call would return the next character in the string. By calling this function repeatedly, the entire source buffer was scanned by the lexer.

Although you could spend all night rigging `GetNextToken ()` itself to handle the new linked list structure, you would be much smarter to simply add the new functionality to `GetNextChar ()`. This way, `GetNextToken ()` can remain *completely* the same—the new underlying method of source code storage will remain entirely transparent. Figure 14.21 illustrates this concept of abstracting the underlying storage method of the source code by isolating the logic in `GetNextChar ()`.

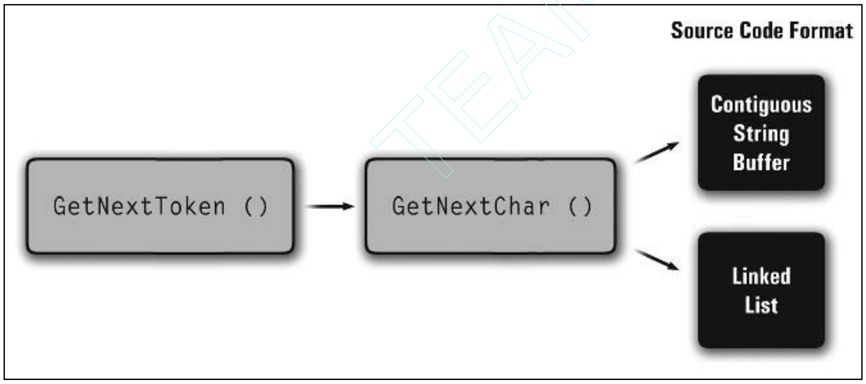


Figure 14.21

By isolating the logic behind reading the next character in `GetNextChar ()`, the rest of the lexer can remain oblivious to the underlying storage method.

The first thing you have to do in order to make this work is add some new fields to the `LexerState` structure. Namely, you need a pointer to the current source line in the `g_SourceCode` linked list at all times. It will also help, for error-handling purposes, to keep the current line number on hand. Here's the new structure layout, with the added fields in bold:

```
typedef struct _LexerState          // The lexer's state
{
    int iCurrLineIndex;             // Current line index
    LinkedListNode * pCurrLine;     // Current line node
                                    // pointer
    char pstrCurrLexeme [ MAX_LEXEME_SIZE ];    // Current lexeme
    int iCurrLexemeStart;           // Current lexeme's
                                    // starting index
    int iCurrLexemeEnd;              // Current lexeme's
                                     // ending index
    int iCurrOp;                     // Current operator
}
    LexerState;
```

Let's take a look at the new version of `GetNextChar ()`, capable now of reading the next character from the source buffer in linked list format:

```
char GetNextChar ()
{
    // Make a local copy of the string pointer, unless we're at the end of the
    // source code
    char * pstrCurrLine;
    if ( g_CurrLexerState.pCurrLine )
        pstrCurrLine = ( char * ) g_CurrLexerState.pCurrLine->pData;
    else
        return '\0';

    // If the current lexeme end index is beyond the length of the string,
    // we're past the end of the line
    if ( g_CurrLexerState.iCurrLexemeEnd >= ( int ) strlen ( pstrCurrLine ) )
    {
    // Move to the next node in the source code list
        g_CurrLexerState.pCurrLine = g_CurrLexerState.pCurrLine->pNext;

    // Is the line valid?
        if ( g_CurrLexerState.pCurrLine )
        {
    // Yes, so move to the next line of code and reset the lexeme
    // pointers
            pstrCurrLine = ( char * ) g_CurrLexerState.pCurrLine->pData;
            ++ g_CurrLexerState.iCurrLineIndex;
            g_CurrLexerState.iCurrLexemeStart = 0;
            g_CurrLexerState.iCurrLexemeEnd = 0;
        }
        else
        {
```

```
    // No, so return a null terminator to alert the lexer that the end
    // of the source code has been reached
            return '\0';
        }
    }

    // Return the character and increment the pointer
    return pstrCurrLine [ g_CurrLexerState.iCurrLexemeEnd ++ ];
}
```

Simply to keep the code readable, the first thing the function does is makes a local copy of the `pCurrLine` pointer in the `g_CurrLexerState` structure. It first makes sure, however, that the current line isn't the end of the source code; if it is, the pointer will be `NULL` and `\0` is returned to `GetNextToken ()` as a sign that the end of the source has been reached.

It's then determined whether the current lexeme end pointer is beyond the end of the current line. If so, the program moves to the next line by reading the `pCurrLine` structure's `pNext` pointer. The next task is to determine whether the line is valid; if it's `NULL`, it means you've reached the end of the source code and `\0` should once again be immediately returned. Otherwise, the `pstrCurrLine` string pointer is updated to point to the new line of code, the `iCurrLineIndex` field of `g_CurrLexerState` is incremented, and the lexeme pointers are both reset to the line's first character.

With any potential line increments taken care of, the current character in the stream is returned and the lexeme end pointer is incremented. `GetNextChar ()` now functions with an entirely different underlying storage structure, but remains identical to `GetNextToken ()`. This means that the entire lexer is now on board with the compiler's method of storing source code, and the majority of your work is done.

New Miscellaneous Functions

With the major tasks out of the way—rewinding the token stream and upgrading the lexer to work with a linked list source buffer—you're ready to finish things up. Fortunately, you have only a few minor tweaks and additions here and there left to deal with.

Adding a Look-Ahead Character

Just as was the case with XASM in Chapter 9, the XtremeScript compiler's parser will need the capability to read the first character of the next token in the stream, also known as the *look-ahead*. Fortunately, between the capability to preserve the current lexer state, as well as `GetNextChar ()`'s capability to continually return new characters regardless of line breaks and node boundaries within the linked list, writing a look-ahead function is no problem. Here's the code to `GetLookAheadChar ()`:

```
char GetLookAheadChar ()
{
    // Save the current lexer state
    LexerState PrevLexerState;
    CopyLexerState ( PrevLexerState, g_CurrLexerState );

    // Skip any whitespace that may exist and return the
    // first non-whitespace character
    char cCurrChar;
    while ( TRUE )
    {
        cCurrChar = GetNextChar ();
        if ( ! IsCharWhitespace ( cCurrChar ) )
            break;
    }

    // Restore the lexer state
    CopyLexerState ( g_CurrLexerState, PrevLexerState );

    // Return the look-ahead character
    return cCurrChar;
}
```

The function begins by preserving the current lexer state in a local `LexerState` instance. It does this because it's going to need to enlist `GetNextChar ()` in order to locate the first character of the next token, but as you just saw, the new version of this function will automatically advance the lexer state every time it's called. By saving the state first, you can call it all you want as long as you remember to restore it before returning the look-ahead. The concept of a look-ahead character is demonstrated in Figure 14.22.

A `while` loop is then entered that reads characters from the source buffer until the first non-whitespace character is found. This is considered the look-ahead, and is returned to the caller (but not before restoring the lexer state, of course).

Handling Invalid Tokens

The lexer prototype built in the last chapter would simply print an error message and exit upon the encounter of an invalid token. Although the compiler will do more or less the same thing, it's

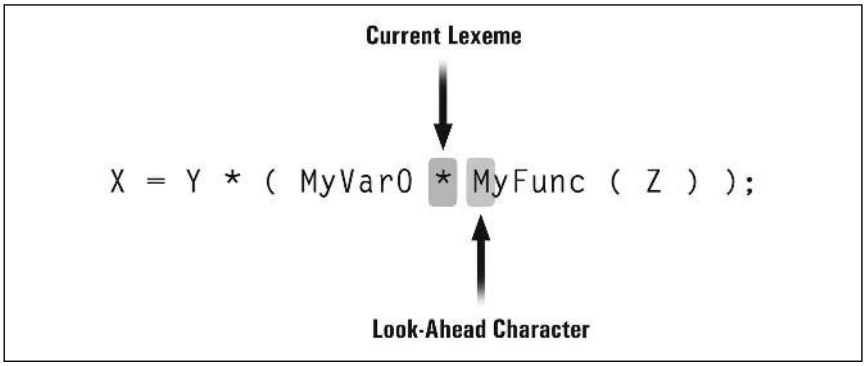


Figure 14.22

Using a look-ahead to read the first character of the next lexeme.

still not the lexer's place to terminate the program and display error messages. That task is handled by the error-handling functions defined in `error.cpp|h`, which you should be mindful of. Because the lexer is now but a single part in a much larger system, it should now simply return an error flag that signifies invalid tokens, allowing the caller (most likely the parser) to handle the error.

To implement this, you need a new token type to represent invalid tokens. Not surprisingly, you can call this `TOKEN_TYPE_INVALID`:

```
#define TOKEN_TYPE_INVALID                1
```

I set this new token's value for 1, so that it would immediately follow `TOKEN_TYPE_END_OF_STREAM`. Although this decision was ultimately arbitrary, I did it so I could group error-related token types together before getting into the valid token types that immediately follow. Of course, this means that the other token type constants had to be renumbered, but because they're constants, this has no effect on the rest of the program. Check out the source on the companion CD to see what I mean.

In addition, the lexer needs to keep track of invalid lexemes so that it can set the token type to `TOKEN_TYPE_INVALID` when the state machine finishes extracting them. To do this, you need a new lexer state called `LEX_STATE_UNKNOWN`:

```
#define LEX_STATE_UNKNOWN                 0
```

Every instance within the lexer's state machine that used to call an error function now simply sets the lexer state to `LEX_STATE_UNKNOWN`. As an example, check out the floating-point lexeme state handler:

```
case LEX_STATE_FLOAT:
    // If a numeric is read, keep the state as-is
    if ( IsCharNumeric ( cCurrChar ) )
    {
        iCurrLexState = LEX_STATE_FLOAT;
    }

    // If whitespace or a delimiter is read, the lexeme is done
    else if ( IsCharWhitespace ( cCurrChar ) || IsCharDelim ( cCurrChar ) )
    {
        iLexemeDone = TRUE;
        iAddCurrChar = FALSE;
    }
    // Anything else is invalid
    else
        iCurrLexState = LEX_STATE_UNKNOWN;
    break;
```

As soon as a non-float character is read, the state transitions to unknown. Upon the next iteration of the machine, this state handler will be invoked:

```
// If an unknown state occurs, the token is invalid, so exit
case LEX_STATE_UNKNOWN:
    iLexemeDone = TRUE;
    break;
```

Once outside of the state machine loop, it's time to assign a token type to the lexeme that was read. The new `LEX_STATE_UNKNOWN` state is easy to convert to a token; you just need to add a new `case` to the `switch` block used to map terminal lexer states to tokens:

```
// Determine the token type
Token TokenType;
switch ( iCurrLexState )
{
    // Unknown
    case LEX_STATE_UNKNOWN:
        TokenType = TOKEN_TYPE_INVALID;
        break;
```

The lexer now gracefully handles invalid tokens without terminating the program, allowing the caller to deal with the problem in a more appropriate manner.

Returning the Current Token

`GetNextToken ()` always returns the current token (whichever one it read), whereas separate functions like `GetCurrLexeme ()` and `GetCurrOp ()` can be used to get the current lexeme string or operator. However, even though GetNextToken () returns it initially, it would be nice to be able to read the token again at any time. As you might imagine, this is an easy feature to add. All you need to do is expand the LexerState function to track the current token, make sure GetNextToken () saves the token type there before returning, and add a new one-line function that returns that saved value. To start, let's make one final addition to the LexerState structure:

```
typedef struct _LexerState          // The lexer's state
{
    int iCurrLineIndex;             // Current line index
    LinkedListNode * pCurrLine;     // Current line node
                                    // pointer
    Token CurrToken;                // Current token
    char pstrCurrLexeme [ MAX_LEXEME_SIZE ];    // Current lexeme
    int iCurrLexemeStart;           // Current lexeme's
                                    // starting index
    int iCurrLexemeEnd;             // Current lexeme's
                                    // ending index
    int iCurrOp;                    // Current operator
}
    LexerState;
```

With the structure now capable of storing the current token, you need to add some code to the end of GetNextToken () to do so:

```
    // Return the token type and set the global copy
    g_CurrLexerState.CurrToken = TokenType;
    return TokenType;
}
```

Lastly, a separate function needs to be created for returning this new value:

```
Token GetCurrToken ()
{
    return g_CurrLexerState.CurrToken;
}
```

Copying the Current Lexeme

GetCurrLexeme () already returns a pointer to the current lexeme string, but the parser may likely have the need to make a physical copy at some point. In these cases, it would be nice to have a function available that will do it in a single call. For this, there's CopyCurrLexeme ():

```
void CopyCurrLexeme ( char * pstrBuffer )
{
    strcpy ( pstrBuffer, g_CurrLexerState.pstrCurrLexeme );
}
```

Error-Printing Helper Functions

The error-handling functions discussed later in the chapter will require that the lexer expose a few key pieces of information to help make its messages more verbose and informative for the users. As with XASM, it's helpful to print the actual line of code where the error was found, as well as the line number, and the pointer to the first character of the offending lexeme. To do this, you need functions for returning these three values.

Returning the current line of code is just a matter of returning the string pointer stored in the current node of the `g_SourceCode` linked list, but it's important to make sure the node is valid before doing so. Here's the source to `GetCurrSourceLine ()`:

```
char * GetCurrSourceLine ()
{
    if ( g_CurrLexerState.pCurrLine )
        return ( char * ) g_CurrLexerState.pCurrLine->pData;
    else
        return NULL;
}
```

If the current node pointer is invalid, a null string pointer is returned. Otherwise, the node's `pData` member is cast to a string pointer and returned.

Next up are functions for returning the current line number (which I refer to in the code as the line index), as well as the starting index of the current lexeme. Both of these are simple, one-line functions, so let's just look at them both:

```
int GetCurrSourceLineIndex ()
{
    return g_CurrLexerState.iCurrLineIndex;
}

int GetLexemeStartIndex ()
{
    return g_CurrLexerState.iCurrLexemeStart;
}
```

Simple, yes, but these will prove invaluable later.

Resetting the Lexer

One last modification to the lexer worth mentioning is that `InitLexer ()` is now known as `ResetLexer ()`. It's the same function, but because the compiler may need to reset the lexer multiple times during its lifespan, I feel the name change is appropriate for the new environment.

The Parser Module

The parser will be left blank for this chapter, because it's an equally large topic unto itself. The next chapter focuses solely on its development, so you'll just have to wait until then.

Error Handling

Error handling is implemented in `error.cpp|h` and consists primarily of two functions for printing the two major types of error messages. Just as was the case in XASM, the XtremeScript compiler differentiates between general errors and errors that relate specifically to the source code, such as syntax errors.

General Errors

Printing a general error is trivial and is handled by the `ExitOnError ()` function:

```
void ExitOnError ( char * pstrErrorMssg )
{
    // Print the message
    printf ( "Fatal Error: %s.\n", pstrErrorMssg );

    // Exit the program
    Exit ();
}
```

It's simply a matter of printing the error message to the screen and calling `Exit ()`, a function defined earlier that lets the compiler clean up after itself just before exiting. Note that the `printf ()` call automatically appends a trailing period to the message, so the error messages will not contain one.

Code Errors

Printing a code error is more complex than that of a general error, because it's helpful to give the users detailed information about the specifics of the error. Like XASM, the XtremeScript compiler will display the current line, print the line number, and use a caret symbol to point out the offending character/lexeme:

```
void ExitOnCodeError ( char * pstrErrorMssg )
{
    // Print the message
    printf ( "Error: %s.\n\n", pstrErrorMssg );
    printf ( "Line %d\n", GetCurrSourceLineIndex () );

    // Reduce all of the source line's spaces to tabs so it takes less space
    // and so the caret lines up with the current token properly
    char pstrSourceLine [ MAX_SOURCE_LINE_SIZE ];

    // If the current line is a valid string, copy it into the local source
    // line buffer
    char * pstrCurrSourceLine = GetCurrSourceLine ();
    if ( pstrCurrSourceLine )
        strcpy ( pstrSourceLine, pstrCurrSourceLine );
    else
        pstrSourceLine [ 0 ] = '\0';

    // If the last character of the line is a line break, clip it
    int iLastCharIndex = strlen ( pstrSourceLine ) - 1;
    if ( pstrSourceLine [ iLastCharIndex ] == '\n' )
        pstrSourceLine [ iLastCharIndex ] = '\0';

    // Loop through each character and replace tabs with spaces
    for ( unsigned int iCurrCharIndex = 0;
          iCurrCharIndex < strlen ( pstrSourceLine );
          ++ iCurrCharIndex )
        if ( pstrSourceLine [ iCurrCharIndex ] == '\t' )
            pstrSourceLine [ iCurrCharIndex ] = ' ';

    // Print the offending source line
    printf ( "%s\n", pstrSourceLine );

    // Print a caret at the start of the (presumably) offending lexeme
    for ( int iCurrSpace = 0;
          iCurrSpace < GetLexemeStartIndex ();
          ++ iCurrSpace )
        printf ( " " );
    printf ( "^\n" );
```

```
        // Print message indicating that the script could not be assembled
        printf ( "Could not compile %s.", g_pstrSourceFilename );

        // Exit the program
        Exit ();
    }
```

The function first prints the error message and the line number. It then statically allocates a local string buffer to hold the current line of code, the pointer to which it gets from `GetCurrSourceLine ()`. Once it has a physical copy, it looks for a trailing line break and clips it by replacing it with a null terminator. It does this because it's better to control the formatting of the message yourself, without having to worry about whether the line of code will impose its own line breaks. The function then scans through each character of the line, replacing tabs with single spaces. You'll see why in a moment.

The offending line of code is then printed. Directly underneath it, a series of spaces are printed on the same line corresponding to the number of characters between the beginning source line and the starting index of the current lexeme. These spaces are immediately followed by a caret, which now points to the beginning of the lexeme where the error occurred. This should make it clear why you had to replace tabs with spaces; even though a tab is expressed on the screen as multiple spaces, it's represented internally as a single `\t` character. If you were to print the code as-is, the tabs would cause the code line to be desynchronized with the caret, and the wrong character would be highlighted for the users.

Cascading Errors

One popular feature of most modern compilers, from assemblers all the way up to C++ and Java compilers, is the *cascading error*. An error-handling system is said to cascade when it continues to parse the source file even after an error was found, in an attempt to list all of a script or program's errors in one shot (the term *cascade* comes from the fact that, more often than not, subsequent errors are simply the result of the first error). I chose not to implement cascading errors in the XtremeScript compiler for a number of reasons:

- They're more complex.
- They aren't necessarily accurate, often making only the first error, or first few errors, worth noting.
- Although they're understandably useful in compilers used for large projects, scripts are smaller, simpler pieces of code almost by nature. It's unlikely that you'll need error handling as robust and verbose as a high-end C++ compiler for writing individual game scripts.

However, it's an interesting topic and one that I'll discuss briefly. To implement a cascading error system, the parser needs to be able to *resynchronize* itself after detecting an error. On a basic level, this means finding the next valid token with which it can pick itself up, dust itself off, and resume a normal parsing process.

For example, imagine the following code fragment:

```
// Declare a function
func Square ( X )
{
    return X * X;
}

// Declare some variables
var MyVar0;
var MyVar1;

// Use the variables and functions
MyVar0 = MyVar1 [ 3 ];    // Error - MyVar1 is not an array
MyVar1 = Square ( 4 );    // Valid
Square ( MyVar0, MyVar1 );    // Error - Square () only accepts one
                  // parameter
```

As you can see, there are two clear errors here; one in which `MyVar1` is treated as an array, and one in which `Square ()` is passed two parameters instead of the one it accepts. In the error system you'll implement in the next chapter's parser, only the first error will be flagged before the program terminates. In a cascading error system, however, both errors would appear.

This is accomplished by resynchronizing the parser at the next valid lexeme. In the case of the first error, the next valid token is the `MyVar1` on the following line. To better illustrate this, the two lines in question are reprinted here, with two lexemes in bold:

```
MyVar0 = MyVar1 [ 3 ];    // Error - MyVar1 is not an array
MyVar1 = Square ( 4 );    // Valid
```

The first bolded lexeme, `[`, is where the error occurs. As soon as the lexer sends the `[` token to the parser, it knows that `MyVar1` is being used as an array illegally. From that point on, there are three tokens left in the statement—`3`, `]`, and `;`. None of these should concern you, because you know that the statement from here on out is invalid. So, a basic strategy for resynchronizing the parser after the detection of an error is to simply consume tokens until the next semicolon is read. The token following that semicolon must be the first token of the next line, which is where the parser will attempt to get back on track.

The parser will read the next line without a problem, because it's perfectly valid. The line after that, however, in which `Square ()` is passed two parameters, presents another error. And because the parser is still active, it will catch it as well as the first one. The end result is two errors printed where a more simplistic error mechanism would print only one.

Of course, this was just a basic strategy; there are more sophisticated methods of error recovery out there. For example, it may be necessary to resynchronize within the current statement, because multiple errors can certainly occur before the next semicolon. Also, remember that not all statements will end in a semicolon; for example, function declarations and `while` loops are two likely candidates for syntax errors, but neither is terminated in the same way a statement is. In these cases, the parser has to be smart enough to finish parsing whatever *type* of statement it's currently processing, in order to intelligently make its way to the next line.

Once you read the next chapter, you should be able to modify the parser you'll build to support this feature in the basic way I've described it here.

The I-Code Module

In between the source code and lexeme and token streams of the front end, and the XVM assembly output of the back end, there's the I-code module. As has been explained a number of times throughout the book, the purpose of intermediate code is to allow the parser and the code emitter to talk to a common structure without having to directly talk to one another. The logic behind the parser is complicated enough as it is; having to directly output ASCII-formatted assembly would make things considerably more difficult. By allowing it to instead interface with an abstracted I-code module through an API of simple functions, the parser can focus almost exclusively on what it does best—parsing the token stream. The XtremeScript compiler I-code module is implemented in `i_code.cpp|h`.

Approaches to I-Code

There are a number of ways to approach I-code. On the one hand, I-code is often implemented as what is known as an *annotated syntax tree*, which is a hierarchical representation of the source code (see Figure 14.23), in a streamlined format that minimizes extraneous data. Another common approach is a linked list or other such aggregate structure of instructions that represent a generalized, abstracted instruction set. This lets the front end reduce the source code to an assembly-style format without being bogged down by the details and nuances of the specific platform.

One of the main ways I-code implementations can be classified is how close they are to one of the compiler's ends. High-level I-code, like annotated syntax trees, are much closer to the original source code—and therefore the front end——and often maintain statements and nested

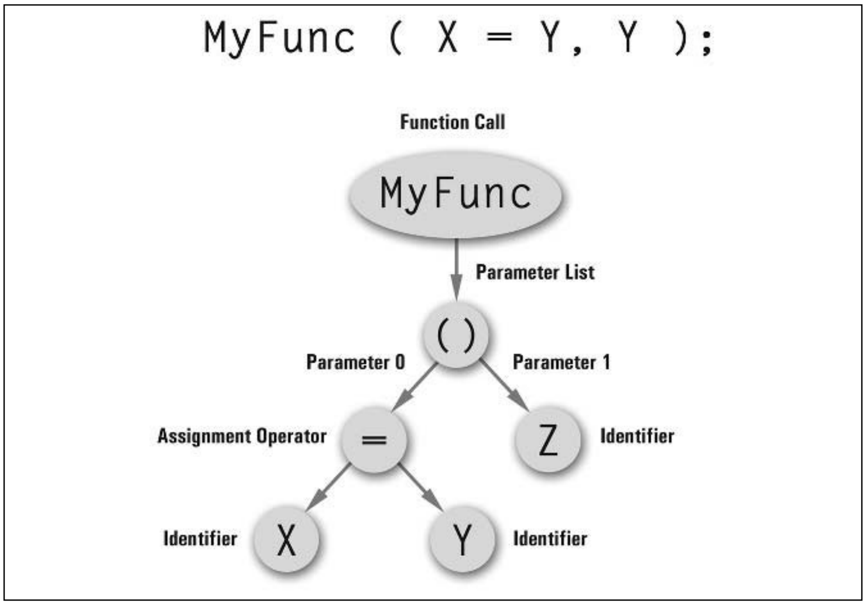


Figure 14.23

Using an annotated source tree as an I-code implementation.

structures that are similar to the source language. Low-level I-code implementations, like lists of pseudo-instructions, are closer to the back end and resemble Assembly far more than they would C++ or Pascal.

To keep things simple but still useful, I've chosen to base XtremeScript's I-code module on the latter of the two options. The intermediate code generated by the parser will be very similar to XVM assembly, but represented in a numeric form like a compiled instruction stream rather than an ASCII-formatted string of characters. The I-code will be stored in a linked list, wherein each node represents a separate instruction. Each instruction will contain an opcode, an operand count, and a list of operands—again, much like the compiled instruction stream generated by XASM and executed by the XVM.

A Simplified Instruction Set

After deciding to go with an assembly-style, instruction-based I-code scheme, the next decision is what the instruction set will look like. If the compiler was targeting the Intel 80x86, for example, you would have a very complex target code to deal with. 80x86 assembly is a complex instruction set with hundreds of instructions, countless rules, exceptions, and idiosyncrasies, and plenty of other issues that make generating valid, functional 80x86 code a very difficult task. So, to help separate the parser and other front-end elements from this complex environment, the I-code module would represent source code using a higher-level, far more simplistic instruction set. The code emitter would then be responsible for translating the I-code's higher-level, simplified language to Intel's language.

For example, the 80x86 has a multiplication operator that differs strongly from the XVM's `Mul` operator (even though they share the same name). Here's an example of multiplying two variables, `X` and `Y`, and storing the result in `X`:

```
MOV    EAX, X
MUL    Y
MOV    X, EAX
```

The first thing to remember is that the 80x86 platform has a number of hardware registers, of which `EAX` is an example. The `MUL` (multiplication) operator *requires* that the destination operand be the `EAX` register. The source, which the destination is multiplied by, can be either another register or a memory location. This is why you can specify `Y` as the operand for `MUL`. Because `EAX` is already specified as the destination operand in all cases, `MUL` only needs to accept a single operand. What this also means is that `X` must first be moved into `EAX` before the multiplication, and that `EAX` must be moved back to `X` afterwards.

This example should make it clear that often times, a target language is inconvenient to work with. There's no arguing that it's a conceptually simple process to simply say `Mul X, Y`, like you could on the XVM. Rather than having to use specific registers, and implied operands, it's easier to just specify exactly what you want and be done with it. This is why it's *far* easier to represent a multiplication operation in this manner within the I-code, and rely on the code emitter module to convert it to valid 80x86 assembly in a later phase. This is demonstrated in Figure 14.24.

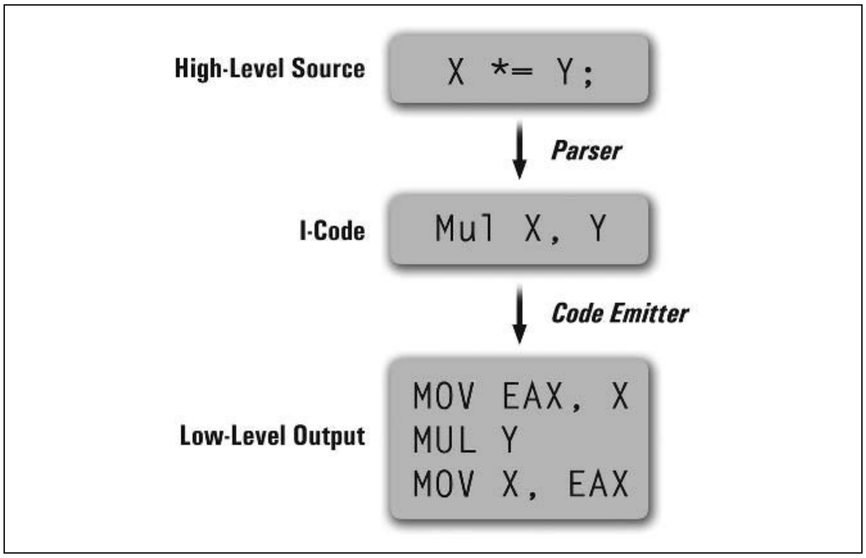


Figure 14.24

Using assembly-style I-code as an intermediate step between high-level source and low-level output.

The XtremeScript I-Code Instruction Set

The funny thing, however, is that the XVM is *already* designed around an intentionally simplistic and easy-to-use instruction set. Although I'm sure it's possible to find ways to make it even simpler (within reason), I designed it intentionally from day one to iron out the difficulties associated with many native hardware assembly languages. Because of this, there's not much you can do to make your I-code language any easier than XVM assembly already is.

Because of this, XtremeScript I-code will more or less mirror XVM assembly, all the way down to the individual instructions and their operands. This design decision may seem to invalidate the very purpose of I-code in the first place—after all, why waste the effort converting something to an "intermediate" code that's actually identical to the target code?

The reason XtremeScript I-code is still more useful than forcing the parser to directly output XVM assembly is because of the interface. As you'll see shortly, the I-code module makes it extremely easy to add instructions to its internal list with only a few function calls. Furthermore, writing directly to the output file brings with it a number of drawbacks; for example, there's no easy or efficient way to shift around large blocks of data, or make changes after something's been written. By writing everything to an intermediate linked-list of instructions, you're free to perform virtually any form of manipulation at any time. In addition, this prevents the parser from having to deal with actual code, which is string-based and messy. It's much easier for the parser to simply say, "move the integer literal value of 2 into the symbol table index 186," than it is to literally spell out `"Mov MyVar, 2`", character by character. It's also far less error prone, because numeric data wrapped in constants is much cleaner and simpler to work with.

As if that wasn't reason enough, there's still the main attraction to I-code—the capability to retarget other platforms. For example, you *could* one day decide that it would be useful for the XtremeScript compiler to generate real, 80x86 machine code. If this was ever decided, it would be a *huge* pain to have to convert the token stream directly to the Intel's far more complex instruction set. By leaving the I-code module in place, the parser and front end can remain entirely unchanged; only the code emitter will require modifications to output code for the new platform. And because XVM assembly is far simpler than 80x86 machine code, it makes for the perfect I-code syntax.

The XtremeScript I-Code Implementation

Implementing I-code in XtremeScript is a lot like implementing the assembled instruction stream was in XASM. The only real difference is that instead of using a statically allocated array to hold the instruction stream, a linked list is used to allow it to grow and shrink dynamically as the parsing process progresses.

Instructions

Each node of this list will represent a single instruction, complete with an opcode and operands. To keep things as simple as possible, these opcodes will map directly to XVM assembly opcodes, so you can copy and paste the list of instruction constants directly from XASM:

```
#define INSTR_MOV               0

#define INSTR_ADD               1
#define INSTR_SUB               2
#define INSTR_MUL               3
#define INSTR_DIV               4
#define INSTR_MOD               5
#define INSTR_EXP               6
#define INSTR_NEG               7
#define INSTR_INC               8
#define INSTR_DEC               9

#define INSTR_AND               10
#define INSTR_OR                11
#define INSTR_XOR               12
#define INSTR_NOT               13
#define INSTR_SHL               14
#define INSTR_SHR               15

#define INSTR_CONCAT            16
#define INSTR_GETCHAR           17
#define INSTR_SETCHAR           18

#define INSTR_JMP               19
#define INSTR_JE                20
#define INSTR_JNE               21
#define INSTR_JG                22
#define INSTR_JL                23
#define INSTR_JGE               24
#define INSTR_JLE               25

#define INSTR_PUSH              26
#define INSTR_POP               27
```

```
#define INSTR_CALL              28
#define INSTR_RET               29
#define INSTR_CALLHOST          30

#define INSTR_PAUSE             31
#define INSTR_EXIT              32
```

This takes care of the I-code instructions, but you of course need operands as well. Like the instructions, you can copy these directly from XASM, but they'll require a bit of modification. Here are the XtremeScript I-code operand types:

```
#define OP_TYPE_INT                 0   // Integer literal value
#define OP_TYPE_FLOAT               1   // Floating-point literal value
#define OP_TYPE_STRING_INDEX        2   // String literal value
#define OP_TYPE_VAR                 3   // Variable
#define OP_TYPE_ARRAY_INDEX_ABS     4   // Array with absolute index
#define OP_TYPE_ARRAY_INDEX_VAR     5   // Array with relative index
#define OP_TYPE_JUMP_TARGET_INDEX   6   // Jump target index
#define OP_TYPE_FUNC_INDEX          7   // Function index
#define OP_TYPE_REG                 9   // Register
```

I-code instruction operands can be integer literals, floating-point literals, indexes into the string table (string literals), indexes into the symbol table (variables), indexes into the symbol table with an offset (arrays indexed with an immediate integer value), indexes into the symbol table with an offset contained in another symbol table offset (arrays indexed with variables), jump targets (the I-code representation of a line label, (which I'll discuss in more detail shortly), indexes into the function table, or register codes (which, for now, always means `_RetVal`).

Thanks to XASM, you can lift these constants almost directly. You now need a data structure to hold their values. Again, like XASM, you need a structure that represents a single I-code instruction's opcode and operand list. The structure is called `ICodeInstr`, and looks like this:

```
typedef struct _ICodeInstr      // An I-code instruction
{
    int iOpcode;                // Opcode
    LinkedList OpList;          // Operand list
}
    ICodeInstr;
```

Unlike XASM, however, you're using another dynamic linked list to hold the operand list. Because of this, you don't need a separate field to store the operand count. `OpList`'s `iNodeCount` member will contain it at all times. The operand list still needs an operand structure to embody each of its nodes, however. For this, you need the `Op` structure:

```
typedef struct _Op              // An I-code operand
{
    int iType;                  // Type
    union                       // The value
    {
        int iIntLiteral;        // Integer literal
        float fFloatLiteral;    // Float literal
        int iStringIndex;       // String table index
        int iSymbolIndex;       // Symbol table index
        int iJumpTargetIndex;   // Jump target index
        int iFuncIndex;         // Function index
        int iRegCode;           // Register code
    };
    int iOffset;                // Immediate offset
    int iOffsetSymbolIndex;     // Offset symbol index
}
    Op;
```

Most of this should look familiar; a `union` combines all of the mutually exclusive fields into a single, overlapping block of memory, and the `iType` function lets you know which field of the `union` is currently active. You'll notice that within the `union`, there's no mention of labels or target instructions, but rather `iJumpTargetIndex`. I'll talk about this more in the next section. `iSymbolIndex` is used to store the index into the symbol table for variables and arrays. `iOffset` and `iOffsetSymbolIndex` are used for array indexing. If the array is indexed with an immediate value, it goes into `iOffset`. If it's indexed by a variable, that variable's symbol table index is stored in `iOffsetSymbolIndex`.

Jump Targets

Instructions aren't quite enough, however. Just as is the case with XVM assembly, the I-code representation of a program needs the capability to express iterative and conditional logic in the form of jump instructions. Of course, in order for a jump instruction to work, it needs a label to jump to. Because labels are generally designed to enhance the readability of a program for a human, the I-code version of the program will only need bare-bone markers, or *jump targets*, that represent a specific node to jump to and are represented by a numeric index. Check out Figure 14.25.

Although you could simply add a flag to the `ICodeInstr` structure to mark certain instructions as jump targets, as well as a second field that contains the target's index, this suffers from some

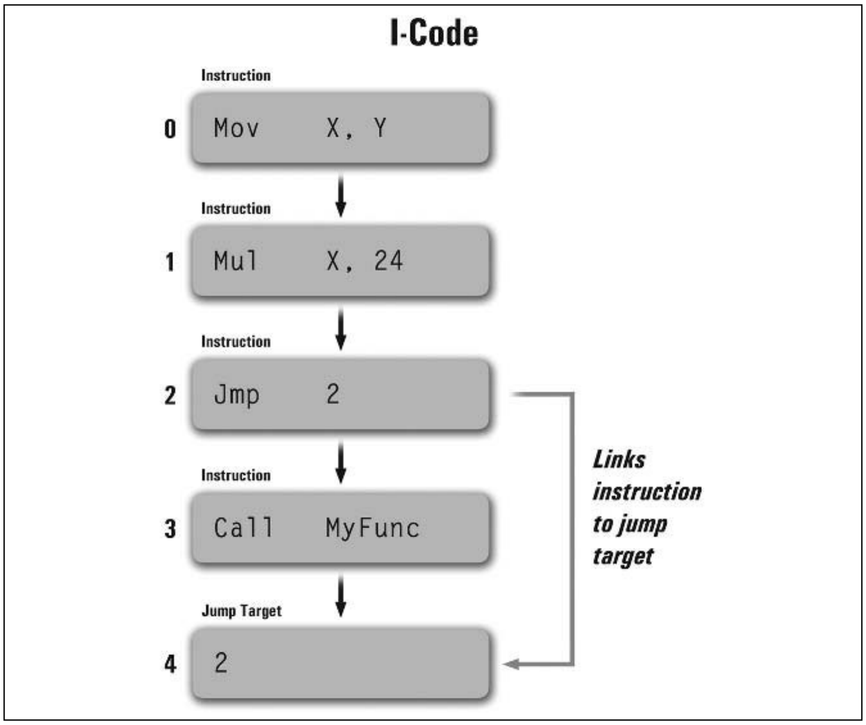


Figure 14.25

Jump targets allow jump instructions to reference their target I-code nodes.

drawbacks. For example, it is possible that at some point, you'll need to insert an instruction arbitrarily into the stream. If, by chance, this insertion must take place in between the jump target and the instruction to which that target is bound, you're hosed—there's no way to separate the target from its instruction, because they both occupy the same structure. The only solution would be to clear the old instruction's jump target flag and set it in the new one, but this is a lot of unnecessary work.

Rather, you can take a lesson from human-readable labels and make them separate I-code nodes unto themselves. This way, no matter how much the neighboring instructions change and evolve, the jump target always remains in place, as a separate, intact entity of its own.

At this point, you can formulate the basis for a general I-code node structure, of which the I-code stream linked list will be composed. You now know that you need at least two structures within each node—one to represent jump targets and one to represent instructions. Because it would be silly to store these separately, you'll once again use a `union`. Here's the `ICodeNode` structure:

```
typedef struct _ICodeNode           // An I-code node
{
    int iType;                      // The node type
    union
    {
        ICodeInstr Instr;            // The I-code instruction
        int iJumpTargetIndex;        // The jump target index
    };
}
    ICodeNode;
```

Now, the ICodeStream linked list found in the FuncNode structure discussed earlier can be filled with ICodeNode structures. Each node is capable of functioning as either a jump target or instruction, allowing for a complete representation of any source program in an abstracted I-code format. Very cool.

Of course, you need to create some new constants to represent instructions and jump targets:

```
#define ICODE_NODE_INSTR        0
#define ICODE_NODE_JUMP_TARGET  1
```

And you're all set!

Source Code Annotation

There is *one* more detail worth mentioning before you get into the nitty-gritties of the I-code module interface (isn't there always?). In addition to instructions and jump targets, there's a third possible node type that I think is important to consider. This third type is known as *source code annotation.*

If you've ever used the Visual C++ disassembler to view the compiler's output, you know what I'm talking about. Because each C++ instruction is compiled down to *N* number of assembly instructions, it can be hard to follow which instructions belong to which parts of the original source code. To remedy this problem, the VC++ disassembler has an option to automatically annotate its assembly output with comments that contain each line of the original source. For example, take the following block of C++ code:

```
main ()
{
    int X, Y;

    Y = 4;
    X = Y * 8;
    Y = X / 2;

    int Z = X + Y;
    return 0;
}
```

With source code annotation turned on, the Microsoft VC++ disassembler produces the following:

```
; 5    :     Y = 4;

   mov   DWORD PTR _Y$[ebp], 4

; 6    :     X = Y * 8;

   mov   eax, DWORD PTR _Y$[ebp]
   shl   eax, 3
   mov   DWORD PTR _X$[ebp], eax

; 7    :     Y = X / 2;

   mov   eax, DWORD PTR _X$[ebp]
   cdq
   sub   eax, edx
   sar   eax, 1
   mov   DWORD PTR _Y$[ebp], eax

; 8    :
; 9    :     int Z = X + Y;

   mov   ecx, DWORD PTR _X$[ebp]
   add   ecx, DWORD PTR _Y$[ebp]
   mov   DWORD PTR _Z$[ebp], ecx
```

As you can see, it's much easier to follow when you can tell exactly which instructions came from which statements.

Because you'll be doing a lot of examinations on the XVM assembly code emitted by the compiler, you'll benefit greatly from this feature. Especially when developing a compiler, it's extremely important to have a way to ensure that the I-code module is being fed the proper instructions from the proper source code.

This feature can be added easily with the addition of a string pointer in the `ICodeNode` `union` and a new node type. Here's the addition to the `ICodeNode` structure:

```
typedef struct _ICodeNode           // An I-code node
{
    int iType;                      // The node type
    union
    {
        ICodeInstr Instr;           // The I-code instruction
        char * pstrSourceLine;      // The source line with
                                    // which this instruction
                                    // is annotated
        int iJumpTargetIndex;       // The jump target index
    };
}
    ICodeNode;
```

Here's the addition of a new constant to reflect the new node type:

```
#define ICODE_NODE_INSTR        0
#define ICODE_NODE_SOURCE_LINE  1
#define ICODE_NODE_JUMP_TARGET  2
```

Simply by allowing certain nodes to contain pointers to source code strings (which can remain in the `g_SourceCode` linked list), the code emitter will have all it needs to generate source code annotated assembly output. For now, however, your work here is done. The structures and constants needed by the I-code module are in place, so all you need now is a set of interface functions for manipulating them easily.

The Interface

The I-code module's interface is responsible for enabling a number of tasks:

- Adding instructions to the end of the current I-code stream.
- Adding operands of all types to those instructions after they've been added.
- Automatically generating the next unique jump target index and adding it to the instruction stream.
- Adding source code annotation.
- Retrieving an I-code node based on its order within the stream.

Once you have functions for each of these tasks, you'll have a completed I-code module that's ready to use.

Adding Instructions

The first and most basic I-code module operation is the addition of an instruction. Remember, *all* I-code must exist within the scope of a specific FuncNode structure in the function table. In other words, code only exists within functions. Because of this, a function for adding an I-code instruction needs both the opcode to add, as well as a function table index to specify the proper scope. This is done with the AddICodeInstr () function:

```
int AddICodeInstr ( int iFuncIndex, int iOpcode )
{
    // Get the function to which the instruction should be added
    FuncNode * pFunc = GetFuncByIndex ( iFuncIndex );

    // Create an I-code node structure to hold the instruction
    ICodeNode * pInstrNode = ( ICodeNode * ) malloc ( sizeof ( ICodeNode ) );

    // Set the node type to instruction
    pInstrNode->iType = ICODE_NODE_INSTR;

    // Set the opcode
    pInstrNode->Instr.iOpcode = iOpcode;

    // Clear the operand list
    pInstrNode->Instr.OpList.iNodeCount = 0;

    // Add the instruction node to the list and get the index
    int iIndex = AddNode ( & pFunc->ICodeStream, pInstrNode );

    // Return the index
    return iIndex;
}
```

As I said, this function accepts a function index, iFuncIndex, and an opcode, iOpcode. The first order of business is retrieving the FuncNode structure of the specified function using GetFuncByIndex (). A new ICodeNode structure is then allocated and initialized by setting its iType field to ICODE_NODE_INSTR and its iOpcode field to the specified opcode. The operand list is cleared by setting its iNodeCount member to zero. Lastly, AddNode () is called, and the index it provides is returned to the caller.

The index returned by `AddICodeInstr ()` is actually of significant importance; because operands will be added to the instruction in subsequent function calls, the caller needs to be able to specify which node index in the stream the operands should be added to.

Adding Operands

Speaking of which, adding operands to preexisting instructions in an I-code stream is the focus of the next set of functions I'm going to discuss. Once an instruction exists in the I-code stream, operands can be added to its `OpList` linked list. This is done with the `AddICodeOp ()` function:

```
void AddICodeOp ( int iFuncIndex, int iInstrIndex, Op Value )
{
    // Get the I-code node
    ICodeNode * pInstr = GetICodeNodeByImpIndex ( iFuncIndex, iInstrIndex );

    // Make a physical copy of the operand structure
    Op * pValue = ( Op * ) malloc ( sizeof ( Op ) );
    memcpy ( pValue, & Value, sizeof ( Op ) );

    // Add the instruction
    AddNode ( & pInstr->Instr.OpList, pValue );
}
```

The function is passed a function index, `iFuncIndex`, which it uses to find the specific I-code stream. It's also passed an instruction index, `iInstrIndex`, which allows it to find the proper instruction within that stream. Lastly, we send it an `Op` structure containing the operand's value, `Value`. A call is made to a function called `GetICodeNodeByImpIndex ()`, which returns a pointer to the I-code node. I'll come back to this function soon, but for now, all you need to know is that it returns a node pointer based on a specific function index and instruction index. The node is found in the instruction stream based on its *implicit index*, which is just another way of saying its physical order in the list.

A new `Op` structure is then allocated to store a physical copy of the one passed in the `Value` parameter. This `Op` structure is ultimately the one that's added to the list with `AddNode ()`. Note that this function doesn't seem to care about the new node's index—this is because there's no need to modify an operand after it's added.

Making Operand Addition Easier

This function is certainly convenient, but it's still a bit of a hassle to have to create a new `Op` structure every time you want to add an operand. If you're adding an integer literal operand, it would be nice to simply pass the function an integer value. If you're adding a symbol table index operand, it would be easier if you could just pass the index itself. To do this, you can create a number of helper functions that will wrap `AddICodeOp ()` to make the addition of specific operand values easier. Let's start with `AddIntICodeOp ()`, which adds integer values as I-code operands:

```
void AddIntICodeOp ( int iFuncIndex, int iInstrIndex, int iValue )
{
    // Create an operand structure to hold the new value
    Op Value;

    // Set the operand type to integer and store the value
    Value.iType = OP_TYPE_INT;
    Value.iIntLiteral = iValue;

    // Add the operand to the instruction
    AddICodeOp ( iFuncIndex, iInstrIndex, Value );
}
```

This function declares a local `Op` structure, sets its `iType` field to `OP_TYPE_INT`, sets its `iIntLiteral` field to the integer value specified by specified `iValue`, and adds the operand by calling `AddICodeOp ()`.

The rest of the functions work in exactly the same way; they only differ by the constant they assign to `iType` and the values they put in the rest of the `Op` structure. Because of this, there's no point in wasting the time and page space involved in printing and dissecting them individually. To check them out for yourself, however, you're encouraged to browse the XtremeScript compiler source provided on the companion CD.

Retrieving Operands

As you'll see when studying the implementation of the code emitter module, it will be necessary to retrieve an I-code's operands based on their index within the list. This is done by the `GetICodeOpByIndex ()` function:

```
Op * GetICodeOpByIndex ( ICodeNode * pInstr, int iOpIndex )
{
    // If the list is empty, return a NULL pointer
    if ( ! pInstr->Instr.OpList.iNodeCount )
        return NULL;

    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = pInstr->Instr.OpList.pHead;
    // Traverse the list until the matching index is found
    for ( int iCurrNode = 0;
          iCurrNode < pInstr->Instr.OpList.iNodeCount;
          ++ iCurrNode )
    {
    // If the index matches, return the operand
        if ( iOpIndex == iCurrNode )
            return ( Op * ) pCurrNode->pData;

    // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The operand was not found, so return a NULL pointer
    return NULL;
}
```

This simple function accepts an `ICodeNode` structure pointer, as well as an index within its operator list. The function then traverses the list, assuming it's not empty, until the specified index matches the current index. If a match is found, the operand structure pointer is returned; otherwise, `NULL` is returned.

Adding Jump Targets

Now that you can add instructions, you need to add jump targets to facilitate looping and branching. Fortunately, this is just as easy as adding instructions was, and is handled with the `AddICodeJumpTarget ()` function:

```
void AddICodeJumpTarget ( int iFuncIndex, int iTargetIndex )
{
    // Get the function to which the source line should be added
    FuncNode * pFunc = GetFuncByIndex ( iFuncIndex );

    // Create an I-code node structure to hold the line
    ICodeNode * pSourceLineNode = ( ICodeNode * )
        malloc ( sizeof ( ICodeNode ) );

    // Set the node type to jump target
    pSourceLineNode->iType = ICODE_NODE_JUMP_TARGET;
    // Set the jump target
    pSourceLineNode->iJumpTargetIndex = iTargetIndex;

    // Add the instruction node to the list and get the index
    AddNode ( & pFunc->ICodeStream, pSourceLineNode );
}
```

Predictably, this function accepts a function index and a jump target index. It calls `GetFuncByIndex ()` to retrieve a pointer to the function node, and then allocates space for the new I-code node. It sets the newly created node's `iType` to `ICODE_NODE_JUMP_TARGET`, and the `iJumpTargetIndex` field to the index specified by the `iTargetIndex` parameter. Finally, it adds the node with a call to `AddNode ()`.

This is a simple enough function, and it's pretty obvious how everything works, but how do you determine the jump target's index? It's extremely important that at least within the same scope, *all* jump targets have unique indexes. Otherwise, chaos will ensue as the compiler and assembler attempt to direct different jumps to the same instruction, and ultimately declare multiple labels with the same name in the resulting assembly output.

To remedy this, you need a function that can guarantee a new, unique jump target index every time it's called. Implementing this is actually rather simple; the I-code module just maintains a global variable called `g_iCurrJumpTargetIndex` and increments it every time a new one is requested. This is handled by the `GetNextJumpTargetIndex ()` function:

```
int GetNextJumpTargetIndex ()
{
    // Return and increment the current target index
    return g_iCurrJumpTargetIndex ++;
}
```

This can now be called just before a call to `AddICodeJumpTarget ()` to ensure that a unique target index is used every time.

Adding Source Code Annotation

The last type of I-code node that can be added to the stream is source code annotation, which simply contains a string pointer that references one of the strings in the `g_SourceCode` linked list. Adding the annotation is a somewhat trivial matter; it's based on the same principals as `AddICodeInstr ()` and `AddICodeJumpTarget ()` before it. Here's the function responsible for it, `AddICodeSourceLine ()`:

```
void AddICodeSourceLine ( int iFuncIndex, char * pstrSourceLine )
{
    // Get the function to which the source line should be added
    FuncNode * pFunc = GetFuncByIndex ( iFuncIndex );

    // Create an I-code node structure to hold the line
    ICodeNode * pSourceLineNode = ( ICodeNode * )
       malloc ( sizeof ( ICodeNode ) );

    // Set the node type to source line
    pSourceLineNode->iType = ICODE_NODE_SOURCE_LINE;

    // Set the source line string pointer
    pSourceLineNode->pstrSourceLine = pstrSourceLine;

    // Add the instruction node to the list and get the index
    AddNode ( & pFunc->ICodeStream, pSourceLineNode );
}
```

As you might expect, it accepts a function index and a source line in the form of a string pointer. It grabs the function's node, uses it to determine which I-code stream to add the node to, allocates a new I-code node, and initializes it. The `iType` field is set to `ICODE_NODE_SOURCE_LINE` and the `pstrSourceLine` string pointer member is set to the pointer specified. And of course, the process is completed with a call to `AddNode ()`.

Retrieving I-Code Nodes

The last function your I-code module interface needs is one to retrieve an entire I-code node. As you'll see when you write the code emitter module, the best way to retrieve I-code nodes is by their *implicit index*. The node's implicit index, unlike most of the other nodes you've dealt with in other tables, is simply its physical order in the list. So, if the node in question is the third node from the head, its implicit index is 2. Let's take a look at `GetICodeNodeByImpIndex ()`, which returns an I-code node based on its implicit index:

```
ICodeNode * GetICodeNodeByImpIndex ( int iFuncIndex, int iInstrIndex )
{
    // Get the function
    FuncNode * pFunc = GetFuncByIndex ( iFuncIndex );

    // If the stream is empty, return a NULL pointer
    if ( ! pFunc->ICodeStream.iNodeCount )
        return NULL;
    // Create a pointer to traverse the list
    LinkedListNode * pCurrNode = pFunc->ICodeStream.pHead;

    // Traverse the list until the matching index is found
    for ( int iCurrNode = 0;
          iCurrNode < pFunc->ICodeStream.iNodeCount;
          ++ iCurrNode )
    {
    // If the implicit index matches, return the instruction
        if ( iInstrIndex == iCurrNode )
            return ( ICodeNode * ) pCurrNode->pData;

    // Otherwise move to the next node
        pCurrNode = pCurrNode->pNext;
    }

    // The instruction was not found, so return a NULL pointer
    return NULL;
}
```

This simple function follows the pattern of most of the compiler's other retrieval functions. A node pointer is used to traverse the list, and at each iteration, the specified index is compared to the current one. If a match is found, the pointer is returned. `NULL` is returned in the event that the specified index does not exist.

The Code-Emitter Module

Code emission is the final step of a compiler, wherein the I-code generated by the parser is finally converted to the compiler's target format. In this case, the target format is an XVM assembly file compatible with the XASM assembler built in Chapter 9. Because the I-code module devised in the last system is so similar to this language, it will be an easy translation. The code emitter is implemented in `code_emit.cpp|h`.

Code-Emission Basics

On a basic level, the code emitter just needs to produce a valid text file that can be fed to the assembler. Its job is nothing more than taking the I-code stream, which is very similar to the compiled instruction stream created by the XASM assembler, and converting it *back* to a text representation. Opcodes are converted back to their instruction mnemonics, symbol table indexes are emitted as variable identifiers, entries in the function table are emitted as formal XASM function declarations, and so on.

Although you could just emit a bare-bones, completely unformatted chunk of borderline unreadable text, there are a number of reasons to expend some extra effort formatting the generated assembly file for both general aesthetics and readability:

- You will most likely find it useful to do some hand-tuning to the compiler's output for certain scripts, mostly for the purpose of optimization.
- The compiler may generate erroneous output, which causes XASM to complain. To get to the root of the problem, it will be invaluable to be able to easily browse the assembly file it generates.
- You can learn a lot about how everything works by simply observing the compiler's output.

In each of these three cases, the common thread is that a human will have to read the compiler's output on at least a semi-regular basis. To this end, it's important that the compiler do everything it can to make the assembly file as human-like as possible, to enhance the reader's comfort and minimize confusion.

Of course, XASM couldn't care less either way. You specifically designed the assembler to filter out all forms of extraneous whitespace, comments, and other such human formatting, just as the compiler will.

The General Format

Before writing any emitter code, it would be a good idea to decide on a single, general format used to create uniform assembly output files all across the board. Here's what I came up with:

```
; Filename.XASM

; Source File: Filename.XSS
; XSC Version: 0.8
;   Timestamp: Thu Sep 05 00:42:46 2002

; ---- Directives ---------------------------------

   ; Directives go here

; ---- Global Variables ---------------------------

   ; Global variable declarations go here
; ---- Functions ----------------------------------

   ; Non-_Main () function declarations go here

; ---- Main ---------------------------------------

   ; _Main ()'s function declaration goes here, if present
```

As you can see, this is designed to mimic the formatting style I've been using throughout the book. Each segment of the file is partitioned in a very visual, verbose manner that helps guide the reader. Each file begins with a standard header, which states the file's name, the name of the .XSS source file from which it was generated, the version of the assembler that created it, and a timestamp.

Immediately following the header are declarations, such as `SetStackSize` and `SetPriority`. Following those are global variable declarations. After the globals come the definitions for each of the script's functions, *except* `_Main ()`. As in my own scripts, the XtremeScript compiler will emit `_Main ()` separately, in its own fenced off area.

Generating this basic skeleton is easy—it's just a series of hard-coded `fprintf ()` calls. The real issue is emitting the code and declarations that lie within each segment. The rest of this section covers the emission of these segments, one by one.

Global Definitions

The first aspects of the code emitter to understand are the few basic global definitions it uses. First up is the global file handle it uses to track the output file as it's written to:

```
FILE * g_pOutputFile = NULL;
```

Next is an array of strings that are used to map I-code instruction opcodes to their human-readable mnemonics. The opcode of each instruction is used as an index into the array, which allows for an easy one-to-one mapping. If you recall Chapter 10, you may notice that I lifted this directly from the original XVM prototype, which used it to print the mnemonic of the instruction it was currently executing:

```
char ppstrMnemonics [][ 12 ] =
{
    "Mov",
    "Add", "Sub", "Mul", "Div", "Mod", "Exp", "Neg", "Inc", "Dec",
    "And", "Or", "XOr", "Not", "ShL", "ShR",
    "Concat", "GetChar", "SetChar",
    "Jmp", "JE", "JNE", "JG", "JL", "JGE", "JLE",
    "Push", "Pop",
    "Call", "Ret", "CallHost",
    "Pause", "Exit"
};
```

Last is a single constant that is used to track the width of tab stops:

```
#define TAB_STOP_WIDTH                      8
```

This will come in handy when aligning the columns of instructions and their operands in the outputted code.

Emitting the Header

The header is probably the easiest part of the output file, and because it comes at the very top, is a good place to start. The header is emitted by the `EmitHeader ()` function:

```
void EmitHeader ()
{
    // Get the current time
    time_t CurrTimeMs;
    struct tm * pCurrTime;
    CurrTimeMs = time ( NULL );
    pCurrTime = localtime ( & CurrTimeMs );

    // Emit the filename
    fprintf ( g_pOutputFile, "; %s\n\n", g_pstrOutputFilename );

    // Emit the rest of the header
    fprintf ( g_pOutputFile, "; Source File: %s\n", g_pstrSourceFilename );
    fprintf ( g_pOutputFile, "; XSC Version: %d.%d\n",
        VERSION_MAJOR, VERSION_MINOR );
    fprintf ( g_pOutputFile, ";   Timestamp: %s\n", asctime ( pCurrTime ) );
}
```

The function first calculates the time and date with the `localtime ()` function. `localtime ()` returns a pointer to a `tm` structure containing a full timestamp based on the current time in milliseconds, which is returned by `time ()` and stored in the `time_t` structure instance `CurrTimeMs`. You can store the result of `localtime ()` in `pCurrTime` for use in a subsequent `fprintf ()` call.

You then emit the filename, which is of course readily available in `g_pstrOutputFilename`, followed by the rest of the header. This includes the original source filename, as found in g_pstrSourceFilename, and the version of the compiler, found in VERSION_MAJOR and VERSION_MINOR (defined in xsc.h):

```
#define VERSION_MAJOR              0
#define VERSION_MINOR              8
```

The final line of the header is the timestamp calculated earlier. To convert the contents of the structure pointed to by pCurrTime to something that can be printed to a file by fprintf (),use the asctime () to convert it to a string representation.

Emitting Directives

The emission of directives is pretty straightforward; the only issue to keep in mind is that if a directive hasn't been defined by the users (via the command-line), it should be left out of the generated code. Directive emission is handled by EmitDirectives ():

```
void EmitDirectives ()
{
    // If directives were emitted, this is set to TRUE so we remember to
    // insert extra line breaks after them
    int iAddNewline = FALSE;

    // If the stack size has been set, emit a SetStackSize directive
    if ( g_ScriptHeader.iStackSize )
    {
        fprintf ( g_pOutputFile, "\tSetStackSize %d\n",
            g_ScriptHeader.iStackSize );
        iAddNewline = TRUE;
    }

    // If the priority has been set, emit a SetPriority directive
    if ( g_ScriptHeader.iPriorityType != PRIORITY_NONE )
    {
        fprintf ( g_pOutputFile, "\tSetPriority " );
        switch ( g_ScriptHeader.iPriorityType )
        {
    // Low rank
            case PRIORITY_LOW:
                fprintf ( g_pOutputFile, PRIORITY_LOW_KEYWORD );
                break;
    // Medium rank
            case PRIORITY_MED:
                fprintf ( g_pOutputFile, PRIORITY_MED_KEYWORD );
                break;

    // High rank
            case PRIORITY_HIGH:
                fprintf ( g_pOutputFile, PRIORITY_HIGH_KEYWORD );
                break;

    // User-defined time slice
            case PRIORITY_USER:
                fprintf ( g_pOutputFile, "%d", g_ScriptHeader.iUserPriority );
                break;
        }
        fprintf ( g_pOutputFile, "\n" );
        iAddNewline = TRUE;
    }

    // If necessary, insert an extra line break
    if ( iAddNewline )
        fprintf ( g_pOutputFile, "\n" );
}
```

The first thing the function does is set a flag called `iAddNewLine` to `FALSE`. This flag is used to determine whether the function should emit a trailing newline after the directives. This will make a bit more sense when you see how an entire file is emitted in the last section, but for now, just think of it like this; if no directives were set by the user, the function shouldn't output anything. If one or both of the directives were set, however, that's one or two more lines emitted by the function that wouldn't have been there anyway. To keep the overall formatting of the file consistent, this extra content should be padded with an extra newline to separate it from whatever might come next. This extra line should be generated *only* if necessary. If this doesn't make perfect sense, yet, however, don't worry. You'll see more of why this is important in a later section. Either way, it's a rather trivial formatting detail that has little to do with the overall theory of the code emitter.

The function first checks the script header's `iStackSize` field. If it's nonzero, the emitter takes that as a sign that the user set it to a specific size that should be reflected in the output file. To emit the directive, a single `fprintf ()` call is made to print the "`SetStackSize`" string, followed by a single space and the `iStackSize` field's value. Note that the `iAddNewLine` flag is set after emitting the directive, and that the directive is inset by a single tab stop.

The next directive is `SetPriority`, whose value is represented within the script header by two separate fields. Before doing anything, the function determines whether the script header's `iPriorityType` field is `PRIORITY_TYPE_NONE`. If so, it's taken as a sign that the user never entered a priority. Otherwise, it's assumed to be the type of priority requested.

`fprintf ()` is called first to emit the "`SetPriority`" string, followed by a space. A `switch` block is then used to emit the proper priority value, depending on the `iPriorityType` field. If it's one of the `PRIORITY_LOW`, `PRIORITY_MED` or `PROIRITY_HIGH` constants, the corresponding `PRIORITY_*_KEYWORD` string constant is emitted. Otherwise, it's a user-defined time slice duration (`PRIORITY_TYPE_USER`), so the script header's `iUserPriority` value is emitted.

Finally, a newline is emitted if the `iAddNewLine` flag was set at any point during the function.

Emitting Symbol Declarations

With the header and directives out of the way, the next stop on your way down the output file are the global variable and array declarations. To emit these declarations, all you need to do is scan through the symbol table, read the relevant nodes, and print them in the style and format of an XVM declaration. For example, a symbol node whose identifier string is "`MyVar`" and whose size is 1 can be emitted like this:

```
Var MyVar
```

A node whose identifier string is "`MyArray`" and whose size is 16 can be emitted like this:

```
Var MyArray [ 16 ]
```

The general formats for variable and array declaration emission are as follows:

```
Var <pstrIdent>
Var <pstrIdent> [ <iSize> ]
```

Of course, there's also the issue of a variable's type, as well as its scope. Because the `iType` variable can differentiate between variables and parameters, you need a third format in case `iType` is equal to `SYMBOL_TYPE_PARAM`:

```
Param <pstrIdent>
```

This process is illustrated in Figure 14.26.

In terms of scope, the key to remember is this: even though global and local declarations are located in different places within the script, they're composed of the exact same token sequences. The only real difference is that global declarations never use the `Param` directive. Because of this, it would be silly to handle local and global symbol declaration emission in separate functions; because the logic is the same in both places, a more intelligent solution would be to simply code a single function that emits symbol declarations within a specified scope. This function is called

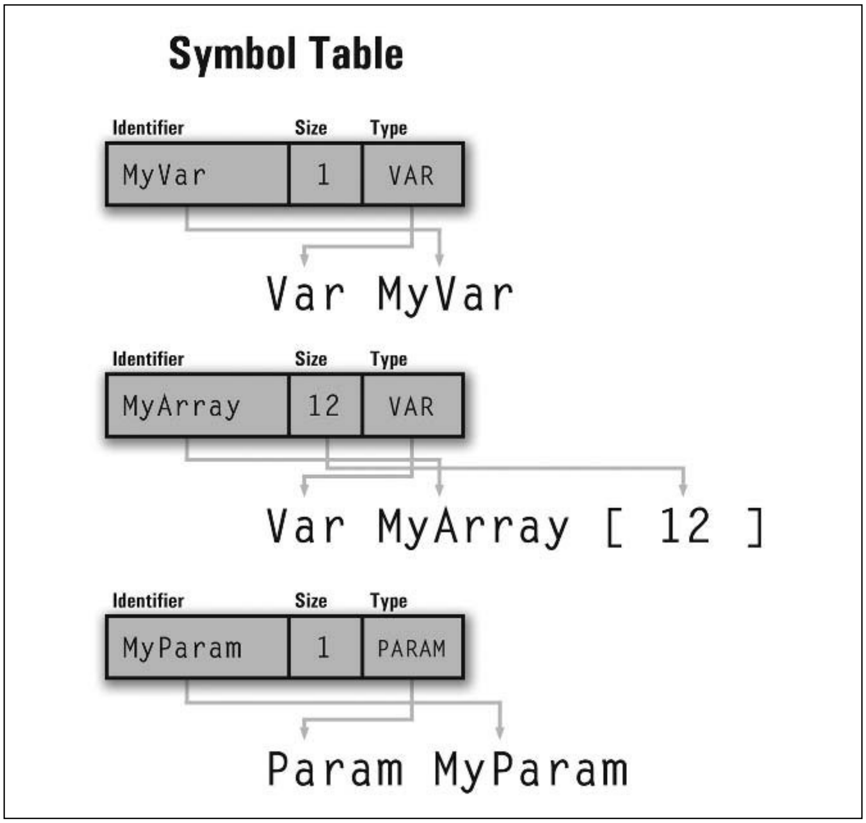


Figure 14.26

Emitting symbol declarations based on the contents of the symbol table.

`EmitScopeSymbols ()` and can be used to emit both the global declarations at the top of the script, and the local declarations within each function:

```
void EmitScopeSymbols ( int iScope, int iType )
{
    // If declarations were emitted, this is set to TRUE so we remember to
    // insert extra line breaks after them
    int iAddNewline = FALSE;

    // Local symbol node pointer
    SymbolNode * pCurrSymbol;

    // Loop through each symbol in the table to find the match
    for ( int iCurrSymbolIndex = 0;
              iCurrSymbolIndex < g_SymbolTable.iNodeCount;
              ++ iCurrSymbolIndex )
    {
    // Get the current symbol structure
        pCurrSymbol = GetSymbolByIndex ( iCurrSymbolIndex );

    // If the scopes and parameter flags match, emit the declaration
        if ( pCurrSymbol->iScope == iScope && pCurrSymbol->iType == iType )
        {
    // Print one tab stop for global declarations, and two for locals
            fprintf ( g_pOutputFile, "\t" );
            if ( iScope != SCOPE_GLOBAL )
                fprintf ( g_pOutputFile, "\t" );

    // Is the symbol a parameter?
            if ( pCurrSymbol->iType == SYMBOL_TYPE_PARAM )
                fprintf ( g_pOutputFile, "Param %s", pCurrSymbol->pstrIdent );

    // Is the symbol a variable?
            if ( pCurrSymbol->iType == SYMBOL_TYPE_VAR )
            {
                fprintf ( g_pOutputFile, "Var %s", pCurrSymbol->pstrIdent );

    // If the variable is an array, add the size declaration
                if ( pCurrSymbol->iSize > 1 )
                    fprintf ( g_pOutputFile, " [ %d ]", pCurrSymbol->iSize );
            }
            fprintf ( g_pOutputFile, "\n" );
            iAddNewline = TRUE;
        }
    }

    // If necessary, insert an extra line break
    if ( iAddNewline )
        fprintf ( g_pOutputFile, "\n" );
}
```

After clearing the `iAddNewLine` flag, the function begins a traversal of the symbol table to find all symbols matching the specified scope. Upon a match, the function ensures that the symbol is also of the specified type. `EmitScopeSymbols ()` allows the caller to emit a scope's variables and parameters separately, which will come in handy in the next section when you emit functions. If the symbol matches both the specified scope and type, it's time to emit it. The first step is to emit the appropriate number of tab stops. The function can be used for both global and local declarations, and this fact is reflected here. Globals and functions are both indented by a single tab. So, a global variable declaration only needs one tab stop to precede it. However, because a local declaration's function is one tab in as well, the declaration *itself* needs two tab stops so it appears to be "within" its surrounding function. Here's an example of what I mean:

```
; ---- Globals ------------------------------------

    Var MyGlobal

; ---- Functions ----------------------------------

    Func MyFunc
    {
        Var MyLocal
    }
```

Notice that the global is inset by only one tab stop, whereas the local is indented by two. After emitting the tab stops, the function checks the specified type. If it's `SYMBOL_TYPE_PARAM`, the `Param` directive is emitted. Otherwise, `Var` is the output. In both cases, the directive is immediately followed by a single space and the symbol's identifier, as found in its node's `pstrIdent` field. At this point, both single variables and parameters have been emitted, but arrays need special attention. This is handled by determining whether a variable symbol's `iSize` node is greater than one. If so, a second call to `fprintf ()` is made to emit the size value enclosed in braces.

This function will emit a contiguous sequence of declarations for all variables and parameters within a given scope. You can directly apply this to the emission of functions, so let's check them out next.

Emitting Functions

Functions are without question the most complex aspect of code emission in the XtremeScript compiler, because they're solely responsible for the emission of actual I-code. A function's declaration takes the following general form:

```
Func <pstrName>
{
    ; Parameter declarations
    ; Local variable declarations

    ; Code
}
```

Everything except the code is a snap; the function declaration itself is just a matter of emitting the Func directive, the function node's pstrName field, and the curly braces. Parameters and local variables can each be emitted with two calls to the EmitScopeSymbols () function developed in the last section. The code, however, is where things get tricky. Because you're emitting instruction mnemonics and operands based on a purely numeric I-code representation, it's almost as if it's the reverse of the process performed by XASM. In this regard, writing the code emitter is very similar to writing a disassembler.

NOTE

In case you aren't familiar with the term, a *disassembler* is a utility that converts the compiled instruction stream of an executable back to assembly language, by replacing each opcode with a mnemonic string, and each operand with its human-readable equivalent. Because of this, it's more or less the opposite of an assembler, hence the name. Disassemblers are useful when reverse-engineering a compiled executable when the source is not accessible.

The Function and Local Symbol Declarations

Functions are emitted with EmitFunc (), which emits a single function based on a function node pointer that is passed from the caller. Let's get started:

```
void EmitFunc ( FuncNode * pFunc )
{
    // Emit the function declaration name and opening brace
    fprintf ( g_pOutputFile, "\tFunc %s\n", pFunc->pstrName );
    fprintf ( g_pOutputFile, "\t{\n" );

    // Emit parameter declarations
    EmitScopeSymbols ( pFunc->iIndex, SYMBOL_TYPE_PARAM );

    // Emit local variable declarations
    EmitScopeSymbols ( pFunc->iIndex, SYMBOL_TYPE_VAR );
```

As I said, the easy part of function emission was taken care of in only a few lines. The Func directive was followed by the function's name, a line break, and an opening curly brace. If the function node's pstrName field pointed to the string "MyFunc", the emitter would produce the following so far:

```
Func MyFunc
{
```

Two calls to `EmitScopeSymbols ()`, used to emit the function's parameters and variables (in that order), are then made. At this point, all that remains is the code. The function node stores this code in its nested `ICodeStream` linked list, so you begin by determining whether it contains anything:

```
// Does the function have an I-code block?

if ( pFunc->ICodeStream.iNodeCount > 0 )
{
```

Once you know there's an I-code stream to process, you can begin a traversal of the list to output each node. Once you have the node, you can use its `iType` field to determine what it is and how to emit it:

```
// Used to determine if the current line is the first
int iIsFirstSourceLine = TRUE;

// Yes, so loop through each I-code node to emit the code
for ( int iCurrInstrIndex = 0; iCurrInstrIndex < pFunc->ICodeStream.iNodeCount; ++
iCurrInstrIndex )
{
    // Get the I-code instruction structure at the current node
    ICodeNode * pCurrNode = GetICodeNodeByImpIndex ( pFunc->iIndex,
        iCurrInstrIndex );

    // Determine the node type
    switch ( pCurrNode->iType)
    {
```

The `iIsFirstSourceLine` flag is yet another formatting-related issue. As you'll see as you get deeper into this function's code, it can be beneficial to determine whether the line currently being printed is the first in the I-code block, to resolve certain vertical whitespace issues. I'll come back to this. In the meantime, you've got a copy of the I-code node pointer and are about to dive into a `switch` block that will let you emit it based on its type.

At this point, there are three I-code node types you could be dealing with:

- **Source code annotation.** Certain I-code nodes are reserved entirely for holding a pointer to a string within the source code linked list. These are simply emitted as comments to help guide a human reader through the assembly output.

- **I-code instruction.** An I-code instruction in the XtremeScript compiler has a one-to-one mapping with the XVM instruction set, so all you have to do here is emit the proper mnemonic and each of its operands.
- **Jump target.** Jump targets are ultimately translated to labels by the code emitter, which must generate a unique label name on the fly. You'll learn how this is done shortly.

Source Code Annotation

Let's start at the top and look at the emission code for a source code annotation node:

```
case ICODE_NODE_SOURCE_LINE:
{
    // Make a local copy of the source line
    char * pstrSourceLine = pCurrNode->pstrSourceLine;

    // If the last character of the line is a line break, clip it
    int iLastCharIndex = strlen ( pstrSourceLine ) - 1;
    if ( pstrSourceLine [ iLastCharIndex ] == '\n' )
        pstrSourceLine [ iLastCharIndex ] = '\0';

    // Emit the comment, but only prepend it with a line break
    // if it's not the first one
    if ( ! iIsFirstSourceLine )
        fprintf ( g_pOutputFile, "\n" );

    fprintf ( g_pOutputFile, "\t\t; %s\n\n", pstrSourceLine );

    break;
}
```

These are easy. The function first makes a local copy of the source line pointer for convenience, and then clips any trailing line breaks that may be present so that it doesn't mess up the formatting you'd like to enforce. You can make direct alterations to the code without making a physical copy first at this point because you're at the end of the compilation pipeline and you'll never need it again. Once you've ensured that the line break is gone, you can check the `iIsFirstSourceLine` flag. If this is the first line of code in the I-code block, you can already rely on the blank line appended to the last emission. If you're inside the I-code block, however, you have to generate your own. Following this vertical whitespace is the commented source note itself, containing the original line of code. Note the use of two tab stops to ensure that the comment appears within its surrounding function.

I-Code Instructions

Instructions are hands down the most complex part about emitting I-code. Fortunately, the process is really just a regurgitation of the ones performed many times during the implementation of XASM and the XVM.

Naturally, the first thing to do when emitting an instruction is to map the opcode to its corresponding string in the mnemonic array declared earlier, and then to print it. This mnemonic should be immediately followed by either one or two tab stops, depending on its length. To understand why this is done, consider the following fragment:

```
Mov    X, Y
Add    X, Z
Jmp    MyLabel
```

This does fine with a single tab stop. The problem occurs when the `CallHost` instruction finds its way into the stream:

```
Mov    X, Y
Add    X, Z
Jmp    MyLabel
CallHost    MyHostFunc
```

Suddenly, the columns are misaligned and all hell is breaking loose! I admit I'm a bit anal when it comes to organization and formatting, but I still stand by the results. By appending `CallHost` with a single tab and `Mov`, `Add`, and `Jmp` by two, you get much cleaner output:

```
Mov         X, Y
Add         X, Z
Jmp         MyLabel
CallHost    MyHostFunc
```

It may be a little on the "spacey" side, but it's a godsend when you're trying to wade through a thousand lines of the stuff and can barely keep your head straight as it is. To combat this, the length of the mnemonic is compared to the `TAB_STOP_WIDTH` constant mentioned earlier. If the mnemonic is greater, a single tab stop is used; otherwise, two are emitted. Here's the next block of code, implementing everything just discussed:

```
case ICODE_NODE_INSTR:
{
    // Emit the opcode
    fprintf ( g_pOutputFile, "\t\t%s", ppstrMnemonics
        [ pCurrNode->Instr.iOpcode ] );
    // Determine the number of operands
    int iOpCount = pCurrNode->Instr.OpList.iNodeCount;

    // If there are operands to emit, follow the instruction with some space
    if ( iOpCount )
    {
    // All instructions get at least one tab
        fprintf ( g_pOutputFile, "\t" );

    // If it's less than a tab stop's width in characters, however, they
    // get a second
        if ( strlen ( ppstrMnemonics [ pCurrNode->Instr.iOpcode ] ) <
            TAB_STOP_WIDTH )
            fprintf ( g_pOutputFile, "\t" );
    }
```

As always seems to be the case, however, the real complications arise when the operands are emitted. As usual, it's because operands come in many forms, each of which must be handled differently. In addition to emitting the operand, it's also important to remember that each operand must be followed by a comma, unless it's the last. Here's the code for looping through each operand in the I-code node's list and emitting them:

```
for ( int iCurrOpIndex = 0; iCurrOpIndex < iOpCount; ++ iCurrOpIndex )
{
    // Get a pointer to the operand structure
    Op * pOp = GetICodeOpByIndex ( pCurrNode, iCurrOpIndex );

    // Emit the operand based on its type
    switch ( pOp->iType )
    {
    // Integer literal
        case OP_TYPE_INT:
            fprintf ( g_pOutputFile, "%d", pOp->iIntLiteral );
            break;

    // Float literal
        case OP_TYPE_FLOAT:
            fprintf ( g_pOutputFile, "%f", pOp->fFloatLiteral );
            break;
    // String literal
        case OP_TYPE_STRING_INDEX:
            fprintf ( g_pOutputFile, "\"%s\"", GetStringByIndex
                ( & g_StringTable, pOp->iStringIndex ) );
            break;

    // Variable
        case OP_TYPE_VAR:
            fprintf ( g_pOutputFile, "%s", GetSymbolByIndex
                ( pOp->iSymbolIndex )->pstrIdent );
            break;

    // Array index absolute
        case OP_TYPE_ARRAY_INDEX_ABS:
            fprintf ( g_pOutputFile, "%s [ %d ]",
            GetSymbolByIndex ( pOp->iSymbolIndex )->pstrIdent, pOp->iOffset );
            break;

    // Array index variable
        case OP_TYPE_ARRAY_INDEX_VAR:
            fprintf ( g_pOutputFile, "%s [ %s ]", GetSymbolByIndex
                ( pOp->iSymbolIndex )->pstrIdent,
                GetSymbolByIndex ( pOp->iOffsetSymbolIndex )->pstrIdent );
            break;

    // Function
        case OP_TYPE_FUNC_INDEX:
            fprintf ( g_pOutputFile, "%s", GetFuncByIndex
                ( pOp->iSymbolIndex )->pstrName );
            break;

    // Register (just _RetVal for now)
        case OP_TYPE_REG:
            fprintf ( g_pOutputFile, "_RetVal" );
            break;

    // Jump target index
        case OP_TYPE_JUMP_TARGET_INDEX:
            fprintf ( g_pOutputFile, "_L%d", pOp->iJumpTargetIndex );
            break;
    }
```

```
    // If the operand isn't the last one, append it with a comma and space
    if ( iCurrOpIndex != iOpCount - 1 )
        fprintf ( g_pOutputFile, ", " );
}
```

This should look pretty straightforward, but here's a quick rundown. Integer operands are printed by simply emitting the `iIntLiteral` field of the `Op` structure. Floats are handled the same way; they come directly out of the `fFloatLiteral` field. Strings are almost emitted in their exact form, but must be surrounded by double-quotes. The string itself is obtained with a call to `GetStringByIndex ()`, using the `iStringIndex` field. Variables are represented simply as their identifier string, `pstrIdent`, so that's all that needs to be emitted. In the case of arrays indexed with absolute values, the identifier string is immediately followed by an integer value, stored in the `iOffset` field, surrounded by braces. The same goes for arrays indexed with variables, except that the indexing variable's identifier is placed in between the braces, instead of an integer index. Function operands (used in the `Call` and `CallHost` instructions) are simply emitted as their `pstrName` string. Register codes are up next; for now, because the XVM only has one register, the code itself is ignored and `_RetVal` is unconditionally emitted.

Last up are jump targets, which are emitted as label names. Because the jump target is simply an integer value, you have to construct a label name on the fly. Fortunately, this is easy to do. Remember, within a given scope, labels have to be unique. Because of this, you can use the jump target's integer index as the basis for labels that will always be unique, because each jump target's index is unique. For example, if you convert the index to a string and prefix it with something like "`_L`", you could generate a limitless amount of unique labels in a single line of code. For example, if you have three jump indexes, 0, 1, and 2, they'll be emitted as the labels `_L0`, `L1`, and `_L2`. The leading underscore is the convention I've used throughout the book to represent special or compiler-generated identifiers, and the `L` of course stands for label.

With the operand emitted, the last step is to immediately follow it with a comma and a space (to help visually separate it from the next operand), unless it's the last one in the list. The instruction is now complete, so you simply tack on a line break and consider it finished:

```
// Finish the line
fprintf ( g_pOutputFile, "\n" );
break;
```

Jump Targets

Luckily, the last I-code node type is extremely simple. Its only job is to convert a jump table index into a label (using the same process devised in the last section) and emitting it in the form of a label declaration:

```
case ICODE_NODE_JUMP_TARGET:
{
    // Emit a label in the format _LX, where X is the jump target
    fprintf ( g_pOutputFile, "\t_L%d:\n", pCurrNode->iJumpTargetIndex );
}
```

It's simply a matter of prefixing the jump target index with `_L` to make a valid label, and then following it with a colon to turn it into a declaration.

Finishing Up

The rest of the operand emission loop and the `EmitFunc ()` function is pretty uneventful. Let's have a quick look:

```
        }
    // Update the first line flag
        if ( iIsFirstSourceLine )
            iIsFirstSourceLine = FALSE;
    }
}
else
{
    // No, so emit a comment saying so
    fprintf ( g_pOutputFile, "\t\t; (No code)\n" );
}

// Emit the closing brace
fprintf ( g_pOutputFile, "\t}" );
```

After emitting the I-code node, regardless of its type, the `iFirstSourceLine` flag is cleared. You'll also notice an `else` clause to the original determination of whether the function had any I-code in the first place; if it doesn't, the emitter will simply generate a "`(No code)`" message in the form of a comment. The function is then wrapped up with the emission of its closing curly brace.

Emitting a Complete XVM Assembly File

With the capability to emit the script's header and directives, as well as its variables and functions, it's time to wrap everything up into a single file that will emit an entire XVM assembly file. This main code emission function is called `EmitCode ()`, and starts by opening the output file and emitting the header:

```
void EmitCode ()
{
    // ---- Open the output file
    if ( ! ( g_pOutputFile = fopen ( g_pstrOutputFilename, "wb" ) ) )
        ExitOnError ( "Could not open output file for output" );

    // ---- Emit the header
    EmitHeader ();
```

Immediately following the header are the directives:

```
// ---- Emit directives
fprintf ( g_pOutputFile, "; ---- Directives ---------------------------\n\n" );
EmitDirectives ();
```

Up next are the script's global variables, which are emitted with a call to `EmitScopeSymbols ()`, with the `iScope` parameter set to the `SCOPE_GLOBAL` constant and the `iType` parameter set to `SYMBOL_TYPE_VAR` (because there's no such thing as a global parameter):

```
// ---- Emit global variable declarations
fprintf ( g_pOutputFile, "; ---- Global Variables ---------------------\n\n" );

// Emit the globals by printing all non-parameter symbols in the global scope
EmitScopeSymbols ( SCOPE_GLOBAL, FALSE );
```

The next segment of the XVM assembly file contains each of its function definitions, with the exception of `_Main ()` if it's present. Even with the aid of `EmitFunc ()`, this is a more complex process than the last three have been, because you need to manually traverse the function list in order to pass the proper function node pointers. Furthermore, you need to keep an eye out for the `_Main ()` function, and suppress its emission. You have to remember to save its pointer for use in the next section. Here's the code:

```
// ---- Emit functions
fprintf ( g_pOutputFile, "; ---- Functions ----------------------------\n\n" );

// Local node for traversing lists
LinkedListNode * pNode = g_FuncTable.pHead;

// Local function node pointer
FuncNode * pCurrFunc;
```

```
// Pointer to hold the _Main () function, if it's found
FuncNode * pMainFunc = NULL;

// Loop through each function and emit its declaration and code, if functions
// exist
if ( g_FuncTable.iNodeCount > 0 )
{
    while ( TRUE )
    {
    // Get a pointer to the node
        pCurrFunc = ( FuncNode * ) pNode->pData;

    // Don't emit host API function nodes
        if ( ! pCurrFunc->iIsHostAPI )
        {
    // Is the current function _Main ()?
            if ( stricmp ( pCurrFunc->pstrName, MAIN_FUNC_NAME ) == 0 )
            {
    // Yes, so save the pointer for later (and don't emit it yet)
                pMainFunc = pCurrFunc;
            }
            else
            {
    // No, so emit it
                EmitFunc ( pCurrFunc );
                fprintf ( g_pOutputFile, "\n\n" );
            }
        }

    // Move to the next node
        pNode = pNode->pNext;
        if ( ! pNode )
            break;
    }
}
```

Begin by setting up a few variables. First is `pNode`, a linked list node pointer that starts off pointing at the head of the function table. Next is `pCurrFunc`, a function node pointer that will point to the current function's node structure. Last is `pMainFunc`, another function node pointer specifically set aside to store a pointer to the `_Main ()` function node if it's found during the traversal of the table. You intentionally set this to `NULL` for now.

The table traversal then begins, assuming it's not empty, and pCurrFunc is set to pNode's current pData member at each iteration. The first thing to determine is whether the current function is defined by the script, or whether it belongs to the host API. Host API functions are simply kept in the function table for the parser's benefit so it can validate function calls as the code is parsed. By the time the code emitter is running, they have no use and are ignored.

Assuming the function isn't part of the host API, it's determined whether the function is _Main (). If not, it's emitted with a call to EmitFunc () and followed by two line breaks. Otherwise, the pointer is saved in pMainFunc for later use. This wraps up the emission of functions.

The last steps are emitting the _Main () function, if present, and closing the output file:

```
// ---- Emit _Main ()
fprintf ( g_pOutputFile, "; ---- Main -------------------------------------" );

// If the last pass over the functions found a _Main () function. emit it
if ( pMainFunc )
{
    fprintf ( g_pOutputFile, "\n\n" );
    EmitFunc ( pMainFunc );
}

// ---- Close output file
fclose ( g_pOutputFile );
```

That's it! You've converted the I-code, symbol table, function table, and string table to a fully formatted and valid XVM assembly file. It's all ready to be fed to XASM, so next you find out how that's done and finish the job.

GENERATING THE FINAL EXECUTABLE

Finally, you're at the last stage of the pipeline. With the exception of the parser, you've seen every step the source code takes as it slips and slides from its initial raw form, to a compiled I-code representation, to a human-readable XVM assembly file generated by the code emitter. Now, with every piece of the puzzle in place, you can make a quick call to the XASM assembler built in Chapter 9 to deliver the coup de grace.

The execution of XASM is handled by the AssmblOutputFile () function, found in xsc.cpp|h. All you're really doing is using the C standard library function spawnv () to invoke a new process. Here's the code:

```
void AssmblOutputFile ()
{
    // Command-line parameters to pass to XASM
    char * ppstrCmmndLineParams [ 3 ];

    // Set the first parameter to "XASM" (not that it really matters)
    ppstrCmmndLineParams [ 0 ] = ( char * ) malloc ( strlen ( "XASM" ) + 1 );
    strcpy ( ppstrCmmndLineParams [ 0 ], "XASM" );

    // Copy the .XASM filename into the second parameter
    ppstrCmmndLineParams [ 1 ] = ( char * )
        malloc ( strlen ( g_pstrOutputFilename ) + 1 );
    strcpy ( ppstrCmmndLineParams [ 1 ], g_pstrOutputFilename );

    // Set the third parameter to NULL
    ppstrCmmndLineParams [ 2 ] = NULL;

    // Invoke the assembler
    spawnv ( P_WAIT, "XASM.exe", ppstrCmmndLineParams );

    // Free the command-line parameters
    free ( ppstrCmmndLineParams [ 0 ] );
    free ( ppstrCmmndLineParams [ 1 ] );
}
```

This function is basically a wrapper for `spawnv ()`, which spawns new processes. If you're not familiar with this function, it's declared in `process.h` and has the following prototype:

```
int spawnv ( int mode, const char * cmdname, const char * const * argv );
```

In a nutshell, the function is designed to load and execute a new process from another. In this case, you can use it to invoke the XASM executable, which you'll provide in the same working directory as the XtremeScript compiler.

`spawnv ()`'s parameters are described in Table 14.3.

In short, this function lets you simulate what would happen if you typed this into the command line:

```
XASM MyFunc.xasm
```

Table 14.3 spawnv () Parameters

Name	Type	Description
mode	Integer	The "execution mode" for the calling process. What this means is basically whether you'll wait idly for XASM to finish. You can pass it P_WAIT to tell the function that you would like to wait until the assembler is done.
cmdname	String	The path of the executable to launch.
argv	String Array	The command-line arguments expressed as an array of string pointers. The last element of this array must be a null pointer so the function can determine how many arguments are being passed.

The AssmblOutputFile () function begins by declaring a string array of three elements called ppstrCmmndLineParams []. You allocate three elements because the argv [] array passed to a console application's main () function always includes the name of the executable as typed at the command line at index zero of the array. The second element in the array is the filename of the .XASM file you want to assemble, and the third is set to NULL so spawnv () can determine when it's processed all of the parameters you want to pass.

Even though it's not necessary, the function sets the first parameter to the string "XASM". The second parameter is set to the output filename created originally by VerifyFilenames (). Notice that you don't explicitly specify an executable filename; you do this because XASM allows the name of the executable to be omitted and uses the name of the .XASM file in its place. Lastly, you set the pointer at index 2 to NULL.

With the command-line arguments in place, you're ready to invoke the assembler. You do this with a call to spawnv (), of course. The first parameter you pass is P_WAIT, a constant that causes the compiler to wait until the new process terminates. This makes the invocation of the assembler very similar to a function call. The next parameter is "XASM.exe", which is of course the assembler itself. As I mentioned earlier, you'll simply place a copy of the executable in the compiler's working directory. The last parameter is of course the command-line parameter array.

By following these steps, the .XASM file created by the code emitter module will be compiled into a fully functional .XSE executable. Back in the compiler's main () function, as you saw earlier, the original .XASM is then deleted.

Wrapping It All Up

At this point, you've seen how every component of the XtremeScript compiler was designed and implemented from the ground up, and for the most part, seen how they fit together. This section covers a few loose ends left over from the previous discussion.

Initiating the Compilation Process

Earlier in the chapter, the compiler's `main ()` function was listed as a general layout of the lifespan of the program. There still remains one function called from `main ()` you haven't seen yet, although it doesn't do much in this incarnation of the compiler. It's called `CompileSourceFile ()` and is defined in `xsc.cpp|h`:

```
void CompileSourceFile ()
{
    // Parse the source file to create an I-code representation
    ParseSourceCode ();
}
```

As you can see, it's currently just one line that calls `ParseSourceCode ()`. You haven't defined this function yet, as it's the focus on the next chapter. For now, just understand that this is where the real action begins. After the loader and preprocessor have done their jobs, `CompileSourceFile ()` calls `ParseSourceCode ()` to create an I-code representation of the code. You'll add a bit more to this function in the next chapter.

Printing Compilation Statistics

As a final touch (which was also present in XASM), I like to display a number of "compilation statistics" that are gathered during the compilation process. They're just an idle novelty in most cases, but they can be rather helpful when debugging. The basic idea is to just print a bunch of miscellaneous totals, such as the number of variables, globals, arrays, functions, and so on. This is handled by the `PrintCompileStats ()` function, found in `xsc.cpp|h`:

```
void PrintCompileStats ()
{
    // ---- Calculate statistics

    // Symbols
    int iVarCount = 0,
        iArrayCount = 0,
        iGlobalCount = 0;
```

```
// Traverse the list to count each symbol type
for ( int iCurrSymbolIndex = 0;
      iCurrSymbolIndex < g_SymbolTable.iNodeCount;
      ++ iCurrSymbolIndex )
{
// Create a pointer to the current symbol structure
    SymbolNode * pCurrSymbol = GetSymbolByIndex ( iCurrSymbolIndex );

// It's an array if the size is greater than 1
    if ( pCurrSymbol->iSize > 1 )
        ++ iArrayCount;

// It's a variable otherwise
    else
        ++ iVarCount;

// It's a global if it's stack index is nonnegative
    if ( pCurrSymbol->iScope == 0 )
        ++ iGlobalCount;
}

// Instructions
int iInstrCount = 0;

// Host API Calls
int iHostAPICallCount = 0;

// Traverse the list to count each symbol type
for ( int iCurrFuncIndex = 1;
      iCurrFuncIndex <= g_FuncTable.iNodeCount;
      ++ iCurrFuncIndex )
{
// Create a pointer to the current function structure
    FuncNode * pCurrFunc = GetFuncByIndex ( iCurrFuncIndex );

// Determine if the function is part of the host API
    ++ iHostAPICallCount;

// Add the function's I-code instructions to the running total
    iInstrCount += pCurrFunc->ICodeStream.iNodeCount;
}
```

```
// Print out final calculations
printf ( "%s created successfully!\n\n", g_pstrOutputFilename );
printf ( "Source Lines Processed: %d\n", g_SourceCode.iNodeCount );
printf ( "            Stack Size: " );
if ( g_ScriptHeader.iStackSize )
    printf ( "%d", g_ScriptHeader.iStackSize );
else
    printf ( "Default" );

printf ( "\n" );
printf ( "              Priority: " );
switch ( g_ScriptHeader.iPriorityType )
{
    case PRIORITY_USER:
        printf ( "%dms Timeslice", g_ScriptHeader.iUserPriority );
        break;
    case PRIORITY_LOW:
        printf ( PRIORITY_LOW_KEYWORD );
        break;
    case PRIORITY_MED:
        printf ( PRIORITY_MED_KEYWORD );
        break;
    case PRIORITY_HIGH:
        printf ( PRIORITY_HIGH_KEYWORD );
        break;
    default:
        printf ( "Default" );
        break;
}
printf ( "\n" );
printf ( "  Instructions Emitted: %d\n", iInstrCount );
printf ( "             Variables: %d\n", iVarCount );
printf ( "                Arrays: %d\n", iArrayCount );
printf ( "               Globals: %d\n", iGlobalCount);
printf ( "       String Literals: %d\n", g_StringTable.iNodeCount );
printf ( "        Host API Calls: %d\n", iHostAPICallCount );
printf ( "             Functions: %d\n", g_FuncTable.iNodeCount );
```

```
    printf ( "      _Main () Present: " );
    if ( g_ScriptHeader.iIsMainFuncPresent )
        printf ( "Yes (Index %d)\n", g_ScriptHeader.iMainFuncIndex );
    else
        printf ( "No\n" );
    printf ( "\n" );
}
```

It should all be pretty self-explanatory. A number of variables are created to hold various totals that are either read directly from the `iNodeCount` of tables or calculated by other means. After all the data is collected, it's printed to the screen in an aligned list.

Hard-coding a Test Script

This chapter's been pretty rough, and it would be a bit of a let down if there wasn't a demo or example of the compiler's capabilities to cap it all off. I must admit, you're at a pretty serious disadvantage without the help of the parser, because you have no real capability to translate code into I-code, which would ultimately become a pair of .XASM and .XSE files. That would be the best way to demonstrate the compiler's power, but you can't do anything like that until the next chapter.

So, in the meantime, we'll just have to make do with what we have and hard-code a script directly into the I-code module and the compiler's tables. You can then let it run as normal, and watch it convert it all into a fully formatted XVM assembly file and ultimately into an .XSE executable.

Because hard-coding the data directly into the compiler's structures is going to be a bit tedious, let's keep things *extremely* simple. You can start off by "hand-compiling" the following high-level code fragment, written in actual XtremeScript:

```
// Declare a global
var MyGlobal;

// Declare a main function
func _Main ()
{
    // Declare some locals
    var X;
    var Y [ 4 ];

    // Perform some basic arithmetic
    MyGlobal = 2;
    X = 8;
    // Calculate 2^8 and put the result in Y [ 1 ]
    Y [ 1 ] = MyGlobal ^ X;
}
```

By hand-compiling this code, meaning to compile it manually without the aid of an actual compiler, you can see pretty easily that the script should compile down to something along these lines:

```
Var MyGlobal

Func _Main
{
    Var X
    Var Y [ 4 ]

    Mov    MyGlobal, 2
    Mov    X, 8
    Exp    MyGlobal, X
    Mov    Y, MyGlobal
}
```

Now that you know what the assembly version of this script fragment should look like, you can hard-code the instructions, operands, functions, and symbols directly into the compiler and see what it spits out. Because `CompileSourceFile ()` isn't really being used for anything just yet, you can put all of our hard-coded logic there.

The Function

This hand-compiled script has only one function, `_Main ()`. The first order of business is hard-coding this function into the function table, like so:

```
// Hard-code a _Main () function into the table and save its index
int iMainIndex = AddFunc ( "_Main", FALSE );
```

Notice that the index returned by `AddFunc ()` is saved in `iMainIndex`. You'll need this later.

The Symbols

This script defines three variables—a global called `MyGlobal`, a local variable called `X`, and a local array of four elements called `Y []`. You can therefore break down the symbols according to Table 14.4.

Table 14.4 Test Script Symbols

Identifier	Size	Scope	Type
`MyGlobal`	1	Global	Variable
`X`		1	`_Main ()` Variable
`Y`		4	`_Main ()` Variable (array)

These can be hard-coded into the table with repeated calls to `AddSymbol ()`. Once again, it's important to save their indexes for later use:

```
// Hard-code symbols into the table and save their indexes
int iMyGlobalIndex = AddSymbol ( "MyGlobal", 1, SCOPE_GLOBAL,
    SYMBOL_TYPE_VAR );
int iXIndex = AddSymbol ( "X", 1, iMainIndex, SYMBOL_TYPE_VAR );
int iYIndex = AddSymbol ( "Y", 4, iMainIndex, SYMBOL_TYPE_VAR );
```

The symbol table is now populated with two variables, one of which is global, and a four-element array.

The Code

You have a function to write your code in, as well as variables for it to work with, so you're ready to hard-code the most important part. Before doing so, however, you need to allocate a few strings to hold the original high-level script discussed earlier. You can then add these to the I-code as annotations, and test the source code annotating functions of the code emitter:

```
// Allocate strings to hold each line of the high-level script
char * pstrLine0 = ( char * ) malloc ( MAX_SOURCE_LINE_SIZE );
strcpy ( pstrLine0, "MyGlobal = 2;" );

char * pstrLine1 = ( char * ) malloc ( MAX_SOURCE_LINE_SIZE );
strcpy ( pstrLine1, "X = 8;" );

char * pstrLine2 = ( char * ) malloc ( MAX_SOURCE_LINE_SIZE );
strcpy ( pstrLine2, "Y [ 1 ] = MyGlobal ^ X;" );
```

These three string buffers now contain the three lines of executable, non-declaration code from the original high-level script we hand-compiled. With these ready to go, let's add them, along with the instructions, to the I-code module.

The First Instruction

Here's the first instruction:

```
Mov    MyGlobal, 2
```

And this is the line of high-level code it was hand-compiled from:

```
MyGlobal = 2;
```

Here are the calls to the I-code module to add both the high-level source code annotation and the low-level instructions:

```
// Hard-code the instructions and source annotation into the I-code module
int iInstrIndex;

// MyGlobal = 2;
AddICodeSourceLine ( iMainIndex, pstrLine0 );
iInstrIndex = AddICodeInstr ( iMainIndex, INSTR_MOV );
AddVarICodeOp ( iMainIndex, iInstrIndex, iMyGlobalIndex );
AddIntICodeOp ( iMainIndex, iInstrIndex, 2 );
```

The first call is made to `AddICodeSourceLine ()`, to add the first source line annotation to the I-code module. This will be displayed directly above the instruction nodes that follow it. So, you call `AddICodeInstr ()` to add a `Mov` instruction to the `_Main ()` function, making sure to save the index. You then follow up with two operands. The first is the `MyGlobal` variable, which you add using `AddVarICodeOp ()`. The second is the integer literal `2`, which you add using `AddIntICodeOp ()`. Notice also that you only need one copy of `iInstrIndex`, because once an instruction is added, you never need to mess with it again and can reuse the same index variable.

The Second Instruction

This is the second instruction:

```
Mov    X, 8
```

This is the source line from which it was hand-compiled:

```
X = 8;
```

Here's the code used to hard-code it into the I-code module:

```
// X = 8;
AddICodeSourceLine ( iMainIndex, pstrLine1 );
iInstrIndex = AddICodeInstr ( iMainIndex, INSTR_MOV );
AddVarICodeOp ( iMainIndex, iInstrIndex, iXIndex );
AddIntICodeOp ( iMainIndex, iInstrIndex, 8 );
```

Once again, `AddICodeSourceLine ()` is called first to add the source line annotation. This is followed by the addition of a second `Mov` instruction with `AddICodeInstr ()`. The instruction is then fleshed out with two operands, the `X` variable and the integer literal `8`.

The Third and Fourth Instructions

Finally, here is the last line of the high-level script:

```
Y [ 1 ] = MyGlobal ^ X;
```

This statement, unlike the last two, hand-compiles down to two instructions rather than one:

```
Exp    MyGlobal, X
Mov    Y, MyGlobal
```

You therefore have to make more calls to the I-code module, but there's still only one source line annotation to add:

```
// Y [ 1 ] = MyGlobal ^ X;
AddICodeSourceLine ( iMainIndex, pstrLine2 );

iInstrIndex = AddICodeInstr ( iMainIndex, INSTR_EXP );
AddVarICodeOp ( iMainIndex, iInstrIndex, iMyGlobalIndex );
AddVarICodeOp ( iMainIndex, iInstrIndex, iXIndex );

iInstrIndex = AddICodeInstr ( iMainIndex, INSTR_MOV );
AddArrayIndexAbsICodeOp ( iMainIndex, iInstrIndex, iYIndex, 1 );
AddVarICodeOp ( iMainIndex, iInstrIndex, iMyGlobalIndex );
```

You have now added an `Exp` instruction for calculating the exponent, as well as a `Mov` for moving the final value from `Y` to `MyGlobal`. This completes the hard-coded I-code module, so you're ready to see what the compiler does with it!

The Results

When the compiler runs, the `CompileSourceFile ()` function will hard-code the data covered previously into the function table, symbol table, and I-code stream. From that point on, it will be as if that was read directly from the source file and converted by the parser. The rest of the compiler has no idea where any of it came from, and doesn't care. Because of this, you can test your compiler framework by examining the output it produces to ensure everything is correct. Make sure you run the compiler with the `-A` command-line option so it doesn't delete the .XASM file it produces.

When all is said and done, the framework should produce this:

```
; TEST.XASM

; Source File: TEST.XSS
; XSC Version: 0.8
;   Timestamp: Tue Sep 10 21:58:53 2002

; ---- Directives -----------------------------------------------------------
; ---- Global Variables -----------------------------------------------------

   Var MyGlobal

; ---- Functions ------------------------------------------------------------
; ---- Main -----------------------------------------------------------------

   Func _Main
   {
      Var X
      Var Y [ 4 ]

      ; MyGlobal = 2;
      Mov      MyGlobal, 2

      ; X = 8;
      Mov      X, 8

      ; Y [ 1 ] = MyGlobal ^ X;
      Exp      MyGlobal, X
      Mov      Y [ 1 ], MyGlobal
   }
```

How cool is this? You've created a perfectly valid, XASM-ready assembly file with full source code annotation. You now *know* the framework for the ever-evolving compiler works (at least, with as much certainty as you can derive from a single, simplistic test). With everything working so far, you can plow through the parser in the next chapter and create a finished, working compiler that's ready for full-on game scripting.

Summary

The clock is ticking, and with every new chapter you plow your way through, you get ever closer to the attainment of scripting mastery. At this point, you have a well-structured and thorough compiler framework that can already generate complete .XSE executables from hard-coded I-code data. The best part is, if you've followed this chapter entirely, you understand *all* of it. Go ahead and check the source—everything has been explained in complete detail. The parser implemented in the next chapter may be the real star of the show here, but what you've done here is a hugely important job that shouldn't be understated. All the parser theory in the world won't mean a thing if you don't have a sturdy foundation upon which to apply it, and that's exactly what you've built in this chapter.

The exciting thing is that, by the end of the next chapter, XtremeScript will be done. All that will be left after that is to apply it to a fully operational game demo, which is icing on the cake. For now, you're encouraged to perhaps take one more quick glance over everything covered here, because it was a decent sized chapter that covered a lot of ground. And of course, even more importantly, check out the source! Even though I went to great lengths to make sure that virtually all of the source this chapter covered was actually printed in the book, there's still no substitute for seeing how it all fits together in the final program.

Now, if you think you're ready, the *real* trials await you in the next chapter…

On the CD

There isn't much in the way of demos for this chapter, but I've included the source to the finished XtremeScript compiler framework in the `Programs/Chapter 14/XtremeScript Compiler/` directory. Remember, without the parser, it's capable of very little. Because of this, the hard-coded script I talked about earlier is included in the source, so you can play around with that and make it compile small chunks of code.

As you might imagine, this is still just a console application, so you won't have much trouble getting it to compile. And, as usual, it comes with Microsoft Visual C++ project and workspace files that'll immediately organize the source files for you. Try hard-coding your own script and see what it produces!

Challenges

Being that this chapter was mostly about preparation for the parser you'll build in the next, there isn't much room for improvement or enhancement just yet. As a result, there's just one challenge for this chapter:

- *Intermediate:* Implement the missing `#include` and `#define` preprocessor directives discussed earlier.

CHAPTER 15

Parsing and Semantic Analysis

"Boy, a month in Europe with Elaine.
That guy's coming home in a body bag."
——*Kramer,* Seinfeld

This is the last of this book's three chapters on the construction of the XtremeScript compiler. You started in Chapter 13 with the development of a complete lexical analyzer module, and integrated it with a full compiler framework in Chapter 14. You now have a compiler that, with the exception of a parser, is finished.

Along the path from the source code to the final output, a number of modules are invoked in a more or less sequential manner. The loader initially reads the source code from its file and stores it in an internal format. The preprocessor then scans through the freshly loaded source and converts it to a more "correct" format. The parser, with help from the lexer, then makes sense of the source code and converts it to its intermediate code format. Lastly, the code emitter converts the I-code into the target format and the process is complete.

The problem is, you have a rather large hole in the otherwise pristine compiler pipeline. In between the lexer and the I-code module, the parser is nowhere to be found. The reason this hole exists is that I find it easier to understand how a parser works when I don't have anything else to worry about. In other words, the considerable complexity of a parser is much less of a challenge when you already have an otherwise complete compiler to test it with. Because you took the time to create everything else the compiler needs in the last chapter, you now have the luxury of passing the results of even the very first parser experiments through the complete compiler pipeline. This means that from the ground up, you'll see immediate results as the parser is incrementally constructed. I hope this gives you some perspective on the otherwise monotonous laundry list of tasks performed in the last chapter; it may have seemed like a lot of useless work, but you'll clearly reap the rewards as you make your way through this chapter. Because of this, however, it's important that you read and fully understand all of Chapter 14 before proceeding.

In this chapter, you're going to

- Learn more about what parsing is, why it's necessary, and how it's done.
- Learn specifically how recursive descent parsing works.
- Complete the XtremeScript compiler you've been developing for the last two chapters by embedding a fully functional parser module between the lexical analyzer and I-code module.

In short, this chapter provides everything you need to complete this ever-evolving scripting system by bridging the gap between high-level and low-level code once and for all.

What Is Parsing?

In almost oversimplified terms, a script's code can be said to exist in three primary forms as it passes through the compiler, and you've studied them exhaustively throughout this book's recent chapters. The code begins as a raw stream of characters presented by the loader and preprocessor modules. The lexical analyzer module then "elevates" this raw stream to a higher level of coherence by grouping related characters into lexemes, which are like the words of a sentence. Finally, as you'll see in this chapter, the parser groups the lexemes into the fundamental building blocks of the source language—statements, declarations, and so on. At this point, the source code can be fully understood. This is demonstrated in Figure 15.1.

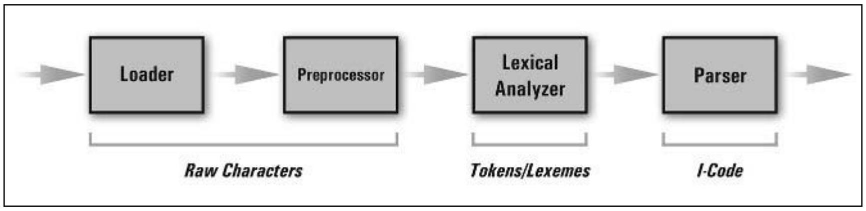


Figure 15.1

The three simplified forms of code as they pass through the compiler.

Specifically, *parsing* is the process of determining patterns in the token stream that correspond to the source language's constructs like statements and declarations. Because the compiler is designed such that the parser module reads from the lexical analyzer module and writes to the I-code module, it will be the final step toward understanding and translating the source code.

Syntactic versus Semantic Analysis

Parsing is also known as *syntactic analysis*, because its primary job is to ensure that the syntax of the source code is correct. The *syntax* of a language refers to the set of legal patterns and sequences its tokens can form to express that language's constructs. For example, the following line of code is syntactically valid:

```
X = Y * 2;
```

Although this is not:

```
* = Y X 2;
```

Note that all I've done in the second line is swap the `*` and `X` lexemes. However, assuming this language is C/C++ or some derivative thereof, the language syntax specifies that an operator (such as `*`) is not a valid L-value; in other words, it can't appear on the left side of an assignment operator. Furthermore, `Y X 2` is not a syntactically valid expression, because the `Y` and `X` operands (as well as the `X` and `2` operands) are not separated by a binary operator (or any operator at all in this case).

Syntax goes a long way towards helping you understand both what a language is saying, as well as whether it's valid. Speaking in terms of syntax alone, you can determine that the two lines of code listed previously are expressions, and, furthermore, that they're valid ones. To understand the shortcomings of syntactic analysis, however, you need to understand exactly how a parser would identify the previous expressions. Here's the valid expression, listed once again:

```
X = Y * 2;
```

In order for the parser to determine that this is an expression, it noted that the line token pattern consisted of an identifier (`X`), the binary assignment operator, a second identifier (`Y`), the binary multiplication operator, and an integer literal (`2`). Based on this information, you might be quick to assume that there's nothing more to say—it's definitely an expression, and it's definitely valid.

There's no arguing that even based on syntax alone, the previous line of code is an expression. You cannot, however, be absolutely positive that it's a *valid* expression. The reason for this is that the identifiers being referenced are more complex than they seem. In addition to being simply an identifier, for example, `X` has a number of other attributes. It can be an array, a parameter, a local variable, or even a class or function. You may have assumed that the expression was valid upon first glance, but what was the block of code in which it appeared?

```
func X ()
{
    var Y [ 16 ];
    X = Y * 2;
}
```

Not quite what you expected, is it? Now, it's clear that you're attempting to "assign" an array (`Y []`) to a function (`X ()`) after multiplying it by 2. Naturally, this doesn't make any sense. This is where *semantic analysis* comes into play.

The *semantics* of a language go beyond mere syntax to explain not what a language must *look* like, but the *context* in which it can be considered valid. To return to the example, the expression is perfectly valid as long as `X` and `Y` are single variables. When `X` is defined as a function and `Y` is an array, however, the expression's validity is lost. Notice that in both situations, the token stream is identical—the expression itself doesn't change—all that's different is the context in which the expression appears. Because of this, the parser is not usually the final step in the front end's pipeline—the *semantic analyzer* is. Of course, I did mention earlier that the parser module would be the final addition to the XtremeScript compiler, so I'll be sure to clarify what I mean by this momentarily.

If the parser is the syntactic analyzer, it ensures that the tokens form valid language constructs such as expressions, statements, and declarations. The semantic analyzer is responsible for validating the context in which these constructs appear. It can perform tasks such as ensuring that an identifier is valid in an expression, like you saw previously, as well as preventing identifier redefinition, ensuring that the value returned from an expression is valid for its destination, and so on.

Expressing Syntax

Semantics are important, but I'm going to start with the basics and focus exclusively on syntactic analysis for the moment. The first step in building a parser of any kind is formally deciding on a language's syntax. Chapter 7 was spent laying out and designing the XtremeScript language, which was an important step, but it didn't go very far to give you a strict, formal description of what is and isn't valid syntax when writing scripts.

This is done by literally laying out the token sequence behind every type of statement, declaration, and expression the language supports. The resulting rules and descriptions of this process are collectively known as a *grammar*. There are a number of official formats for expressing grammars, but I'll focus only on two of the most prominent—*syntax diagrams* and *Backus-Naur Form*.

Syntax Diagrams

A *syntax diagram*, also known as a *flow diagram*, is very similar to a standard flowchart or even a state diagram. It visually describes the sequence in which tokens will be encountered as specific types of statements are parsed in a language. Rather than blabber on endlessly about the what's, why's, and how's, however, let's just check out Figure 15.2, which depicts a syntax diagram for an XtremeScript variable declaration.

Even without an explanation, this should make some level of intuitive sense right off the bat. What this diagram is saying specifically is that a variable is declared as the `var` keyword, followed by an identifier, followed by an *optional* array size enclosed in braces. The beauty of a state diagram is its simple, straightforward nature; it spells out what it's trying to say using the source language itself. Beyond the boxes, however, the arrows provide significant insight into the flow of the diagram as well, by allowing you to follow all of the possible paths from the first token to the last.

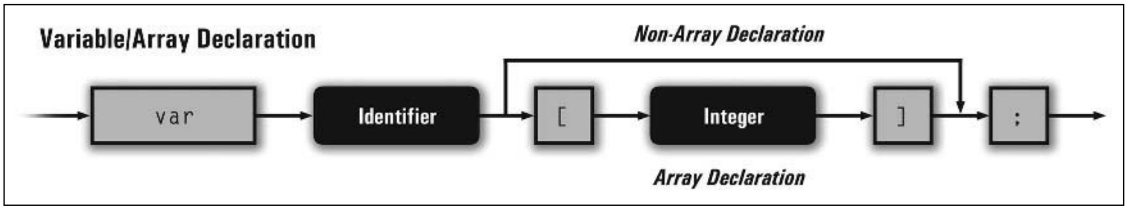


Figure 15.2

The syntax diagram for an XtremeScript variable declaration.

Notice that until the optional array notation is reached, there's only one arrow to follow from one token to the next. Once the identifier is passed, however, the path forks to allow one of two possibilities. Lastly, note the difference between the rectangular and rounded nodes. Tokens enclosed in a rectangle refer to literal strings that must appear as-is, exactly, such as `var` (a reserved word), and the `[]` braces (delimiters). Rounded token boxes refer to user-defined lexemes such as identifiers and integer literals.

Backus-Naur Form

Backus-Naur Form, or BNF, is a more text-oriented way to specify the grammar of a language. As I did with syntax diagrams, I'll start things off with an example and save the discussion for afterwards. Here's the same variable declaration from the previous syntax diagram, expressed in BNF:

```
VarDecl ::= 'var' Ident | 'var' Ident '[' Int ']'
```

Compared to its equivalent syntax diagram, this may require a bit more explanation. As you can see, the BNF version almost looks like code of its own—in fact, BNF is indeed its own language. Specifically, it belongs to a class of languages called *metalanguages*—languages used to define other languages.

To understand BNF, it's crucial to understand its two most fundamental elements—*terminals* and *non-terminals*. A terminal (short for *terminal symbol*) is an element of the language that is irreducible—because of this, terminals usually correspond directly to tokens. A non-terminal (short for *non-terminal symbol*), on the other hand, is an element of the language that is composed of some sequence or pattern of terminals. In the previous example, `VarDecl` is a non-terminal defined by the pattern of terminals on the right side of the `::=` operator. Of course, non-terminals don't have to be defined simply by terminals; in fact, the most common definitions in a BNF grammar will involve defining non-terminals by other non-terminals, or even recursively with alternative forms of themselves.

To explain the example in more detail, `VarDecl` is a non-terminal because it can be reduced to the constituent parts listed on the right side of the `::=` operator. In this example, it just so happens that each of these constituent parts is a terminal, and is thus irreducible, although this is not often the case. For example, **`'var'`** corresponds to the literal string `var`, as in the reserved word `var` token. `Ident`, on the other hand, refers to any valid identifier, and is therefore not a literal string but a user-defined lexeme. The difference between these two types of terminals is analogous to the rectangular and rounded nodes in Figure 15.2.

Also important is the use of logical operators to denote alternative forms of the same non-terminal. Because, as you know, a `var` declaration can be either a single variable or an array, you use the logical or | operator to denote two separate possibilities. This grammar states that `VarDecl` can be defined as either of those two sequences (although they're entirely mutually exclusive—it's strictly one or the other).

To wrap this section up, let's look quickly at what the grammar might look like if you threw some extra non-terminals in. Although these additions aren't necessary in this specific example, they help demonstrate the flexibility of BNF more clearly. In this example, you'll take the '`'['` `Int` `']'`' a non-terminal of its own, so it can be nested in the `VarDecl` definition:

```
ArraySize ::= '[' Int ']'
VarDecl ::= 'var' Ident | 'var' Ident ArraySize
```

`ArraySize` is defined as an integer literal value enclosed in braces, just as is required by array declaration notation, and `VarDecl` has been redefined more concisely as either the `var` keyword followed by an identifier, or the `var` keyword followed by an identifier and the `ArraySize` non-terminal.

Although BNF is indeed a structured and readable method of defining a grammar, its real attraction is the fact that parser-generating programs usually use text files containing BNF grammar definitions as their input. In other words, by deriving your language's BNF grammar, you can use a parser-generation program like yacc or Bison to actually create a fully functional parser for that language in minutes.

Choosing a Method of Grammar Expression

For the purpose of XtremeScript, to keep consistent with the continuing trend of simplicity and straightforward solutions, I've decided to go with syntax diagrams as the method of expressing the language's formal syntax. This allows me to keep the discussion visual, lets you clearly see the physical flow of the syntax, and just makes things cleaner and easier to follow. Furthermore, because you'll be writing the parser by hand, rather than automatically generating it, there's no practical reason to favor BNF over its alternatives.

Parse Trees

One thing that will become more and more clear as you formally define the syntax of the language is that its definition is strongly hierarchical. Non-terminals are based on terminals and non-terminals, many of which are based on terminals and non-terminals of their own. Because of this, a tree is implicitly formed by the syntax of the language.

To understand this better, let's temporarily add a simple keyword to the language for defining integer constants. It will be called `const`, and its syntax diagram is depicted in Figure 15.3. An example of its syntax looks like this:

```
const MyAge = 20;
```

Although the syntax diagram illustrates the flow of a `const` declaration in a linear fashion, it can also be converted to a simple tree structure, as demonstrated in Figure 15.4.

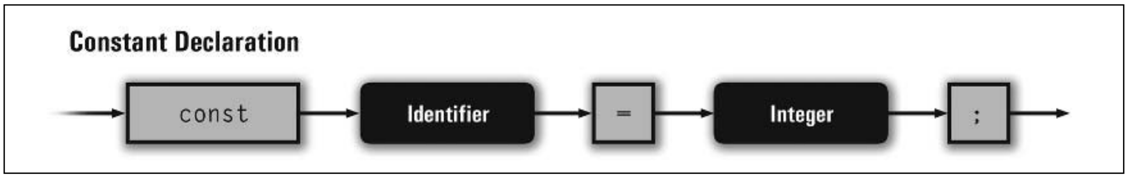


Figure 15.3

The syntax diagram for a hypothetical integer constant-defining keyword.

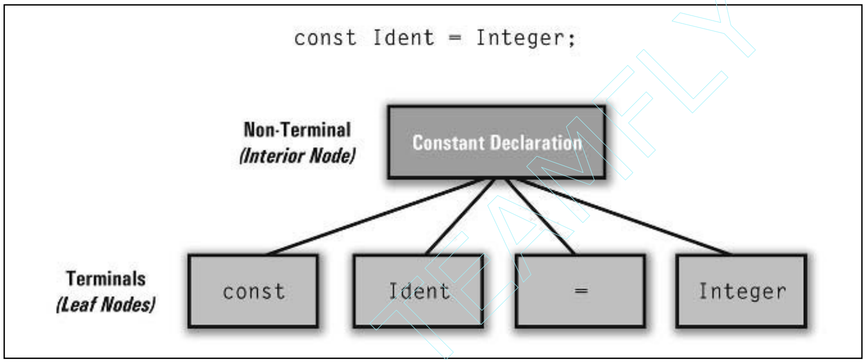


Figure 15.4

The initial conversion of the `const` *syntax diagram to a tree.*

The general constant declaration is the somewhat abstract root of the tree, whereas its child nodes are each of the terminal symbols—`const`, an identifier, =, and an integer literal. This particular tree is a bit messy, however; it's bogged down by useless nodes that only serve to get in the way. You can prune the tree a bit to remove the implicit and therefore needless `const` and = nodes, leaving only those nodes that contain real information. A new, more concise syntax tree for the `const` definition is found in Figure 15.5.

This new tree describes only what you need, and does its job well. By virtue of the Constant Declaration node alone, you know it must contain identifier and integer literal nodes, so the

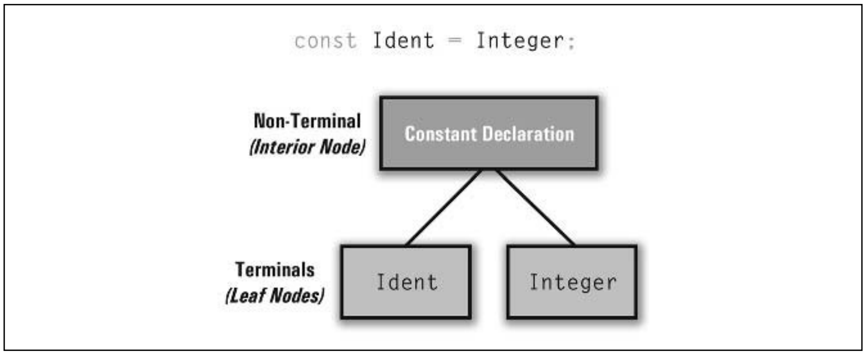


Figure 15.5

A new, more concise syntax tree for `const`.

`const` and = nodes are therefore implied. To make this example a bit more interesting, however, let's expand the `const` keyword to define entire arrays of integer constants. The syntax diagram for this new version appears in Figure 15.6. Here's an example of its usage:

```
const MyArray [ 4 ] = { 0, 16, -4, 8192 };
```

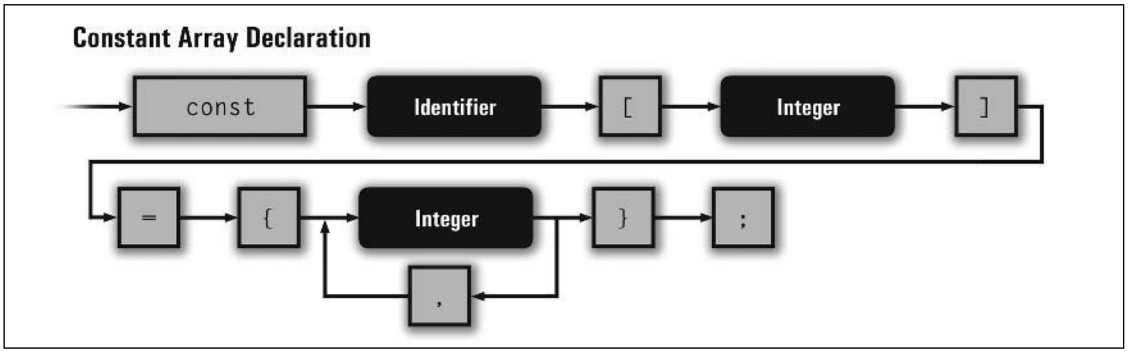


Figure 15.6

The syntax diagram for the array-based version of `const`*.*

For simplicity's sake, notice that the new version of `const` isn't optional in its support for array constants, so you don't have to worry about including the previous definition as an alternative path. This new version is a much clearer example of the tree-like structure of a language's grammar; check out what `const`'s syntax tree looks like now (note that I've already pruned the useless nodes this time).

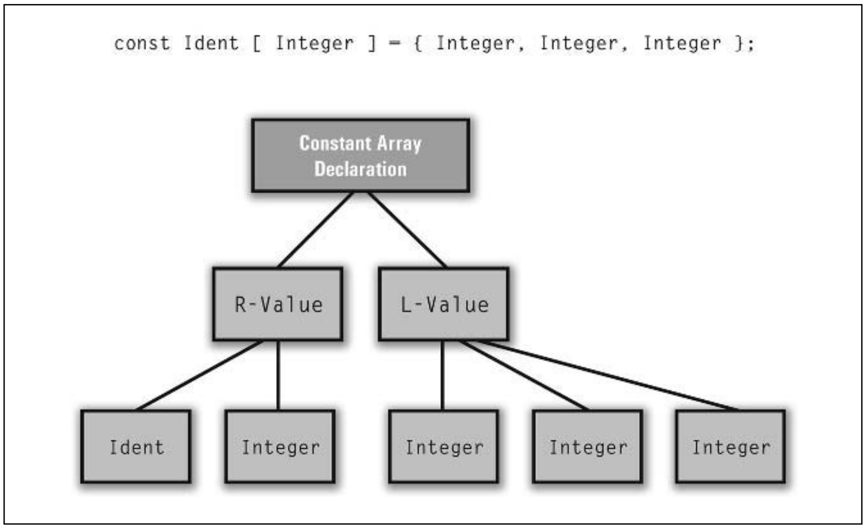

const Ident [Integer] = { Integer, Integer, Integer };
Constant Array Declaration
R-Value
L-Value
Ident
Integer
Integer
Integer
Integer

Figure 15.7

The pruned syntax tree of the new `const` *keyword.*

Note that now, due to their respective added complexity, I've abstracted the identifier and array size to an "L-Value," and the array of values to an "R-Value". Within the syntax tree, the `Ident` node under L-Value corresponds to the constant's identifier, and the `Int` corresponds to its size. Under the R-Value, I simply filled in three values; there could actually be any number of leaf nodes here, each corresponding to one of the constant's values. Although the trees you've seen so far have specifically related to the static syntax of this non-terminal symbol, a *parse tree* is the representation of the actual source code. For example, consider the following instance of the `const` keyword:

```
const MyArray [ 4 ] = { 0, 16, -4, 8192 };
```

Just as the syntax tree is a hierarchical view of an otherwise linear syntax diagram, the parse tree is the hierarchical version of a linear statement in the source code. This declaration might be represented in the form of the parse tree shown in Figure 15.8.

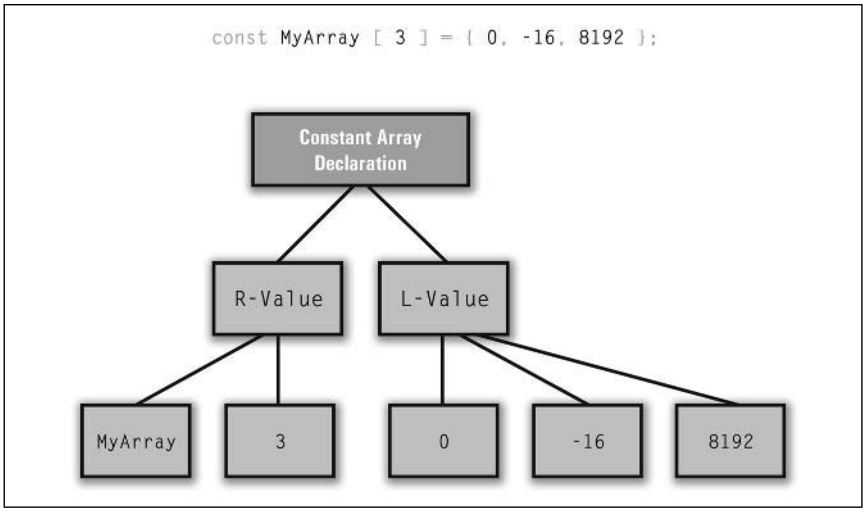

const MyArray [3] = { 0, -16, 8192 };
Constant Array Declaration
R-Value
L-Value
MyArray
3
0
-16
8192

Figure 15.8

The parse tree for an instance of the `const` *keyword.*

In a traditional compiler, the parser is responsible for generating a parse tree similar to this one by scanning through the token stream and picking up the patterns defined by the grammar's non-terminal symbols. The result is a highly structured, easily-traversable tree that represents the program in a purely hierarchical manner. At the root node is the program itself, which branches off into its highest level statements—probably declarations of functions, globals, and constants. A global or constant definition will most likely be a leaf node, whereas function declarations will branch off into a number of child nodes, each of which will contain statements and declarations in the least-nested scope. An example of a simplified but complete parse tree appears in Figure 15.9.

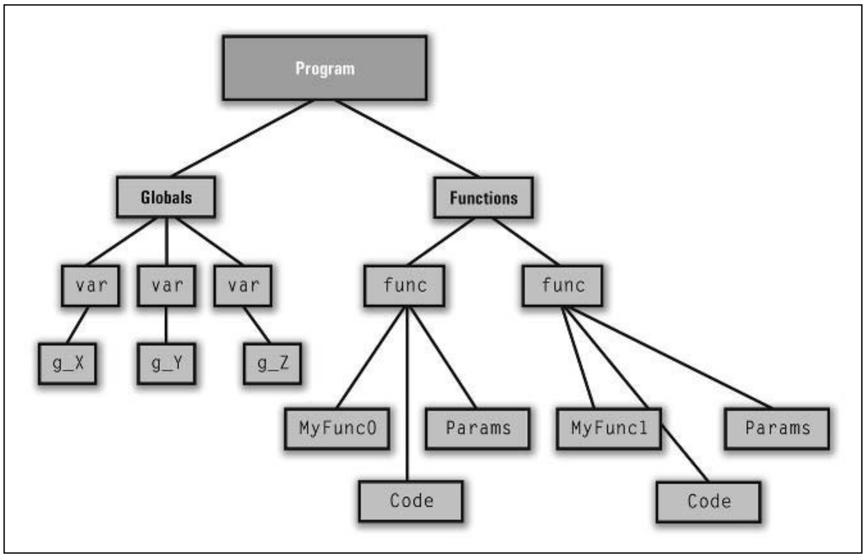


Figure 15.9

A simplified but complete parse tree.

Once a program or script has been converted into a parse tree, it can be easily scanned and analyzed by other modules, such as the semantic analyzer. Semantics are easy to verify in a parse tree, because the analyzer can rest assured that the tree is free of syntax errors, and can focus entirely on traversing the nodes and ensuring that they can legally appear in their specific context.

The XtremeScript compiler will not create a parse tree, however. Although it certainly has its uses, the language is just simple enough to be translated to XVM assembly without the use of such a tree. Rather, the parser will directly generate I-code based on the token stream it reads and perform on-the-fly semantic analysis that combines the roles of both a syntactic and semantic analyzer into a single module. In the case of a simple script compiler, I personally find this approach to be adequately structured and readable, while maintaining simplicity on all levels.

How Parsing Works

Despite the sheer volume of compiler-related algorithms, data structures, and formalisms discussed so far, the parsing of a high-level language like C or XtremeScript may still seem like a mysterious and insurmountable task. After all, it's one thing to parse the simple and predictable format of an assembly language like XVM assembly, but how can you apply these principals to something as complicated as this?

```
func FuncX ( U, V )
{
    var Z [ 8 ];
    if ( U > V )
    {
        Z [ 0 ] = FuncX ( U / 2, V / 2 );
        Z [ 1 ] = 0;
    }
    else
    {
        Z [ 0 ] = 0;
        Z [ 1 ] = FuncY ( U * V );
    }
    return U * ( V << 8 ) + ( Z [ 0 ] + 3.14159 * FuncY ( Z [ 1 ] / V ) );
}
```

It looks like a formidable challenge, and indeed it is, but the key is to approach it in an incremental manner that slowly builds the parser up by adding support for more and more of the language. This task will be made easier by the fact that you'll be designing the parser with the *recursive descent* algorithm, which is an intuitive and straightforward parsing method.

Recursive Descent Parsing

A recursive descent parser is so named because it constructs the parse tree by *recursively descending* from the root node to the leaf nodes. However, because you won't be explicitly building a parse tree in the compiler, you can forget about that part of the definition and instead focus exclusively on the apparently recursive nature of this algorithm.

Non-Terminal Symbol Parsing Functions

The key to the recursive descent parser is assigning separate functions to parse each of its non-terminals. For example, the parsing of the `while` statement generally involves three things—parsing the `while` keyword, parsing the expression that determines under what conditions the loop will execute, and finally, parsing the block of code within the loop. You can wrap all of this into a single function called `ParseWhile ()`.

One thing you'll quickly realize, however, is that only the `while` keyword can be parsed easily. The parsing of expressions and blocks of statements is rather complex. Furthermore, expressions and statement blocks are hardly specific to the `while` loop—in fact, most other language constructs involve them in some way. For example, the `if` structure uses an expression to determine whether to execute its true block, and the true block is a code block onto itself. Furthermore, code blocks can appear anywhere—with or without a loop or other block construct. For example, the following code is perfectly legal:

```
func X ()
{
    {
        DoStuff ();
    }
    return DoEvenMoreStuff ();
}
```

So, it's clear that `while` loops aren't the only places you'll need the ability to parse expressions and code blocks. And because it's obvious that both of these operations will be complex, it's a good idea to abstract them into their own functions anyway. So, you'll add the `ParseExpr ()` and `ParseBlock ()` functions to perform these tasks. Now, `ParseWhile ()` will simply call these two functions when it reaches their respective segments of the source code, and the `while` loop will be parsed. Right off the bat, you can already see that nested function calls to specialized parsing functions will play a big role in the recursive descent algorithm. What happens, however, if another `while` loop appears in the first `while` loop's block? Naturally, something like this is legal:

```
while ( X > 0 )
{
    while ( Y > 0 )
    {
    }
}
```

How is `ParseStatement ()` going to handle it, though? Simple—by calling `ParseWhile ()`. However, because `ParseStatement ()` was originally called the first instance of `ParseWhile ()`, the second call is now recursive. Because these parse functions can call themselves (or in this case, indirectly call themselves), the language can support any arbitrary level of nesting. As you can probably imagine, this recursive approach will lend itself well to expression parsing, which quickly becomes a convoluted affair when operator precedence and nested parentheses join the fray. To wrap this all up, check out Figure 15.10, which depicts the path of execution in a recursive descent parser as it parses the nested `while` loops listed previously.

Of course, I still haven't discussed exactly what these parsing functions will do internally, or what sort of output they'll produce and where they'll put it. For now, you're just getting a feel for the general process, which will help you understand the details and code presented later in this chapter more easily. The coverage of the XtremeScript parser will be slow-paced and incremental, however—I'll intentionally start the discussion with the easiest parts of the module, which you'll have no trouble understanding. From there, I'll move on to elementary expressions, after which you'll have the prerequisite knowledge to understand virtually everything else.

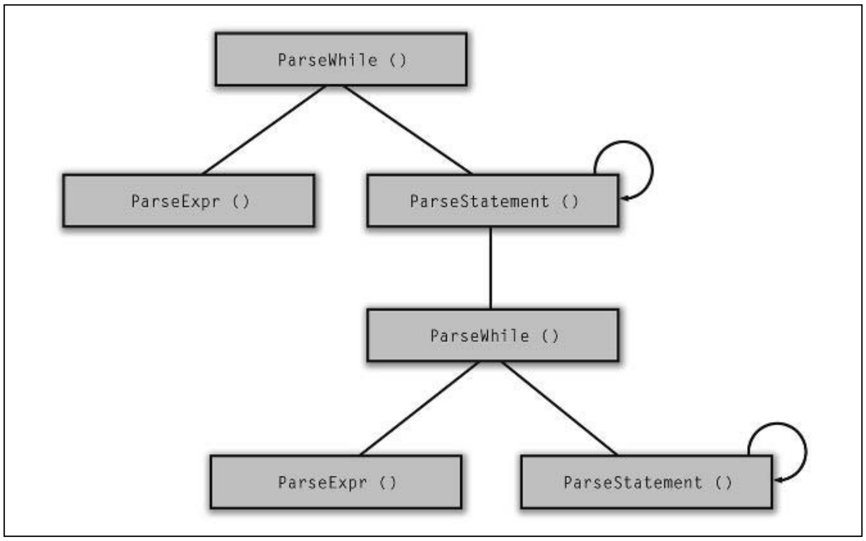


Figure 15.10

The path of execution for a recursive descent parser as it parses nested `while` loops.

The XtremeScript Parser Module

The XtremeScript parser module will be implemented in `parser.cpp|h`, and will consist primarily of functions for parsing specific non-terminals of the XtremeScript grammar as expressed by the syntax diagrams. By putting all of these together, you'll have a complete parser that understands the entire language and can translate token and lexeme streams into I-code that the code emitter can convert to XVM assembly. Sound good? Then let's get started.

The Basics

Before you can get into the parser's code, you need to get a few miscellaneous details in place.

Tracking Scope

Just like XASM, the XtremeScript parser will need the capability to track the script's scope. For example, when a function declaration is being parsed, it's important to know whether the scope is currently global, because that's the only time a function declaration is valid. Within a function, it's important to know which index into the function table it's associated with so that its local symbol declarations can be properly bound to it. For this, you'll declare a global called `g_iCurrScope`:

```
int g_iCurrScope;                // The current scope
```

As you might have already guessed, this variable will work just like the `iScope` field of the `SymbolNode` structure discussed in the last chapter—a value of zero means the scope is currently global, whereas any positive, nonzero value is interpreted as an index into the function table corresponding to the current function. Check out Figure 15.11.

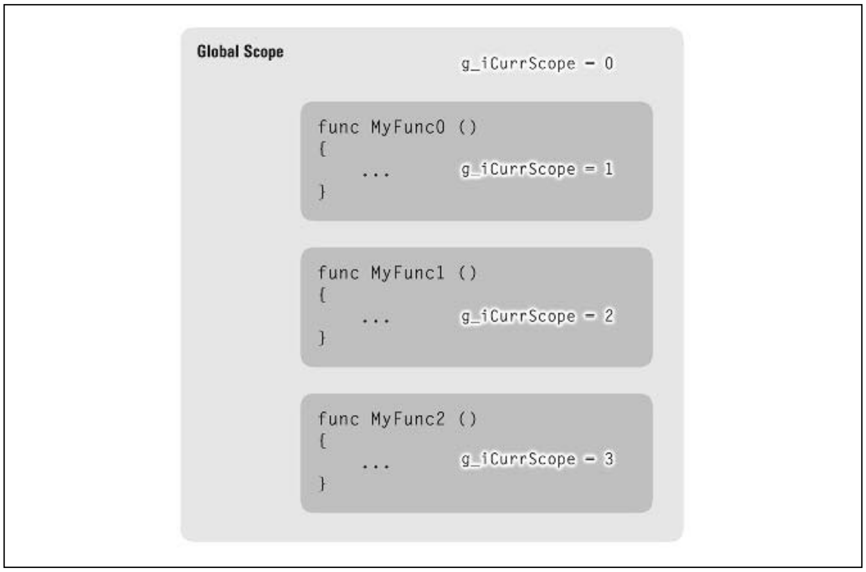


Figure 15.11

The values of `g_iCurrPath` as the source code is parsed.

It's also important to notice that tracking the scope of the script is the current brush with semantic analysis; the scope in which a token is encountered is one important aspect of the token's context.

Reading Specific Tokens

`GetNextToken ()` is designed to be an easy and fast way to read the next token, but there will be many times when it's not quite enough. In addition to reading tokens, you'd like to read *specific* tokens. For example, when parsing a function declaration, the token that comes after the `func` token *must* be an identifier. Anything else is invalid, and should cause an appropriate error to be displayed. Rather than constantly calling `GetNextToken ()`, comparing it to the desired token, and displaying an error, it would be nice to be able to call another function that does all of this for you. The `ReadToken ()` function solves this problem.

`ReadToken ()` is really just a wrapper for `GetNextToken ()`. However, unlike `GetNextToken ()`, it accepts a `Token` parameter specifying which token should appear next. It then reads the token, and compares the two. If they don't match, it means the token was erroneous and automatically displays an appropriate error message. Let's check it out:

```
void ReadToken ( Token ReqToken )
{
    // Determine if the next token is the required one
    if ( GetNextToken () != ReqToken )
    {
        // If not, exit on a specific error
        char pstrErrorMssg [ 256 ];
        switch ( ReqToken )
        {
            case TOKEN_TYPE_INT:
                strcpy ( pstrErrorMssg, "Integer" );
                break;
            case TOKEN_TYPE_FLOAT:
                strcpy ( pstrErrorMssg, "Float" );
                break;
            case TOKEN_TYPE_IDENT:
                strcpy ( pstrErrorMssg, "Identifier" );
                break;
            case TOKEN_TYPE_RSRVD_VAR:
                strcpy ( pstrErrorMssg, "var" );
                break;
            case TOKEN_TYPE_RSRVD_TRUE:
                strcpy ( pstrErrorMssg, "true" );
                break;
            case TOKEN_TYPE_RSRVD_FALSE:
                strcpy ( pstrErrorMssg, "false" );
                break;
            case TOKEN_TYPE_RSRVD_IF:
                strcpy ( pstrErrorMssg, "if" );
                break;
            case TOKEN_TYPE_RSRVD_ELSE:
                strcpy ( pstrErrorMssg, "else" );
                break;
            case TOKEN_TYPE_RSRVD_BREAK:
                strcpy ( pstrErrorMssg, "break" );
                break;
            case TOKEN_TYPE_RSRVD_CONTINUE:
                strcpy ( pstrErrorMssg, "continue" );
                break;
            case TOKEN_TYPE_RSRVD_FOR:
                strcpy ( pstrErrorMssg, "for" );
                break;
            case TOKEN_TYPE_RSRVD_WHILE:
                strcpy ( pstrErrorMssg, "while" );
                break;
            case TOKEN_TYPE_RSRVD_FUNC:
                strcpy ( pstrErrorMssg, "func" );
                break;
            case TOKEN_TYPE_RSRVD_RETURN:
                strcpy ( pstrErrorMssg, "return" );
                break;
            case TOKEN_TYPE_OP:
                strcpy ( pstrErrorMssg, "Operator" );
                break;
            case TOKEN_TYPE_DELIM_COMMA:
                strcpy ( pstrErrorMssg, "," );
                break;
            case TOKEN_TYPE_DELIM_OPEN_PAREN:
                strcpy ( pstrErrorMssg, "(" );
                break;
            case TOKEN_TYPE_DELIM_CLOSE_PAREN:
                strcpy ( pstrErrorMssg, ")" );
                break;
            case TOKEN_TYPE_DELIM_OPEN_BRACE:
                strcpy ( pstrErrorMssg, "[" );
                break;
            case TOKEN_TYPE_DELIM_CLOSE_BRACE:
                strcpy ( pstrErrorMssg, "]" );
                break;
            case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
                strcpy ( pstrErrorMssg, "{" );
                break;
            case TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE:
                strcpy ( pstrErrorMssg, "}" );
                break;
            case TOKEN_TYPE_DELIM_SEMICOLON:
                strcpy ( pstrErrorMssg, ";" );
                break;
            case TOKEN_TYPE_STRING:
                strcpy ( pstrErrorMssg, "String" );
                break;
        }
```

```
            // Finish the message
            strcat ( pstrErrorMssg, " expected" );
            // Display the error
            ExitOnCodeError ( pstrErrorMssg );
        }
    }
```

The function is long, but simple. As I mentioned, it reads the token, compares it to the token specified by `ReqToken`, and formulates the proper error message if they don't match. For each token type, it creates a string that mentions the token by name, and appends `" expected"` on the end. It then passes it to `ExitOnCodeError ()`. As can be seen in Table 15.1, this is an easy and quick way to read tokens and ensure that a verbose error message will be presented automatically if they are not found.

As you'll soon see, `ReadToken ()` will be an invaluable and frequently used addition to the parser.

Table 15.1 Sample `ReadToken ()` Error Messages

`ReqToken` Value	Error Message
`TOKEN_TYPE_RSRVD_VAR`	`"var expected"`
`TOKEN_TYPE_STRING`	`"String expected"`
`TOKEN_TYPE_INT`	`"Integer expected"`
`TOKEN_TYPE_DELIM_COMMA`	`", expected"`

The Parsing Strategy

The strategy from here on out is twofold: first, you need to formally map out the exact grammar of XtremeScript with syntax diagrams. Then, armed with this specification to work from, you'll code a number of parsing functions that can parse each of the grammar's non-terminal symbols. Most of the calls between these functions will be nested, and many will be fully recursive. Because of this, the syntax flow and layout of these functions may be a bit hard to follow at first. Just go slowly, maintain your focus, and it will all make sense eventually.

As far as actually coding the parser module, you're going to do it in a number of incremental steps. You'll soon find that XtremeScript declarations are the easiest part of the parser, so that's where you'll start. Using the compiler framework from the last chapter, you'll build progressively more sophisticated parser modules that can handle more and more of the language. After mastering declarations, you'll start with simple expressions, and then move on to the entire expression vocabulary of the language, and finally wrap it all up with general statements like loops, branching, and assignments. The result will be a finished parser module that completes the XtremeScript compiler, and with it, the entire XtremeScript system. Each separate parser module will be available on the accompanying CD as well (along with its own copy of the compiler framework, so they'll run right away).

Each of these parsing functions will be responsible for three major tasks (although this will vary slightly from function to function). They'll each start by parsing the incoming token and lexeme streams to determine which language construct is forming. Once this is identified, they'll perform on-the-fly, somewhat ad hoc semantic analysis by ensuring that the context in which the construct appears is valid. Lastly, they'll convert the construct directly to its I-code equivalent. By performing each of these three steps, the parser will single-handedly bridge the gap between the lexical analyzer module and the I-code module. Check out Figure 15.12.

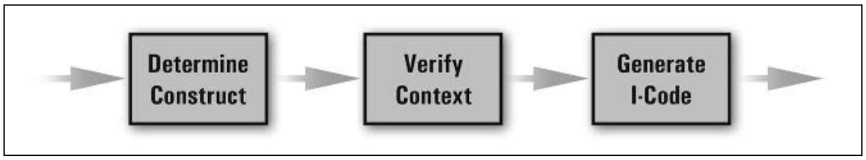


Figure 15.12

The three major aspects of a parse function's logic.

You'll hopefully see, as this chapter progresses, why it was so important to build a sturdy and complete compiler framework before diving into the parser. Developing even a recursive descent parser is hard work that can seem extremely complex at first to a beginner. Having to deal with both the parser's intrinsic complexity, as well as endless details of a complete compiler at the same time is a recipe for disaster. By getting all of the bookkeeping and grunt work out of the way ahead of time, however, you can now devote 100 percent of your brainpower to solving this final problem. Well, not exactly 100 percent—chances are if you could do that you'd be destroying major landmarks and shooting lasers out of your eyes. But it will be close enough.

PARSING STATEMENTS AND CODE BLOCKS

Although declarations will be the first major parsing task, you'll actually start with something a bit subtler to get the juices flowing—basic statements and code blocks. What's great about these two elements of the language is that they're almost nonexistent; they bring with them almost no real substance, and are thus easy ways to get the first fragments of the parser in place.

For now, you can specifically limit yourself to empty statements; obviously, any non-empty statement will bring with it considerable extra complexity. For now, you just want a "boiler plate" upon which the rest of the language's statement types can be implemented.

Syntax Diagrams

The first thing to do is devise an initial syntax diagram that lays out the exact syntax for empty statements and code blocks. Figure 15.13 contains this diagram.

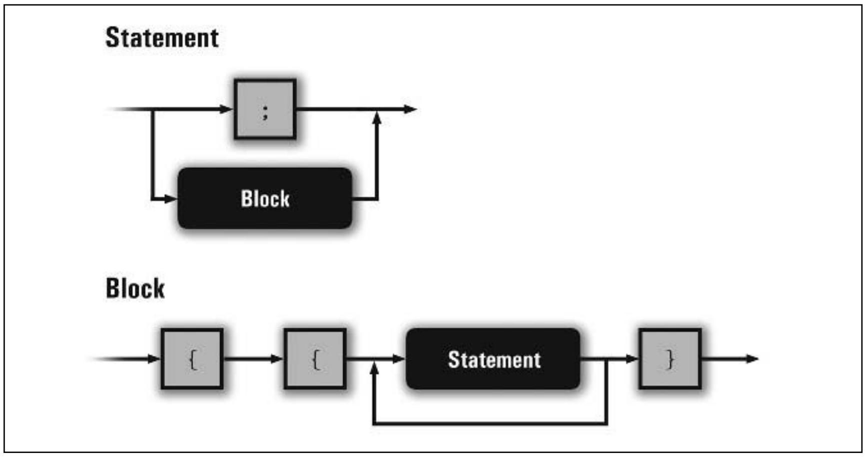


Figure 15.13

The syntax diagrams for empty statements and code blocks.

This is an understandably simple diagram, but it's still very effective in its description. A *Statement* is currently defined as an empty statement, which consists solely as a semicolon. Note that I didn't explicitly define a non-terminal called an *Empty Statement* anywhere; rather empty statements are part of an overall Statement non-terminal. In the future, you'll add more statement types to Statement.

The next noteworthy point is the arrows. Notice that even in the case of the single-terminal statement diagram, an arrow comes in from the left and exits to the right. This represents the fact that the diagram can "fit in" to any preexisting flow of syntax; without the arrow on the right side, you'd be stating that a Statement can only occur at the very beginning of a program; without the arrow on the left, you'd be stating that it can only occur at the very end.

Lastly, and most importantly, notice that this is your first encounter with recursion in this language's syntax. What this diagram says is that a Statement can consist of either a single semicolon or a Block, and that a Block can consist of one or more Statements enclosed in curly braces. Because both of these non-terminals include each other, a parser that implements them will support infinite levels of repetition and recursion. For example, these diagrams alone support the following blocks of code:

```
;

{
}

;
{
    ;
}

{;} ;;; {}{}{} ;;; {;}

{
    ;
    {
        {
            ;
        }
        ;
    }
}
```

In contrast, the following code blocks are illegal:

```
{
   ;

{
    {
         {{
    }
}
```

It can be tricky to grasp at first, but the basic summary is this—Blocks are *types* of Statements, but they can also *contain* Statements. This recursive relationship gives the compiler the flexibility to parse arbitrary levels of nesting. Let's take a look at some code.

The Implementation

Remember, the next step after creating syntax diagrams is committing them to code by implementing a parsing function for each non-terminal. The grammar so far has two non-terminals—Statement and Code Block. These will map directly to the two parsing functions you'll write in this section, `ParseStatement ()` and `ParseBlock ()`.

It's also important to remember that these particular functions won't produce I-code of any sort. As you can imagine, code blocks and statements that don't actually do anything have no I-code equivalent; much like comments and whitespace, semicolons and curly braces exist primarily for delimiting purposes and can thusly be discarded as they're parsed.

ParseSourceCode ()

If you remember from the last chapter, there was a function defined in `xsc.cpp` called `CompileSourceFile ()` that was responsible for initiating the compilation process. Its sole purpose at the time was to call `ParseSourceCode ()`, which generated the script's I-code equivalent by parsing its token stream. Before you can go any farther, you have to implement this function, because it's ultimately what will manage the parsing process.

On the subject of terminology, it's important to note before you go any further that a *statement* is defined in the XtremeScript language as virtually every language construct it supports. For example, a function or variable declaration is a type of statement (regardless of scope), an assignment is a statement, and `for` and `while` loops are statements as well. Because of this, the *real* job of the parser is to simply loop through each statement in the script and parse it depending on its type. For this reason, `ParseSourceCode ()` can manage the entire parsing process simply by repeatedly calling `ParseStatement ()` until the end of the token stream is reached. With that in mind, here's the code:

```
void ParseSourceCode ()
{
    // Reset the lexer
    ResetLexer ();

    // Set the current scope to global
    g_iCurrScope = SCOPE_GLOBAL;

    // Parse each line of code
    while ( TRUE )
    {
        // Parse the next statement and ignore an end of file marker
        ParseStatement ();
```

```
            // If we're at the end of the token stream, break the parsing loop
            if ( GetNextToken () == TOKEN_TYPE_END_OF_STREAM )
                break;
            else
                RewindTokenStream ();
        }
    }
```

That wasn't so bad, huh? The function starts with a call to `ResetLexer ()` to prep the lexical analyzer module before everything begins. It then sets the current scope to `SCOPE_GLOBAL`, which makes sense because a script never starts out inside a function. It then enters a loop that parses statements with a call to `ParseStatement ()` until the next token read is `TOKEN_TYPE_END_OF_STREAM`. At this point, the function knows that the end of the source file is reached, and exits.

Statements

`ParseStatement ()` is the only function called by `ParseSourceCode ()`, because when you really get down to it, every line of code in the script is technically a statement of some sort. Because of this, it's in charge of all subsequent branches to other parse functions; `ParseSourceCode ()` may be the overall parsing process manager, but `ParseStatement ()` is really the one calling the shots.

Currently, all the statement parser does is consume semicolons and call `ParseBlock ()` when an opening curly brace is encountered. Anything else is considered invalid input and flags the appropriate error. It also checks for the `TOKEN_TYPE_END_OF_STREAM` token, which would flag an unexpected end-of-file. As you'll see, this logic alone is enough to fully implement the first two syntax diagrams, simple as they may be. Here's the code:

```
void ParseStatement ()
{
    // If the next token is a semicolon, the statement is empty so return
    if ( GetLookAheadChar () == ';' )
    {
        ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
        return;
    }

    // Determine the initial token of the statement
    Token InitToken = GetNextToken ();

    // Branch to a parse function based on the token
    switch ( InitToken )
    {
        // Unexpected end of file
        case TOKEN_TYPE_END_OF_STREAM:
            ExitOnCodeError ( "Unexpected end of file" );
            break;

        // Block
        case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
            ParseBlock ();
            break;

        // Anything else is invalid
        default:
            ExitOnCodeError ( "Unexpected input" );
            break;
    }
}
```

The logic here is simple. The first thing the function does is use the look-ahead to determine whether a semicolon appears to be the next token. If so, it makes sure with a call to `ReadToken ()` and returns immediately. This is how empty statements are supported. If the look-ahead isn't a semicolon, you know you aren't dealing with an empty statement and read the statement's first token. This token is used as the criteria for determining which type of statement is up next, but because you currently just parse blocks, the only token you worry about is `TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE`. Anything other than the curly brace is considered invalid and flags an `"Unexpected input"` error, with the exception of the end-of-stream flag, which flags an `"Unexpected end of file"` error.

Note that even though you check for the end-of-stream flag in `ParseSourceCode ()`, it's important to check for it in `ParseStatement ()` as well. If an end-of-stream occurs during `ParseSourceCode ()`'s main loop, it represents a valid ending to the file because, for reasons you'll see more clearly later on, `ParseSourceCode ()` only handles statements in the global scope. Other functions in the parser will make calls to `ParseStatement ()`, however, and when they do, it's important that you be on the lookout for the end-of-stream flag. Once you're actually within a specific statement parsing function, however, you can rest assured that all instances of `TOKEN_TYPE_END_OF_STREAM` will simply register as an invalid token, and cause an error as well. Because of this, you can be sure that at no point in the parser's lifespan will an end-of-stream go unhandled.

Blocks

Blocks are handled by the `ParseBlock ()` function, which performs the simple task of parsing every statement within a pair of curly braces. The great thing about this function is that even when the parser reaches its final, most sophisticated state, this will still be a profoundly simple function that's little more than a single loop. Let's look at the code first, and discuss it afterwards:

```
void ParseBlock ()
{
    // Read each statement until the end of the block
    while ( GetLookAheadChar () != '}' )
        ParseStatement ();

    // Read the closing curly brace
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE );
}
```

You might be wondering why this function doesn't start by reading the opening curly brace. After all, that's the first token of a block's syntax, right? Rememeber, however, that `ParseStatement ()` has already consumed this token. The opening curly brace is how it knew to call `ParseBlock ()` in the first place, so it's already been read from the stream. This will be a continuing trend with all parsing functions—by the time the function is active, the first token of the syntax diagram it implements has been read. All this function does is repeatedly call `ParseStatement ()` until the look-ahead character reveals that a `}` token might be on the horizon. At this point, it terminates the loop and validates the presence of the token with a call to `ReadToken ()`. That's it!

Going back to the previously mentioned issue of unexpected end-of-file encounters, this is a perfect example of why `ParseStatement ()` needs to keep watch for the `END_OF_STREAM` token. In the event that an EOF occurs during one of the `ParseStatement ()` calls made by `ParseBlock ()`, it means that the file ended before the block was closed with a closing curly brace. Because this is syntactically invalid, you need to make sure to alert the users.

Lastly, stop for a moment and think about the implications of this function calling `ParseStatement ()`, which is the very function that called *it*. Remember, the recursive nature of this relationship allows statements and blocks to be infinitely nested in any order.

At this point, you have enough code to properly parse and understand the basic structure of the language—semicolon-terminated statements and code blocks. Even if the statements were empty, it's still a start, and will provide a solid footing for the remainder of the chapter. Of course, the parser still doesn't produce anything—it consumes and even understands its input, but simply isn't capable of interpreting anything complex enough to warrant the generation of I-code or table entries. Fortunately, the next section will rectify this.

Parsing Declarations

Taking a small step up from the decidedly dull world of empty statements and code blocks brings you to the language's declarations. XtremeScript currently supports two fundamental types of declarations (although you'll be adding a third before this section is over), variables and arrays (data declarations), and functions (logic/code declarations). Variables are declared with the `var` keyword, and can be optionally followed by an integer array size enclosed in curly braces. Functions are declared with the `func` keyword, and consist of an identifier, a parentheses-enclosed parameter list containing zero or more parameters, and a code block. Thanks to the last section, you're now capable of parsing code blocks, which means you're already part of the way there.

Function Declarations

You can start with function declarations first, because the parser currently has no notion of scope—something you need to fix immediately. Even variables, which I'll cover in this section, need some form of scope in order to be properly added to the symbol table, and without an understanding of function declarations, you can't do that.

The syntax for XtremeScript functions is presented in Figure 15.14.

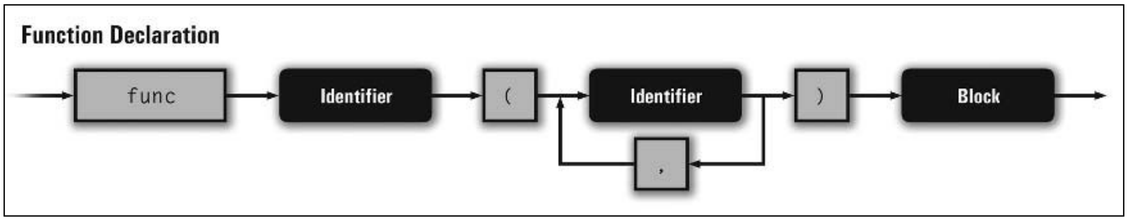


Figure 15.14

The syntax diagram for function declarations.

What this diagram is saying is that a function declaration starts with the `func` keyword, is followed by an identifier, and ends with a parameter list and a code block. The parameter list is enclosed by parentheses, within which zero or more parameters are housed, each of which is followed by a comma, except for the last one.

Before a function can be parsed, however, `ParseStatement ()` needs to be updated to acknowledge its existence. Remember, the current statement parser only understands empty statements, blocks, and the end-of-stream flag. Anything else is considered invalid and causes an error, which currently includes the `func` keyword. So, the first step in handling functions is adding an extra `case` to the `switch` block used to branch to the proper parsing function:

```
void ParseStatement ()
{
    // If the next token is a semicolon, the statement
    // is empty so return
    if ( GetLookAheadChar () == ';' )
    {
        ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
        return;
    }

    // Determine the initial token of the statement
    Token InitToken = GetNextToken ();

    // Branch to a parse function based on the token
    switch ( InitToken )
    {
        // Unexpected end of file
        case TOKEN_TYPE_END_OF_STREAM:
            ExitOnCodeError ( "Unexpected end of file" );
            break;

        // Block
        case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
            ParseBlock ();
            break;

        // Function definition
        case TOKEN_TYPE_RSRVD_FUNC:
            ParseFunc ();
            break;

        // Anything else is invalid
        default:
            ExitOnCodeError ( "Unexpected input" );
            break;
    }
}
```

This alters the syntax diagram for the Statement non-terminal, so let's have a look at the updated version, shown in Figure 15.15.

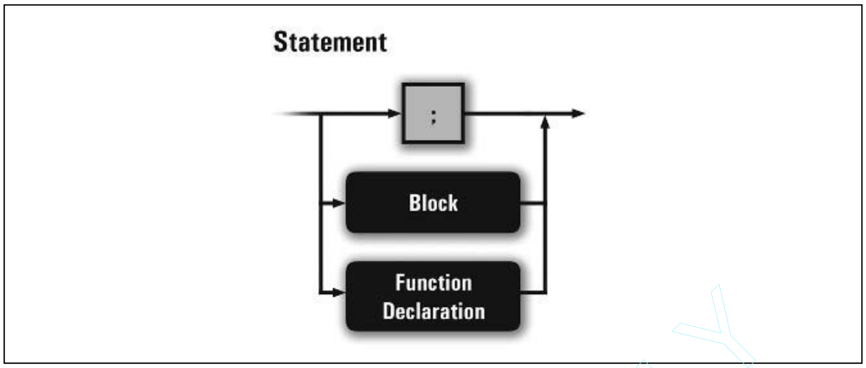


Figure 15.15

The Statement syntax diagrams with added support for function declarations.

As you can see, the parsing function you'll use to handle function declarations is called `ParseFunc ()`. It should be pretty clear that you'll be continually updating the Statement non-terminal as each new statement type is added. With that out of the way, let's talk about what this new function will do. Parsing the `func` token is obviously easy, as is the identifier. And the code block parsing logic is already done—all you have to do is make a call to `ParseBlock ()`. The only tricky part here will be parsing the parameter list, but it's nothing particularly difficult when you get right down to it.

Overall, however, functions do bring with them a reasonable level of complexity, so let's take it one step at a time. Rather than dump the entire `ParseFunc ()` function on you at once, you can step through it gradually. There are really three major aspects to parsing a function—the name, the parameter list, and the code block that comprises the function's body. I'll now discuss each of these three components separately.

Parsing and Verifying the Function Name

The first task in parsing a function is verifying its name and using it to add the function to the function table. Not surprisingly, this comprises the first chunk of code in the `ParseFunc ()` function. Let's have a look:

```
void ParseFunc ()
{
    // Make sure we're not already in a function
    if ( g_iCurrScope != SCOPE_GLOBAL )
        ExitOnCodeError ( "Nested functions illegal" );

    // Read the function name
    ReadToken ( TOKEN_TYPE_IDENT );
    // Add the non-host API function to the function
    // table and get its index
    int iFuncIndex = AddFunc ( GetCurrLexeme (), FALSE );

    // Check for a function redefinition
    if ( iFuncIndex == -1 )
        ExitOnCodeError ( "Function redefinition" );

    // Set the scope to the function
    g_iCurrScope = iFuncIndex;
```

By the time `ParseFunc ()` is called, the `func` keyword has already been read by `ParseStatement ()`. After all, that's how it knew to call this function in the first place, right? Because of this, the first thing it does is attempt to read an identifier token. Before doing so, however, it makes sure that the current scope is `SCOPE_GLOBAL`; if you aren't in the global scope, it can only mean you're in a function. And because XtremeScript doesn't support nested functions, this results in an error.

Once the scope has been verified as global, the function's identifier is read and a new entry in the function table is immediately created with a call to `AddFunc ()`. You tell `AddFunc ()` the name of the new function by passing it the current lexeme with a call to `GetCurrLexeme ()`, which of course returns the identifier. You also pass it `FALSE` for the `iIsHostAPI` parameter, to let it know that this is a script-defined function, not a host API function.

`AddFunc ()` returns an index to the newly created function, which you store in `iFuncIndex`. The first thing to do at this point is find out if the index is -1; if so, it's a flag that a function with the specified name already exists in the table. Because this means a function redefinition has occurred, an error is presented to the users. If the index is valid, however, you know the function was added properly, and immediately assign it to `g_iCurrScope` to ensure that all subsequent statements, including function declarations, will be aware that you're currently inside a function.

Parsing the Parameter List

This takes care of the function name, so the first component of a function declaration is behind us. Next up is the parameter list. Although this is mostly a straightforward affair, there is one caveat that you can't forget to address. As you learned in Chapters 8 and 9, parameters can be passed using two distinct conventions: right-to-left, and left-to-right. Generally, because (Western) people read from left-to-right, parameters are defined in that order and it makes the most intuitive sense to push them onto the stack that way as well. However, from the function's perspective, this results in a reversed parameter list, because the last parameter in the list is the first one available relative to the top of the stack (which is a result of the stack's *last in, first out* nature). Because of this, a specific convention must be agreed upon beforehand; once a language has defined such a convention, parameters are pushed onto the stack using the chosen order, and are popped off within the function's code in the reverse order.

XtremeScript will pass parameters in the left-to-right order, which means functions will have to read them from right to left. What this means is that the parser, I-code module, and code emitter must all make sure that parameters are read within the function from right to left. The solution here is to add the parameters to the symbol table in the reverse order. If you recall the last chapter, you'll remember that the code emitter's `EmitScopeSymbols ()` function reads from the symbol table in a sequential manner; it moves from the first node to the last, which means that parameter declarations are emitted in the same order in which they're added to the table. This means that if you want those declarations to appear in the right-to-left order, it's necessary to add them to the symbol table backwards.

The problem with this is that the lexer module will pass you the lexemes for each parameter's identifier in a rigid left-to-right order, as it's read from the source code. There's no way to make the lexer "jump around" within the source, so there's no way to start with the last parameter and lex your way back to the first. Instead, you have to buffer the parameters locally as they're read, so they're kept in an array in left-to-right order. Then, after reading them, you can cycle through the array backwards and add the resulting sequence of identifiers produced in this second loop to the symbol table.

Here's the next segment of `ParseFunc ()`, which parses and processes the parameter list:

```
// Read the opening parenthesis
ReadToken ( TOKEN_TYPE_DELIM_OPEN_PAREN );

// Use the look-ahead character to determine if the
// function takes parameters
if ( GetLookAheadChar () != ')' )
{
    // If the function being defined is _Main (), flag an error since
    // _Main () cannot accept paraemters
    if ( g_ScriptHeader.iIsMainFuncPresent &&
         g_ScriptHeader.iMainFuncIndex == iFuncIndex )
    {
        ExitOnCodeError ( "_Main () cannot accept parameters" );
    }

    // Start the parameter count at zero
    int iParamCount = 0;
```

```
        // Create an array to store the parameter list locally
        char ppstrParamList [ MAX_FUNC_DECLARE_PARAM_COUNT ][ MAX_IDENT_SIZE ];

        // Read the parameters
        while ( TRUE )
        {
            // Read the identifier
            ReadToken ( TOKEN_TYPE_IDENT );

            // Copy the current lexeme to the parameter list array
            CopyCurrLexeme ( ppstrParamList [ iParamCount ] );

            // Increment the parameter count
            ++ iParamCount;

            // Check again for the closing parenthesis to see
            //  if the parameter list is done
            if ( GetLookAheadChar () == ')' )
                break;

            // Otherwise read a comma and move to the next parameter
            ReadToken ( TOKEN_TYPE_DELIM_COMMA );
        }

        // Write the parameters to the function's symbol table in
        // reverse order, so they'll be emitted from right-to-left
        while ( iParamCount > 0 )
        {
            -- iParamCount;
            // Add the parameter to the symbol table
            AddSymbol ( ppstrParamList [ iParamCount ], 1, g_iCurrScope,
                SYMBOL_TYPE_PARAM );
        }

        // Set the final parameter count
        SetFuncParamCount ( g_iCurrScope, iParamCount );
    }

    // Read the closing parenthesis
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );
```

You can begin by attempting to read an opening parenthesis token. If it's found, you immediately use the look-ahead to determine whether a closing parenthesis follows it. If so, the parameter list is empty and you can skip past the parameter parsing logic entirely. If not, you have to make sure the current function isn't `_Main ()`, because `_Main ()` can't legally accept parameters. Otherwise, you create a local variable called `iParamCount`, which tracks the number of parameters the function accepts, sets it to zero, and begins a parameter-parsing loop. You also declare a local array called `ppstrParamList []`, which will store the identifier strings of each parameter you parse. To dimension this array, you create a new constant called `MAX_FUNC_DECLARE_PARAM_COUNT`, which sets a maximum number of parameters that a function declaration can contain:

```
#define MAX_FUNC_DECLARE_PARAM_COUNT        32
```

As you can see, I have mine set to 32. This is yet another case of overkill, because it's unlikely that a function (especially in the context of scripting) will ever need more than six to eight parameters at most.

At each iteration of the loop, you attempt to read an identifier, which is always the first (and sometimes only) token of a parameter. If it's found, you add it to the `ppstrParamList []` array with `CopyCurrLexeme ()` and increment the parameter count.

You can once again consult with the look-ahead to find out whether a closing parenthesis appears to be the next token. If not, you read a comma token and the loop completes the iteration. If so, however, you break the loop and make a call to `SetFuncParamCount ()` in order to update the function's entry in the table to reflect the parameter count you gathered during the loop and saved in `iParamCount`.

You now have the parameter list in the local array, so it's time to move backwards through its elements and add them in reverse order to the symbol table. This is done with a `while` loop, which decrements `iParamCount` at each iteration and uses it as an index into the array. The string at that index is the identifier of the parameter you're adding, so you pass it to `AddSymbol ()`. You must set the symbol size to 1 (because there's no such thing as a parameter array), pass the scope you set earlier in `g_iCurrScope` so the symbol table knows the parameter is local to this function, and finish the call with the `SYMBOL_TYPE_PARAM` flag so the symbol is recorded specifically as a parameter.

The parameter list parsing-process is complete, so you should finish up by validating the closing parenthesis with `ReadToken ()`.

At this point, the function's parameter count has been stored along with the name in the function table, and each of its parameters have been stored in the symbol table in right-to-left order as local variables within the function's scope. In other words, the parameter list has been fully parsed and processed.

Parsing the Function's Body

The last order of business is parsing the function's body. Fortunately, function bodies are really just code blocks, and because you've already written `ParseBlock ()`, all you need to do is call it. Of course, before doing so, you need to use `ReadToken ()` to ensure that an opening curly brace is next in the token stream. If so, `ParseBlock ()` handles the rest. Here's the code:

```
    // Read the opening curly brace
    ReadToken ( TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE );

    // Parse the function's body
    ParseBlock ();

    // Return to the global scope
    g_iCurrScope = SCOPE_GLOBAL;
}
```

At this point, it's important to understand what's going on. You're currently in a statement parsing loop run by `ParseSourceCode ()`, which is continually calling `ParseStatement ()` until the end of the file is reached. During one of the invocations of `ParseStatement ()`, a `func` token was found, which caused `ParseFunc ()` to be called, which brings you to the present moment. Now, however, you're calling `ParseBlock ()` from *within* `ParseFunc ()`, which means that the overall statement parsing loop managed by `ParseSourceCode ()` is halted until the block is fully parsed. When `ParseBlock ()` returns, it will return to `ParseFunc ()`, which will return to `ParseStatement ()`, which will return to `ParseSourceCode ()`. In addition to simply reinforcing both the highly nested and recursive nature of this parsing method, it also demonstrates that the parsing of the function's block takes place entirely within the confines of `ParseFunc ()`. Because this is such a visual process, refer to Figure 15.16 for a better idea of what's happening.

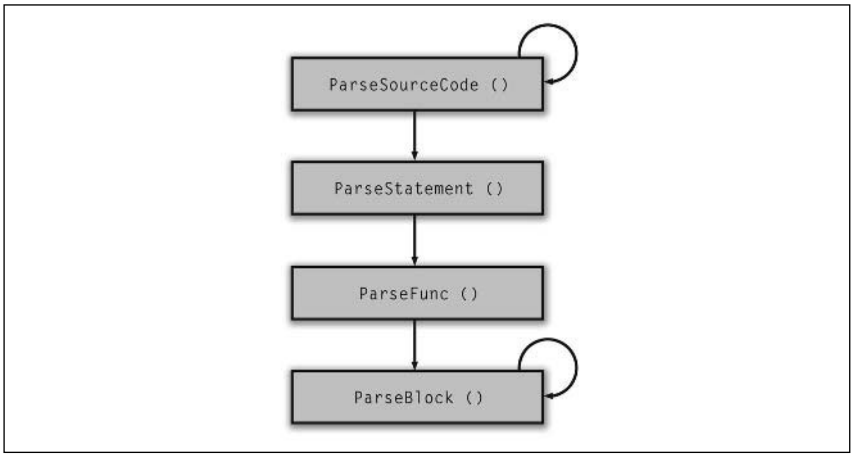


Figure 15.16

The parsing of a function's block takes place within `ParseFunc ()`, which takes place within `ParseStatement ()`, which takes place within `ParseSourceCode ()`. Whew!

Once ParseBlock () returns, you know the entire function body has been handled. Of course, at this point, all it can contain are empty statements and nested blocks, but even after the rest of the statement types are added, ParseFunc () will remain the same. Because you're now back outside the function's body, you can set the scope back to SCOPE_GLOBAL. Remember, if the user attempts to nest a function call, the offending func token will be found *inside* the nested call to ParseBlock (), which is why you never have to make any permanent changes to the g_iCurrScope variable. You should only change it once before making the call, and immediately change it back afterwards.

NOTE

One extremely important note to remember about the XtremeScript compiler is that it does *not* allow the forward referencing of functions like XASM did. ParseSourceCode () is called only once, which means only a single pass is made over the source code. Because of this, it's inconvenient to anticipate and retroactively verify forward referencing, and I left it out entirely to keep things clean and simple. This does go to show, however, that single-pass compilers have their weaknesses. Many languages suffer from this same setback, which means that functions must be declared in order of their usage; a function call can only be made *below* that function's definition in the source code. C++ eliminates this problem without a second pass by requiring any forward-referenced functions to be declared above the program's main source code with function prototypes. You may want to try adding such a feature on your own.

As a final note, now that the parser has a notion of scope, you need to make a change to ParseBlock (). Because code blocks can only appear within functions (including blocks that *are* the function), you need to ensure that they never appear in the global scope. ParseBlock () now looks like this:

```
void ParseBlock ()
{
    // Make sure we're not in the global scope
    if ( g_iCurrScope == SCOPE_GLOBAL )
        ExitOnCodeError ( "Code blocks illegal in global scope" );

    // Read each statement until the end of the block
    while ( GetLookAheadChar () != '}' )
        ParseStatement ();

    // Read the closing curly brace
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_CURLY_BRACE );
}
```

Nothing's changed except the initial `g_iCurrScope` check. If the scope is currently global, an error is presented to alert the users that the block is illegal.

Variable and Array Declarations

Now that you can determine whether you're inside a function, and get a hold of current function's index at any time, you're ready to tackle variable declarations. Figure 15.17 depicts the now familiar syntax diagram for the declaration of variables and arrays.

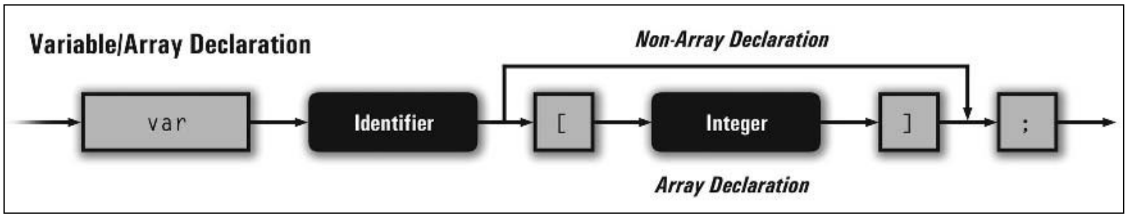


Figure 15.17

The syntax diagram for variable and array declarations.

As you should know well at this point, the diagram tells you that a variable is declared with a `var` token followed by an identifier. If the variable is intended to be an array, an optional integer index follows the identifier, enclosed in brackets. Regardless of which path is taken, however, both end with a semicolon.

You may be wondering why you need to explicitly mention the semicolon in the declaration. Because all statements end with it, why not just automatically check for it after calling the proper `Parse* ()` function in `ParseStatement ()`? The reason for this is that only *some* statements end in a semicolon. Remember, function declarations are statements too, and the parser you just finished didn't check for a semicolon because the syntax doesn't require it to do so. Because of this, the check for the terminating token is done on an individual statement basis.

As was the case with functions, however, and as will be the case for all subsequent statement types, the first stop in the implementation process is `ParseStatement ()`, where the new statement type will be recognized by its `switch` block with the addition of a new case for the `var` token. As you might imagine, variable declarations are parsed with `ParseVar ()`, so let's add it:

```
void ParseStatement ()
{
    // If the next token is a semicolon, the statement
    // is empty so return
    if ( GetLookAheadChar () == ';' )
    {
        ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
        return;
    }

    // Determine the initial token of the statement
    Token InitToken = GetNextToken ();

    // Branch to a parse function based on the token
    switch ( InitToken )
    {
        // Unexpected end of file
        case TOKEN_TYPE_END_OF_STREAM:
            ExitOnCodeError ( "Unexpected end of file" );
            break;

        // Block
        case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
            ParseBlock ();
            break;

        // Function definition
        case TOKEN_TYPE_RSRVD_FUNC:
            ParseFunc ();
            break;

        // Variable/array declaration
        case TOKEN_TYPE_RSRVD_VAR:
            ParseVar ();
            break;

        // Anything else is invalid
        default:
            ExitOnCodeError ( "Unexpected input" );
            break;
    }
}
```

Figure 15.18 depicts the latest version of the Statement non-terminal, now with support for variable/array declarations. With `ParseStatement ()` updated to understand the initial token of variable declarations, you can add the declaration-parsing function.

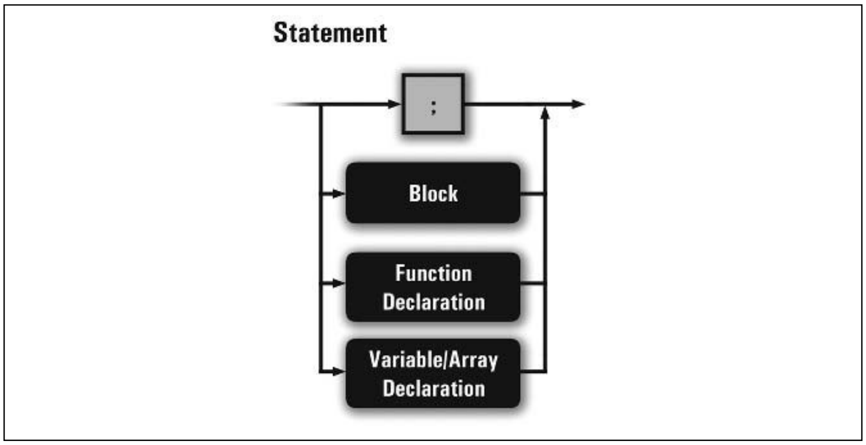


Figure 15.18

The new version of the Statement diagram, now with variable/array declaration support.

Here's the code to do so:

```
void ParseVar ()
{
    // Read an identifier token
    ReadToken ( TOKEN_TYPE_IDENT );

    // Copy the current lexeme into a local string buffer
    // to save the variable's identifier
    char pstrIdent [ MAX_LEXEME_SIZE ];
    CopyCurrLexeme ( pstrIdent );

    // Set the size to 1 for a variable (an array will
    // update this value)
    int iSize = 1;

    // Is the look-ahead character an open brace?
    if ( GetLookAheadChar () == '[' )
    {
        // Verify the open brace
        ReadToken ( TOKEN_TYPE_DELIM_OPEN_BRACE );

        // If so, read an integer token
        ReadToken ( TOKEN_TYPE_INT );

        // Convert the current lexeme to an integer to get the size
        iSize = atoi ( GetCurrLexeme () );
```

```
            // Read the closing brace
            ReadToken ( TOKEN_TYPE_DELIM_CLOSE_BRACE );
        }

        // Add the identifier and size to the symbol table
        if ( AddSymbol ( pstrIdent, iSize, g_iCurrScope, SYMBOL_TYPE_VAR ) == -1 )
            ExitOnCodeError ( "Identifier redefinition" );

        // Read the semicolon
        ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
    }
```

As you can see, it's a pretty simple process, and definitely easier than function declaration parsing. Because ParseStatement () already consumed the var token, the identifier is up next. You must verify its presence with ReadToken (), use CopyCurrLexeme () to make a physical copy of the identifier string, and save the string in the locally declared pstrIdent string buffer.

This finishes the parsing of a single variable's declaration, but because you have to be ready for arrays as well, your job isn't done. A local flag called iSize is declared and set to one, representing the fact that you're still assuming the declaration is for a single variable. You then once again read the handy look-ahead character to determine whether an opening brace token is up next, verifying it with ReadToken () if so. ReadToken () is then called again to read the integer array size, which is converted to a real integer value with atoi () and saved in iSize. The closing brace is then verified with a third call to ReadToken () and the parsing is complete.

At this point, you have all the information you need to register the symbol with the symbol table. To do this, AddSymbol () is called, along with the variable's identifier (stored in pstrIdent), size (stored in iSize), scope, and the SYMBOL_TYPE_VAR flag. You should now understand why you had to make a physical copy of the identifier—if an array was found, the next call to ReadToken (), which would have verified the opening curly brace, would've overwritten the current lexeme string with [and, in turn, deprived you of a copy of the variable's identifier. By copying it locally first, you're free to call the lexer as much as you want without disturbing any important data.

AddSymbol () returns an index to the newly created symbol, but if that index is -1, it's a sign that a symbol with the specified name already exists within the same scope, or an overlapping scope in the case of globals. When this occurs, an error must be flagged that lets the users know that the identifier was redefined. To complete the process, the semicolon is read with ReadToken ().

As variable declarations are parsed, the symbol table is populated with information regarding both global and local data. Remember, ParseVar () can be called in any scope, so you don't need to write any extra code to handle global or local variables specifically. By the time the compiler reaches the end of the source file, a record of the script's entire collection of variables will have been assembled.

Host API Function Declarations

The last type of declaration to cover might not seem immediately obvious. What's a host API declaration? If you recall the development of XASM in Chapter 9, you'll remember that host API function calls were obvious to the assembler because they were also used in the context of a `CallHost` instruction. As a result, their differentiation from script-defined functions was obvious, because script functions were called with the `Call` instruction.

You don't use any "instructions" to call functions in a high-level language such as XtremeScript, however. Function calls simply consist of the function's name and parameter list. Because of this, there's no easy way to know whether a given function belongs to the host API. For example, consider the following code:

```
func Square ( X )
{
    return X * X;
}

func _Main ()
{
    var U;
    var V;

    U = Square ( 64 );
    V = SomeOtherFunc ( U );
}
```

Although you can easily determine that `Square ()` is a script-defined function due to its preceding declaration, you have no way of telling what `SomeOtherFunc ()` is. From the perspective of the compiler, it could be anything—a misspelled version of a real function, a completely non-existent function, or a host API function that is indeed real, but not within the script. The problem is, the parser will have no choice but to assume that the function call is invalid. This cuts you off from the host API entirely, and renders the entire scripting system useless.

One way to solve the problem is simply to consider all function calls valid, regardless of whether the name is found in the function table. This way, you can assume that any unknown functions are defined by the host API, and everything will work out. The downside here, however, is that this allows completely nonexistent functions to be called without the compiler issuing an error. This means that simple misspellings on behalf of the user, such as `Squsre ()` instead of `Square ()`, will go unnoticed and lead to enigmatic logic errors.

The only safe way to resolve this situation is to give the script writer some way to formally declare a host API function ahead of time. Although it's still possible to define a function that the host application never defines, at least this rules out the possibilities of accidental misspellings that the compiler doesn't flag. To do this, you need to make a small addition to the XtremeScript language by adding the `host` keyword.

The host Keyword

The purpose of `host` is to allow the script writer to declare a host API function before its subsequent use. Functions declared by `host`, even though they don't have a body or even a parameter list, are added to the function table. This allows the parser to verify that a call is indeed valid, whether it's to a script-defined or host-defined function.

The syntax of the `host` keyword is simple. Here's an example:

```
host MyHostAPIFunc ();
```

From this point onward, the function table will have a record of a host API function called `MyHostAPIFunc ()`. Notice that I also enforce the `()` notation at the end of the declaration; even though this isn't necessary, I think it makes the whole thing more readable. Figure 15.19 contains the `host` keyword's syntax diagram.

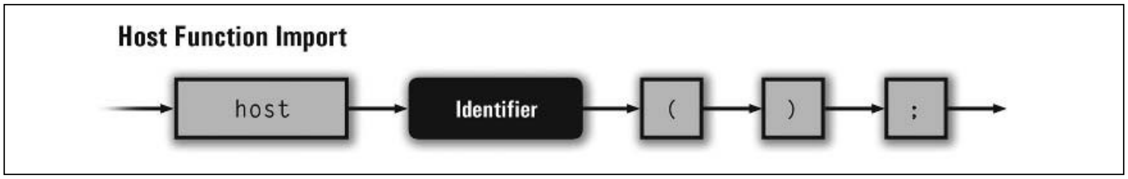


Figure 15.19

The syntax diagram for the `host` *keyword.*

This directive would have been added to the language along with the rest of the specification in Chapter 7, but I felt that the perspective gained in Chapters 9 through 11 in regards to the host API and its inner workings were necessary first. So, I deferred its introduction until now. What this does mean, however, is that the lexer needs to understand a new reserved word.

Upgrading the Lexer

The current lexical analyzer module has no idea that the `host` keyword has been added to the language, and will end up thinking it's an identifier. To alleviate this, you just need to make a few superficial changes. Start by adding the `TOKEN_TYPE_RSRVD_HOST` constant to the token type constant list:

```
#define TOKEN_TYPE_RSRVD_HOST            16
```

Then, under the `LEX_STATE_IDENT` case in the `switch` block that `GetNextToken ()` uses to convert the terminal lexer state to a token type, you simply add this small block of code:

```
// host
if ( stricmp ( g_CurrLexerState.pstrCurrLexeme, "host" ) == 0 )
    TokenType = TOKEN_TYPE_RSRVD_HOST;
```

That's all it takes. The lexer now understands the new keyword, and you're ready to implement it. You're encouraged to check out the source, however, to see it in the context of the rest of the lexer's code.

Parsing and Processing the host Keyword

All that's left at this point is to add a `ParseHost ()` function that will parse and process the `host` keyword and add its function to the function table. Of course, the first step in doing this is once again making changes to `ParseStatement ()`, so that it will understand the initial `host` token:

```
void ParseStatement ()
{
    // If the next token is a semicolon, the statement
    // is empty so return
    if ( GetLookAheadChar () == ';' )
    {
        ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
        return;
    }

    // Determine the initial token of the statement
    Token InitToken = GetNextToken ();

    // Branch to a parse function based on the token
    switch ( InitToken )
    {
        // Unexpected end of file
        case TOKEN_TYPE_END_OF_STREAM:
            ExitOnCodeError ( "Unexpected end of file" );
            break;

        // Block
        case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
            ParseBlock ();
            break;
        // Function definition
        case TOKEN_TYPE_RSRVD_FUNC:
            ParseFunc ();
            break;

        // Host API function import
        case TOKEN_TYPE_RSRVD_HOST:
            ParseHost ();
            break;

        // Variable/array declaration
        case TOKEN_TYPE_RSRVD_VAR:
            ParseVar ();
            break;

        // Anything else is invalid
        default:
            ExitOnCodeError ( "Unexpected input" );
            break;
    }
}
```

Figure 15.20 is a more recent version of the ever-evolving Statement non-terminal syntax diagram.

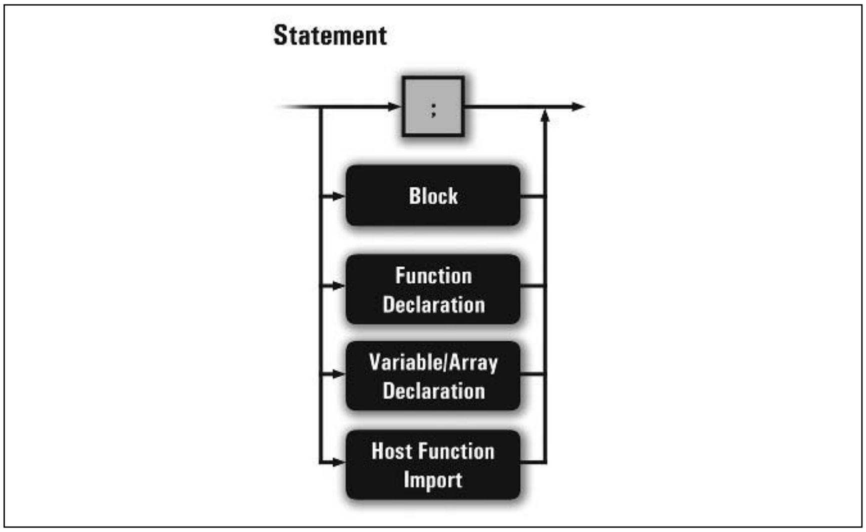


Figure 15.20

The Statement non-terminal syntax diagram, with support for host function imports.

Note that I refer to it as a "host API function import". If you recall from Chapter 11, the process of exposing a function on behalf of the host application is called *exporting*. This means that from the script's perspective, the function is being *imported*. Anyway, whenever the `host` token appears as the initial token of a new statement, `ParseHost ()` is called to parse the declaration. Let's take a look:

```
void ParseHost ()
{
    // Read the host API function name
    ReadToken ( TOKEN_TYPE_IDENT );

    // Add the function to the function table with the host API flag set
    if ( AddFunc ( GetCurrLexeme (), TRUE ) == -1 )
        ExitOnCodeError ( "Function redefinition" );

    // Make sure the function name is followed with ()
    ReadToken ( TOKEN_TYPE_DELIM_OPEN_PAREN );
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );

    // Read the semicolon
    ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
}
```

This is probably the simplest of all the declaration parsing functions. It begins by reading the identifier token, which comes directly after the `host` token. This token's corresponding lexeme is the function name, which is passed to `AddFunc ()` to create the function. Note that now, you pass `TRUE` as the `iIsHostAPI` parameter so it's known that this function is not script-defined. Remember, however, that you're storing host API functions and script-defined functions in the same table. Because of this, name clashes cannot exist—if a script function called `MyFunc ()` is entered into the table, and a host API function is declared with the same name, a function redefinition error will result. This is a good thing—because host API and script-defined function calls look identical, there was no way to tell which function was being called.

NOTE

Of course, function overloading would allow you to differentiate between two functions of the same name, which would be one way to allow host API and script-defined functions to share identifiers. The problem with this, however, is determining which function is being called based on the parameter list alone. Remember, XtremeScript is a completely typeless language, which means that unless the parameter lists were different sizes, it would be impossible to tell one function from the other based on the data types alone.

Once the function name has been parsed and added to the function table with the host API flag, two more tokens are read to ensure that the statement ends with a `()`. Finally, a semicolon is read, and the declaration is fully parsed.

Testing Code Emitter Module

So far, the parser is shaping up quite nicely. It understands the fundamental structure of a script through its support for code blocks and [empty] statements, and can now both parse and process the full set of XtremeScript declarations. This includes both local and global variables and arrays, functions with parameter lists, and host API function import declarations with the newly added `host` keyword. At this point, even though you still aren't generating I-code of any sort, a script written using only the statements the parser currently understands *will* produce visible output.

To test this, check out the following script. Although it consists only of declarations and empty code blocks, you can actually see its output in the form of an equivalent XVM assembly file:

```
/*
    XtremeScript declaration test.
*/

// Import a host API function
host MyHostAPIFunc ();

// Declare some globals
var GlobalX;
var GlobalY;

// Create a simple test function
func MyFunc ( X, Y )
{
    // Declare some locals
    var U;
    var V;
}

// Declare a _Main () function
func _Main ()
{
    // Declare some locals
    var LocalX;
    var LocalY;
}
```

By saving this file as `declare.xss` and running it through the `Programs/Chapter 15/15-01/` version of the compiler on the CD with the `-A` switch, you'll get the following output:

```
; DECLARE.XASM

; Source File: TEST.XSS
; XSC Version: 0.8
;   Timestamp: Fri Sep 13 14:53:08 2002

; ---- Directives -------------------------------------------
; ---- Global Variables -------------------------------------

   Var GlobalX
   Var GlobalY

; ---- Functions --------------------------------------------

   Func MyFunc
   {
      Param X
      Param Y

      Var U
      Var V

      ; (No code)
   }

; ---- Main -------------------------------------------------

   Func _Main
   {
      Var LocalX
      Var LocalY

      ; (No code)
   }
```

Is that cool or what? The high-level language is now officially being translated to low-level code (in the form of directives, at least)! Note how the parameter list is automatically translated to `Param` declarations, and how the functions and scope levels were faithfully translated as well.

Notice also that the `host` declaration seems to have disappeared. This is because such declarations only exist for the compiler's benefit, not the assembler's. However, by giving the compiler a record of which functions are which, it will know whether to ultimately emit the function call with the `Call` instruction or the `CallHost` instruction.

Parsing Simple Expressions

Everything you've done so far has been reasonably static, and the things that aren't have been mostly non-recursive. For example, a `var` declaration is simply the `var` keyword, followed by an identifier, followed by a semicolon. Array notation notwithstanding, that's the exact form in which all `var` declarations will appear. `host` declarations are even simpler; there aren't any alternative forms of any kind to worry about there. Even function declarations are pretty straightforward, even if their arbitrarily sized parameter lists are more "dynamic" than the other declarations of the language.

Expressions, on the other hand, are quite a bit different than anything you've encountered so far. In addition to being arbitrarily long, they're highly recursive; there are operator precedence levels to deal with, nested sub-expressions within parentheses, and non-arithmetic operators like relational and logical operators. All of these factors mean one thing—your first encounter with significantly complex parsing logic.

Because of the obvious complexity in parsing expressions, you may be wondering why I am having you tackle the problem now. After all, wouldn't it make more sense to get loops, branching, and other such language features out of the way first? Unfortunately, this is more or less impossible; after all, loops, branches, assignments, function calls, and almost every other aspect of XtremeScript require the capability to parse expressions in some capacity. Because of this, you'll do well to get them out of the way now.

An Expression Parsing Strategy

So how does one go about parsing an expression? For example, imagine the following:

```
X = Y * ( Z / 3.14159 + MyFunc ( U, V ) ^ Theta ) - Phi % Gamma;
```

Looks pretty intimidating, doesn't it? I mean it's like a train wreck of operators, parentheses, and even function calls. Somehow, despite the nesting and recursion, you need to parse this thing in purely sequential order, from left to right. This can be a considerable challenge when you're new at this stuff, so you can start small and work your way up incrementally.

Parsing Addition and Subtraction

Let's start at the very bottom. Specifically, with this:

```
16 + 32
```

Here you have two operands, separated by the + operator. You know, as a well-trained, arithmetic-loving human, that this expression is saying "add 16 to 32." You also know, thanks to the human brain's modest computation facilities, that the sum is 48. But how can you get the parser to do the same thing?

You can start by applying the same approach used for the rest of the parsing tasks. For example, in terms the compiler can understand, the previous expression is actually just three tokens:

```
TOKEN_TYPE_INT
TOKEN_TYPE_OP
TOKEN_TYPE_INT
```

So, a simple parsing strategy for a two-operand expression would be to read the first token, which corresponds to the first operand, read the second token, which corresponds to the operator, and read the third token, which corresponds to the second operand. Once you've read these tokens in, you can convert the first and third tokens (the operands) to integers, and use the second token (the operator) to determine which operation should be performed with these two values. You can call `GetCurrOp ()` after reading the second token, and because it will return `OP_TYPE_ADD`, you'll know to add the two integer values. That wasn't so bad, right? Check out Figure 15.21.

In fact, you can even apply this to entire chains of addition and subtraction operators, like so:

```
16 + 32 - 4 + 256 - 72
```

With an only slightly modified game plan, you can handle this new, obviously more complex expression. The secret is realizing that it really *isn't* more complex—aside from the repetition, it's

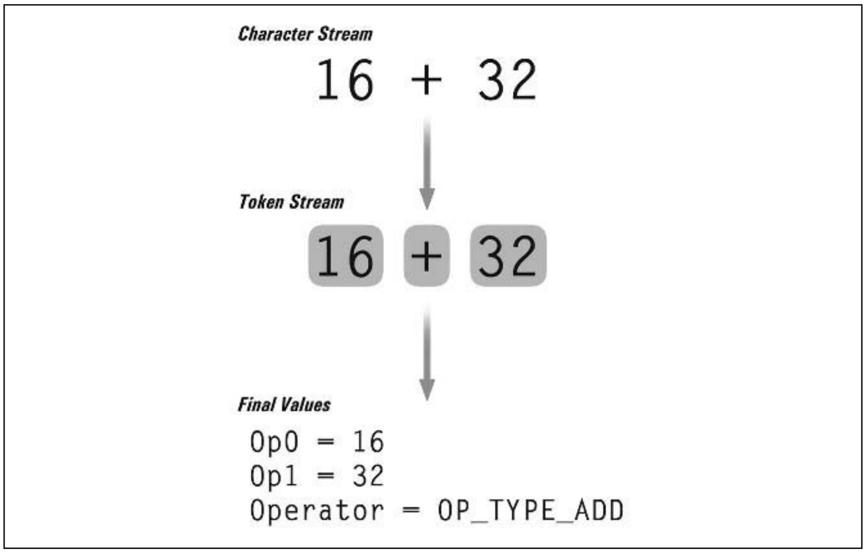


Figure 15.21

Parsing a two-operand addition expression.

the same thing. The parsing process would begin just as it did in the last example—16, +, and 32 would be read from the token stream, and after the integer lexemes were converted to their actual values, the addition operator would be applied to them. This would yield 48. From here, you can simply continue the process, by conceptually "collapsing" 16 + 32 into 48. In other words, you can now think of the expression like this:

```
48 - 4 + 256 - 72
```

Only the bold part represents a change; the rest of the expression remains unchanged. If you repeat the process, you'll read the lexemes 48, -, and 4. You now have 44, which once again prompts you to collapse two operands and an operator into a single operand. 48 - 4 is now 44, which leaves you with yet another, more compact, version of the expression:

```
44 + 256 - 72
```

Again, you repeat the process, and perform the 44 + 256 operation, yielding a sum of 300:

```
300 - 72
```

At this point, you're back to an instance of the first example—two operands separated by a single operator. By subtracting 72 from 300, you get the result:

```
228
```

Presto! Check out Figure 15.22 for a more visual idea of how this process of incrementally "collapsing" the expression works.

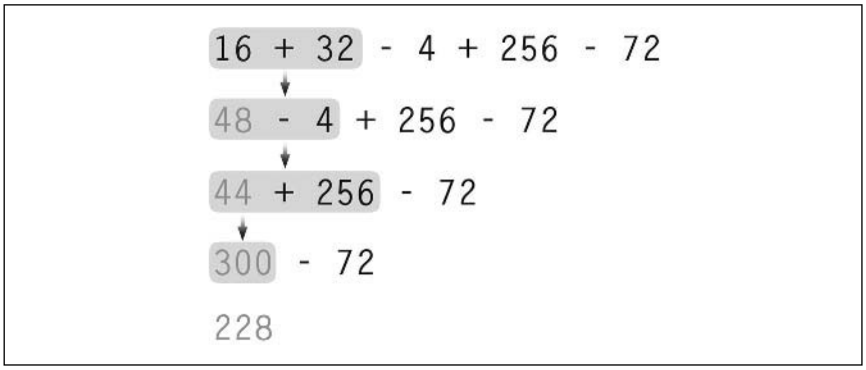


Figure 15.22

Parsing an expression with repetitious collapsing.

Multiplication, Division, and Operator Precedence

The straight left-to-right approach has served you well so far, allowing you to easily chomp your way from one end of the expression to the other, while keeping a constantly updated result value handy until you reach the last operation. This simple method breaks down, however, when the multiplication and division operators are thrown into the mix. This is because operators of the same precedence levels are meant to be handled in a sequential, left-to-right order. Imagine if you tried parsing the following expression using the current technique:

```
16 + 32 * 2
```

You'd first add 16 to 32, and then multiply the resulting 48 by 2. The "result" would be 96, even though the real result is 80. The pure left-to-right method doesn't take operator precedence into account, which results in the operators being applied in the wrong order. What's worse is that you couldn't change this if you wanted to. Even with the look-ahead and the capability to read and subsequently rewind the token stream, there's no way to know enough about the rest of the expression to make educated decisions at each step of the way.

What you need is a way to put certain operands on the "back burner," so to speak, until you can be sure that there isn't an operator of higher precedence that needs to be dealt with first. In purely conceptual terms, what you need to do is read 16, 32, and the addition operator, but hold off on the operation for a moment. Instead, you'll move on the next operator, which is multiplication, and perform the 32 * 2 operation. Then, with the result of 64 already calculated, you'll move back to 16 and perform the addition. This will of course yield a sum of 80, which is the correct result.

So how is this done in practice? One way is to create a temporary register variable that will store the operand associated with the lower-precedence operator until the appropriate time. You could thus save 16 in this register, perform the multiplication of 32 and 2, and then refer back to the register to complete the expression. Unfortunately, this doesn't help you much in a situation like this:

```
16 + 32 * 4 / 256 - 72 * 65536 + 2 * 4
```

You're going to need quite a few extra registers to handle this situation. What you need instead is a structure that will automatically grow to accommodate new operands as they're parsed, allowing you to store an arbitrary amount of such values until the proper time. If you haven't noticed already, this situation sounds suspiciously similar to a problem you had with the stack frames and return addresses associated with function calls. And if you recall, as I certainly hope you do, you'll remember that the solution came in the form of a stack.

Stack-Based Expression Parsing

As an expression is being parsed, you've already seen that you'll run into the problem of operator precedence quickly. In such a situation, operands associated with lower-precedence operators must be stored for later use, in a specific order, so they can be dealt with at the proper time. Fortunately, the stack provides an elegant, flexible, and straightforward way to do exactly this.

Let's go back to the original example, and see how it can be solved using stacks:

```
16 + 32 * 2
```

For the purpose of this example, you'll use two separate stacks. One will store the operands, and the other will store the operators. The following is a walk-through of the process of parsing the previous expression with these stacks.

16 is read as the first token, and is pushed onto the operand stack. The next token is +, which is pushed onto the operator stack. Up next is 32, which is pushed onto the operand stack. Figure 15.23 demonstrates the current state of the stacks at this point.

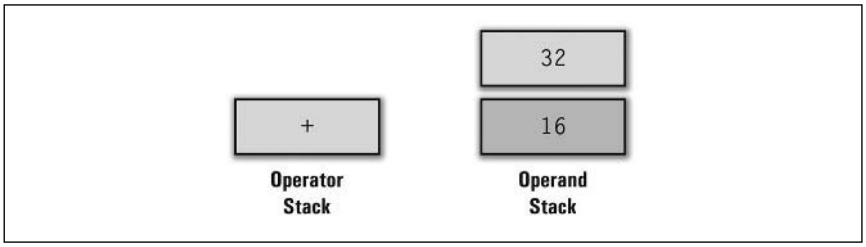


Figure 15.23

The operand and operator stacks after parsing 16 + 32.

Next you read the * token, which is pushed onto the operator stack. Finally, 2 is read, which is pushed onto the operand stack. You now have the situation shown in Figure 15.24.

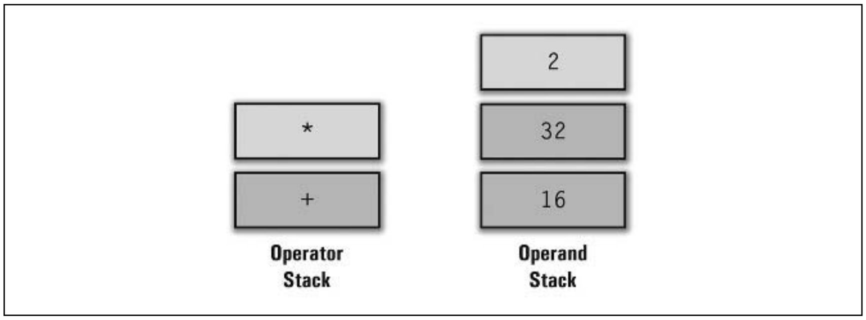


Figure 15.24

*The operand and operator stacks after parsing 16 + 32 * 2.*

When the multiplication operator is encountered, you've reached the highest precedence level and can perform the operation. This is done by popping the top element off the operator stack, as well as popping the top two elements off the operand stack. This gives you 32, 2, and the * operator. The operation is performed, and the resulting value is pushed onto the operand stack. This leaves the stacks in the form depicted by Figure 15.25.

One operation remains, so you pop the next value off the top of the operator stack (which empties it), as well as the next two values off the operand stack. This gives you 64, 16, and +. You add 64 and 16, yielding a sum of 80, and the expression is complete.

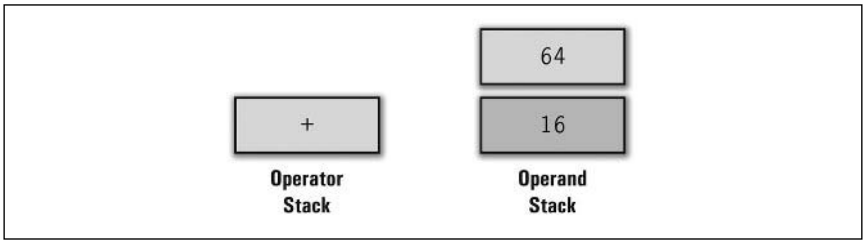


Figure 15.25

The operand and operator stacks after performing `32 * 2`*.*

This section covers the theory and code behind parsing expressions supporting:

- Integer and floating-point literal values.
- Basic arithmetic operators with the proper precedence rules: +, -, *, /.
- The unary negation and plus operator.
- Nesting with parentheses.
- Variable and array references.

This initial version of the expression parser doesn't support the assignment operator, so expressions won't "go anywhere". Rather, an expression on its own will be considered a valid statement by the parser. For example, the following statement:

```
4 + 2;
```

will be reduced to the following XVM assembly fragment:

```
Push    4
Push    2
Pop     _T0
Pop     _T1
Add     _T0, _T1
Push    _T0
```

Although the code might not make too much sense yet, you get the picture (right?).

Understanding the Expression Parser

The expression parser in the XtremeScript compiler is quite simple if you understand how it works, but therein lies the challenge. The recursive nature of expression parsing is such that if you don't understand what's going on, you'll be utterly lost; if you do understand, however, it's like second nature. So, to help make things a bit easier to swallow, you should start by breaking up the expressions you want to parse into a number of separate, recursively related entities:

- **Expressions.** An *expression* is the highest-level abstraction, and represents all expressions the parser can handle.

- **Sub-expressions.** *Sub-expressions*, in the context of this first parser, are synonymous with expressions. The next version of the parser will differentiate between the two, but for now, they're identical. A sub-expression is composed of a number of *terms*, each separated by + or - operators. For example, `X + Y - Z` is an example of a sub-expression. `X`, `Y`, and `Z` are the terms.
- **Terms.** *Terms* are the constituents of sub-expression, lying between the plus and minus operators. A term itself is composed of a number of *factors*, each of which is separated by * and / operators. Therefore, `U * V / W` is an example of a term, and `U`, `V` and `W` are the factors.
- **Factors.** A *factor* is the lowest-level entity in an XtremeScript expression, representing a single value and an optional unary operator. `16`, `-7`, `MyVar`, and `-MyArray [ 0 ]` are examples of factors. The real kicker, however, is that a factor can start with an opening parenthesis. In such a case, the factor is actually a complete nested expression. I'll talk more about this in a second.

Breaking expressions into the entities listed previously isn't simply a way to make things easier to grasp. The real reason you make these separations is that it gives you a way to take the recursive nature of expressions into account, with operator precedence. Now that you have at least some idea of the terms used to describe these entities (even if you don't quite understand what's going on just yet), you're ready to learn about how they relate to each other. For the purpose of the following examples, consider the following expression:

```
-X + Y / ( 2 * MyVar - MyArray [ 2 + 4 / Z ] ) + 17;
```

An expression, as you saw, represents an entire expression. In this case, it represents the entire expression listed here. A sub-expression is currently just a synonym for an expression, so therefore, the expression listed here is also a sub-expression.

In simplistic terms, you can describe the sub-expression as three terms:

```
-X
Y / ( 2 * MyVar - MyArray [ 2 + 4 / Z ] )
17
```

You can make this distinction by grouping each element of the expression separated by either a + or - operator, and not nested within parentheses. Even though the nested expression contains a number of + and -'s of its own, don't count those just yet. Lump them together into a single term.

Within the second term, there are two factors:

```
Y
( 2 * MyVar - MyArray [ 2 + 4 / Z ] )
```

The same rule applies here; any element of the expression separated by the `/` or `*` operator that's not nested within parentheses is considered a separate factor. Because factors are the lowest-level entities within this expression, you can evaluate them. The first is `Y`, which is simply a variable. The second is a nested expression within parentheses.

What I'm driving at here is the basis for the expression-parsing method. Using the entities you've defined, you can split expressions up into increasingly low-level components. Starting from expressions at the top, and working your way down to factors at the bottom, you can recursively evaluate expressions (or more specifically, generate expression evaluating code). The beauty of this approach, however, is that a factor can contain a top-level expression *within* it. Because of this, the lowest-level entity can "wrap around" back to the highest-level, thus creating a circular relationship. To understand this better, think back to the description of statements and blocks of code; a block of code consists of statements, but statements can also be blocks of code. This creates a circular relationship that allows infinite nesting of blocks and statements. Because an expression is ultimately just a series of factors, and because a factor can also *be* an expression, it means that expressions can contain nested expressions to any arbitrary depth.

If you understood how statements and blocks relate to one another circularly, the recursion behind expression nesting should make perfect sense. The other aspect of this approach to parsing, however, is respecting operator precedence levels. This is the reason for the multiple layers of generality that separate high-level expressions from low-level factors. In between you have sub-expressions and terms. It's no coincidence that a sub-expression is composed of terms separated by plus and minus operators, nor is it by chance that terms consist of factors separated by multiplication and division operators. This is done specifically to follow the precedence of operators.

As an expression is being analyzed, the parser will begin at the topmost level—the expression. It then works its way down to the sub-expression level, which currently involves no work because you consider the two entities to be the same thing. From here, its job is to add or subtract each term. It moves from left to right, performing addition and subtraction as it encounters each operator. For example, in the following expression:

```
10 + 27 - 16 + 2
```

The parser will consider `10` to be the first term and `27` to be the second. It will add them together, and subtract the third term, `16`. The final step is adding the last term `2`. However, not all terms are simple integers. Specifically, a term can be any number of factors, separated by multiplication and division operators. So, as another example, consider the following term:

```
128 * 4 / 3
```

The parser will handle this by multiplying the first two factors, `128` and `4`, and dividing the result by the third factor, `3`. The upshot to all of this is that sub-expressions are parsed first. This is done by parsing each term and adding or subtracting it. Terms are parsed by multiplying or dividing each of the factors they contain. Let's apply this to an example that combines the previous two. Consider the following expression:

```
10 + 128 * 4 - 16 + 10 / 2
```

Here you see multiple levels of operator precedence, which means things will be more complicated this time around. Or will they? You can solve this expression quite easily using only the techniques you've seen so far. The key is understanding when these techniques are to be used. You can step through the parser's attack on this expression to understand exactly what needs to take place.

The parser starts with the token 10. It starts at the expression level, which is currently treated as a sub-expression. Because you're at the sub-expression level, you're looking for terms to add and subtract. This means you need to parse 10 as a term. To do this, you need to shift to the term level. At the term level, you're looking for factors to multiply and divide by one another. You therefore need to parse 10 as a factor.

Parsing factors is simple. In this case, it's the integer literal value 10, so you push it onto the stack. Because the factor is the lowest level of the expression, you're done with 10 and can begin unwinding back to the higher levels. This means initially moving back to the term level, where you're multiplying and dividing factors. You look ahead to see whether a * or / operator is next. It isn't—the + operator is next—so you know you're done with the term. This takes you back to the sub-expression level, which involves adding and subtracting terms. 10 was the first term, which is now fully parsed, so the next move is determining whether a + or - operator follows. It does, so you move on to the next term with the intent of adding it to the last. 128 is the next token, so you parse it as a term. Parsing a term means parsing a number of factors separated by * and / operators. You parse the factor, 128, and push its value onto the stack. You then return to the term level, where you look for a multiplication or division operator. You find one, so you parse the next factor, which is 4. This is another integer value, so you push it onto the stack as well. The stack now consists of 10, 128, and 4. You pop the two top elements and multiply them, and then push them back onto the stack.

This initial fragment of the parser's approach to evaluating the expression hopefully gets the main point across—that you work from the top down, starting with the expression and parsing your way to each individual factor. Along the way, you subsequently process terms and sub-expressions in reverse order, preserving operator precedence. Multiplication and division are *always* evaluated first, followed by addition and subtraction. This is because a sub-expression contains terms, which in turn contains factors. Because you first work your way to the lowest level, and execute operators on the way *back* to the higher levels, the precedence levels are not violated.

Coding the Expression Parser

As you might have guessed, you can code an expression parser by creating `Parse* ()` functions for each of the expression entities covered in the last section. Specifically, you need `ParseExpr ()` for parsing expressions, which calls `ParseSubExpr ()` for parsing sub-expressions, which subsequently calls `ParseTerm ()` for parsing terms, which finally calls `ParseFactor ()` for parsing factors.

A quick summary of each function is as follows: `ParseExpr ()` is called whenever an expression needs to be parsed. Currently, its only job is to call `ParseSubExpr ()`. `ParseSubExpr ()` is responsible for parsing terms and the addition and subtraction operators between them. It parses each term with a call to `ParseTerm ()`, looks for an appropriate operator following it, and calls `ParseTerm ()` for the second operand if it finds one. `ParseTerm ()` is very similar to `ParseSubExpr ()`, except that it parses factors and the multiplicative operators. The process is the same, however; `ParseFactor ()` is called for each factor. `ParseFactor ()` is perhaps the most interesting of all. It's in charge of parsing the current factor.

This factor may be a literal integer or floating-point value, in which case it's directly pushed onto the stack. It may also be a variable, which is pushed onto the stack as well. If it's an array index, however, the process is slightly more complex. First, the array's base index is pushed onto the stack (in other words, the zero index—`Array [ 0 ]`). Then, `ParseExpr ()` is recursively called from within `ParseFactor ()` to parse the expression in between the `[]` braces. This allows entire expressions to be embedded within array references. The last type of factor the current parser can handle is the nested expression. If the `(` token is detected, `ParseExpr ()` is called, which starts the whole process over again.

To put it simply, the expression parser will interact heavily with the stack. For example, when parsing the binary `*` operator, two operands are pushed onto the stack. They're then popped off, multiplied together, and pushed back on. Why the seemingly redundant pushing and popping? The reason is that it gives the parser a chance to push entire expressions onto the stack before popping the two operands off. The result of *any* expression is always stored in the top element of the stack, which means that if two expressions are parsed in succession, the top two elements are each of their results. You can then pop them off, perform whatever operation is necessary, and push the result.

One important question that hasn't been resolved yet, however, is what exactly these top two elements will be popped *into*. If you were coding for a real processor, you'd simply pick two hardware registers and use them as the destination for each pop. You would then perform the necessary operation using these two registers as the operands. Unfortunately, the XVM only has the `_RetVal` register. Aside from being one register short of the two you'd need to support binary operations, `_RetVal` may be in use at the time of the expression's evaluation, and you certainly wouldn't want to corrupt its value by overwriting it with your own data.

I solved this problem by "simulating" a pair of general-purpose registers called `_T0` and `_T1` (`T` standing for "temporary"). This is accomplished by forcing the declaration of `_T0` and `_T1` as globals in every script. In other words, all XVM assembly scripts produced by the XSC compiler contain this at the top of their global definitions:

```
Var _T0
Var _T1
```

Now, after pushing two operands onto the stack, you can pop them into `_T0` and `_T1` and have them readily available for whatever binary operation you need to perform. To create these variables in the first place, however, I've chosen to hard-code them in `CompileSourceFile ()`, found in `xsc.cpp`:

```
void CompileSourceFile ()
{
    // Add two temporary variables for evaluating expressions
    g_iTempVar0SymbolIndex = AddSymbol ( TEMP_VAR_0, 1, SCOPE_GLOBAL,
        SYMBOL_TYPE_VAR );
    g_iTempVar1SymbolIndex = AddSymbol ( TEMP_VAR_1, 1, SCOPE_GLOBAL,
        SYMBOL_TYPE_VAR );

    // Parse the source file to create an I-code representation
    ParseSourceCode ();
}
```

Here you're manually adding two entries to the symbol table, using `TEMP_VAR_0` and `TEMP_VAR_1` as the identifiers. These are string constants defined in `xsc.h`:

```
#define TEMP_VAR_0                    "_T0"      // Temporary variable 0
#define TEMP_VAR_1                    "_T1"      // Temporary variable 1
```

The indexes into the table returned by these two calls are stored in the globals `g_iTempVar0SymbolIndex` and `g_iTempVar1SymbolIndex`, which allows you to refer to them anywhere in the program.

With the simulated temporary registers in hand, you're ready to code the expression parser. This of course begins with `ParseExpr ()`, a function that's called whenever an expression needs to be parsed. By calling this function, code for evaluating the expression will be generated. The code is specifically designed to *always* leave the expression's result on the top of the stack. So, for example, if you're handling the binary division operator, the general structure would be:

```
Expr0 / Expr1
```

Where Expr0 is the first operand and Expr1 is the second. This would be parsed by calling ParseExpr () to parse the first operand. The top element of the stack now contains the result of this expression (or at least, it will at runtime). The division operator would then be parsed and saved in a local variable. A second call would be made to ParseExpr (), and the new top of the stack contains the value of the second operand. The top two elements are popped into _T0 and _T1, and the division is performed. The result of this division is pushed onto the stack, and that's it. Here's the code for ParseExpr ():

```
void ParseExpr ()
{
    // Parse the subexpression
    ParseSubExpr ();
}
```

Of course, for all it does, its job is pretty simple. It really just defers its workload to ParseSubExpr (), whose code is listed here:

```
void ParseSubExpr ()
{
    int iInstrIndex;

    // The current operator type
    int iOpType;

    // Parse the first term
    ParseTerm ();

    // Parse any subsequent + or - operators
    while ( TRUE )
    {
        // Get the next token
        if ( GetNextToken () != TOKEN_TYPE_OP ||
             ( GetCurrOp () != OP_TYPE_ADD &&
               GetCurrOp () != OP_TYPE_SUB ) )
        {
            RewindTokenStream ();
            break;
        }

        // Save the operator
        iOpType = GetCurrOp ();
```

```
            // Parse the second term
            ParseTerm ();

            // Pop the first operand into _T1
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );

            // Pop the second operand into _T0
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

            // Perform the binary operation associated with the specified operator
            int iOpInstr;
            switch ( iOpType )
            {
                // Binary addition
                case OP_TYPE_ADD:
                    iOpInstr = INSTR_ADD;
                    break;

                // Binary subtraction
                case OP_TYPE_SUB:
                    iOpInstr = INSTR_SUB;
                    break;
            }
            iInstrIndex = AddICodeInstr ( g_iCurrScope, iOpInstr );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );

            // Push the result (stored in _T0)
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
        }
    }
```

Aside from some declarations, this function starts by calling `ParseTerm ()` to parse the first operand. After this call, code has been generated that will place the value of this operand on the top of the stack. The function then enters a loop that parses any subsequent + or - operators, as well as each operand along the way. If such an operator isn't found, the token stream is rewound and the loop breaks. Otherwise, the two operands are popped into `_T0` and `_T1`, and code is generated to perform either an addition or subtraction based on the operator token. After the operation is performed, the value is pushed back onto the stack.

`ParseTerm ()` was called by `ParseSubExpr ()` to handle each operand in between its additive operators, so let's take a look at it now:

```
void ParseTerm ()
{
    int iInstrIndex;

    // The current operator type
    int iOpType;

    // Parse the first factor
    ParseFactor ();

    // Parse any subsequent * or / operators
    while ( TRUE )
    {
        // Get the next token
        if ( GetNextToken () != TOKEN_TYPE_OP ||
             ( GetCurrOp () != OP_TYPE_MUL &&
               GetCurrOp () != OP_TYPE_DIV ) )
        {
            RewindTokenStream ();
            break;
        }

        // Save the operator
        iOpType = GetCurrOp ();

        // Parse the second factor
        ParseFactor ();

        // Pop the first operand into _T1
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );

        // Pop the second operand into _T0
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
        // Perform the binary operation associated with the specified operator
        int iOpInstr;
        switch ( iOpType )
        {
            // Binary multiplication
            case OP_TYPE_MUL:
                iOpInstr = INSTR_MUL;
                break;

            // Binary division
            case OP_TYPE_DIV:
                iOpInstr = INSTR_DIV;
                break;
        }
        iInstrIndex = AddICodeInstr ( g_iCurrScope, iOpInstr );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );

        // Push the result (stored in _T0)
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
    }
}
```

Everything here is more or less identical to the logic behind ParseSubExpr (); the only differences of course are that different instructions are generated for the operators (Mul and Div instead of Add and Sub), and that ParseFactor () is called for each operand instead of ParseTerm (). Speaking of which, check out ParseFactor () now.

ParseFactor () is a particularly large function, so rather than dump the whole thing out and let you wade through it alone, we can step through it piece by piece together. Starting from the top:

```
void ParseFactor ()
{
    int iInstrIndex;
    int iUnaryOpPending = FALSE;
    int iOpType;

    // First check for a unary operator
    if ( GetNextToken () == TOKEN_TYPE_OP &&
         ( GetCurrOp () == OP_TYPE_ADD ||
           GetCurrOp () == OP_TYPE_SUB ) )
    {
        // If it was found, save it and set the unary operator flag
        iUnaryOpPending = TRUE;
        iOpType = GetCurrOp ();
    }
    else
    {
        // Otherwise rewind the token stream
        RewindTokenStream ();
    }
```

Factors can be preceded by unary operators, so the first thing the function does is check for one. You're currently just supporting the unary + and -, so those are the only checks that are made. The result is saved in `iOpType`, and the `iUnaryOpPending` flag is set. If an operator wasn't found, the token stream is rewound. This next block is pretty big, so bear with me:

```
// Determine which type of factor we're dealing with based on the next token
switch ( GetNextToken () )
{
    // It's an integer literal, so push it onto the stack
    case TOKEN_TYPE_INT:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
        AddIntICodeOp ( g_iCurrScope, iInstrIndex, atoi ( GetCurrLexeme () ) );
        break;

    // It's a float literal, so push it onto the stack
    case TOKEN_TYPE_FLOAT:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
        AddFloatICodeOp ( g_iCurrScope, iInstrIndex,
            ( float ) atof ( GetCurrLexeme () ) );
        break;

    // It's an identifier
    case TOKEN_TYPE_IDENT:
    {
        // First find out if the identifier is a variable or array
        SymbolNode * pSymbol = GetSymbolByIdent ( GetCurrLexeme (),
            g_iCurrScope );
        if ( pSymbol )
        {
```

```
// Does an array index follow the identifier?
if ( GetLookAheadChar () == '[' )
{
    // Ensure the variable is an array
    if ( pSymbol->iSize == 1 )
        ExitOnCodeError ( "Invalid array" );

    // Verify the opening brace
    ReadToken ( TOKEN_TYPE_DELIM_OPEN_BRACE );

    // Make sure an expression is present
    if ( GetLookAheadChar () == ']' )
        ExitOnCodeError ( "Invalid expression" );

    // Parse the index as an expression recursively
    ParseExpr ();

    // Make sure the index is closed
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_BRACE );

    // Pop the resulting value into _T0 and use it as the index
    // variable

    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex,
        g_iTempVar0SymbolIndex );

    // Push the original identifier onto the stack as an array,
    // indexed with _T0

    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddArrayIndexVarICodeOp ( g_iCurrScope, iInstrIndex,
        pSymbol->iIndex, g_iTempVar0SymbolIndex );
}
else
{
    // If not, make sure the identifier is not an array, and push
    // it onto the stack
    if ( pSymbol->iSize == 1 )
    {
```

```
                    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
                    AddVarICodeOp ( g_iCurrScope, iInstrIndex,
                        pSymbol->iIndex );
                }
                else
                {
                    ExitOnCodeError ( "Arrays must be indexed" );
                }
            }
        }
        else
        {
            // It's not a variable or array
            ExitOnCodeError ( "Unknown identifier" );
        }

        break;
    }

    // It's a nested expression, so call ParseExpr () recursively and validate
    // the presence of the closing parenthesis

    case TOKEN_TYPE_DELIM_OPEN_PAREN:
        ParseExpr ();
        ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );
        break;

    // Anything else is invalid

    default:
        ExitOnCodeError ( "Invalid input" );
}
```

Phew! As you can probably tell, this is the part of the function that parses each individual factor type and emits the code for representing it within the assembly script. A `switch` block is used with `GetNextToken ()` as the criteria to determine what sort of factor is being parsed. In the case of `TOKEN_TYPE_INT` and `TOKEN_TYPE_FLOAT`, the job is easy; the literal value is simply pushed onto the stack. Identifiers are a bit trickier though, because, as usual, they can be either variables or arrays. Once again, the look-ahead comes to the rescue.

If an open bracket is found, the identifier is probably an array. The first check here is to make sure that the identifier's symbol table record is indeed of the array type; otherwise, an error is flagged. If the symbol is a valid array, `ParseExpr ()` is called again to parse the expression that lies in between the braces. The closing brace is then validated. The array index specified by the expression is then pushed onto the stack.

In the case of single variables, a similar initial check is made to ensure that the variable isn't actually an array. If not, the variable is simply pushed onto the stack, and the job is done.

The last factor type to consider is that of the nested expression, denoted by an opening parenthesis. This is a simple case to handle; `ParseExpr ()` is called to handle the expression, and `ReadToken ()` is used to make sure the expression's closing parenthesis is present.

This brings you to the last section of the code, responsible for handling any pending unary operators:

```
// Is a unary operator pending?
if ( iUnaryOpPending )
{
    // If so, pop the result of the factor off the top of the stack
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

    // Perform the unary operation
    int iOpIndex;
    switch ( iOpType )
    {
        // Negation
        case OP_TYPE_SUB:
            iOpIndex = INSTR_NEG;
            break;
    }

    // Add the instruction's operand
    iInstrIndex = AddICodeInstr ( g_iCurrScope, iOpIndex );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

    // Push the result onto the stack
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
}
```

If a negation operator is present, the value on top of the stack (the value of the factor) is popped into _T0, negated with a Neg instruction, and pushed back on. For simplicity's sake, I've left out the unary + operator; I hardly consider it common enough to worry about here, even though it's accepted by the syntax.

That's all the code you need to parse simple expressions, but as usual, you need to update ParseStatement () to recognize them:

```
// Expression
case TOKEN_TYPE_INT:
case TOKEN_TYPE_FLOAT:
case TOKEN_TYPE_OP:
case TOKEN_TYPE_DELIM_OPEN_PAREN:
case TOKEN_TYPE_IDENT:
{
    // Annotate the line
    AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

    // Rewind the token stream so the first token of the expression becomes
    // available again
    RewindTokenStream ();

    // Parse the expression and put its result on the stack
    ParseExpr ();

    break;
}
```

I just took the brute force approach and caused a whole group of initial tokens to invoke the expression parser. Let's finish things up with a simple example. Imagine that the following line of code is encountered by the expression parser:

```
2 * 2 + 4 * 4;
```

Here you have two multiplications nested within a single addition. Because you know the multiplicative factors will be parsed before the additive terms, the output shouldn't be too surprising:

```
; 2 * 2 + 4 * 4;
Push    2
Push    2
Pop     _T0
Pop     _T1
Mul     _T0, _T1
```

```
Push    _T0
Push    4
Push    4
Pop     _T0
Pop     _T1
Mul     _T0, _T1
Push    _T0
Pop     _T0
Pop     _T1
Add     _T0, _T1
Push    _T0
```

Quite a bit of code for such a simple statement, eh? Unfortunately, such is the nature of a non-optimizing compiler. Fortunately, the code it does emit is quite easy to read, allowing you to follow its output easily. As you can see, the 2s are pushed, popped and multiplied, followed by the 4s. At this point, `2 * 2` and `4 * 4` reside on the stack in the top and second-to-the-top positions, at which point they're popped into the temporary registers, added together, and pushed back in the form of the sum. As expected, the result of this expression lies on the top of the stack, ready for use by a larger piece of code.

Parsing Full Expressions

This section completes the expression parser you started in the last section, resulting in a new version of the parser with the following features:

- Integer, floating-point, and string literal values.
- The full set of arithmetic and bitwise operators with parenthetic nesting.
- Logical and relational operators.
- The built-in `TRUE` and `FALSE` constants.
- Variable and array references, as well as function calls.
- Unary negation, plus, logical not, and bitwise not.

Parsing expressions that support the full set of XtremeScript operators isn't a trivial task, but a lot of it builds on what you learned in the last section. Let's step through the major changes and additions, step by step.

New Factor Types

The current set of factors supported by the parser is somewhat lacking; it takes more than integers, floats, and variables to get the job done in a real-world scripting project. Fortunately, expanding the `ParseFactor ()` function is perhaps the easiest way to expand the parser, because factors lie at the bottom of the expression entity hierarchy and therefore don't require any further parsing. All you need to do is determine the factor type's value, and push it onto the stack.

The new factor types are: string literal values, function calls, and the `TRUE` and `FALSE` constants that are directly supported by the XtremeScript language. Let's start by looking at the new code that's been inserted into `ParseFactor ()`'s main `switch` block:

```
// It's a true or false constant, so push either 0 and 1 onto the stack
case TOKEN_TYPE_RSRVD_TRUE:
case TOKEN_TYPE_RSRVD_FALSE:
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex,
        GetCurrToken () == TOKEN_TYPE_RSRVD_TRUE ? 1 : 0 );
    break;

// It's a string literal, so add it to the string table and push the resulting
// string index onto the stack
case TOKEN_TYPE_STRING:
{
    int iStringIndex = AddString ( & g_StringTable, GetCurrLexeme () );
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddStringICodeOp ( g_iCurrScope, iInstrIndex, iStringIndex );
    break;
}

// It's an identifier
case TOKEN_TYPE_IDENT:
{
    // First find out if the identifier is a variable or array
    SymbolNode * pSymbol = GetSymbolByIdent ( GetCurrLexeme (), g_iCurrScope );
    if ( pSymbol )
    {
        // Does an array index follow the identifier?
        if ( GetLookAheadChar () == '[' )
        {
            // Ensure the variable is an array
            if ( pSymbol->iSize == 1 )
                ExitOnCodeError ( "Invalid array" );

            // Verify the opening brace
            ReadToken ( TOKEN_TYPE_DELIM_OPEN_BRACE );
            // Make sure an expression is present
            if ( GetLookAheadChar () == ']' )
                ExitOnCodeError ( "Invalid expression" );

             // Parse the index as an expression recursively
            ParseExpr ();

            // Make sure the index is closed
            ReadToken ( TOKEN_TYPE_DELIM_CLOSE_BRACE );

            // Pop the resulting value into _T0 and use it as the index
            // variable
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
            AddVarICodeOp ( g_iCurrScope, iInstrIndex,
                g_iTempVar0SymbolIndex );

            // Push the original identifier onto the stack as an array, indexed
            // with _T0
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
            AddArrayIndexVarICodeOp ( g_iCurrScope, iInstrIndex,
                pSymbol->iIndex, g_iTempVar0SymbolIndex );
        }
        else
        {
            // If not, make sure the identifier is not an array, and push it
            // onto the stack
            if ( pSymbol->iSize == 1 )
            {
                iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
                AddVarICodeOp ( g_iCurrScope, iInstrIndex, pSymbol->iIndex );
            }
            else
            {
                ExitOnCodeError ( "Arrays must be indexed" );
            }
        }
    }
    else
    {
        // The identifier wasn't a variable or array, so find out if it's a
        // function
        if ( GetFuncByName ( GetCurrLexeme () ) )
        {
            // It is, so parse the call
            ParseFuncCall ();

            // Push the return value
            iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
            AddRegICodeOp ( g_iCurrScope, iInstrIndex, REG_CODE_RETVAL );
        }
    }

    break;
}
```

The `TRUE` and `FALSE` comments are the first newcomers, and are handled easily thanks to the lexer's capability to directly return the `TOKEN_TYPE_RSRVD_TRUE` and `TOKEN_TYPE_RSRVD_FALSE` tokens. Because these constants directly correspond to one and zero, the values are immediately converted as they're parsed, and the proper integer value is pushed onto the stack.

Strings are the next addition, and are rather easy to parse. The lexer directly returns strings and automatically trims their double quotes, so all you have to do is add the string to the table and push it onto the stack.

The real changes are in the `TOKEN_TYPE_IDENT` clause. If the identifier doesn't turn out to be an integer, the parser concludes that it must be a function name and attempts to call it with a new function called `ParseFuncCall ()`. You'll see how this function is implemented in just a moment, but for now, all you need to know is that it fully parses function calls and stores the value in `_RetVal` (not on the stack, like other parse functions have thus far). That's why the call is followed by code for pushing `_RetVal` onto the stack.

Parsing Function Calls

Function calls are parsed in a manner somewhat similar to function declarations, with the major difference being that each parameter is treated as an expression, rather than a solitary identifier. Because of this, the parsing process is fairly straightforward, and is contained in a function called `ParseFuncCall ()` that you saw in the last section. Here's the code:

```
void ParseFuncCall ()
{
    // Get the function by its identifier
    FuncNode * pFunc = GetFuncByName ( GetCurrLexeme () );
    // It is, so start the parameter count at zero
    int iParamCount = 0;

    // Attempt to read the opening parenthesis
    ReadToken ( TOKEN_TYPE_DELIM_OPEN_PAREN );

    // Parse each parameter and push it onto the stack
    while ( TRUE )
    {
        // Find out if there's another parameter to push
        if ( GetLookAheadChar () != ')' )
        {
            // There is, so parse it as an expression
            ParseExpr ();

            // Increment the parameter count and make sure it's not
            // greater than the amount accepted by the function (unless it's
            // a host API function
            ++ iParamCount;
            if ( ! pFunc->iIsHostAPI && iParamCount > pFunc->iParamCount )
                ExitOnCodeError ( "Too many parameters" );

            // Unless this is the final parameter, attempt to read a comma
            if ( GetLookAheadChar () != ')' )
                ReadToken ( TOKEN_TYPE_DELIM_COMMA );
        }
        else
        {
            // There isn't, so break the loop and complete the call
            break;
        }
    }

    // Attempt to read the closing parenthesis
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );

    // Make sure the parameter wasn't passed too few parameters (unless
    // it's a host API function)
    if ( ! pFunc->iIsHostAPI && iParamCount < pFunc->iParamCount )
        ExitOnCodeError ( "Too few parameters" );
    // Call the function, but make sure the right call instruction is used
    int iCallInstr = INSTR_CALL;
    if ( pFunc->iIsHostAPI )
        iCallInstr = INSTR_CALLHOST;

    int iInstrIndex = AddICodeInstr ( g_iCurrScope, iCallInstr );
    AddFuncICodeOp ( g_iCurrScope, iInstrIndex, pFunc->iIndex );
}
```

In a nutshell, the logic simply scans through each parameter and calls `ParseExpr ()` to parse it. It also continually checks the current number of parameters parsed in order to make sure that more parameters than the function accepts aren't found. When it's done, it compares the two values again to make sure that the function wasn't passed too few parameters, either. The function finishes by inserting a `Call` instruction to complete the process.

New Unary Operators

Rounding out the additions to `ParseFactor ()` are the new unary operators. In addition to the unary negation - operator of the last expression parser, the new version includes both logical and bitwise not. Bitwise not is a snap to implement—the code is the same as negation, except you use the `Not` instruction instead of `Neg`. The real challenge is adding *logical* not. The reason for this is actually self-explanatory; because a "logical not" involves actual logic, you need to add jumps and labels to route the flow of execution to the right place based on the value of the factor.

To implement this operator, the parser uses `GetNextJumpTargetIndex ()` (a function described in the last chapter) to get the next two jump target indexes. Using these indexes, a small system of jumps is set up that will cause the script to push zero onto the stack if the factor is nonzero, and one if it isn't. Simply put, the following example line of XtremeScript:

```
! X;    // Logical not X
```

should be compiled down to:

```
    Push    X
    Pop     _T0, X
    JE      _T0, 0, True
    Push    0
    Jmp     Exit
True:
    Push    1
Exit:
```

What you've seen here will play a large role in the logical operators you'll develop in the next section, so make sure you understand what's going on. Just to reiterate, the idea here is to generate code that implements the logic behind the operator. In this case, because you want to push the logical not of the factor onto the stack, you want to push zero when the factor is nonzero, and one otherwise. Think of it as a "logical opposite".

New Binary Operators

XtremeScript's binary operator set is also filled out in the new expression parser. The remaining arithmetic operators like negation and exponentiation are added, as well as the full array of bitwise operators. Fortunately, the new code isn't really new at all. Because XVM assembly offers such a rich assortment of binary operation instructions, every XtremeScript operator maps directly to one such instruction. You're already doing this with the basic arithmetic supported in the last parser, so the new operators are simply a rehash of the logic behind the old ones. Rather than waste the paper space here with redundant information, you can see the additional operators for yourself in the source code on the companion CD. Check out this second version of the expression parser in the `Programs/Chapter 15/15_03/` folder.

Logical and Relational Operators

Logical and relational operators are a definite departure from the implementation of the binary operators you've seen so far. Just like the logical not unary operator I recently covered, logical and relational operators require actual *logic* to be inserted into the compiled assembly script in order to push the proper values onto the stack. The key to remember is that all logical and relational operators produce one of two values and two values only—true and false, or, more specifically, one and zero.

This is why actual logic must be coded into the executable script. Because there's no purely mathematical way to filter all input values into either true or false (at least, not a particularly convenient one), you have to use conditional logic based on labels and jumps to allow the flow of the script itself to do it for you.

NOTE

Before going any farther, it's important to note that for simplicity's sake, I'm compressing XtremeScript's operator precedence levels into four tiers. There's the level of lowest precedence, wherein relational and logical operators reside. Right above them are the addition, subtraction, and string concatenation operators. Up next are the remaining binary operators. Finally, the unary operators maintain the highest precedence. Although this does mean that certain C and C++ operator practices can't be reliably translated to XtremeScript, everything will still work out fine as long as you use parentheses to manually resolve any ambiguities.

The Logical And Operator

As an example of a logical operator, let's look at logical and. Due to the compression of the XtremeScript operator precedence levels, you're going to handle this operator in `ParseExpr ()`, where the lowest-precedence level operators are handled. Here's the code for converting a binary and operator expression into assembly:

```
case OP_TYPE_LOGICAL_AND:
{
    // Get a pair of free jump target indexes
    int iFalseJumpTargetIndex = GetNextJumpTargetIndex (),
        iExitJumpTargetIndex = GetNextJumpTargetIndex ();

    // JE _T0, 0, False
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JE );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iFalseJumpTargetIndex );

    // JE _T1, 0, False
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JE );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iFalseJumpTargetIndex );

    // Push 1
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex, 1 );

    // Jmp Exit
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iExitJumpTargetIndex );

    // L0: (False)
    AddICodeJumpTarget ( g_iCurrScope, iFalseJumpTargetIndex );

    // Push 0
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
    // L1: (Exit)
    AddICodeJumpTarget ( g_iCurrScope, iExitJumpTargetIndex );

    break;
}
```

The basic logic here is as follows. Given an example line of XtremeScript like the following:

```
X && Y;    // Logical X and Y
```

Assembly code should be generated that adheres to the following format:

```
    JE    _T0, 0, False
    JE    _T1, 0, False
    Push  1
    Jmp   Exit
True:
    Push   0
Exit:
```

Simply put, if either operand is zero, the overall operation must be false. Otherwise, it's true.

Relational Greater Than or Equal

Moving right along, let's check in on the relational operators and see how >= works. First off, here's the code:

```
// Pop the first operand into _T1
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );

// Pop the second operand into _T0
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

// Get a pair of free jump target indexes
int iTrueJumpTargetIndex = GetNextJumpTargetIndex (),
    iExitJumpTargetIndex = GetNextJumpTargetIndex ();

// Generate a JGE instruction
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JGE );
```

```
// Add the jump instruction's operands (_T0 and _T1)
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );
AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iTrueJumpTargetIndex );

// Generate the outcome for falsehood
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );

// Generate a jump past the true outcome
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iExitJumpTargetIndex );

// Set the jump target for the true outcome
AddICodeJumpTarget ( g_iCurrScope, iTrueJumpTargetIndex );

// Generate the outcome for truth
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_PUSH );
AddIntICodeOp ( g_iCurrScope, iInstrIndex, 1 );

// Set the jump target for exiting the operand evaluation
AddICodeJumpTarget ( g_iCurrScope, iExitJumpTargetIndex );
```

Once again, you start off by getting two free jump target indexes. One is jumped to in the case of a true outcome, and the other marks the end of the operator's assembly representation. Now, assuming that both operand expressions have been parsed and pushed onto the stack, the operands are popped into the temporary registers. These registers are then used as the criteria for a `JGE` instruction, which jumps to the true label if the first operand is greater or equal to the second, which in turn pushes 1 onto the stack. It jumps to the exit operand otherwise, but not before pushing zero onto the stack.

In short, the following XtremeScript expression:

```
X >= Y;    // Is X greater than or equal to Y?
```

should become the following XVM assembly fragment:

```
    Push    X
    Push    Y
    Pop     _T1
    Pop     _T0
    JGE     _T0, _T1, True
    Push    0
    Jmp     Exit
True:
    Push    1
Exit:
```

Presto!

The Rest

You've seen an example of logical operators, and one of the relationals. Once you understand how these two work, you're definitely prepared to understand the rest. Again, to save space in an already rather large chapter, I've omitted the remaining operators in print and instead encourage you to check them out yourself in the source code on the companion CD.

L-Values and R-Values

You will read about the assignment statement in an upcoming section, but before you get there, let's briefly discuss the concept of *L-values* and *R-values*. The *L* and *R* in the terms refer to left and right, and thus correspond to which side of the assignment operator a value is found. In order for a value to be a valid L-value, it must not be a constant. Because it's impossible to "assign a value" to the number five, for example, it's vital that all values on the left side of the operator, meaning, the values that are being "assigned," are variables and can thus be altered. R-values, on the other hand, can be virtually anything, because the value itself is all you're worried about in their case. Figure 15.26 shows the syntax diagram for an XtremeScript L-value.

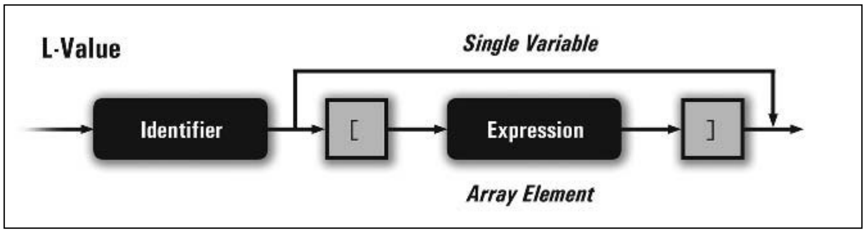


Figure 15.26

The syntax diagram for an L-value.

A Standalone Runtime Environment

So far, despite the considerable capabilities you've built into the evolving parser module, you haven't yet generated anything that's visibly executable. Sure, you could call a host API function that prints a string from within an expression, but you haven't really dealt with any code that's truly "alive". Without looping, branching, or even just the capability to assign values and expressions to variables, the code really doesn't do much yet.

The next section will change all that, with the implementation of all the missing features I mentioned previously. Before you go ahead and do that, however, it's important to note that you don't have a particularly convenient or readily available venue for testing the output of the compiler. Although the XVM is indeed finished, it's not much good without a host application to support it. What you need is a standalone runtime environment that will execute the code quickly and easily and provide just enough output functionality to let you watch the scripts as they execute.

Fortunately, this will be easy to set up. All you need to do is create a simple program that "wraps" the virtual machine. By exposing a basic, bare-bones host API that gives you just enough power to output text to a console, you can write scripts of all kinds and watch them run. The effort and attention to detail you put into the development of the XVM and its integration interface is about to pay off—as you'll soon see, creating this standalone VM will be trivial at best. The next two sections will cover the development of this program, but you can check it out now if you're interested on the accompanying CD in the `Programs/Chapter 15/XVM Console/` directory.

The Host Application

All you really need is a simple command-line program that can load a single script into memory, execute it until a key is pressed, and provide that script with a simple console output API so it can display text as it runs.

As I'm sure you'd agree, there's nothing particularly daunting about writing this program. In fact, all you need is a single `main ()` function for its entire core logic. The real work goes on within the XVM, which is of course already finished and ready to go. You'll create the following host application program in the source file `console.cpp`. Figure 15.27 depicts the file layout of the XVM console.

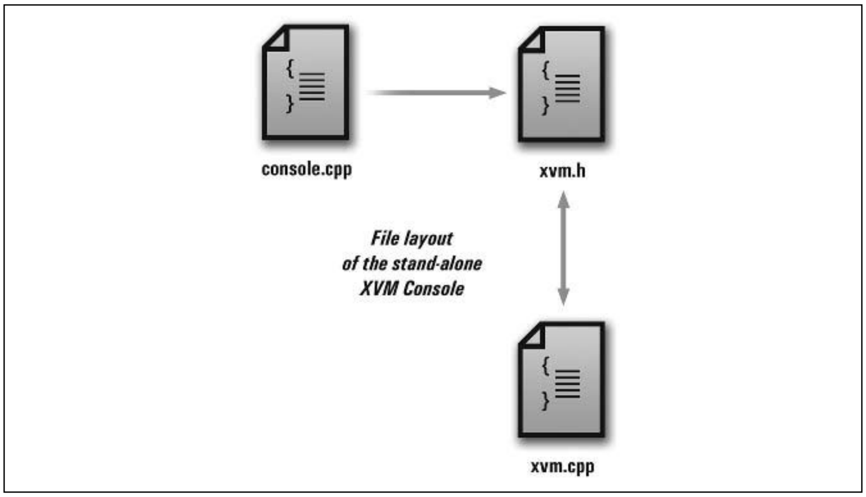


Figure 15.27

The file layout of the XVM console.

Of course, in order to use the XVM, you'll need to link `console.cpp` with `xvm.cpp` and include `xvm.h`. This is done easily in Visual C++ by simply loading both `console.cpp` and `xvm.cpp` into the same Console Application project. Both the project and workspace files for accomplishing this are located in the `Programs/Chapter 15/XVM Console/Source/` directory.

The rest of this section outlines the decidedly simple process of building the standalone runtime environment. Everything here will be a cake walk, so feel free to skim it if you'd just like to get back to the development of the parser module. Just make sure you're familiar with how it works, because you'll be using the finished product for the rest of the chapter.

Reading the Command Line

This runtime environment will be simple and concise, but there's no need to make it crude. Because of this, it allows the user to input the script filename through the command line, and prints usage information in the event that a filename was not found. This all takes place in the first segment of the program's `main ()` function:

```
main ( int argc, char * argv [] )
{
   // Make sure a filename was passed
   if ( argc < 2 )
   {
      // Print the logo and usage info
      printf ( "XVM Console\n" );
      printf ( "Stand-Alone Console-Based Runtime Environment\n" );
      printf ( "Written by Alex Varanese\n" );
      printf ( "\n" );
      printf ( "Usage:\tXVMCONSOLE Script.XSE\n" );
      printf ( "\n" );
      printf ( "Notes:\n" );
      printf ( "\t- A file extension is required.\n" );
      printf ( "\t- Scripts without a _Main () function will not
                 execute.\n" );
      printf ( "\n" );

      // Exit the program
      return 0;
   }
```

Loading the Script

Once you know a filename is present on the command line, you can start the XVM with a call to `XS_Init ()` and load the script. Remember, it's important to save the script's index, and for the sake of completeness, you need to check for load errors as well:

```
// Initialize the runtime environment
XS_Init ();

// Declare the thread indexes
int iThreadIndex;

// An error code
int iErrorCode;

// Load the specified script
iErrorCode = XS_LoadScript ( argv [ 1 ], iThreadIndex,
    XS_THREAD_PRIORITY_USER );

// Check for an error
if ( iErrorCode != XS_LOAD_OK )
{
   // Print the error based on the code
   printf ( "Error: " );
   switch ( iErrorCode )
   {
      case XS_LOAD_ERROR_FILE_IO:
         printf ( "File I/O error" );
         break;
      case XS_LOAD_ERROR_INVALID_XSE:
         printf ( "Invalid .XSE file" );
         break;
      case XS_LOAD_ERROR_UNSUPPORTED_VERS:
         printf ( "Unsupported .XSE version" );
         break;
      case XS_LOAD_ERROR_OUT_OF_MEMORY:
         printf ( "Out of memory" );
         break;
      case XS_LOAD_ERROR_OUT_OF_THREADS:
         printf ( "Out of threads" );
         break;
   }
   printf ( ".\n" );
   return 0;
}
```

Running the Script

Once the thread is in memory, it's time to run it. The script is initially started with a call to `XS_StartScript ()`, and kept in motion with repeated calls to `XS_RunScripts ()`. You call `XS_RunScripts ()` repeatedly in a `while` loop that runs until a key is pressed. This way, any scripts that involve infinite loops (intentionally or otherwise) can be kept under control by the user but left to run as long as desired. Once you're done, you make a single call to `XS_ShutDown ()`, and everything packs up and goes home. Here's the final block of the core application:

```
    // Start up the script
    XS_StartScript ( iThreadIndex );

    // Run the script until a key is pressed
    while ( ! kbhit () )
        XS_RunScripts ( 200 );

    // Free resources and perform general cleanup
    XS_ShutDown ();

    return 0;
}
```

The Host API

So you can load programs into memory and run them until a key is pressed, but they still can't talk to you. To do this, you need a function for printing text strings. Unfortunately, the XASM assembler only understands escape sequences for double-quotes, and because `printf ()` expects newlines and tabs to appear as `\n` and `\t`, you can't directly print such characters with a general string printing function. You therefore need to write two others for doing exactly that. Overall, this means you need three functions: `PrintString ()` for printing strings, and `PrintNewline ()` and `PrintTab ()` for printing their respective control characters.

PrintString ()

As mentioned previously, you'll wrap `printf ()` to do the printing:

```
void HAPI_PrintString ( int iThreadIndex )
{
    // Read in the parameters
    char * pstrString = XS_GetParamAsString ( iThreadIndex, 0 );

    // Print the string
    printf ( "%s", pstrString );

    // Return to the XVM
    XS_Return ( iThreadIndex, 1 );
}
```

This simple function operates in three steps. First, it reads a single string parameter with `XS_GetParamAsString ()`, which it then prints with `printf ()`. Lastly, it uses the `XS_Return ()` macro to terminate the function. Remember, the function itself has to clean up the parameters on the stack, so you pass 1 to the macro to tell it that the function takes one parameter.

NOTE

Remember, it's good practice to prefix host API functions with `HAPI_` or some other descriptive tag so you can prevent name clashes with the rest of your program. And if nothing else, it just enhances readability.

PrintNewline () and PrintTab ()

The last two functions are even simpler. Because these don't accept any parameters, they're practically empty:

```
void HAPI_PrintNewline ( int iThreadIndex )
{
   // Print the newline
   printf ( "\n" );

   // Return to the XVM
   XS_Return ( iThreadIndex, 0 );
}

void HAPI_PrintTab ( int iThreadIndex )
{
   // Print the tab
   printf ( "\t" );
```

```
    // Return to the XVM
    XS_Return ( iThreadIndex, 0 );
}
```

Remember, however, that even without parameters it's vital to return from the function with XS_Return (). Forgetting to do so will lead to a corrupted stack and most likely crash the machine.

Registering the API

The last step is of course to register the three functions you just created. As you should remember from Chapter 11, this is done with the XS_RegisterHostAPIFunc () function; you pass it the function pointer, the desired function name, and the scope among the currently active threads, and it will add the function to its internal host API table:

```
// Register the console output API
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "PrintString", HAPI_PrintString );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "PrintNewline", HAPI_PrintNewline );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "PrintTab", HAPI_PrintTab );
```

I made the functions global with the XS_GLOBAL_FUNC flag, but this was an arbitrary decision. I could've passed it the script's thread index instead, and the result would've been the same. A host API function's scope really doesn't matter when there's only one thread running.

That takes care of it—you've created a simple, but complete, runtime environment that's ready to use. In the coming sections, as you add increasingly sophisticated features to the parser module, you'll be able to use this program to get immediate feedback. Once again, for future reference, the XVM console is located on the companion CD under Programs/Chapter 15/XVM Console/.

Parsing Advanced Statements and Constructs

With expressions out of the way, along with the basic stuff like code blocks, statements, and declarations, you're ready to deliver the coup de grace and knock the parser out once and for all. This final section will actually be surprisingly straightforward, at least for the most part, when compared to the complexities of full expression parsing.

You're going to round out the language implementation here by adding loops, branching, and assignment statements. Remember, because assignments, loops, and branching constructs all require either an arithmetic, logical, or relational expression to function properly (or a combination of the three), you had to make sure the parser is capable of understanding them first.

Assignment Statements

I intentionally decided not to support C/C++-style assignments, as they can appear anywhere in an expression and often lead to confusion. Rather, you're taking a simpler route and making assignments their own specific type of statement. This lends itself to a cleaner language that's easier to parse. Of course, you'll still support the full range of assignment shorthand operators supported by languages like C and C++, such as += and `&=`.

The syntax of an assignment is quite simple. It's really just an identifier of some sort, be it a variable or array, followed by one of the XtremeScript assignment operators, followed by an expression. The variable or array on the left-hand side of the assignment operator is the *L-value*, and the expression on the right-hand side is the *R-value*. Although an R-value can be virtually anything, an L-value must always be either a variable or array; obviously it doesn't make sense to "assign" one literal value to another.

Figure 15.28 depicts the assignment statement's syntax diagram.

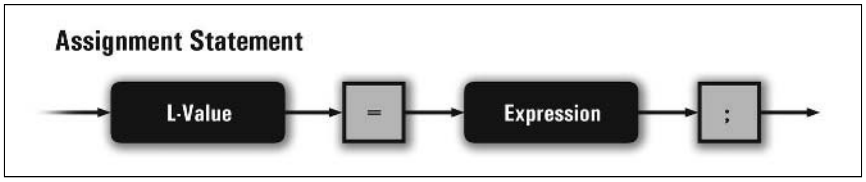


Figure 15.28

The syntax diagram for an assignment statement.

The parsing strategy for such a diagram is clearly simple. The L-value is parsed using the same logic used to parse variables and arrays in the last sections. The assignment operator is then read and verified, and finally, the expression is parsed using the now-complete expression-parsing functions.

The assembly representation of an assignment is even simpler. It really just boils down to code that evaluates the expression and pushes it onto the stack, followed by another piece of code that pops it off the stack and copies it into the destination. If the = operator is used, the `Mov` instruction can implement it in assembly. If += is used, `Add` performs the assignment, and so on.

As always, the first step in implementing a new statement type is adding a new `case` to `ParseStatement ()`. Determining whether a specific token is the initial token of an assignment is a bit more involved than usual, however:

```
// Assignment
case TOKEN_TYPE_IDENT:
{
    // What kind of identifier is it?
    if ( GetSymbolByIdent ( GetCurrLexeme (), g_iCurrScope ) )
```

```
        {
            // It's an identifier, so treat the statement as an assignment
            ParseAssign ();
        }
        else
        {
            // It's invalid
            ExitOnCodeError ( "Invalid identifier" );
        }
        break;
    }
```

Once again the Statement syntax diagram grows, as shown in Figure 15.29.

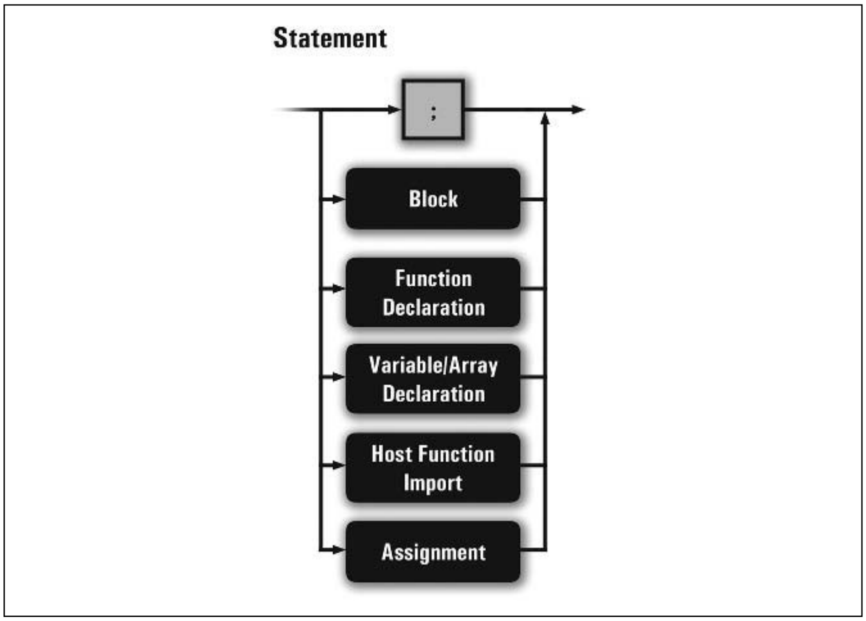


Figure 15.29

The syntax diagram for Statements with assignments added.

If an identifier is read, you can assume you're dealing with an assignment expression. It's important to verify that it's a variable or array first, however, so you call `GetSymbolByIdent ()` and make sure the pointer it returns isn't null. `ParseAssign ()` is then called to handle the parsing process, which I'll cover momentarily. If the identifier isn't found, an invalid identifier error is flagged. Let's continue by breaking `ParseAssign ()` down, piece by piece:

```
void ParseAssign ()
{
    // Make sure we're inside a function
    if ( g_iCurrScope == SCOPE_GLOBAL )
        ExitOnCodeError ( "Assignment illegal in global scope" );

    int iInstrIndex;

    // Assignment operator
    int iAssignOp;

    // Annotate the line
    AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

    // ---- Parse the variable or array

    SymbolNode * pSymbol = GetSymbolByIdent ( GetCurrLexeme (), g_iCurrScope );

    // Does an array index follow the identifier?
    int iIsArray = FALSE;
    if ( GetLookAheadChar () == '[' )
    {
        // Ensure the variable is an array
        if ( pSymbol->iSize == 1 )
            ExitOnCodeError ( "Invalid array" );

        // Verify the opening brace
        ReadToken ( TOKEN_TYPE_DELIM_OPEN_BRACE );

        // Make sure an expression is present
        if ( GetLookAheadChar () == ']' )
            ExitOnCodeError ( "Invalid expression" );

        // Parse the index as an expression
        ParseExpr ();

        // Make sure the index is closed
        ReadToken ( TOKEN_TYPE_DELIM_CLOSE_BRACE );

        // Set the array flag
        iIsArray = TRUE;
    }
    else
    {
        // Make sure the variable isn't an array
        if ( pSymbol->iSize > 1 )
           ExitOnCodeError ( "Arrays must be indexed" );
    }
```

The function begins by making sure the current scope isn't global, and declaring a few variables. `iInstrIndex` will be used when generating the statement's I-code to keep track of the current instruction node. `iAssignOp` will also be used later to keep track of which particular assignment operator was found.

The line is then annotated, and a pointer to the symbol corresponding to the identifier is stored locally with `GetSymbolByIdent ()`. The L-value may be parsed at this point, but as always, there's the issue of array notation. To find out whether the symbol is actually an array, you use the look-ahead to determine whether an opening brace token appears to be next. In the meantime, the `iIsArray` flag is declared and set to `FALSE`.

If so, you first ensure that the variable in question is indeed an array by comparing `pSymbol->Size` to 1. If so, an invalid array error is flagged. Otherwise, you continue by verifying that the opening brace token is valid, and once again using the look-ahead to make sure a closing brace doesn't immediately follow. If it did, it would mean that the expression had been omitted, like this:

```
MyArray [] = 256;
```

which obviously doesn't make sense. `ParseExpr ()` is then called to parse the expression between the braces, and `ReadToken ()` is used to ensure that the closing brace follows. At this point, it's a pretty safe bet that you're dealing with an array, so the `iIsArray` flag is set to `TRUE`.

Even if the look-ahead character doesn't reveal an opening brace, however, you still can't be sure that you're done processing the L-value. In the absence of a brace, you assume you're not dealing with an array. To make sure this is correct, you once again compare `pSymbol->iSize` to 1. If it's greater, an array has been used as the L-value without specifying an index. This results in the flagging of another error.

TIP

Perhaps an easier way to determine whether an array should be parsed is to immediately consult `pSymbol->iSize`, rather than reading the look-ahead. This way, you can take action depending on what the L-value *should* be, rather than what it appears to be.

The L-value is taken care of, so you can move on to parsing the assignment operator itself. Because XtremeScript supports more than just the = operator for assignment, there are a number of possibilities here:

```
// ---- Parse the assignment operator
if ( GetNextToken () != TOKEN_TYPE_OP &&
     ( GetCurrOp () != OP_TYPE_ASSIGN &&
       GetCurrOp () != OP_TYPE_ASSIGN_ADD &&
       GetCurrOp () != OP_TYPE_ASSIGN_SUB &&
       GetCurrOp () != OP_TYPE_ASSIGN_MUL &&
       GetCurrOp () != OP_TYPE_ASSIGN_DIV &&
       GetCurrOp () != OP_TYPE_ASSIGN_MOD &&
       GetCurrOp () != OP_TYPE_ASSIGN_EXP &&
       GetCurrOp () != OP_TYPE_ASSIGN_CONCAT &&
       GetCurrOp () != OP_TYPE_ASSIGN_AND &&
       GetCurrOp () != OP_TYPE_ASSIGN_OR &&
       GetCurrOp () != OP_TYPE_ASSIGN_XOR &&
       GetCurrOp () != OP_TYPE_ASSIGN_SHIFT_LEFT &&
       GetCurrOp () != OP_TYPE_ASSIGN_SHIFT_RIGHT ) )
     ExitOnCodeError ( "Illegal assignment operator" );
else
    iAssignOp = GetCurrOp ();
```

Once you know you have a valid operator, you call `GetCurrOp ()` to save it in `iAssignOp`. This allows you to generate the proper assignment instruction later. The value expression and semicolon are parsed next:

```
// ---- Parse the value expression

ParseExpr ();

// Validate the presence of the semicolon
ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
```

The last step is generating the I-code for the assignment. At this point, the item on the top of the stack is the result of the value expression, and if the L-value was an array, the index value is directly under it. You can therefore pop the value expression into `_T0` and the array index (if present) into `_T1`. From there, you just need to emit the code to assign the value and you're done:

```
// Pop the value into _T0
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

// If the variable was an array, pop the top of the stack into _T1 for use as
// the index
```

```
if ( iIsArray )
{
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar1SymbolIndex );
}

// ---- Generate the I-code for the assignment instruction

switch ( iAssignOp )
{
    // =
    case OP_TYPE_ASSIGN:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_MOV );
        break;
    // +=
    case OP_TYPE_ASSIGN_ADD:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_ADD );
        break;
    // -=
    case OP_TYPE_ASSIGN_SUB:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_SUB );
        break;
    // *=
    case OP_TYPE_ASSIGN_MUL:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_MUL );
        break;
    // /=
    case OP_TYPE_ASSIGN_DIV:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_DIV );
        break;
    // %=
    case OP_TYPE_ASSIGN_MOD:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_MOD );
        break;
    // ^=
    case OP_TYPE_ASSIGN_EXP:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_EXP );
        break;
    // $=
    case OP_TYPE_ASSIGN_CONCAT:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_CONCAT );
        break;
    // &=
    case OP_TYPE_ASSIGN_AND:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_AND );
        break;
    // |=
    case OP_TYPE_ASSIGN_OR:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_OR );
        break;
    // #=
    case OP_TYPE_ASSIGN_XOR:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_XOR );
        break;
    // <<=
    case OP_TYPE_ASSIGN_SHIFT_LEFT:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_SHL );
        break;
    // >>=
    case OP_TYPE_ASSIGN_SHIFT_RIGHT:
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_SHR );
        break;
}

// Generate the destination operand
if ( iIsArray )
    AddArrayIndexVarICodeOp ( g_iCurrScope, iInstrIndex, pSymbol->iIndex,
        g_iTempVar1SymbolIndex );
else
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, pSymbol->iIndex );

// Generate the source
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
```

With the result of the expression in `_T0` and the array index in `_T1`, you're ready to generate the assignment code. The first step is generating the proper instruction, which corresponds directly with the operator that was used. = results in a `Mov`, += results in `Add`, -= results in `Sub`, and so on. Because you stored the operator in `iAssignOp`, you can easily make this determination using a `switch` block.

The next step is generating the proper destination operand. Once again, the `iIsArray` flag is checked to determine whether to generate code for a variable or code for an array. If the flag is clear, `AddVarICodeOp ()` is called with `pSymbol->iIndex` to generate a variable operand. Otherwise, AddArrayIndexVarICodeOp () is called to generate an array indexed with a variable. This function is passed pSymbol->iIndex as well as g_iTempVar1SymbolIndex.

Finally, you generate the source operand, which is just _T0. As an example, check out the following fragment of XtremeScript code:

```
var MyArray [ 4 ];
var Radius;

Radius = 4;
MyArray [ 1 ] = 3.14159 * Radius ^ 2;
```

When compiled, XSC produces this:

```
Var MyArray [ 4 ]
Var Radius

;     Radius = 4;
Push        4
Pop         _T0
Mov         Radius, _T0

;     MyArray [ 1 ] = 3.14159 * Radius ^ 2;
Push        1
Push        3.141590
Push        Radius
Pop         _T1
Pop         _T0
Mul         _T0, _T1
Push        _T0
Push        2
Pop         _T1
Pop         _T0
Exp         _T0, _T1
Push        _T0
Pop         _T0
Pop         _T1
Mov         MyArray [ _T1 ], _T0
```

First the variables are declared, and then Radius is assigned 4, and finally, MyArray [1] is assigned the result of the expression. As you can see, the expression ends with the result being popped into _T0, and the array index being popped into _T1.

Function Calls

Even though you've already written logic to call functions from within an expression, you still need to support statements that are themselves single function calls. Fortunately, this is extremely simple. The `ParseFuncCall ()` function you wrote for the expression parser already encapsulates virtually all of the logic you need. All you need to do is update `ParseStatement ()` a bit, and you can leverage the existing code to do the job (as shown in Figure 15.30).

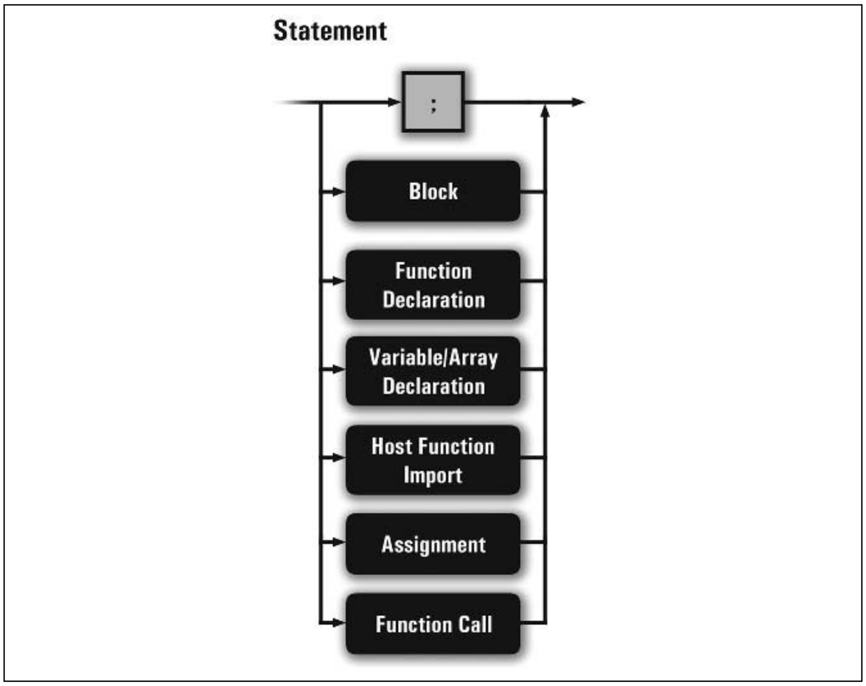


Figure 15.30

The syntax diagram for Statements with function calls added.

In fact, because the initial token in a function call is the function's name, you can add it to the `TOKEN_TYPE_IDENT case` you created in the last section for handling variables and arrays in assignment statements. In the event that the identifier isn't found in the symbol table, you can look for it in the function table and treat it like a function call. From there, all you have to do is annotate the source line, call `ParseFuncCall ()`, and verify the trailing semicolon. You don't even need to worry about return values.

Here's the updated identifier `case` in `ParseStatement ()`:

```
case TOKEN_TYPE_IDENT:
{
    // What kind of identifier is it?
    if ( GetSymbolByIdent ( GetCurrLexeme (), g_iCurrScope ) )
```

```
        {
            // It's an identifier, so treat the statement as an assignment
            ParseAssign ();
        }
        else if ( GetFuncByName ( GetCurrLexeme () ) )
        {
            // It's a function

            // Annotate the line and parse the call
            AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );
            ParseFuncCall ();

            // Verify the presence of the semicolon
            ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );
        }
        else
        {
            // It's invalid
            ExitOnCodeError ( "Invalid identifier" );
        }

        break;
    }
```

And just like that, you can make function calls in the form of statements. For example, take a look at this script fragment:

```
host PrintString ();

func PrintStringWrap ( String )
{
    PrintString ( String );
}

func _Main ()
{
    PrintStringWrap ( "This is a script-defined function." );
    PrintString ( "This is a host API function." );
}
```

The host API function `PrintString ()` is imported, followed by the definition of a script-defined function called `PrintStringWrap ()` that wraps the host API version of the function to print a string as well. Within `_Main ()`, both functions are called via function call statements. Here's an excerpt of the compiled code:

```
;     PrintStringWrap ( "This is a script-defined function." );
Push        "This is a script-defined function."
Call        PrintStringWrap

;     PrintString ( "This is a host API function." );
Push        "This is a host API function."
CallHost    PrintString
```

Cool, huh? The code emitter automatically knows to differentiate between `Call` and `CallHost`.

return

Parsing `return` is understandably simple. In fact, the entire statement consists solely of the `return` keyword, and optional expression, and a semicolon. Check out Figure 15.31.

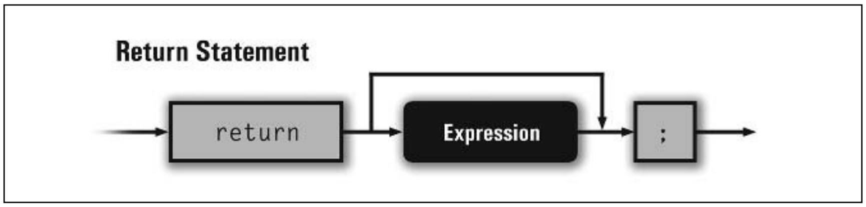


Figure 15.31

The syntax diagram for a `return` statement.

The assembly representation of `return` is extremely simple as well. If an expression is present, its evaluation code is generated first, followed by a `Pop` instruction that pops the result into the `_RetVal` register. The `Ret` instruction is then used to return from the function.

The one caveat is the `_Main ()` function, however. Like in C, a `return` statement in `_Main ()` actually has the effect of terminating the script entirely, because `_Main ()` has no caller to return to. Because of this, you must generate an `Exit` instruction *instead* of `Ret` if the `return` statement is found in the `_Main ()` function. However, in both cases, an expression can be returned; if the function returning is `_Main ()`, the result of the expression is the exit code and is an operand for the `Exit` instruction. If not, it's popped into `_RetVal`, because `Ret` doesn't accept any operands.

Because of this, you need to make a number of checks throughout the function to find out of the current function is `_Main ()` or not (which can be easily done by comparing `g_iCurrScope` to the script header's `_Main ()` index) and act accordingly.

As always, let's start by adding the proper update to `ParseStatement ()`, as shown in the code listing here and in Figure 15.32:

```
// return
case TOKEN_TYPE_RSRVD_RETURN:
    ParseReturn ();
    break;
```

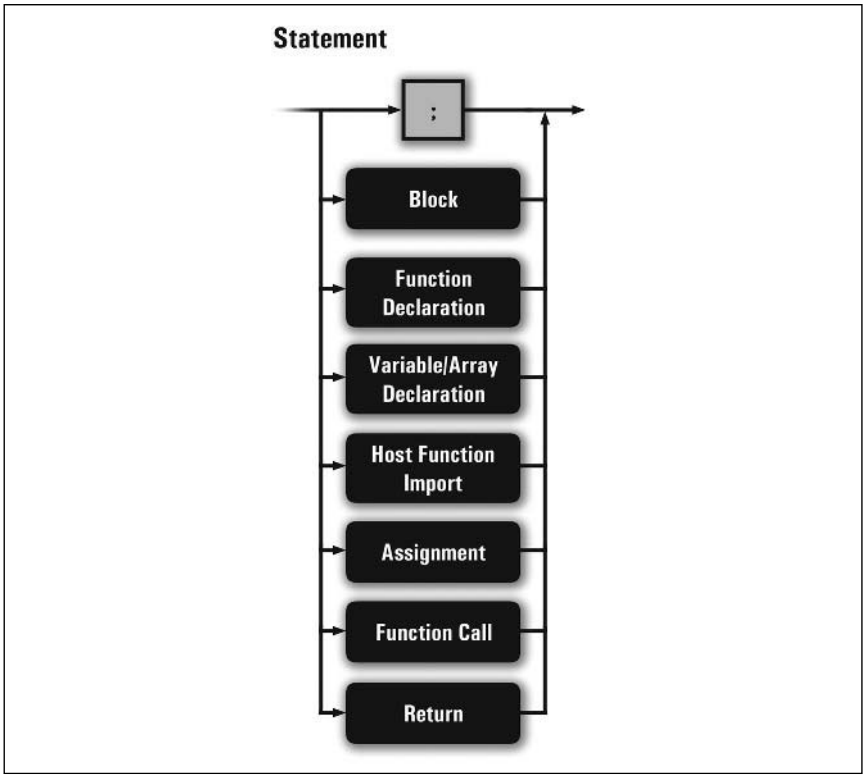


Figure 15.32

The syntax diagram for Statements with `return` *taken into account.*

With that out of the way, let's check out `ParseReturn ()`:

```
void ParseReturn ()
{
    int iInstrIndex;

    // Make sure we're inside a function
    if ( g_iCurrScope == SCOPE_GLOBAL )
        ExitOnCodeError ( "return illegal in global scope" );
```

```
// Annotate the line
AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

// If a semicolon doesn't appear to follow, parse the
// expression and place it in _RetVal
if ( GetLookAheadChar () != ';' )
{
    // Parse the expression to calculate the return value and
    // leave the result on the stack.
    ParseExpr ();

    // Determine which function we're returning from
    if ( g_ScriptHeader.iIsMainFuncPresent &&
         g_ScriptHeader.iMainFuncIndex == g_iCurrScope )
    {
        // It is _Main (), so pop the result into _T0
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex,
            g_iTempVar0SymbolIndex );
    }
    else
    {
        // It's not _Main, so pop the result into the _RetVal register
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
        AddRegICodeOp ( g_iCurrScope, iInstrIndex, REG_CODE_RETVAL );
    }
}
else
{
    // Clear _T0 in case we're exiting _Main ()
    if ( g_ScriptHeader.iIsMainFuncPresent &&
         g_ScriptHeader.iMainFuncIndex == g_iCurrScope )
    {
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_MOV );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex,
            g_iTempVar0SymbolIndex );
        AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
    }
}
```

```
    if ( g_ScriptHeader.iIsMainFuncPresent &&
         g_ScriptHeader.iMainFuncIndex == g_iCurrScope )
    {
        // It's _Main, so exit the script with _T0 as the exit code
        iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_EXIT );
        AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
    }
    else
    {
        // It's not _Main, so return from the function
        AddICodeInstr ( g_iCurrScope, INSTR_RET );
    }
}
```

Of course, because `return` is used to return from functions, it must be found *within* one. If not, an error is flagged alerting the users that `return` is illegal in the global scope. After annotating the source line, `ParseExpr ()` is called to parse the expression, which will leave the result on the stack. If the current function is `_Main ()`, a `Pop` instruction is generated and pops that result into `_RetVal`, allowing you to follow up with a `Ret` instruction to complete the process. Otherwise, the value is popped into `_T0` for use with `Exit`. If an expression didn't follow `return`, and you're currently inside `_Main ()`, `_T0` is cleared to allow the `Exit` instruction's operand to default to zero.

Here's an example of a function that uses `return`:

```
func Square ( X )
{
    return X ^ 2;
}
```

Here's its compiled output:

```
Func Square
{
    Param X

    ;    return X ^ 2;
    Push       X
    Push       2
    Pop        _T1
    Pop        _T0
    Exp        _T0, _T1
    Push       _T0
    Pop         _RetVal
    Ret
}
```

As you can see, the `X ^ 2` expression is emitted first, followed by the `Pop _RetVal` and `Ret` instructions, which is everything you need.

while Loops

Expression parsing and assignment statements represented the line-by-line nature of the language—individual statements that perform specific tasks on their own. With the exception of function calls, which are possible at this point, the parser has no real notion of code blocks that perform a common task or are in some way related. An implementation of the `while` loop will be the first divergence from this trend.

Fortunately, by now, you've developed so many parsing functions that can be so easily "black boxed," that implementing loops and branching will more or less be a matter of snapping together preexisting components to parse more complex structures.

Implementing constructs that are larger than a single statement, such as the `while` loop, is a twofold process. The first step is understanding how the structure itself is parsed, which is more or less trivial. Beyond that, however, is an understanding of how the I-code, and ultimately the resulting assembly language, is *arranged* to represent the structure without the aid of the high-level language. Fortunately, Chapter 8 prepared you for exactly this. You'll want to make sure you've read it by now if you haven't already.

while Loop Assembly Representation

`while` loops are represented in assembly language in a fairly intuitive manner.The general assembly representation of loop-like structures was covered in Chapter 8, but I'll follow it up with a more focused study here. `while` loops break down to two major structures—the conditional expression, which is evaluated just before the execution of each iteration, and the code block that implements the loop's intended functionality (see Figure 15.33). A nice feature of the high-level syntactic layout of `while` loops is that they more or less mirror their assembly equivalents.

Due to the sequential flow of an assembly language script (jump instructions notwithstanding), it makes intuitive sense that the expression that determines whether the next iteration of the loop should execute needs to appear before the loop body. An assembly-coded `while` loop therefore begins with the code to implement its conditional expression, which ends by pushing the result of the expression (an either zero or nonzero value, corresponding to false and true, respectively) onto the stack. This value is then popped off into the `_T0` register and compared to zero in a

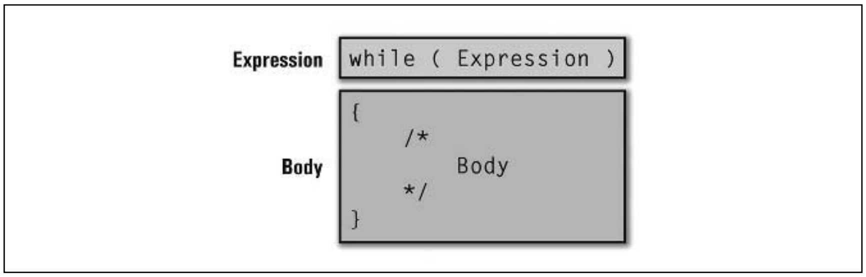


Figure 15.33

The syntactic layout of the `while` loop.

conditional jump and will either fall through into the loop if the expression evaluates to true, or jump to a label set beyond the last instruction of the loop body if the expression evaluates to false. This allows the first iteration of the loop to execute if the expression is true, and results in the loop being skipped entirely otherwise. The only problem is that only the first iteration will execute.

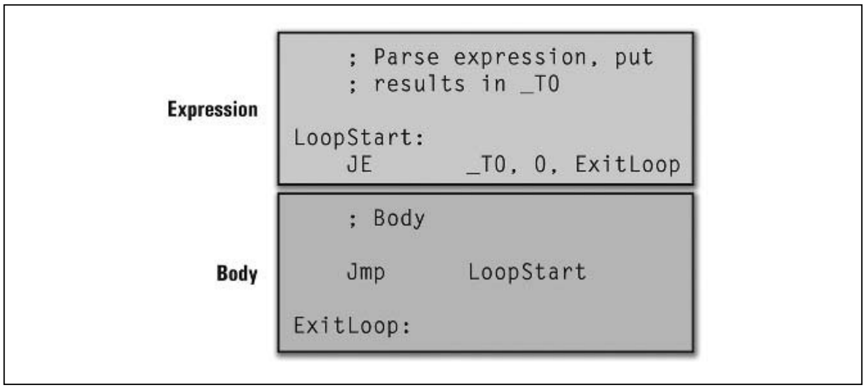
NOTE

I mention that true is represented with nonzero, whereas false is represented by zero. Although this is normally a strict definition in the case of native hardware, it's a slightly simplified way to explain what's going on within the XVM. Remember, as you saw in Chapter 10, true is actually represented by either a numeric nonzero value or a non-empty string. This allows string values to be used in jumps, which is certainly important when a `while` or `if` block involves such data types.

To remedy this, another label must be generated just above the code that evaluates the loop's expression. Every time an iteration of the loop executes, it makes an unconditional jump to this label. The flow of this assembly language representation of the loop is as follows:

- When the loop initially begins executing, it will evaluate its expression and push the result onto the stack.
- The result will be popped off the stack into `_T0` and used as the criteria for a conditional jump that will jump to a label beyond the end of the loop in the event of a zero result. Otherwise, if the result is nonzero, the jump does not occur and execution "falls through" into the body of the loop.
- The loop body executes, completing a single iteration.
- The last instruction of the loop is an unconditional jump placed just above the loop's expression evaluating code, which causes the process to repeat from the first step.

To understand this more clearly, check out Figure 15.34, which illustrates this process graphically.



```
    ; Parse expression, put
    ; results in _T0

LoopStart:
    JE        _T0, 0, ExitLoop
    ; Body

    Jmp       LoopStart

ExitLoop:
```

Figure 15.34

The assembly-language representation of a `while` loop.

Parsing while Loops

Now that you understand the theory and assembly language representation behind the `while` loop, you can write a `ParseWhile ()` function that will parse it. To kick things off, check out the `while` loop's syntax diagram in Figure 15.35.

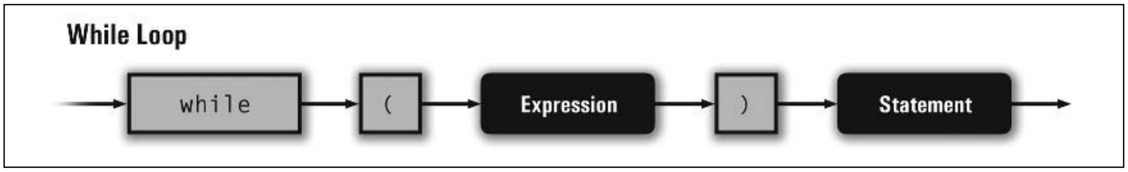


Figure 15.35

The `while` loop's syntax diagram.

As always, the first step in adding any new feature to the parser is updating `ParseStatement ()` so that it can intercept the initial token. For the sake of brevity, I'm only going to list `ParseStatement ()`'s `switch` block:

```
// Branch to a parse function based on the token
switch ( InitToken )
{
    // Unexpected end of file
    case TOKEN_TYPE_END_OF_STREAM:
        ExitOnCodeError ( "Unexpected end of file" );
        break;
    // Block
    case TOKEN_TYPE_DELIM_OPEN_CURLY_BRACE:
        ParseBlock ();
        break;

    // Function definition
    case TOKEN_TYPE_RSRVD_FUNC:
        ParseFunc ();
        break;

    // Host API function import
    case TOKEN_TYPE_RSRVD_HOST:
        ParseHost ();
        break;

    // Variable/array declaration
    case TOKEN_TYPE_RSRVD_VAR:
        ParseVar ();
        break;

    // while loop block
    case TOKEN_TYPE_RSRVD_WHILE:
        ParseWhile ();
        break;

    // Anything else is invalid
    default:
        ExitOnCodeError ( "Unexpected input" );
        break;
}
```

Figure 15.36 updates the Statement syntax diagram.

You can start by taking a look at `ParseWhile ()`:

```
void ParseWhile ()
{
    int iInstrIndex;

    // Make sure we're inside a function
    if ( g_iCurrScope == SCOPE_GLOBAL )
        ExitOnCodeError ( "Statement illegal in global scope" );
```

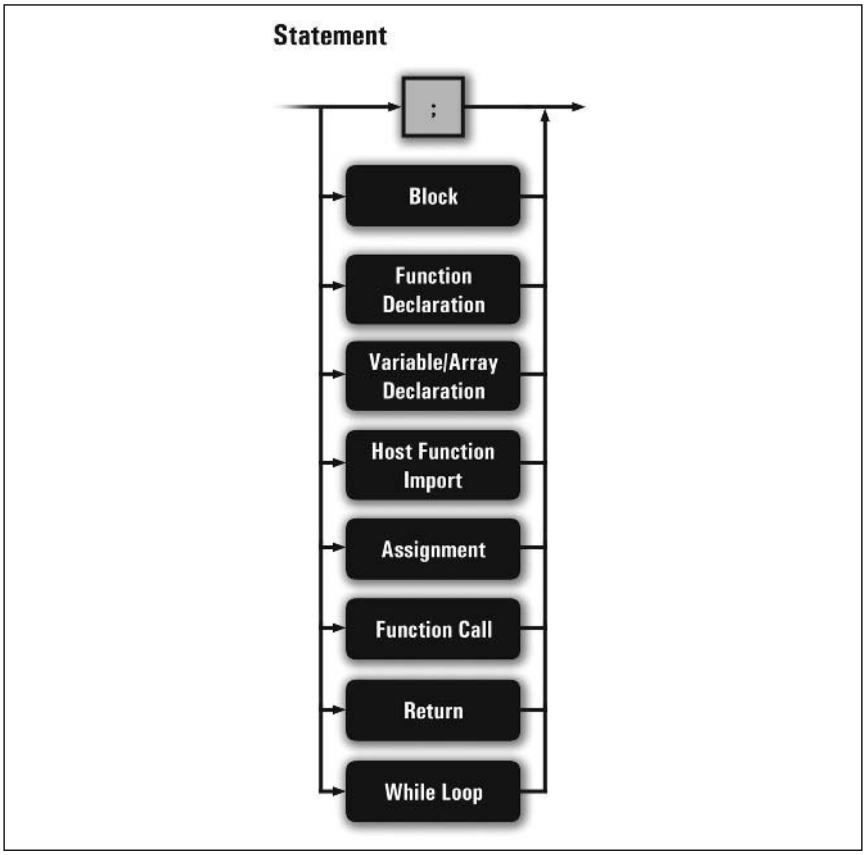


Figure 15.36

The Statement syntax diagram, updated to include `while` loops.

```
// Annotate the line
AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

// Get two jump targets; for the top and bottom of the loop
int iStartTargetIndex = GetNextJumpTargetIndex (),
    iEndTargetIndex = GetNextJumpTargetIndex ();

// Set a jump target at the top of the loop
AddICodeJumpTarget ( g_iCurrScope, iStartTargetIndex );

// Read the opening parenthesis
ReadToken ( TOKEN_TYPE_DELIM_OPEN_PAREN );

// Parse the expression and leave the result on the stack
ParseExpr ();
```

```
    // Read the closing parenthesis
    ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );

    // Pop the result into _T0 and jump out of the loop if it's nonzero
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JE );
    AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
    AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iEndTargetIndex );

    // Parse the loop body
    ParseStatement ();

    // Unconditionally jump back to the start of the loop
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iStartTargetIndex );

    // Set a jump target for the end of the loop
    AddICodeJumpTarget ( g_iCurrScope, iEndTargetIndex );
}
```

The first task is to make sure you're not in the global scope, because the `while` loop must appear inside a function. You then send the I-code module the source line annotation for the first line, which will allow the code emitter to write the loop's expression to the file just above the code that evaluates it. You then generate two new jump targets, and store them in `iStartTargetIndex` and `iEndTargetIndex`. As you can probably guess, these two targets point to the top and bottom of the loop. You make a call to `AddICodeJumpTarget ()` immediately, because the starting jump target must be set before any of the loop's I-code is generated. You can hold off on setting the end target until the rest of the parsing process is complete, however, because you need to make sure it's the last I-code node you generate if you want it to properly represent the end of the loop.

You're ready to generate the I-code for parsing the expression, so `ReadToken ()` is called to verify the presence of the opening parenthesis. The expression's I-code is generated with a call to `ParseExpr ()`, and you once again use `ReadToken ()` to ensure that the closing parenthesis is there.

It's now time for some manual I-code generation. The first instruction you need to immediately follow the expression evaluation is `Pop _T0`, because you need the `_T0` register to hold the result of the expression. This is done with a call to `AddICodeInstr ()` to set the `Pop` instruction, and a follow-up call to `AddVarICodeOp ()` to set `_T0` as the instruction's operand.

Once the value is in _T0, you can generate the jump instruction that will determine whether to execute the next iteration of the loop. You therefore generate a JE instruction (jump if equal) that essentially looks like this:

```
JE    _T0, 0, <Loop End Jump Target>
```

In other words, if the result of the loop's expression was zero (false), exit the loop. You can now parse the loop body, so you call ParseStatement () to generate its I-code. You then append the loop body's I-code with a Jmp (unconditional jump) instruction that branches to the loop's starting jump target. This wraps up the loop, so you can now safely generate the loop's ending jump target, because you know no more loop code will be produced. This is done by passing iEndJumpTarget to AddICodeJumpTarget ().

You might be wondering why ParseWhile () calls ParseStatement () for the loop body instead of ParseBlock (). The reason for this is that, like C, a single-statement loop body doesn't have to be enclosed in curly braces. Of course, if an opening curly brace is found, ParseStatement () knows to call ParseBlock () anyway. This allows you to easily support true C-style loop syntax. Slick, eh?

Here's a simple example of using while in a script:

```
while ( true )
{
    X = Y;
}
```

Here's the compiled output:

```
    ;     while ( true )
_L0:
    Push       1
    Pop        _T0
    JE         _T0, 0, _L1

    ;        X = Y;
    Push     Y
    Pop        _T0
    Mov        X, _T0
    Jmp        _L0
_L1:
```

Notice the automatic generation of unique line labels and their placement within the code. _L0 comes before the expression is evaluated, whereas _L1 lies just outside the loop.

break

Once inside a loop, it might become necessary to pull the panic switch and immediately terminate it. Fortunately, XtremeScript supports C's `break` statement for doing just this. At first glance, `break` seems like it should be an easy addition—after all, it's just an unconditional jump to the loop's ending jump target, right? For the most part, this is correct—`break` is indeed rather simple to implement. There is one serious caveat, however. How will `break`'s parsing function know which jump target to branch to?

Remember, `break` is a statement, just like anything else. This means that the only time you'll parse it is from `ParseStatement ()`, which is called from `ParseWhile ()`. Unfortunately, the jump targets are stored as local variables and are inaccessible from even a nested parse function. You could simply make them global, which would work on some levels, but that too suffers from a fatal flaw. By making the `while` loop's jump targets global, a nested loop will permanently overwrite the targets of its parent loop. This would cause a problem in the case of something like this script fragment:

```
while ( X < Y )
{
    while ( U > V )
    {
        break;
    }
    break;
}
```

The first `while` would save its jump targets in two global values and call `ParseStatement ()` to generate the I-code for its body. Within this call, the nested `while` would cause another instance of `ParseWhile ()` to be invoked, which would end up overwriting the first `while` loop's jump targets. This is okay though, because when `ParseStatement ()` is called, which will end up calling `ParseBreak ()` to handle the `break` statement, all you need are the jump targets of the innermost loop, because that's always the one you're in. The problem occurs when the nested loop terminates, leaving you once again in the outer loop. This time, when the next `break` is encountered, the jump target will incorrectly point to the end of the now terminated inner loop.

The Loop Stack

At this point, it should be clear that the solution to this problem is to push loops' jump targets onto a global stack. This way, loops can be nested indefinitely, and each set of jump targets will remain intact. This is analogous to the way you use the XVM's runtime stack to track the return addresses of functions, regardless of their nested or even recursive nature.

The first step in implementing this solution is declaring a global instance of the `Stack` structure you created in the last chapter called `g_LoopStack`:

```
Stack g_LoopStack;
```

This loop needs to be initialized when the parser starts, so you can add the following line of code to `ParseSourceCode ()`, just before it enters its statement parsing loop:

```
InitStack ( & g_LoopStack );
```

Finally, the stack needs to be freed after the loop, so `ParseSourceCode ()` now ends with this:

```
FreeStack ( & g_LoopStack );
```

You then need to crack open `ParseWhile ()` and make a few changes. Specifically, you need to push the loop's jump targets onto the stack just before parsing the body with `ParseStatement ()`. You then need to pop the targets off afterwards, so in case the loop is nested, the targets of its outer loop will once again be the stack's top element.

Because you're tracking two values (for the two jump targets), you should create a structure to wrap them. This will allow you to deal with single elements on the stack. It will also leave things open ended, so you'll have the option to add additional information somewhere down the line if the need ever arises. The structure will simply be called `Loop`, and will represent a "loop instance":

```
typedef struct Loop                     // Loop instance
{
    int iStartTargetIndex;              // The starting jump target
    int iEndTargetIndex;                // The ending jump target
}
    Loop;
```

Of course, all you need at the moment are the two targets, so that's all the structure contains. With the structure ready to go, you can add the proper code to `ParseWhile ()` so that its nested call to `ParseStatement ()` will have easy access to the proper jump targets in the event that a `break` statement is parsed. Here's the code:

```
// Create a new loop instance structure
Loop * pLoop = ( Loop * ) malloc ( sizeof ( Loop ) );

// Set the starting and ending jump target indexes
pLoop->iStartTargetIndex = iStartTargetIndex;
pLoop->iEndTargetIndex = iEndTargetIndex;

// Push the loop structure onto the stack
Push ( & g_LoopStack, pLoop );
```

```
// Parse the loop body
ParseStatement ();

// Pop the loop instance off the stack
Pop ( & g_LoopStack );
```

Quite simply, the code allocates a new `Loop` structure to hold the loop instance, writes the jump targets to it, and pushes onto the stack. `ParseStatement ()` is then called, as usual, but with the added benefit of the loop stack. When the function returns, you immediately pop the loop instance off to allow any outer loops to regain their position at the top of the stack.

Parsing break

With the loop stack up and running, you have all the information you need to implement `break`. Not surprisingly, this starts by adding its respective `case` to `ParseStatement ()`'s `switch` block:

```
// break
case TOKEN_TYPE_RSRVD_BREAK:
    ParseBreak ();
    break;
```

There's really no need to add another update to the Statement syntax diagram for now; `break` is indeed another statement type, but it's such an obvious addition that it would just be a waste of space. `ParseBreak ()` is a pretty straightforward function, so the simplistic syntax diagram displayed in Figure 15.37 shouldn't be a surprise. Let's check out the code:

```
void ParseBreak ()
{
    // Make sure we're in a loop
    if ( IsStackEmpty ( & g_LoopStack ) )
        ExitOnCodeError ( "break illegal outside loops" );

    // Annotate the line
    AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

    // Attempt to read the semicolon
    ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );

    // Get the jump target index for the end of the loop
    int iTargetIndex = ( ( Loop * )
        Peek ( & g_LoopStack ) )->iEndTargetIndex;
    // Unconditionally jump to the end of the loop
    int iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iTargetIndex );
}
```

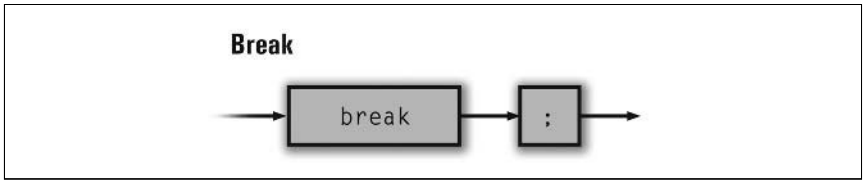


Figure 15.37

The syntax diagram for `break`.

You first ensure that the statement hasn't occurred outside of a function, which is of course illegal. The source code is then annotated, and the trailing semicolon is verified with `ReadToken ()`. You then use the `Peek ()` function to read the top loop instance and extract the `iEndTargetIndex` field. You save this locally in `iTargetIndex`, and use it to generate an unconditional jump to the end of the loop.

As an example, let's look at the script fragment again:

```
while ( X < Y )
{
    while ( U > V )
    {
        break;
    }
    break;
}
```

When compiled, it will produce this. Notice that each `break`'s `Jmp` is linked to the proper label:

```
    ;      while ( X < Y )
_L0:
    Push        X
    Push        Y
    Pop         _T1
    Pop         _T0
    JL          _T0, _T1, _L2
    Push        0
    Jmp         _L3
_L2:
    Push        1
_L3:
    Pop         _T0
    JE          _T0, 0, _L1

    ;      while ( U > V )
_L4:
    Push        U
    Push        V
    Pop         _T1
    Pop         _T0
    Pop         _T0
    JE          _T0, 0, _L5

    ;      break;
    Jmp         _L5
    Jmp         _L4
_L5:

    ;      break;
    Jmp         _L1
    Jmp         _L0
_L1:
```

continue

As you can probably imagine, `continue` is a snap once `break` has been implemented. Because it's virtually the same process, let's just blaze through the code, starting with `ParseStatement ()`'s obligatory addition:

```
// continue
case TOKEN_TYPE_RSRVD_CONTINUE:
    ParseContinue ();
    break;
```

`ParseContinue ()` is almost `ParseBreak ()` verbatim; the only real change is that you're reading the loop's starting target index, rather than the ending index (see Figure 15.38):

```
void ParseContinue ()
{
    // Make sure we're inside a function
    if ( IsStackEmpty ( & g_LoopStack ) )
        ExitOnCodeError ( "continue illegal outside loops" );
    // Annotate the line
    AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );

    // Attempt to read the semicolon
    ReadToken ( TOKEN_TYPE_DELIM_SEMICOLON );

    // Get the jump target index for the start of the loop
    int iTargetIndex = ( ( Loop * )
        Peek ( & g_LoopStack ) )->iStartTargetIndex;

    // Unconditionally jump to the end of the loop
    int iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iTargetIndex );
}
```

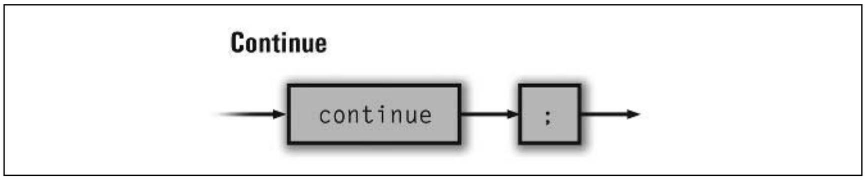


Figure 15.38

The syntax diagram for the `continue` *statement.*

Pretty simple, huh? Let's take a look at an example:

```
while ( true )
{
    continue;
}
```

When compiled, the output will look like this:

```
    ;     while ( true )
_L0:
    Push         1
    Pop          _T0
    JE           _T0, 0, _L1

    ;        continue;
    Jmp          _L0
    Jmp          _L0
_L1:
```

for Loops

Although `for` loops were mentioned in Chapter 7's XtremeScript language specification, I won't be implementing them here. Rather, they're left as a roughly intermediate-level challenge to you. I did this for a number of reasons. First of all, the `for` loop is really just a different way to package the `while` loop. For example, the following `for` loop:

```
for ( X = 0; X < 16; ++ X )
{
    // Loop body
}
```

Can be easily recoded as a `while` loop, like this:

```
X = 0;
while ( X < 16 )
{
    // Loop body
    ++ X;
}
```

Because of this, there's no particularly dire reason to implement `for` at all, really. Anything you can do with `for` can be easily done with `while`, although `for` syntax it can be more convenient and readable at times.

This fact leads you to my second reason, which is that `for` loops *can* be implemented entirely as a preprocessing step. It may sound strange at first, but it's entirely possible with only some basic lexical analysis and string copying to physically convert `for` loops to equivalent `while` loops before the parser even sees the code. If you chose to implement `for`, you might want to investigate this as a possibility.

Branching with if

The second and last control structure you'll be implementing here is `if`, which of course allows you to perform conditional logic. As you did with the `while` loop, the first step in understanding how `if` is compiled is represented in assembly language. With this in mind, developing a parsing strategy will be trivial.

if Block Assembly Representation

The high-level syntactic order of an `if` block is quite simple; a conditional expression starts the block, which is immediately followed by a true block and a false block. If the expression evaluates to true, the flow of execution "falls into" the true block, which resides just under the expression, and skips the false block when it reaches the end. Otherwise, the true block is skipped and the false block is executed. When the false block terminates, execution continues sequentially, because the rest of the code lies directly below it. The false block is of course optional, however, and is facilitated with the `else` keyword. Figure 15.39 depicts the syntactic flow of an `if` block.

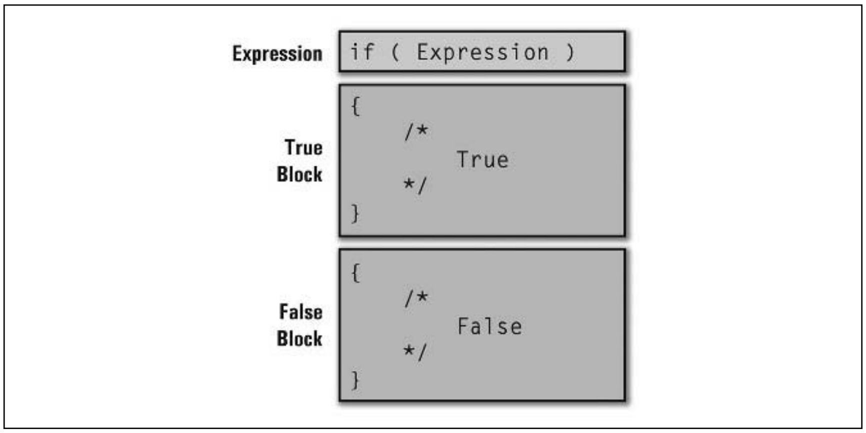


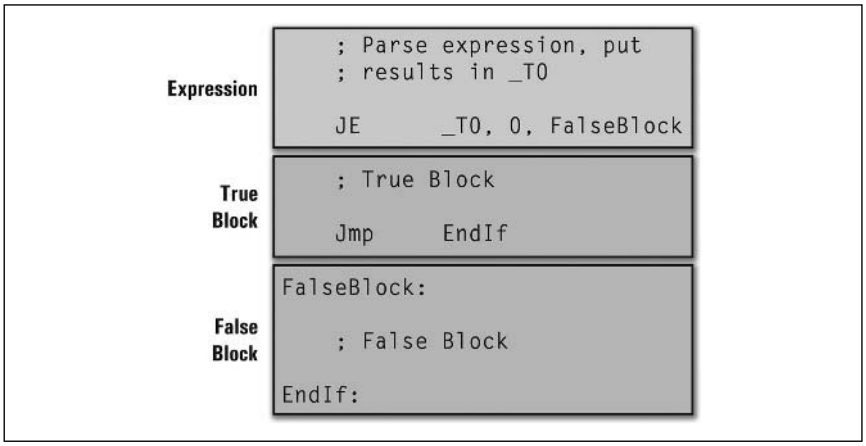
Figure 15.39

The `if` block's syntactic layout.

Chapter 8 discussed the two primary methods by which an `if` block can be organized in assembly, which revolved around the placement of the true and false blocks. Although the discussion there hinged on the fact that one method forced you to inverse the conditional expression, whereas one didn't, this particular point is moot in the case of the compiler, because the actual comparison is simply comparing the final result of the expression to zero. Because of this, you can maintain the conventional order of the true block coming before the false block without hassle.

Like `while`, the first block of code to be generated for an `if` block in assembly language is responsible for evaluating the conditional expression that drives it, and for leaving the result on the top of the stack. Also like `while`, a nonzero expression represents truth, and a zero expression results falsehood. Because of this, the top stack element can be used as the criteria for an unconditional jump that will allow you to route the flow of execution through and around the appropriate blocks.

Such a jump immediately follows the evaluation of the expression. Specifically, you use a `JE` (Jump if Equal) instruction that compares the result of the expression to zero. Because you're testing for equality with zero, the instruction should jump to the false block in the event that the operands match. Otherwise, execution can fall into the true block. Once you're done executing this block, however, it's important that you make an unconditional jump *over* the false block, because you certainly don't want both blocks to execute. As stated earlier, the false block can terminate as-is, because execution will flow back into the otherwise sequential order of the script. Figure 15.40 illustrates the resulting code's general form.



```
    ; Parse expression, put
    ; results in _T0

    JE      _T0, 0, FalseBlock
```

```
    ; True Block

    Jmp     EndIf
```

```
FalseBlock:

    ; False Block

EndIf:
```

Figure 15.40

The `if` *block's assembly representation.*

Parsing if Blocks

Now that you understand the form the emitted code should take, you can put together a parser rather easily. It's primarily a matter of emitting the code blocks in the right order and keeping track of the jump targets. Figure 15.41 presents the syntax diagram for `if` blocks.

Here's the addition you make to `ParseStatement ()` (reflected in Figure 15.42):

```
// if block
case TOKEN_TYPE_RSRVD_IF:
    ParseIf ();
    break;
```

Let's step through `ParseIf ()` section by section:

```
void ParseIf ()
{
    int iInstrIndex;

    // Make sure we're inside a function
    if ( g_iCurrScope == SCOPE_GLOBAL )
        ExitOnCodeError ( "if illegal in global scope" );

    // Annotate the line
    AddICodeSourceLine ( g_iCurrScope, GetCurrSourceLine () );
```

The function starts with the obligatory proceedings. First the scope is checked to make sure an `if` block isn't being used outside of a function, and the current line is added to the I-code as source annotation.

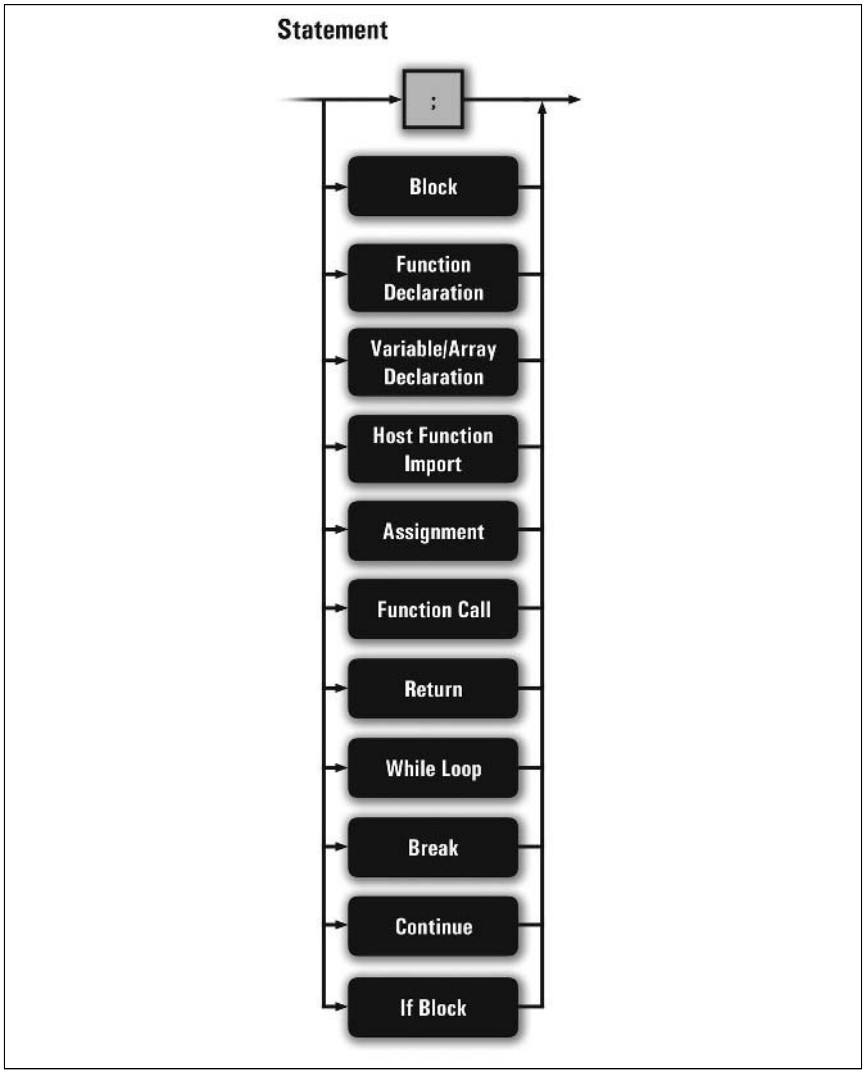


Figure 15.41

The Statement syntax diagram, updated to include `if` blocks.

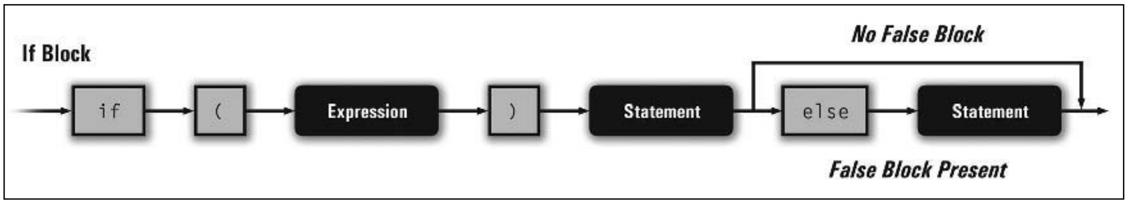


Figure 15.42

The syntax diagram for `if` blocks.

```
// Create a jump target to mark the beginning of the false block
int iFalseJumpTargetIndex = GetNextJumpTargetIndex ();

// Read the opening parenthesis
ReadToken ( TOKEN_TYPE_DELIM_OPEN_PAREN );

// Parse the expression and leave the result on the stack
ParseExpr ();

// Read the closing parenthesis
ReadToken ( TOKEN_TYPE_DELIM_CLOSE_PAREN );
```

The next step is creating the jump target that will mark the beginning of the false block. You have to do this now, because the jump instruction generated after evaluating the expression needs a target to jump to. Remember, you can add a jump node to the I-code before you add its target node, just like a jump instruction can appear before the definition of its label. The expression is handled next, which is simply a matter of reading both the opening and closing parentheses, and calling `ParseExpr ()` in between.

```
// Pop the result into _T0 and compare it to zero
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_POP );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );

// If the result is zero, jump to the false target
iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JE );
AddVarICodeOp ( g_iCurrScope, iInstrIndex, g_iTempVar0SymbolIndex );
AddIntICodeOp ( g_iCurrScope, iInstrIndex, 0 );
AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex, iFalseJumpTargetIndex );
```

At this point, the expression has been parsed, and the I-code for evaluating it has been generated. You can now generate a jump instruction to alter the flow of the script's execution based on the result of this evaluation. This is done by popping the stack's top value into `_T0` and jumping to the `iFalseJumpTargetIndex` target you created earlier. Again, you haven't placed this target yet; you're only generating code to jump to it.

```
// Parse the true block
ParseStatement ();

// Look for an else clause
if ( GetNextToken () == TOKEN_TYPE_RSRVD_ELSE )
```

```
{
    // If it's found, append the true block with an
    // unconditional jump past the false block
    int iSkipFalseJumpTargetIndex = GetNextJumpTargetIndex ();
    iInstrIndex = AddICodeInstr ( g_iCurrScope, INSTR_JMP );
    AddJumpTargetICodeOp ( g_iCurrScope, iInstrIndex,
        iSkipFalseJumpTargetIndex );

    // Place the false target just before the false block
    AddICodeJumpTarget ( g_iCurrScope, iFalseJumpTargetIndex );

    // Parse the false block
    ParseStatement ();

    // Set a jump target beyond the false block
    AddICodeJumpTarget ( g_iCurrScope, iSkipFalseJumpTargetIndex );
}
else
{
    // Otherwise, put the token back
    RewindTokenStream ();

    // Place the false target after the true block
    AddICodeJumpTarget ( g_iCurrScope, iFalseJumpTargetIndex );
}
```

The final step is the generation of each block. You parse and generate the true block first, with a simple call to `ParseStatement ()`. Again, you parse it as a statement rather than a block, because this gives the parser the capability to interpret both single-lines and full blocks.

The false block is a bit trickier, because it's optional. To determine whether a false block is present, you use `GetNextToken ()` to find out if the `TOKEN_TYPE_RSRVD_ELSE` token is next in the stream. If not, you'll use `RewindTokenStream ()` to put it back. The look-ahead won't help you in this situation because simply reading an "e" wouldn't be enough to determine whether `else` truly followed. For example, take the following block of code:

```
var Exp;
if ( Exp < 16 )
    TextBox ( "Your character is low on experience points." );
Exp += NewLevelExp;
```

Here, the token following the true block (which is a single line in this example) is `Exp`, which begins with `E`. Even though this doesn't necessarily have anything to do with the `if` block that preceded it, the parser will interpret it as the start of an `else` clause going by the look-ahead alone. It will then attempt to parse the block, resulting in confusing compile-time errors for the users.

The first step in parsing the false block is generating an unconditional jump past it. This is done because it will immediately follow the true block, which must skip past it. The reason you do this in the generation of the false block, rather than that of the true block, is that you only want this particular jump instruction to appear in the event that an `else` clause exits. Otherwise, it's omitted entirely. The target used to jump past the false block is called `iSkipFalseJumpTargetIndex`, and is only created with `GetNextJumpTargetIndex ()`. It's not actually placed until after the block has been parsed and generated.

Next, the `iFalseJumpTargetIndex` target you created earlier is added to the I-code stream, so the `if`'s original jump can reach it in the event that the conditional expression evaluates to false. With the target in place, it's safe to parse and generate the false block itself, which is done with another call to `ParseStatement ()`. Lastly, now that the false block has been generated, you generate the jump target stored in `iSkipFalseJumpTargetIndex`, which the true block jumps to.

As I mentioned, if the `else` token wasn't found, the stream is rewound. In this case, the false jump target is emitted by itself; this allows the `if`'s initial jump instruction to bypass the true block, whether or not a false block lies beyond it.

Check out this example:

```
if ( X )
    Y = X;
else
    X = Y;
```

Here's its compiled output:

```
    ;    if ( X )
    Push      X
    Pop       _T0
    JE        _T0, 0, _L0

    ;    Y = X;
    Push      X
    Pop       _T0
    Mov       Y, _T0
    Jmp       _L1
_L0:
    ;       X = Y;
    Push       Y
    Pop        _T0
    Mov        X, _T0
_L1:
```

X is pushed onto the stack and popped into _T0. _T0 is then compared to zero, and if it's equal, a jump is made to _L0, which marks the top of the false block. Otherwise, the execution falls into the true block and executes sequentially until its last line, when an unconditional jump to _L1 is made to avoid the false block.

Syntax Diagram Summary

Syntax diagrams have served you well—they've provided a visual blueprint for an entire parser module, and you've been able to follow them accurately. To sum things up, however, let's take a look at Figure 15.43, which presents a single syntax diagram that encompasses the entire XtremeScript language. Think of this as a visual reference for the language's syntax.

The Test Drive

You now have an entire working compiler, so it would be pretty silly not to have some fun with it. To make sure everything is functioning properly, let's write a few demo scripts that test various aspects of XtremeScript. On the most obvious level, there's the compiler itself, which must be vigorously tested because it's such an error-prone component of the system. Next is the assembler, which is being fed the compiler's output directly. Because XASM has its own strictly imposed rules, you can ensure that everything coming out of the compiler is correct. Lastly, the .XSE generated by XASM is put to the ultimate test by letting it run inside the XVM. In a lot of ways, the XVM's behavior is the easiest to debug, because you can directly watch it as it executes. If something isn't working properly, you'll see it immediately. Of course, there are plenty of under-the-hood bugs that can go unnoticed by the eye, so you have to watch the step.

Hello, World!

The quintessential "Hello, world!" is probably the best way to christen a new compiler; it may be about the simplest program imaginable, but there's just something extremely cool about running such an infamous beginner programming lesson in a language you designed and/or implemented yourself. Ladies and gentlemen, I give you "Hello, world!"—XtremeScript style.

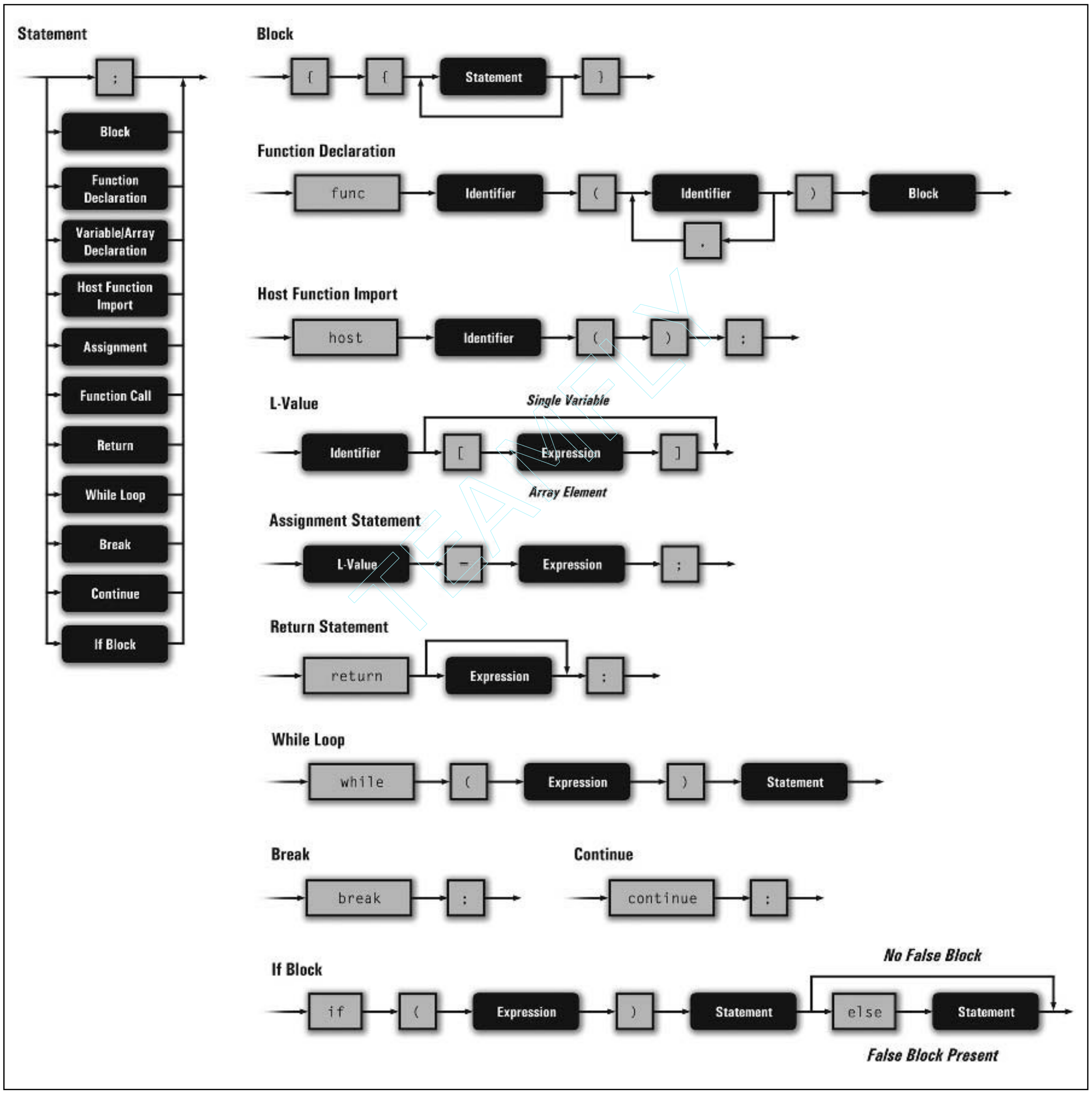


Figure 15.43

A syntax diagram for the entire XtremeScript language.

```
/*
    Hello, world!
*/

host PrintString ();

func _Main ()
{
    PrintString ( "Hello, world!" );
}
```

Surreal, huh? Remember of course that because you're running this on the XVM console, you need to import the `PrintString ()` function. By saving it as `hello.xss` and passing it through the compiler like so:

```
XSC hello.xss -A
```

you can create both an .XSE and the .XASM file from which it was assembled. The XVM assembly produced by the compiler looks like this:

```
; HELLO.XASM

; Source File: HELLO.XSS
; XSC Version: 0.8
;   Timestamp: Sat Sep 14 17:10:36 2002

; ---- Directives -----------------------------------
; ---- Global Variables -----------------------------

    Var _T0
    Var _T1

; ---- Functions ------------------------------------
; ---- Main -----------------------------------------

    Func _Main
    {
  ;    PrintString ( "Hello, world!" );

        Push         "Hello, world!"
        CallHost   PrintString
        Push       _RetVal
        Pop        _T0
    }
```

And of course, by running it in the XVM console, you'll get the following:

```
Hello, world!
```

Drawing Rectangles

I personally find coding for the XVM console to be a fun little exercise; it reminds me of the text-mode demo programs you find in the older books on languages such as Pascal and C. In addition to Hello, world!, however, I remember a lot of the older books presenting example programs that drew shapes using asterisks. So, just for fun, let's write a little script that does the same thing.

The program will of course be very simple; you'll use two global variables to define the X and Y dimensions of the rectangle, and then use two nested `while` loops to do the actual drawing. You'll make heavy use of the XVM's `PrintString ()` and `PrintNewline ()` host API functions as well. Here's the high-level .XSS script:

```
/*
   Rectangle drawing
*/

// Import the host API functions
host PrintString ();
host PrintNewline ();

// Make the size of the rectangle global
var g_XSize;
var g_YSize;

func _Main ()
{
   // Create some variables for tracing the shape
   var X;
   var Y;

   // Set the rectangle size to 32x16
   g_XSize = 32;
   g_YSize = 16;

   // Y-loop
   Y = 0;
```

```
    while ( Y < g_YSize )
    {
        // X-loop
        X = 0;
        while ( X < g_XSize )
        {
            // Draw the next asterisk
            PrintString ( "*" );

            // Move to the next column
            X += 1;
        }

        // Move to the next row
        PrintNewline ();
        Y += 1;
    }
}
```

After drawing each row of `XSize` asterisks, a call is made to `PrintNewline ()` to move to the next line. `X` is incremented at each iteration of the X-loop, and `Y` is incremented at each iteration of the Y-loop. Both are compared to `XSize` and `YSize`, respectively, to determine when the loop should terminate. This file can be saved as `rectangle.xss` and passed through the XSC compiler like this:

```
XSC rectangle.xss -A
```

Remember, you're continuing to use the `-A` switch to preserve the assembly output so you can examine it. The compiler will produce `rectangle.xasm`, which looks like this:

```
; RECTANGLE.XASM

; Source File: RECTANGLE.XSS
; XSC Version: 0.8
;   Timestamp: Mon Sep 16 20:59:57 2002

; ---- Directives -----------------------------------------
; ---- Global Variables -----------------------------------

    Var _T0
    Var _T1
    Var g_XSize
    Var g_YSize

; ---- Functions -----------------------------------------
; ---- Main ----------------------------------------------

    Func _Main
    {
        Var X
        Var Y

  ;     g_XSize = 32;
        Push        32
        Pop         _T0
        Mov         g_XSize, _T0

  ;     g_YSize = 16;
        Push        16
        Pop         _T0
        Mov         g_YSize, _T0

  ;     Y = 0;
        Push        0
        Pop         _T0
        Mov         Y, _T0

  ;     while ( Y < g_YSize )
    _L0:
        Push        Y
        Push        g_YSize
        Pop         _T1
        Pop         _T0
        JL          _T0, _T1, _L2
        Push        0
        Jmp         _L3
    _L2:
        Push        1
    _L3:
        Pop         _T0
        JE          _T0, 0, _L1
;          X = 0;
        Push       0
        Pop        _T0
        Mov        X, _T0

;          while ( X < g_XSize )
  _L4:
        Push       X
        Push       g_XSize
        Pop        _T1
        Pop        _T0
        JL         _T0, _T1, _L6
        Push       0
        Jmp        _L7
  _L6:
        Push       1
  _L7:
        Pop        _T0
        JE         _T0, 0, _L5

;              PrintString ( "*" );
        Push       "*"
        CallHost   PrintString

;              X += 1;
        Push       1
        Pop        _T0
        Add        X, _T0
        Jmp        _L4
  _L5:

;          PrintNewline ();
        CallHost   PrintNewline

;          Y += 1;
        Push       1
        Pop        _T0
        Add        Y, _T0
        Jmp        _L0
  _L1:
  }
```

Lastly, by running `rectangle.xse` in the XVM, you get this, a 32x16 rectangle of asterisks:

```
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
********************************
```

Text games ahoy!

The Bouncing Head Demo

Hello, world! and rectangle drawing might be a nice dose of nostalgia, but they don't exactly put XtremeScript through its paces. It would be nice to write a more graphical demo that actually involves real-world examples of iteration, branching, and other staples of high-level programming. By writing a script that manages a decent amount of data, has to run at a high enough speed to keep the screen updated on a per-frame basis, and has a reasonably complex task to perform, you can be [almost] sure that the compiler, assembler, and virtual machine are all working properly.

The bouncing alien head demo you created and recoded in multiple scripting languages back in Chapter 6 is the perfect candidate. It requires you to manage the positions and frames of each on-screen sprite, must run fast enough to update the screen on a regular basis, and is driven by logic that's just complicated enough to give the compiler a workout without having to spend six months on it.

However, the less-than-glamorous reality of the compiler is that its output is unoptimized in every sense of the word, and the runtime performance of executables built with it will demonstrate this fact. Although what I said in Chapter 7 is true—that the speed difference between compiler and hand-assembly versions of the same script will be negligible—this is only the case when the compiler in question performs at least basic optimizations. Given the complexity of writing an optimizing compiler, as well as the fact that XSC was only one of many components described in this book, you'll have to settle for a compiler whose sole goal is simply to work properly. Fortunately, XSC does that.

For an idea of what the demo will look like ahead of time, check out Figure 15.44.

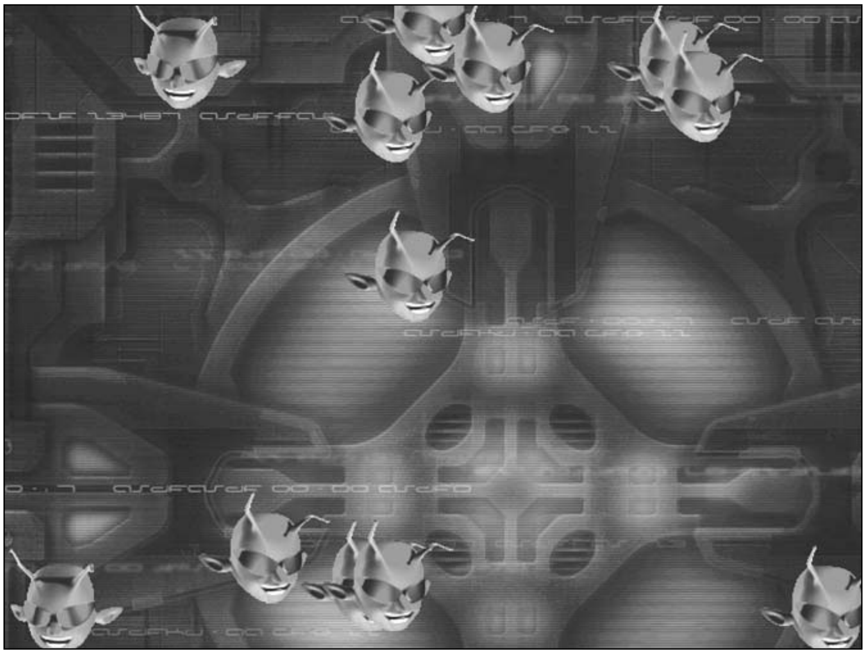


Figure 15.44

The bouncing head demo, now scripted with XtremeScript.

Anatomy of the Program

The demo you're going to put together in this section is rather simple. Its primary job is to display a background image and a number of bouncing alien head sprites, each of which rotates in a bitmapped animation. The movement of the sprites is simple bouncing ball logic; each sprite has an X and Y location, as well as an X and Y velocity; when the sprite collides with one of the screen's boundaries, the sign of its X or Y velocity is flipped to simulate a "bounce."

Chapter 6 covered four versions of this program. The first was entirely coded in C, and therefore had little to do with scripting per se. The remaining demos used the Lua, Python, and Tcl scripting languages to rewrite the program's core logic, thus demonstrating the process behind their integration with the host application.

To put it simply, this chapter's implementation of the demo will consist of two major parts. The first is of course the host application, whose job is to perform low-level tasks like loading graphics and managing the program's main loop, as well as to expose a host API. The second is the script, which will focus on the actual functionality and logic of the demo. It will also help with the program's initialization.

Specifically, the script will expose two functions, an `Init ()` function that is called once, at the start of the program, to set everything up; and `HandleFrame ()`, which is called once per frame and is responsible for moving the sprites around and drawing the new frame's contents.

Although I'll cover it in a bit more detail, the host API will be simple as well. Its primarily job is providing an abstracted interface to the underlying operating system's relevant features—in this case, graphics, timing, and so on.

NOTE

You may be wondering why I am still calling the `HandleFrame ()` function once per frame, rather than taking advantage of XtremeScript's capability to run in parallel with the main loop via repeated calls to `XS_RunScripts ()` with a specified time slice. I did this because the demos in Chapter 6 worked this way, and I wanted XtremeScript's capabilities to mimic their overall functionality. This makes them easier to compare.

Simulating Structures

One important issue worth mentioning before continuing involves the structures used in the Chapter 6 demos to track each alien sprite as the program executes. All three of the languages you used provided some way to create and manage structures that resembled C's `struct`. This was naturally a useful feature, because each sprite maintains an X and Y location, an X and Y velocity, and the direction in which its animation spins. Expressed with pseudo-code, this would form a structure along these lines:

```
struct Alien
{
    var X, Y;
    var XVel, YVel;
    var SpinDir;
}
```

Unfortunately, XtremeScript's only notion of aggregate data structures comes in the form of simple, one-dimensional arrays. This prevents you from easily representing a group of alien sprites, because each element of the array is larger than a single variable.

Even without structures, this problem could be solved fairly easily with the help of two-dimensional arrays. For example, you could allocate storage for the on-screen aliens with something like this:

```
var Aliens [ MAX_ALIEN_COUNT ][ 5 ];
```

Each element of this array is actually five elements, which allows you to store `X` in element 0, `Y` in element 1, `XVel` in element 2, `YVel` in element 3, and `SpinDir` in element 4. Figure 15.45 demonstrates this idea of using an array to simulate a structure.

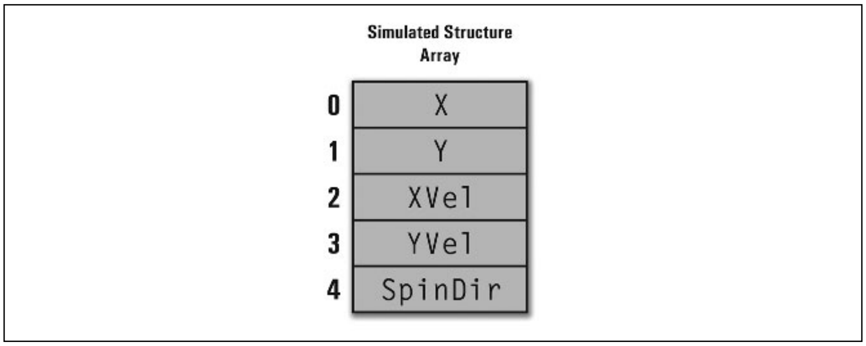


Figure 15.45

A structure simulated with an array.

Without explicit support for two-dimensional arrays, however, the end result of this approach can be simulated. After all, any N-dimensional array is stored in memory in a purely linear fashion; the concept of multiple dimensions is really just an abstraction supported by a language's notation and syntax. Imagine an array like this:

```
var Aliens [ MAX_ALIEN_COUNT * 5 ];
```

Even without N-dimensional notation, you have the same number of elements to work with as you did with the array's two-dimensional counterpart. Now, alien 0 takes up elements 0 through 4 (the first five), alien 1 is represented by elements 5 through 9 (the second five), and so on. Each alien then has a "base index" within the array, which corresponds to the index where its simulated structure starts. Each alien can then be accessed as `ALIEN_INDEX * 5`. This is the solution you'll take when you commit these scripts to XtremeScript and XVM assembly. Figure 15.46 illustrates this final structure.

The Host Application

This particular demo doesn't need much in the way of host application support. All it really needs is a modest API for accessing graphics and other miscellaneous functions, and for the host to perform some basic initialization and the loading of the necessary graphics.

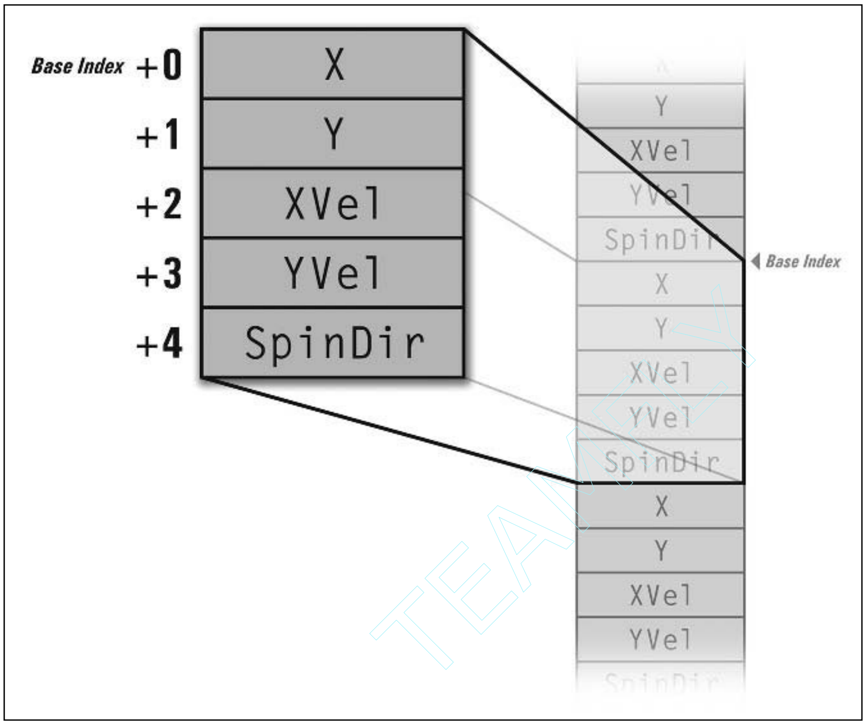


Figure 15.46

An array of simulated structures is actually just one big array.

Specifically, the host application will need to do the following:

- Define the host API's functions.
- Initialize the XVM, register the host API, and shut everything down when the program ends.
- Load the necessary graphics.
- Load the script, and call it on a regular basis within the main loop.

The Host API

The host API's primary functions are graphical, but it also needs to perform a few non-graphical tasks. The API will consist of five functions, which perform the following:

- Blit a sprite to the back buffer given an X, Y coordinate and the index of the sprite into an array of animation frames maintained by the host.
- Blit the preloaded background image to the back buffer.
- Blit the back buffer to the screen.
- Get a random number between a minimum and maximum.
- Return the state of a timer maintained by the host, based on a timer index.

Defining a Host API Function

As you learned in Chapter 11, a host API function is a typical C function that follows a specific prototype:

```
void FuncName ( int iThreadIndex )
```

This signature allows the XVM to pass the function the index of the thread that called it, which is used within the function for various tasks such as reading parameters and returning values.

Parameters are always read with one of the `XS_GetParamAs* ()` functions, which returns the parameter at the specified index in the form of a specific C data type. These functions can return integer, floating-point, and string values. Values are returned to the caller with the `XS_Return* ()` macros, which wrap similarly named functions, but also include a built-in `return` keyword that allows the macro to physically return from the function. Even if a value is not returned, `XS_Return ()` must be used, because all of the macros accept both thread index and parameter count arguments, which are used to help the XVM clear the host API function's stack frame.

BlitSprite ()

The `BlitSprite ()` function blits the specified sprite to the specified X, Y location in the back buffer. This means that the function requires two parameters and returns nothing. The function logic is just a call to `W_BlitImage ()`:

```
void HAPI_BlitSprite ( int iThreadIndex )
{
    // Read in parameters
    int iIndex = XS_GetParamAsInt ( iThreadIndex, 2 );
    int iX = XS_GetParamAsInt ( iThreadIndex, 1 );
    int iY = XS_GetParamAsInt ( iThreadIndex, 0 );

    // Blit sprite
    W_BlitImage ( g_AlienAnim [ iIndex ], iX, iY );

    // Return nothing
    XS_Return ( iThreadIndex, 3 );
}
```

The `iIndex`, `iX`, and `iY` parameters are read using `XS_GetParamAsInt ()`, because they're all integers. The `iThreadIndex` parameter is passed, along with an integer index. The thread index lets the XVM know which thread stack to read the parameter from, and the index specifies the exact desired parameter. Notice that the functions are being read in reverse order, from index 2 to index 0. This is because, as discussed in Chapter 9, by reading parameters from right-to-left within the function, you can let the caller use the traditional left-to-right convention.

After calling `W_BlitImage ()`, the function uses `XS_Return ()` to return nothing and clean up its three parameters.

CAUTION

It's *extremely* important that you *always* end a function with the `XS_Return ()` macro and the proper number of functions. By not cleaning up the function's stack frame properly, the thread's runtime stack will become corrupted and chaos will ensue.

BlitBG ()

`BlitBG ()` is just a simple function that accepts no parameters and returns no values. Its sole concern is blitting the background image to the screen with a call to `W_BlitImage ()`:

```
void HAPI_BlitBG ( int iThreadIndex )
{
    // Blit the background image
    W_BlitImage ( g_BG, 0, 0 );

    // Return nothing
    XS_Return ( iThreadIndex, 0 );
}
```

Remember, even when returning nothing, `XS_Return ()` should be called.

BlitFrame ()

After blitting sprites and background images with the last two functions, the back buffer will contain the next frame. This can be drawn to the screen with `BlitFrame ()`, which wraps the Wrappuh API function of the same name:

```
void HAPI_BlitFrame ( int iThreadIndex )
{
    // Blit the frame to the screen
    W_BlitFrame ();

    // Return nothing
    XS_Return ( iThreadIndex, 0 );
}
```

GetRandomNumber ()

In order to make the aliens bounce around in reasonably interesting ways, they should be initially placed in random locations and given random velocities. Any form of random number generation within the script will be performed with a call to `GetRandomNumber ()`, which returns a random number between `iMin` and `iMax`:

```
void HAPI_GetRandomNumber ( int iThreadIndex )
{
    // Read in parameters
    int iMin = XS_GetParamAsInt ( iThreadIndex, 1 );
    int iMax = XS_GetParamAsInt ( iThreadIndex, 0 );

    // Return a random number between iMin and iMax
    XS_ReturnInt ( iThreadIndex, 2, ( rand () % ( iMax + 1 - iMin ) ) + iMin );
}
```

Once again, you're reading parameters, so `XS_GetParamAsInt ()` is used. You're also returning a value this time, so `XS_ReturnInt ()` is used instead of `XS_Return ()`. Of course, it's still important to pass the parameter count. `XS_ReturnInt ()`'s third argument is the return value.

GetTimerState ()

The movement and animation of the alien heads will be synced up to two timers, both of which are maintained by the host application. In order to read their states (which are 0 or 1), `GetTimerState ()` is used. Like `GetRandomNumber ()`, this function returns a value as well:

```
void HAPI_GetTimerState ( int iThreadIndex )
{
    // Read in the parameters
    int iIndex = XS_GetParamAsInt ( iThreadIndex, 0 );

    // Determine the timer to read based on the index
    int iTimerState = 0;
    switch ( iIndex )
    {
        case 0:
            iTimerState = W_GetTimerState ( g_AnimSpeed );
            break;
        case 1:
            iTimerState = W_GetTimerState ( g_MoveSpeed );
            break;
    }

    // Return the state of the timer
    XS_ReturnInt ( iThreadIndex, 1, iTimerState );
}
```

The parameter it reads with `XS_GetParamAsInt ()` is an index corresponding to a specific timer. This index is then used in a `switch` block to read the timer's state. The value is returned with `XS_ReturnInt ()`. `g_AnimSpeed` and `g_MoveSpeed` are both handles to internal Wrappuh API timers, so check out the source on the companion CD if you want to learn more.

Initialization and Shutdown

Although the XVM's initialization procedure is entirely contained within the `XS_Init ()` function, another vital aspect of initializing the runtime environment is registering the host API. Because of this, I created a function called `InitXVM ()` that wraps these two jobs into a single call:

```
void InitXVM ()
{
    // Initialize the XVM
    XS_Init ();

    // Register the host API with the XVM
    XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetRandomNumber",
        HAPI_GetRandomNumber );
    XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "BlitBG", HAPI_BlitBG );
    XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "BlitSprite", HAPI_BlitSprite );
    XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "BlitFrame", HAPI_BlitFrame );
    XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetTimerState",
        HAPI_GetTimerState );
}
```

Because you'll be loading only one script for the demo, there's no need to worry about function visibility within the host API, so I defined everything as `XS_GLOBAL_FUNC`. Notice also that I decided to drop the `HAPI_` extension within the script; API functions will be known to the scripts with simpler names.

Shutting down the XVM is simply a matter of calling `XS_ShutDown ()`, but because I like to be neat and consistent about everything, I wrapped it in a corresponding `ShutDownXVM ()` function:

```
void ShutDownXVM ()
{
   XS_ShutDown ();
}
```

Loading the Necessary Graphics

The loading of the demo's graphics is really just handled with a few calls to my wrapper API's `W_LoadImage ()` function. You can check out the source on the companion CD if you want to see the details of how the demo deals with this, but there's not much worth explaining here. Suffice it to say, the host application loads the required graphics and makes them globally available to the rest of the program.

Handling the Script

Lastly, there's the issue of the script itself. The script is initially loaded with a call to `XS_LoadScript ()`, which loads the contents of the specified .XSE file into the next free thread:

```
int iThreadIndex;
if ( XS_LoadScript ( "script.xse", iThreadIndex, XS_THREAD_PRIORITY_USER ) )
   W_ExitOnError ( "Could not load script." );
```

You declare `iThreadIndex` to store whatever thread index is used by the function. Because this will be a single-threaded application, you just say `XS_THREAD_PRIORITY_USER` and forget about it. Technically you could pass anything here, because the thread priority is irrelevant. If the function returns a nonzero value, an error has occurred, so the Wrappuh API function `W_ExitOnError ()` is invoked to display an error message in a message box and terminate the program.

Notice you're loading a script called `script.xse`. As you'll see in the next section, you'll write two versions of the script; one in the high-level XtremeScript language, and another in the low-level XVM assembly language. `script.xse` will contain the high-level script, whereas `asm_script.xse` will contain its low-level counterpart.

Once the script is in memory, it must be started:

```
XS_StartScript ( iThreadIndex );
```

The thread is now active in the eyes of the XVM, which allows you to call its functions. When you want to shut down, you can stop the script with `XS_StopScript ()`:

```
XS_StopScript ( iThreadIndex );
```

You don't actually have to do this, because the XVM will shut down either way, but I've included it for illustrative purposes. Normally, this function only applies when a script needs to be stopped at an arbitrary time.

The last aspect of the host's interaction with the script will be the calling of its functions. As you'll see in the next section, the script will define two functions: `Init ()`, whose job is to initialize the script, and `HandleFrame ()`, which is called once per frame and is responsible for drawing and updating the contents of the screen. `Init ()` is called once before entering the main loop, whereas `HandleFrame ()` is called repeatedly until the program terminates:

```
// Let the script initialize the rest
XS_CallScriptFunc ( iThreadIndex, "Init" );
```

```
// Start the main loop
MainLoop
{
    // Start the current loop iteration
    HandleLoop
    {
        // Let XtremeScript handle the frame
        XS_CallScriptFunc ( iThreadIndex, "HandleFrame" );

        // Check for the escape key and exit if it's down
        if ( W_GetKeyState ( W_KEY_ESC ) )
            W_Exit ();
    }
}
```

`XS_CallScriptFunc ()` is used in both cases instead of `XS_InvokeScriptFunc ()`, because you want these functions to execute one time and immediately return. At each iteration of the loop, `HandleFrame ()` is given a chance to draw the next frame and move the alien sprites around. The rest of this section focuses on how these two functions are implemented within the script.

The Low-Level XVM Assembly Script

The first version of the script will be written in XVM assembly language. Although this makes the overall logic considerably more complex, it also yields the fastest possible results by ensuring that nothing is being done unless it absolutely has to. As you've seen, this starkly contrasts with the high-level compiler, which tends to emit far more code than is technically necessary to complete even small tasks.

This subsection dissects the layout and functionality of the assembly language version of the script. I'll run through it segment by segment, so you can see not only how everything works, but specifically how it's implemented in XtremeScript.

Constants

There are a number of constant values that will be used throughout the script, so it's always a good idea to commit them to globally available constants. However, just like the languages you learned about in Chapter 6, XtremeScript doesn't have a `const` keyword or any other method for declaring constant values. So, also like Chapter 6, you'll simulate constants with global variables and `THIS_NAMING_CONVENTION`. Unfortunately, XtremeScript imposes further limitations, which keeps you from initializing these variables with their values in the global scope. You'll therefore have to offload the definition of the constants to the `Init ()` function.

Here are the constants the script will use, in the form of their global declarations:

```
Var ALIEN_COUNT    ; Number of aliens on-screen

Var MIN_VEL        ; Minimum velocity
Var MAX_VEL        ; Maximum velocity

Var ALIEN_WIDTH          ; Width of the alien sprite
Var ALIEN_HEIGHT         ; Height of the alien sprite
Var HALF_ALIEN_WIDTH     ; Half of the sprite width
Var HALF_ALIEN_HEIGHT    ; Half of the sprite height

Var ALIEN_FRAME_COUNT    ; Number of frames in the
                         ; animation
Var ALIEN_MAX_FRAME      ; Maximum valid frame

Var ANIM_TIMER_INDEX     ; Animation timer index
Var MOVE_TIMER_INDEX     ; Movement timer index
```

These contents allow you to track the total number of aliens bouncing around, their minimum and maximum velocities (which will be assigned on a per-alien basis in the `Init ()` function), the sprites' dimensions, the total number of animation frames, and the indexes of the timers the host application will provide for timing the speed of the aliens' animation and movement.

Global Variables

The script needs a small amount of global data, declared with the following code fragment in the global scope:

```
Var Aliens [ 60 ]      ; Sprites

Var CurrAnimFrame      ; Current frame in the alien
                       ; animation
```

The `Aliens []` array stores the 12 on-screen alien sprites. It's declared with 60 elements so that each of the 12 sprites can store its five fields (12 * 5 = 60). `CurrAnimFrame` tracks the current frame of the animation, which continually cycles from 0 to `ALIEN_MAX_FRAME` to simulate a constantly spinning object.

Init ()

The `Init ()` function is responsible for initializing the rest of the script, and in the case of XtremeScript, for defining the constants as well. Beyond this, its main job is cycling through the

Aliens [] array and updating each sprite's pseudo-structure. It also resets CurrAnimFrame to zero. Remember, XtremeScript variables are not initialized and therefore contain unpredictable garbage values until they're explicitly defined.

The process of initializing the Aliens [] array is simple but may not appear immediately straightforward. Because you're working with a one-dimensional array, each alien appears at its index multiplied by five. Therefore, in addition to maintaining an alien counter that increments from 0 to 11 (for the 12 aliens), you also need a separate counter that is incremented by 5 at each iteration of the loop, so you can keep track of the current alien's base index. From this point, each "field" of the alien's simulated structure is just an offset applied to the base address. The alien's X component resides at BaseIndex, the Y is stored at BaseIndex + 1, XVel is at BaseIndex + 2, and so on.

Let's start at the top of the function:

```
Func Init
{
    ; ---- Declare locals
    ; Counters
    Var CurrAlienIndex
    Var CurrArrayIndex

    ; Alien array element fields
    Var X
    Var Y
    Var XVel
    Var YVel
    Var SpinDir
```

This section of the code declares the local variables you'll be using for the rest of the function. CurrAlienIndex is used in the Aliens [] initialization loop to keep track of the current alien, whereas CurrArrayIndex is used to point to the current element within the array. X, Y, XVel, Yvel, and SpinDir are used to temporarily store the values of each field. You'll see more of how these are used as you move through the function.

The next step is defining each of the constants:

```
; ---- Initialize the "constants"
Mov         ALIEN_COUNT, 12
Mov         MIN_VEL, 4
Mov         MAX_VEL, 16
Mov         ALIEN_WIDTH, 128
Mov         ALIEN_HEIGHT, 128
Mov         HALF_ALIEN_WIDTH, ALIEN_WIDTH
Div         HALF_ALIEN_WIDTH, 2
Mov         HALF_ALIEN_HEIGHT, ALIEN_HEIGHT
Div         HALF_ALIEN_HEIGHT, 2
Mov         ALIEN_FRAME_COUNT, 32
Mov         ALIEN_MAX_FRAME, ALIEN_FRAME_COUNT
Dec         ALIEN_MAX_FRAME
Mov         ANIM_TIMER_INDEX, 0
Mov         MOVE_TIMER_INDEX, 1
```

Next up is the definition of the globals. Aside from the `Aliens []` array, which I'll talk about next, this just means setting `CurrAnimFrame` to zero:

```
; Set the current animation frame to zero
Mov         CurrAnimFrame, 0
```

The `Aliens []` array is all that remains. You start by setting both `CurrAlienIndex` and `CurrArrayIndex` to zero, and declare a label to represent the top of the loop:

```
; ---- Initialize the alien array
Mov         CurrAlienIndex, 0
Mov         CurrArrayIndex, 0
InitLoopStart:
```

You're now inside the loop, so you can start initializing the current alien's fields. This is done with two calls to `GetRandomNumber ()`, one of the host API functions defined previously. You'll want to pass it the dimensions of the screen, minus the halved width of the alien head, so the aliens will all appear in valid places, so these values must be pushed onto the stack (and in the proper order):

```
; ---- Initialize the current alien

; Set the X, Y location
Push        0
Mov         X, 639
Sub         X, HALF_ALIEN_WIDTH
Push        X
CallHost    GetRandomNumber
Mov         X, _RetVal

Push        0
Mov         Y, 479
Sub         Y, HALF_ALIEN_HEIGHT
```

```
Push          Y
CallHost      GetRandomNumber
Mov           Y, _RetVal

Mov           Aliens [ CurrArrayIndex ], X
Inc           CurrArrayIndex
Mov           Aliens [ CurrArrayIndex ], Y
Inc           CurrArrayIndex
```

The `GetRandomNumber ()` function was specifically written to read its parameters in reverse order; that is, `Y` is considered parameter 0, whereas `X` is considered parameter 1. This affords you, the caller, the luxury of passing the parameters in the natural `X, Y` order. Notice also that you use `Inc` to increment `CurrArrayIndex` after setting each field. This allows you to be sure that the next field you access will be the right one. Also, by the time you're done with all five fields, `CurrArrayIndex` will be automatically positioned at the base index of the next alien. This means you don't have to explicitly add five after each iteration of the loop.

The X and Y velocities are then set, using the same technique described previously. The only major difference here is that `MIN_VEL` and `MAX_VEL` are passed to `GetRandomNumber ()`:

```
; Set the X and Y velocities
Push          MIN_VEL
Push          MAX_VEL
CallHost      GetRandomNumber
Mov           XVel, _RetVal

Push          MIN_VEL
Push          MAX_VEL
CallHost      GetRandomNumber
Mov           YVel, _RetVal

Mov           Aliens [ CurrArrayIndex ], XVel
Inc           CurrArrayIndex
Mov           Aliens [ CurrArrayIndex ], YVel
Inc           CurrArrayIndex
```

Lastly, the alien's spin direction is set. This determines which direction he'll spin as he bounces around:

```
       ; Set the spin direction
       Push          0
       Push          2
```

```
        CallHost     GetRandomNumber
        Mov          SpinDir, _RetVal

        Mov          Aliens [ CurrArrayIndex ], SpinDir
        Inc          CurrArrayIndex

        ; ---- Move to the next alien
        Inc          CurrAlienIndex
; Keep looping until the last alien is reached
JL          CurrAlienIndex, ALIEN_COUNT, InitLoopStart
```

After the last increment of `CurrArrayIndex`, you'll be at the first element of the next alien, which means you can finish the loop by simply incrementing `CurrAlienIndex`. This value is then compared to `ALIEN_COUNT`, the total number of aliens in the scene, to determine whether to jump back to the top of the loop.

HandleFrame ()

The second and final function defined in the script is responsible for handling each frame by drawing it to the back buffer, blitting the final result to the screen, and moving everything around. This function can be boiled down to two main loops: the first loop draws each of the 12 sprites to the screen, whereas the second moves it along its path based on its velocity and checks for collisions.

Drawing and Blitting the Frame

The first of the two tasks performed by `HandleFrame ()` is drawing and blitting the frame to the screen. This starts with a host API call to the `BlitBG ()` function:

```
CallHost     BlitBG
```

The next step is cycling through each of the 12 alien sprites and blitting them to the screen. Again, the traversal of the `Aliens []` array is dependent on two separate indexes: one for determining the current alien, and one for tracking the current physical field. Let's look at the first block of the code, which starts the loop and reads the alien's X, Y coordinates from the array:

```
Mov          CurrAlienIndex, 0
Mov          CurrArrayIndex, 0

DrawLoopStart:
      ; Get the X, Y location
      Mov           X, Aliens [ CurrArrayIndex ]
      Inc           CurrArrayIndex
      Mov           Y, Aliens [ CurrArrayIndex ]
```

Notice that the second Mov instruction isn't followed by an Inc. This is because when drawing the sprites, you don't need to know their velocities. All you care about is their X, Y locations, which reside within the pseudo-structure at offsets 0 and 1, and the direction in which they're spinning, which is found at offset 4. Because of this, offsets 2 and 3 are of no use and must be skipped. Therefore, after the first Inc, you move from offset 0 to 1. Because the next offset of interest is 4, you need to use the Add instruction to move ahead by three elements:

```
       ; Get the spin direction and determine the final frame
       ; for this sprite based on it
       Add          CurrArrayIndex, 3
       Mov          SpinDir, Aliens [ CurrArrayIndex ]
       Inc          CurrArrayIndex
       JE           SpinDir, 1, InvertFrame
       Mov          FinalAnimFrame, CurrAnimFrame
       Jmp          SkipInvertFrame
InvertFrame:
       Mov          FinalAnimFrame, ALIEN_MAX_FRAME
       Sub          FinalAnimFrame, CurrAnimFrame
```

This block of code determines which frame should be drawn for this particular sprite based on its spin direction. The basic algorithm here is that if the spin direction is set to zero, the value of CurrAnimFrame is used. Otherwise, the value of CurrAnimFrame is "inverted" by subtracting it from ALIEN_MAX_FRAME, which, in effect, causes the animation to run in reverse and thus make the alien appear as if he's spinning in the opposite direction. The pseudo-code for this process looks like this:

```
if ( CurrAnimFrame == 0 )
    AnimFrame = CurrAnimFrame;
else
    AnimFrame = ALIEN_MAX_FRAME - CurrAnimFrame;
```

Based on this, combined with your understanding of how if is represented in assembly, the previous assembly code should make sense. The last block of code blits the current sprite using the X, Y coordinates you read from the Aliens [] array, along with the animation frame you calculated based on CurrAnimFrame and the alien's spin direction:

```
      ; Blit the sprite
      Push          FinalAnimFrame
      Push          X
      Push          Y
      CallHost      BlitSprite
      ; Move to the next alien
      Inc           CurrAlienIndex
; Keep looping until the last alien is reached
JL            CurrAlienIndex, ALIEN_COUNT, DrawLoopStart
```

Once the frame drawing process is complete, you can call the host API function `BlitFrame ()` to blit the final frame to the screen:

```
; ---- Blit the completed frame to the screen
CallHost   BlitFrame
```

Updating the Sprites and Animation

The second phase of `HandleFrame ()` is updating the animation, moving the sprites along their paths, and checking for collisions with the boundaries of the screen.

```
       Push         ANIM_TIMER_INDEX
       CallHost     GetTimerState
       JE           _RetVal, 0, SkipIncFrame

       Inc          CurrAnimFrame
       JL           CurrAnimFrame, ALIEN_MAX_FRAME, SkipWrapFrame
       Mov          CurrAnimFrame, 0
SkipWrapFrame:
SkipIncFrame:
```

NOTE

Notice that both `SkipWrapFrame` and `SkipIncFrame` point to the same instruction, and could therefore be condensed into a single label. I chose to keep them separate for the purpose of readability, however, because they're the targets of two separate jumps in two separate contexts. It would be a lot less clear if these two unrelated processes (checking the animation timer and checking the frame wraparound) both jumped to the same place. Furthermore, because both of these labels are translated to the same target instruction index and subsequently discarded by the assembler anyway, you don't incur a runtime performance hit or any other form of overhead. For this reason, I suggest using multiple labels to enhance readability, even in production code.

Updating the animation involves the script's first encounter with timers, so the first step is pushing `ANIM_TIMER_INDEX` onto the stack and calling `GetTimerState ()`. This will return the status of the animation timer, which you can use to determine whether the frame needs to be updated. If not, you jump to `SkipIncFrame`, which skips the frame increment. Otherwise, the frame is incremented with an `Inc` instruction. However, you need to wrap the frame increment around to zero once it reaches ALIEN_MAX_FRAME, so after each increment you compare the new frame to the maximum. If it's less, a jump is made to SkipClipFrame, which prevents the frame index from wrapping around. Otherwise, you set it to zero.

The last major task is moving the sprites along their paths, which is done in sync with the movement timer. Therefore, this code begins with another host API call to GetTimerState (), this time with the MOVE_TIMER_INDEX:

```
; ---- Move the sprites along their paths
Push         MOVE_TIMER_INDEX
CallHost     GetTimerState
JE           _RetVal, 0, SkipMoveSprites

Mov          CurrAlienIndex, 0
Mov          CurrArrayIndex, 0

MoveLoopStart:
```

Of course, CurrAlienIndex and CurrArrayIndex are reset to zero as well, because this is a new, separate loop. Once inside the loop, the first order of business is reading the X, Y location and X, Y velocities of the current sprite:

```
; Save the base array index of the element so you can access it later
Push         CurrArrayIndex

; ---- Update the sprites

; Get the X, Y location
Mov          X, Aliens [ CurrArrayIndex ]
Inc          CurrArrayIndex
Mov          Y, Aliens [ CurrArrayIndex ]
Inc          CurrArrayIndex

; Get the X and Y velocities
Mov          XVel, Aliens [ CurrArrayIndex ]
Inc          CurrArrayIndex
Mov          YVel, Aliens [ CurrArrayIndex ]
Inc          CurrArrayIndex

Add          X, XVel
Add          Y, YVel
```

Strangely, the first instruction in this block of code pushes CurrArrayIndex onto the stack. You'll see why this is done shortly. For now, the real purpose of this code is setting the X, Y, Xvel, and YVel locals with the appropriate values. XVel and YVel are then added to X and Y, respectively, which moves the sprite along its path.

Now that you've moved the sprite, you need to make sure it hasn't gone past any boundaries. If it has, you register this as a collision by inverting the velocity corresponding to the axis on which the collision occurred. So, if the sprite's Y coordinate is suddenly less than 0, the Y velocity's sign is inverted so the next frame will cause the sprite to move the opposite direction. The one extra detail here is that the boundaries are not 0, 0, and 639, 479. Rather, half of the sprite's width is subtracted from zero and 639, and half of the sprite's height is subtracted from 0 and 479. This effectively lets the sprite's move partially off-screen on all boundaries, which allows the moment of impact to be centered within the sprite, rather than in one of its corners. Here's the code:

```
    ; ---- Determine if a boundary was hit
    Mov          BoundX, 0
    Sub          BoundX, HALF_ALIEN_WIDTH
    JG           X, BoundX, SkipX0VelFlip
    Neg          XVel
SkipX0VelFlip:
  Mov          BoundX, 640
  Sub          BoundX, HALF_ALIEN_WIDTH
  JL           X, BoundX, SkipX1VelFlip
  Neg          XVel
SkipX1VelFlip:
  Mov          BoundY, 0
  Sub          BoundY, HALF_ALIEN_HEIGHT
  JG           Y, BoundY, SkipY0VelFlip
  Neg          YVel
SkipY0VelFlip:
  Mov          BoundY, 480
  Sub          BoundY, HALF_ALIEN_HEIGHT
  JL           Y, BoundY, SkipY1VelFlip
  Neg          YVel
SkipY1VelFlip:
```

It's a simple matter of comparing X and Y to the values placed in BoundX and BoundY. You use the BoundX and BoundY locals so you can perform the subtraction of HALF_ALIEN_WIDTH from each boundary. An obvious (albeit slight) optimization is to store these values in constants, but I think this helps to more clearly illustrate the algorithm. If sprite's X or Y location is beyond its respective boundary, its corresponding velocity is inverted with the Neg instruction, which flips its sign. Otherwise, the Neg is jumped past to the nearest Skip*VelFlip label.

Now that you have the updated sprite locations and velocities calculated in the local variables, you need to store them in the `Aliens []` array so they'll be available for the next frame. However, after all of the array reading you've done, `CurrArrayIndex` has been incremented beyond the base index of the alien. Because you need to write back to the `X`, `Y`, `Xvel`, and `YVel` fields, you need to restore the base index. This is why you pushed it onto the stack originally; you can now simply pop it off, back into `CurrArrayIndex`, and you're ready to go:

```
    ; --- Restore the base index and write the updated values
    Pop      CurrArrayIndex

    Mov      Aliens [ CurrArrayIndex ], X
    Inc      CurrArrayIndex
    Mov      Aliens [ CurrArrayIndex ], Y
    Inc      CurrArrayIndex

    Mov      Aliens [ CurrArrayIndex ], XVel
    Inc      CurrArrayIndex
    Mov      Aliens [ CurrArrayIndex ], YVel
    Add      CurrArrayIndex, 2

    ; Move to the next alien
    Inc      CurrAlienIndex
; Keep looping until the last alien is reached
JL      CurrAlienIndex, ALIEN_COUNT, MoveLoopStart
```

And there you have it. The base index is restored, the relevant fields are written back to the array, and the loop moves on. This wraps up `HandleFrame ()`, and the script in general, for that matter.

Aside from walking you through the development of this script, this section was intended to show you first hand that writing scripts in pure, hand-written assembly can be a tedious process. The logic implemented here would be considerably more compact and concise if it was expressed in a high-level language, which is of course such a language's primary advantage. Scripting, by its very nature, is usually meant to be abstract and simplified. Assembly-style scripting is therefore not very conducive to this philosophy. As a script writer, your focus should be spent on your script's logic, not its implementation.

Of course, by the same token, scripting must be fast if it has any chance of keeping up with a game engine. Because of this, being comfortable with assembly can be a valuable skill, especially in the case of performance-critical scripts that will run on a frequent, or even frame-by-frame, basis.

The XVM assembly version of the script will be saved as `asm_script.xasm` and assembled by XASM to `asm_script.xse`.

The High-Level XtremeScript Script

XtremeScript is very similar to C in most respects, which means that writing the script you labored over in the last section will be a breeze this time through. Most of C's familiar amenities, such as `while` loops, expressions, and so on, are readily at your disposal. You can capitalize on these features thoroughly to express the script's logic in a far more succinct manner. This section is shorter as well; because I've already discussed the logic and algorithms behind the script, you can simply focus on the code itself this time around.

Constants and Globals

The high-level version of the script uses the same constants and globals as its assembly counterpart, and because even the syntax of such declarations is the same in both languages (minus the addition of semicolons in XtremeScript, and the fact that keywords are written entirely in lowercase to mimic the C convention), there's no need to waste the space reprinting them here.

Importing the Host API

Unlike XVM assembly, which can differentiate between a script call and a host API call by simply determining whether `Call` or `CallHost` was used, XtremeScript allows all function calls to be expressed with the same syntax, and thus needs some explicit cues from the users to determine which calls are which. So, the `host` keyword is used to import the host API's functions:

```
host GetRandomNumber ();
host BlitBG ();
host BlitSprite ();
host BlitFrame ();
host GetTimerState ();
```

Init ()

Let's jump right into the `Init ()` function. As was the case last time, you begin by defining the script's constants, because even XtremeScript can't do so in the global scope:

```
func Init ()
{
      // ---- Initialize the "constants"
      ALIEN_COUNT = 12;
      MIN_VEL = 4;
      MAX_VEL = 16;
      ALIEN_WIDTH = 128;
      ALIEN_HEIGHT = 128;
```

```
       HALF_ALIEN_WIDTH = ALIEN_WIDTH / 2;
       HALF_ALIEN_HEIGHT = ALIEN_WIDTH / 2;
       ALIEN_FRAME_COUNT = 32;
       ALIEN_MAX_FRAME = ALIEN_FRAME_cOUNT - 1;
       ANIM_TIMER_INDEX = 0;
       MOVE_TIMER_INDEX = 1;
```

The first noteworthy difference between what's going on here and what went on the assembly version is the expressions used to define the constants. In assembly, the definition of `HALF_ALIEN_WIDTH` as `ALIEN_WIDTH` divided by two required multiple instructions, whereas you can do it all in a single line here.

The animation frame counter is then set to zero, which, no matter what language you're using, is a simple affair:

```
// ---- Initialize the globals
CurrAnimFrame = 0;
```

The `Aliens []` array is initialized next, which is where XtremeScript's high-level, C-style syntax really gets a chance to shine. Notice how much shorter and clearer everything is, now that you're using a language with explicit support for loops, function calls, and expressions:

```
// ---- Initialize each alien

CurrAlienIndex = 0;
CurrArrayIndex = 0;
while ( CurrAlienIndex < ALIEN_COUNT )
{
       // Set the X, Y location
       X = GetRandomNumber ( 0, 639 - ALIEN_WIDTH );
       Y = GetRandomNumber ( 0, 479 - ALIEN_HEIGHT );

       // Set the X, Y velocity
       XVel = GetRandomNumber ( MIN_VEL, MAX_VEL );
       YVel = GetRandomNumber ( MIN_VEL, MAX_VEL );

       // Set the spin direction
       SpinDir = GetRandomNumber ( 0, 2 );

       // Write the values to the array
       Aliens [ CurrArrayIndex ] = X;
       Aliens [ CurrArrayIndex + 1 ] = Y;
```

```
        Aliens [ CurrArrayIndex + 2 ] = XVel;
        Aliens [ CurrArrayIndex + 3 ] = YVel;
        Aliens [ CurrArrayIndex + 4 ] = SpinDir;

        // Move to the next alien
        CurrAlienIndex += 1;
        CurrArrayIndex += 5;
}
```

Although the assembly version of the loop is using `Inc` and `JL` instructions to regulate iterations, `while` allows you to do everything with a single conditional expression. Furthermore, you no longer have to deal with the intricacies of pushing parameters and dealing with the `_RetVal` register. Instead, everything is done with a traditional, C-style function call. Lastly, the interaction with the `Aliens []` array is far simpler and more straightforward as well. Now you can directly embed the addition of the offset into the expression, which is not only clearer, but also temporary. Unlike `Inc`, adding an offset to `CurrArrayIndex` only affects its value within the context of the expression, saving you the trouble of having to incrementally step through the array after each read and write.

HandleFrame ()

Aside from declaring the pertinent local variables and such, the first thing `HandleFrame ()` does is draw the next frame to the back buffer and blit it to the screen. Here's the entire frame-drawing process:

```
BlitBG ();
// ---- Blit each sprite
CurrAlienIndex = 0;
CurrArrayIndex = 0;
while ( CurrAlienIndex < ALIEN_COUNT )
{
        // Get the X, Y location
        X = Aliens [ CurrArrayIndex ];
        Y = Aliens [ CurrArrayIndex + 1 ];

        // Get the spin direction and determine the final
        // frame for this sprite based on it
        SpinDir = Aliens [ CurrArrayIndex + 4 ];
        if ( SpinDir )
            FinalAnimFrame = ALIEN_MAX_FRAME - CurrAnimFrame;
        else
            FinalAnimFrame = CurrAnimFrame;

        // Blit the sprite
        BlitSprite ( FinalAnimFrame, X, Y );

        // Move to the next alien
        CurrAlienIndex += 1;
        CurrArrayIndex += 5;
}

// Blit the completed frame to the screen
BlitFrame ();
```

Again, you can't help but appreciate the huge gains in clarity and brevity that are attributed to high-level code. In only a few lines, you're expressing the exact logic necessary to draw each sprite in the `Aliens []` array and blit the results to the screen. Notice that now, the logic for calculating the final animation frame based on `SpinDir` is almost identical to the pseudo-code example listed in the assembly section. Also, look at how much easier it is to access arbitrary fields of the pseudo-structure; you can simply say `Aliens [ CurrArrayIndex + 4 ]` to access the fourth offset past the base index.

And, of course, the final step is updating the animation, moving everything around, and taking collisions into account. Because this step requires the most conditional logic out of any major task in the script, this is where you'll notice the biggest differences between the assembly version and the high-level version. Here's the code for incrementing the current animation frame and wrapping it around to zero if necessary:

```
// Increment the current frame in the animation
if ( GetTimerState ( ANIM_TIMER_INDEX ) )
{
        CurrAnimFrame += 1;
        if ( CurrAnimFrame >= ALIEN_FRAME_COUNT )
            CurrAnimFrame = 0;
}
```

How simple is that? Two `if`s is all it takes to get the job done. And now, for the crown jewel of it all, check out the code for moving the sprites around and handling collisions:

```
// Move the sprites along their paths
if ( GetTimerState ( MOVE_TIMER_INDEX ) )
{
    CurrAlienIndex = 0;
    CurrArrayIndex = 0;
    while ( CurrAlienIndex < ALIEN_COUNT )
    {
           // Get the X, Y location
           X = Aliens [ CurrArrayIndex ];
           Y = Aliens [ CurrArrayIndex + 1 ];

           // Get the X, Y velocities
           XVel = Aliens [ CurrArrayIndex + 2 ];
           YVel = Aliens [ CurrArrayIndex + 3 ];

           // Increment the paths of the aliens
           X += XVel;
           Y += YVel;
           Aliens [ CurrArrayIndex ] = X;
           Aliens [ CurrArrayIndex + 1 ] = Y;

           // Check for wall collisions
           if ( ( X > 640 - HALF_ALIEN_WIDTH ) || ( X < -HALF_ALIEN_WIDTH ) )
                 XVel = -XVel;
           if ( ( Y > 480 - HALF_ALIEN_HEIGHT ) || ( Y < -HALF_ALIEN_HEIGHT ) )
                 YVel = -YVel;
           Aliens [ CurrArrayIndex + 2 ] = XVel;
           Aliens [ CurrArrayIndex + 3 ] = YVel;

           // Move to the next alien
           CurrAlienIndex += 1;
           CurrArrayIndex += 5;
     }
}
```

Pretty slick, huh? The once-lumbering conditional logic has been reduced to two `if`s, whose expressions now consist of two nested sub-expressions separated by the `||` operator. Remember, because you took the simplified route and generalized the relational and logical operators into the same level of precedence, it's important to use parentheses to assert the proper level of priority. You want to evaluate the relational > and < operators first, and then `||` the results. Either way, though, this is a huge syntactic improvement over assembly. XtremeScript is clean, clear, and easy to use.

The XtremeScript version of the script is saved as `script.xss` and compiled by XSC to `script.xse`.

The Results

Unfortunately, XtremeScript's impressive usability comes at a significant price. The simple fact of the matter is that in the absence of any form of code optimization on behalf of the compiler, the high-level equivalent to a hand-coded assembly script will be hugely inefficient and run at a fraction of the speed. You've seen the evidence for this throughout the chapter—the amount of stack manipulation associated with the compilation of even the simplest expression can be staggering.

This is the reason I wanted to make sure you've seen and understood the coding of this simple demo in both XtremeScript and XVM assembly. By compiling the high-level demo with the compiler's `-A` switch, you can compare the compiler's assembly output to your own assembly code, and will undoubtedly notice a truly massive difference. I can't even begin to list it here in the book, because it would consume far too many pages. And of course, the reality of the results is undeniable when the two demos run in succession. The assembly version is definitely fast enough for most purposes, but the code generated by XSC will need a lot of work before it can be easily applied to real-world game projects.

Optimization

As I've mentioned before, and will mention again, optimization is a hugely complex, math-heavy topic. There are countless reasons why it's necessarily out of this book's league. All is not lost, however. This section provides a brief rundown of some possible avenues to follow if you'd like to attempt to optimize the XSC parser and code emitter.

When you really get down to it, what are the main elements of the XtremeScript language? There are functions, variables, `if` and `while` constructs, and that sort of thing. Everything else, really, falls into the domain of expressions. If you take the time to analyze XSC's assembly output of this demo, you'll find that things like function calls and `if` and `while` are implemented in a rather efficient manner, which shouldn't be surprising. After all, all a function call consists of is the pushing of values onto the stack and the execution of the `Call` instruction. Function calls don't get any simpler than that, and that's exactly what XSC produces. `if` and `while` are also quite simple; they're nothing but jumps and labels. Implementing an `if` or `while` loop by hand in assembly would vary only slightly from the raw output of the compiler.

What ultimately slows everything down are the *expressions* that drive everything. The *expressions* that represent the parameters pushed onto the stack before a function call. The expression that defines the condition by which `if` will execute its true or false block, as well as the expression that a `while` loop uses to determine whether to continue iterating. Expressions are unrelated to the constructs in which they're used, but due to their ubiquity, are unavoidable. In short, if you want to increase the compiler's output quality, expressions are public enemy number one.

Fortunately, it won't take a particularly massive amount of brainpower to determine at least basic optimizations. Any ad hoc optimization you can notice will help, so give it a shot! To get you started, here are a few general tips to keep in mind:

- The stack is utilized to an almost criminal degree when parsing an expression, which is the primary reason that everything is running so slowly. Looking through the demo's assembly output, you'll find that there are even times when values are pushed onto the stack, only to be immediately popped off. This is obviously unnecessary; the trick is getting the compiler to notice this fact as well.
- The stack can often be bypassed entirely. In many cases, such as direct assignment and other simple expressions, values can be directly loaded into `_T0` and `_T1`, or even directly into their destination variables themselves.
- Different types of expressions can be parsed and converted to assembly in different ways. For example, an expression with only two values can be parsed without the stack entirely; the operands can instead go directly into `_T0` and `_T1`. The negation of a value can also be done in many cases by simply loading `_T0` directly and using the `Neg` instruction. The key is noticing patterns or other red flags in an expression as a whole *before* parsing it. You might want to consider the idea of storing each statement in a local I-code buffer before the parsing phase, so you can attempt to notice certain types of expressions and take their specific forms into account.
- I implemented XSC with `_T0` and `_T1` because binary operators will never require more than two operands. Imagine, instead, however, defining a whole array of temporary register variables, and using them instead of the stack for most operations. This would allow operands and values to move directly into variables rather than flowing in and out of the stack, and thus allow the execution to perform the operation faster. Ultimately, this could result in huge speed gains. If a large expression exhausts the array, you can always fall back on the stack, but because most expressions are rather short (using only a handful of operators), the array would handle most situations nicely.

As it stands, however, the compiler is definitely too slow for certain purposes. For example, it wouldn't be a good idea to run an XSC-compiled script on a per-frame basis in a high-speed first-person shooter or racing game.

This doesn't preclude the use of the high-level scripts in all situations, however. RPG cut scenes and dialogue sequences are a great example of an application that isn't speed critical and often requires a great deal of logic to be performed. It's often necessary to check large numbers of game flags and their relationships when managing the flow and progression of an RPG's more cinematic elements, which makes them a prime candidate for XtremeScript's graceful ability to handle complicated logic easily. Puzzle games, adventure games, and non-real time strategy games can also benefit from compiled scripts in the same way. Such games often idle for long periods of time, waiting for the player to react, and also involve lots of complex logic. XtremeScript would once again provide a perfectly adequate solution in these cases.

Summary

This is it! After all the buildup and anticipation, you've *finally* created a real, fully working scripting system. The completion of this module has something of a domino effect on the system overall—by completing the parser module, you subsequently complete the compiler, which, being the last component of XtremeScript, completes the entire system overall.

What you've done here is no trivial task. You've designed two complete languages from the ground up—a low-level assembly language, and a high-level, C-style language. You've now implemented them both as well, and created a full-featured, seamlessly embeddable runtime environment in which they can execute. A complete game-scripting toolset is now at your disposal, and you've been there every step of the way (assuming you haven't been skipping around like some degenerate hoodlum).

With custom-built tools this powerful at your fingertips, there are no limits. XtremeScript is easily capable of expressing virtually any form of scripting logic, allowing the characters, weapons, and environments of your games to behave with extreme precision and total control (performance issues notwithstanding). In fact, this is the subject of the next chapter. Now that you're finished with XtremeScript, it's time to put it to use and script a real, complete game with it. You'll see how the scripting of a game project is approached, and learn how to intelligently use the system you've spent so much time developing.

On the CD

This chapter saw you through the development of XtremeScript's parser module, which evolved over the course of four incarnations. Each of these versions is presented separately on the CD in the `Programs/Chapter 15/` folder for you to study and play around with:

- **`15_01/`** contains the initial parser module, which interprets code blocks, empty statements, and the full assortment of XtremeScript declarations, via the `var`, `func`, and `host` keywords.
- **`15_02/`** contains the second parser module, upgraded to support simple expressions in the form of statements.
- **`15_03/`** rounds out expression parsing by further upgrading the parser to support the entire XtremeScript operator set (except for assignment operators), including logical and relational operators.

- `15_04/` is the final and complete parser module, which subsequently completes the compiler. It adds the full range of XtremeScript statements: assignments, loops, branching, and so on.
- `XVM Console/` is a standalone version of the XVM that exposes a simple console output API, used for testing scripts as XSC compiles them. This is also where you'll find the source and executables for the Hello, world! and rectangle demos.
- `Alien Demo/` is the bouncing alien head demo you created to test the scripting system overall. This folder contains both versions of the script—the high-level XtremeScript version and the low-level XVM assembly version. It also contains the compiler-generated .XASM file. In addition, you'll find the executable version of the scripts, and the DirectX/Win32 host application.

Each of the parser modules is accompanied by its own separate compiler framework, making the modules completely self-contained. You can freely run them without the help or presence of the others, allowing you to focus on specific phases of the parser's development.

CHALLENGES

Even in the case of this relatively simplistic implementation, a parser is a complicated piece of software. As such, there are about a million things that can be done differently along the way. Because of this, you'll have plenty of challenges to play with in this chapter, including the handful of small language features that weren't included in the parser module.

- *Beginner:* Using the logic behind `ParseFunc ()`, the function declaration parser, expand the `host` keyword to allow parameters to be defined in between the `(` and `)`, just like a script-defined function. This can come in handy by allowing the compiler to verify the parameter list passed in host API calls.
- *Intermediate:* Expand the `var` declaration to allow a comma-delimited list of variables to be declared at once, like `var X, Y, Z;`. Again, the logic behind `ParseFunc ()` can be duplicated to implement this.
- *Intermediate:* Expand the `var` declaration to allow variables to be defined as they're declared, like `var Pi = 3.14159;`. This can be added easily, mostly by duplicating the logic behind `ParseAssign ()` and merging it with `ParseVar ()`.
- *Advanced:* Implement the `for` loop, possibly with the preprocessing method described earlier.
- *Advanced:* Implement the ++ and `--`, both in the prefix and postfix forms. This isn't quite as easy as it sounds; remember, these operators actually affect the variables themselves, not just their value in a temporary sense. If `Y` ++ appears in an expression, the value of `Y` is permanently incremented.

Part Seven

Completing Your Training

CHAPTER 16

Applying the System to a Full Game

"I told many, many people."
—*Jeremy Goodwin,* Sports Night

XtremeScript is now a finished, ready-to-use scripting system. From start to finish, you've seen how every aspect of each of its three major components—the assembler, virtual machine, and compiler—are assembled. All that's left is applying your work to an actual game, to get a feel for how scripting really works. The process of doing so is the focus of this chapter.

In this chapter, you're going to:

- Design and plan a simple game.
- Discuss the details involved in implementing the game's engine.
- Apply scripting to key elements of the game's design.
- Use the XtremeScript system you've developed over the course of this book to implement these scripted elements.

As you can probably imagine, this chapter is the real payoff. All the technology and theory in the world doesn't matter if it can't be easily and directly applied to a game, which is why this book just wouldn't be complete without coverage of how scripting is actually used.

To do this, you're going to start by designing a simple game, and discussing its development. I'll start with the initial layout and planning stages, and then talk about how its code, graphics, sound, and other assets fit together to create a complete game engine. You'll then augment the game engine by embedding the XtremeScript virtual machine in it, and use the assembler and compiler to write scripts that control the behavior of the game's enemies.

Introducing Lockdown

I wanted to create a game for this chapter that was simple and easy to both implement and describe. On the other hand, however, I wanted something that was interesting and actually somewhat engaging, and more than anything else, needed enough complexity to justify scripting in the first place. For example, although games like *Pong* and *Breakout* are often good ways to illustrate the complete process of designing a game, the opportunities to apply scripting to their logic aren't exactly abundant.

The Premise

What I ended up settling on is a basic but reasonably cool little game called *Lockdown*. The name comes from the fact that it takes place in a prison-like fortress where your goal is to collect four scattered keys and use them to activate some underlying machinery that allows you to escape. Your character is a levitating droid-type thing designed somewhat after the probe droids sent by the Empire to Hoth in *The Empire Strikes Back.* You float around the fortress, picking up keys, and battling your way to freedom. Along the way, other, different colored droids use varying methods of attack to slow you down and ultimately destroy you. I spent about a week developing the game from start to finish; it took a little under seven days to get from the initial ideas to a finished production.

It shouldn't come as a surprise that storyline and setting weren't a big priority. Although I'm normally a huge proponent of immersive, cinematic, story- and character-driven games like *Metal Gear Solid* and *Halo,* the focus here is simply getting something finished and working, so you can test your scripting system on it.

Same Old Story

Speaking of game storylines and settings, I was at E3 this year (2002 at the time of this writing), and I must sadly admit that the game industry overall seems to be in a *huge* storytelling rut. The level of technology that the average game developer can leverage these days is enough to turn even the most "out there" game world into near-perfect reality, but it's as if there's no one with anything original to say anymore. I swear to God, if I hear about *one* more game whose "plot" involves "a once prosperous land that's been ravaged by the forces of darkness," I'm going to throw my computer out the window, shave my head, and join Green Peace. To any developers that may be listening: *the "forces of darkness" need a day off.* Give the dark, demon-ridden medieval setting a rest and try something new. What about a heavily stylized, *Grand Theft Auto*-style game that focuses on the mafia during prohibition? Or perhaps a game based in a futuristic environment—but one that's only marginally more advanced than the present day—like the setting in *Minority Report?* The point is, there are a million unexplored avenues that could be taken when designing a game world and the story that unfolds within it, so *try them.* There's no law stating that every game needs to drop the player hip-deep into skulls and dungeons. The problem with anything under the umbrella of pop culture, including mainstream video games, however, is that people are more interested in following the leader than they are with doing something unique and original. Instead of breaking new ground and challenging ourselves, all we're doing is driving an increasingly tired gimmick into the ground until it reaches critical mass and becomes a joke. Anyway, I just needed to get that out of my system. Now, you can enjoy the rest of the chapter.

Initial Planning and Setup

Lockdown is a simple game, so there wasn't a whole lot that needed to be sorted out beforehand. I had an idea in my head and knew what it took to make it happen. However, it doesn't take much for an attitude like that to degenerate into full-on cockiness, so I decided to avoid the unfortunate fate that waits all unprepared game developers and take the time to do some formal planning.

The planning of a sufficiently simple game can be reduced to the following major steps:

- **Game logic and storyboarding.** Sure, there's a premise, but you're asking for trouble if you write even one line of code before fully understanding every detail of your game. This is done by writing your ideas down in text files, jotting notes on paper, and sketching out concept art and storyboards.
- **Assessing your asset requirements.** A game's *assets* are the media and resources that drive its logic and content. This can range from scripts to sprites to sound samples to CD audio tracks to full motion video. Asset requirements are very specific—saying something like "I need a room with cool lighting" is virtually meaningless. Rather, it's important to articulate your exact requirements down to a near-pixel level. For example, you might instead say "I need a room with cool lighting, so that'll entail a number of full-screen background images for the room itself, a number of frames of animation for doors, and perhaps additional sprites that can be superimposed over the background to represent dynamic wear and effects like bullet holes or track marks." By the time asset planning is finished, you should know exactly how many resource files you'll need and exactly how they'll be arranged and organized.
- **Planning the code.** Once you understand your game to the fullest extent possible, and have laid out exactly what assets you'll need, you're ready to start thinking about code. This phase involves designing the structure of a sprite engine, thinking about how resources will be loaded and stored, and working the role of the scripting system into the grand scheme of things. The result of this phase should be a framework that you can immediately convert to a general code "skeleton," which can then be filled out to create the final game.

Let's now quickly run through what happened during these phases.

Phase One—Game Logic and Storyboarding

The premise of Lockdown has already been established, but it's a complete understanding of the game's details that's truly important. To give you an idea of how vital this distinction is, consider the following. Here, in a single paragraph, is complete synopsis of the Lockdown game.

> *Lockdown takes place in a prison-like fortress inhabited by floating droids. There are three types of these droids, each of which attacks the player in a different way. The player is also a droid, and is equipped with a built-in laser cannon that can be used to ward off the attackers. In addition to destroying the evil droids, the player's goal is to collect four colored keys, each of which resides in one of the fortress's corners, and use them to activate their corresponding key panels in the fortress's center room. When all four panels are activated, the player's droid can escape lockdown and the game has been won.*

Sounds reasonable, right? I mean, I've explained the setting, the player's goal, and the opposing forces, all in reasonable detail, haven't I? Although this would certainly be enough to explain how the game works to a person, it's hardly what the average software engineer would call a "complete specification". Imagine actually sitting down in front of your compiler and attempting to write a game with nothing more than this!

For example, this little synopsis makes no mention of a title screen or interface. For all we know, the game's action begins as soon as the player invokes the executable, and immediately terminates when the objective is fulfilled. We don't know what sort of damage is taken on behalf of both the player and enemy droids when they're attacked. Do they immediately die after one hit, or can they take a bit of punishment before going down for the count? And how exactly *do* they die? Does the droid's machinery fall apart, does it just disappear altogether, or does its destruction result in a violent explosion? We have no idea what these droids are supposed to look like, how exactly the fortress should be designed, or where anything is. We know the keys are found in the corners and must be dropped off in the center room, but we don't know anything about the architecture in between these points. Are they connected with long tunnels, a chain of singular rooms, or perhaps a sewage system?

As you can hopefully see by now, you need a lot more information than you have at this point. Although I won't belabor you with a *complete* game specification, I am going to walk you through enough of the game's details to understand the rest of the chapter and make sense of the overall project.

The Fortress

As has been mentioned, the game takes place within a prison-like fortress that houses a number of keys and the enemy droids that are out to destroy the player. What this fortress actually looks like, however, is important. Because I didn't want to spend any more time than was absolutely necessary, I decided against any form of scrolling and instead took the top-down, 2D, screen-by-screen approach used in games like *The Legend of Zelda*. The benefits of this approach are many; I can focus my graphical efforts on a few full-screen backgrounds, rather than fifty thousand tiles for a scrolling tile engine, the actual coded logic behind screen-by-screen traversal of a game world is much easier, and lastly, it makes the game a bit easier to play. Many top-down scrolling games suffer from the problem of enemies and other hazards "rushing in" from the side of the screen, because the player's view restricts him or her from seeing enough of what's ahead. By the time the player is able to react, these obstacles have already done their damage. By limiting the immediate action to a single screen, the player is always aware of the surroundings and can play accordingly.

This means that the fortress is really just a two-dimensional array of rooms. Because these rooms need to be connected somehow, I decided to give each room four doors; one facing in each cardinal direction. These doors are automatically opened when you approach them, allowing you to zip around the environment without stopping or slowing down. The rooms, when seen altogether, form a modest but reasonable game world that's just large enough to make it worth playing, but small enough to make the game easy to produce. I wanted to make sure I didn't commit to anything that would push the production of the game past a total of six or seven days.

Naturally, the best way to get the idea for the layout of the fortress out of my head and into some tangible form was a quick and dirty sketch, as can be seen in Figure 16.1.

As you can see, the fortress is five rooms wide and five rooms tall. In each corner you'll notice a flat, circular object. You'll also notice that each of these four rooms is labeled *"Key"*. This of course refers to the fact that the four keys players collect throughout the course of the game are stored in these rooms. The circular objects would become the pedestals upon which the keys are stored.

The center room, marked *"Key Room"*, is where the players drop off the keys as they collect them. There are four panels on the floor of this room, each of which of course corresponds to a specific

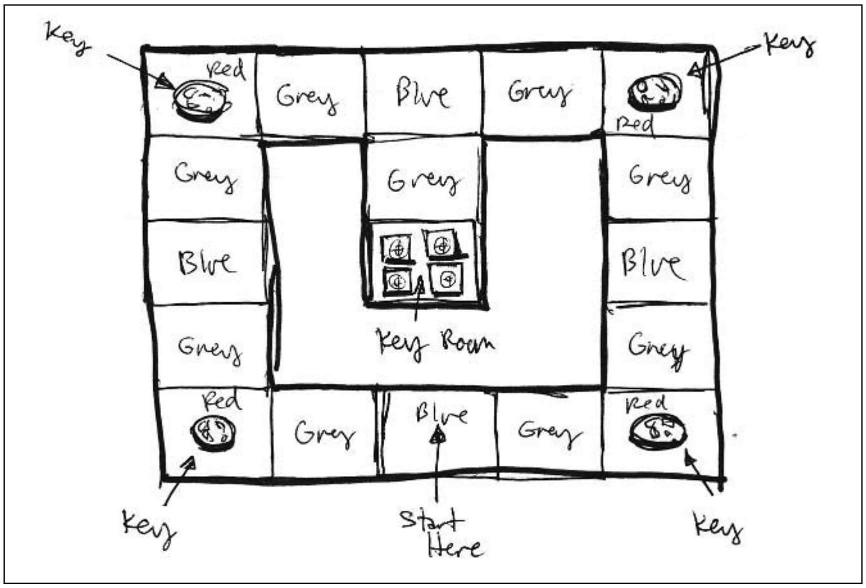


Figure 16.1

A rough sketch of the fortress.

key. Each time a key is used to activate a panel, it lights up with the color of the key. The player wins when all four panels are illuminated.

I should also mention that as an extra atmospheric effect, I decided to make the light in each room flicker at random, resulting in a subtle but effective visual cue in the style of games like *Resident Evil*.

The Enemy Droids

The last detail to cover on the sketch of the fortress map is the fact that every room is marked with one of three colors: blue, grey, and red. These correspond to the colors of the three types of enemy droids that inhabit the fortress. Whenever the player enters a new room, a new random population of droids is spawned to attack the player, and by giving each room a specific droid type, you can "guard" sensitive areas of the game; for example, you can place the most advanced droids in the key rooms, but allow the sophistication of the droids to drop off a bit as the players move farther away from those rooms.

I designed the droids to be simple but self-contained. They're based primarily around a spherical "body", which houses the unit's brain and laser cannon. Jutting out of this central component are three small "grabber claws" that round out the design and make it seem more complete. These design ideas were reflected in more quick-and-dirty sketches, which ended up being the concept art for the exact look of the droid. Figure 16.2 depicts a sketch that represents the nearly final

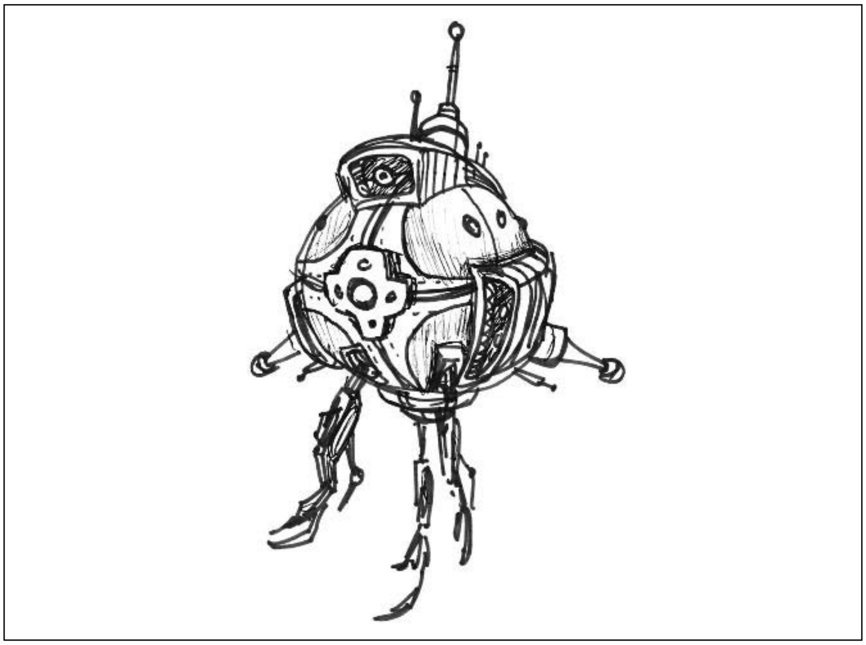


Figure 16.2

A quick-and-rough sketch of the enemy droids.

droid design; I ended up making a number of changes in the final model, but this was a reasonably close approximation.

The aesthetic differences from one droid to another are actually quite simple. In another decision made by deadlines, I decided not to waste the time designing three genuinely unique droid types, and to instead just vary the color. The blue droid is the weakest, the grey droid is more powerful, and the red droid is the deadliest of all. Aside from color, however, the real difference between each droid is its behavior, which is where the scripting system will come in. Each droid will be associated with a particular script, which is executed when that droid is on-screen. Because each room in the fortress will contain only one unique droid type, this means you'll only have one droid-related script running at any one time.

The Blue Droids

The blue droid is the weakest and least intelligent of all three. Its single method attack is moving randomly around the room in a vague attempt to collide with the players. This brings up an important point to remember; the players are damaged by contact with enemy droids. The blue droid appears in the rooms of lowest security—in other words, those that aren't particularly close to more sensitive locations like key rooms.

The Grey Droids

The grey droid is a definite step up from its little blue brother. Although its movement is still more or less random, the grey droid can fire its laser and will do so in the general direction of the player on a frequent basis. Therefore, despite its less-than-brilliant maneuvering, a group of grey droids will bombard the players with lasers and produce a formidable challenge. Grey droids always appear on the outskirts of importance; rather than directly guarding anything, they appear just outside of the rooms that house something important.

The Red Droids

Last up is the red droid, sitting at the top of the fortress food chain. The red droid further improves upon the grey droid by combining its ability to fire its laser with movement that actually makes sense. The red droids constantly reevaluate their location in the room and use that data to move themselves closer to the players. This results in a pack of droids that not only shoot at the players, but follow their movements as well. This makes the red droids the "guardians" of the fortress, which is why they're always found in the key rooms.

The Player

The player is a droid as well, which allows you to reuse the droid design. To differentiate the player from the enemies, however, he's white in color. The player droid is of course controlled by the keyboard, allowing the users to move him around and fire his laser at will. This section will cover the major aspects of controlling the player droid, but the majority of what I'll discuss here applies to the enemies as well. I'll explain this relationship in more detail as the chapter progresses.

Movement and Firing

The two primary actions of the player are moving and firing. The player droid can move in any of eight directions—north, south, east, and west, along with the four diagonals. To cut down on the number of sprites I had to draw, however, I decided to limit the player's firing options—although you can move in eight directions, you can only shoot in the four cardinal directions. It's a lame restriction I know, but it saved some time.

The Laser

Speaking of firing, the lasers themselves are more or less what you'd expect; long strips of color that move quickly through the room and cause damage to whatever they run into. Because both the player and enemies have the ability to shoot lasers, I took another cue from *Star Wars* and made the player's laser a yellowish green, and the enemies' a pinkish red. They also make different sounds.

Lasers can move in any of the four cardinal directions, and have a long enough range to always cross the width or height of the room, regardless of where they were fired from. The only thing that can stop them is a collision with a droid.

Graphically, the laser is represented with a number of sprites. Right off the bat, there's the issue of drawing the laser for each of the four directions (although it could be done with only two). In addition, however, I threw in an extra effect that causes the beam to quickly transform from a bulbous, blob-like mass as it's first fired to a thin, focused beam. To do this, four sprites were needed for each direction, which I sketched out in Figure 16.3.

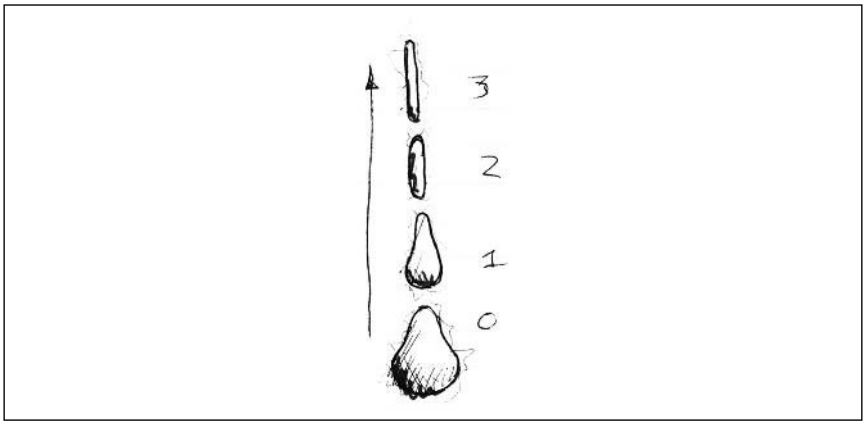


Figure 16.3

A sketch of the four frames of animation depicting a focusing laser.

Damage and Destruction

Naturally, a big part of the game is taking damage and occasionally being destroyed. Because of this, each droid in the game maintains an "energy level" that determines how close it is to destruction. The maximum amount of energy allowed is eight points. Furthermore, because the game doesn't feature power-ups of any kind, I decided to constantly replenish the player droid's energy on a slow but regular basis. Once approximately every three seconds, the player will recover another point of energy. As you'll see, though, this hardly makes the game easy when the action gets hot and heavy enough.

Eventually, however, many droids will meet an untimely demise. This is handled with both the visual and auditory aspects of an explosion; a fiery animation replaces the droid's on-screen presence, accompanied by the proper sound effects.

NOTE

You'll also notice as you play that droids are immediately repopulated if you leave and reenter the room after destroying them. Although this doesn't make a great deal of logical sense, and could be considered a minor annoyance in some cases, it adds an extra challenge when the player has to backtrack. Besides, if it's good enough for *Castlevania: Symphony of the Night*, it's good enough for Lockdown.

The Keys

The last major in-game components of Lockdown are the keys. There are four keys in all, each of which is necessary to complete the game. As has been mentioned a number of times already, the keys are stored in the fortress's four corner rooms. The keys are differentiated by color—red, yellow, green, and blue. These colors correspond to the colored lights on the four key panels located in the central key room. The object of the game is to carry each of the four keys, in any arbitrary order, into the key room and use them to activate their respective panels.

The visual design of the keys went back and forth a number of times as the game progressed, starting with the handful of initial ideas in Figure 16.4. I started out with a more traditional jailor's key style, but with something of a radial, Aztec spin. I bounced around through some further ideas, one of which reminded me of some of the newer keys they're using for luxury cars these days. I ended up deciding on a much different design, however, looking more like some sort of abstract emblem than a key.

Although the final design of the key didn't show up in any of the sketches, Figure 16.5 presents one that came pretty close.

Lastly, there were the issues of the key pedestal and the colored panels. Fortunately, these were much simpler to design and were done immediately, as shown in Figure 16.6.

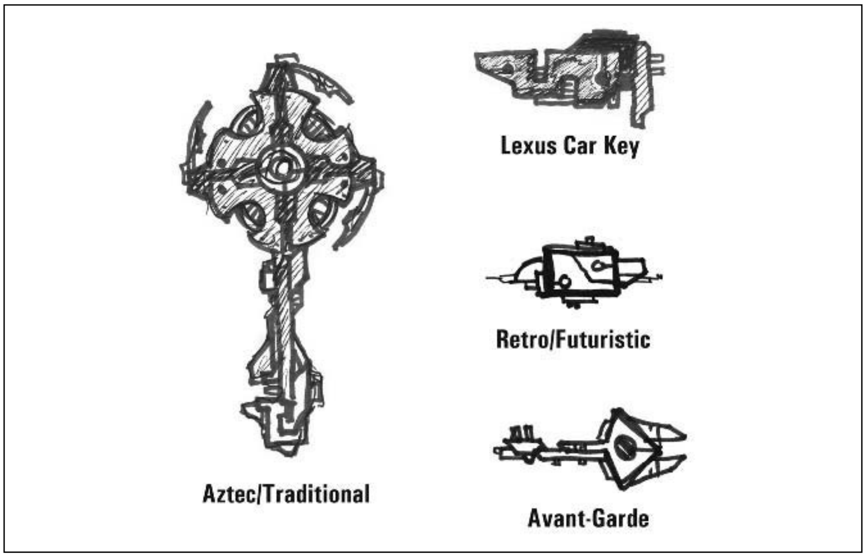


Figure 16.4

Early concept sketches for the keys.

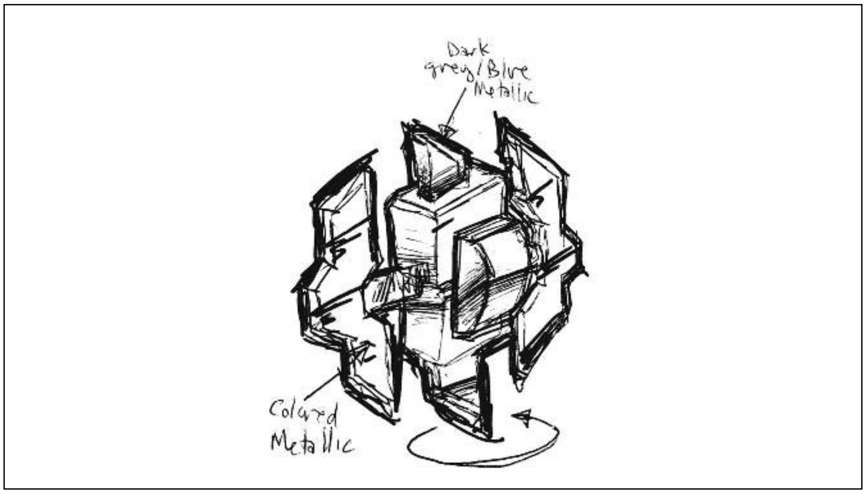


Figure 16.5

The last concept sketch for the key design, coming close to the final version.

The Overall Package

Lastly, it was important to sketch out what the average game screen would look like, especially with the interface superimposed over it. The end result is what I call "the overall package," and attempts to prototype what the game will actually look like when running. Figure 16.7 is a sketch of the overall package I was going for. Note the minimalist interface; all you need is a readout of your energy and the keys you've collected.

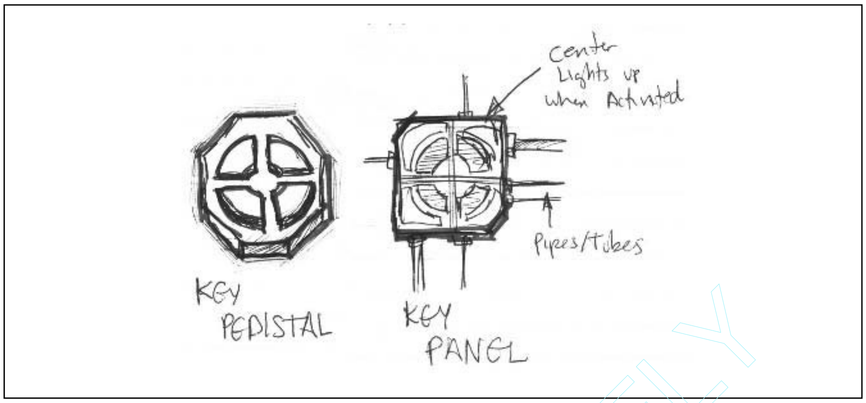


Figure 16.6

The concept sketches for the key pedestals and panels.

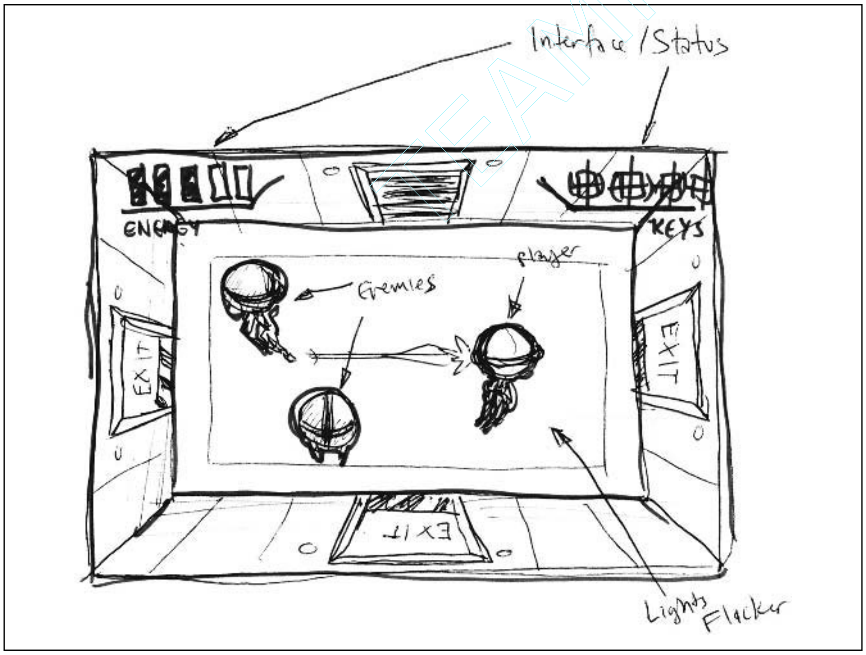


Figure 16.7

The "overall package" of the game—the interface and a typical game screen.

Phase Two—Asset Requirement Assessment

So you know what everything needs to look like, more or less. The reality of graphics, however, is that even simple objects are often reduced to countless individual bitmaps, all of which must be stored and managed somehow.

Ultimately, the assets of Lockdown were reduced to three major groups—graphics, sound, and scripts.

NOTE

The wrapper API I developed for use with this book was designed to be as simple as possible. This meant that using *bitmap templates*, a common technique wherein multiple individual sprites and bitmaps (often frames of an animation) are stored in a single file, was foregone in favor of simply loading individual bitmaps directly from their files and into memory. The result of this decision is that the `Gfx/` directory of Lockdown has quite a few more files than it would have otherwise. To make things manageable, however, I've enforced a strict and verbose directory structure that keeps the files organized.

Graphics

The graphics of the game are stored in the `Gfx/` directory, so feel free to check out the individual .BMP files as you read (as stated in the "On the CD" section at the end of the chapter, you can find the finished, ready-to-play Lockdown game in `Programs/Chapter 16/Lockdown/Executable/`).

The Fortress

The first and most important graphical step was creating the fortress. Because the player is never in more than one room at once, this really just boiled down to the room graphics. Although the logical choice for generating the room graphics would be rendering them in a 3D modeling package, I just ended up doing it by hand, entirely in Photoshop. The final room is composed of a large grated floor, surrounded by four dark walls and the bluish light emanating from the sconces mounted on the sides of each door. Overall I wanted something moody and atmospheric, and when combined with the flickering light effect found in the final game, I think I got it. Figure 16.8 contains the background used as the basis for all of the rooms.

The Droids

Unlike the fortress, which was mostly a static and unchanging image, the droids are in constant motion. To make things easy on myself, I modeled and rendered them in 3ds max, allowing me to easily change their colors, generate sprites from any angle, and alter or animate the lighting. Having a flexible 3D model of a game's characters also makes static title screens very easy to pump out. Figure 16.9 is a rendering of the basic droid model, and Figure 16.10 is the game's title screen.

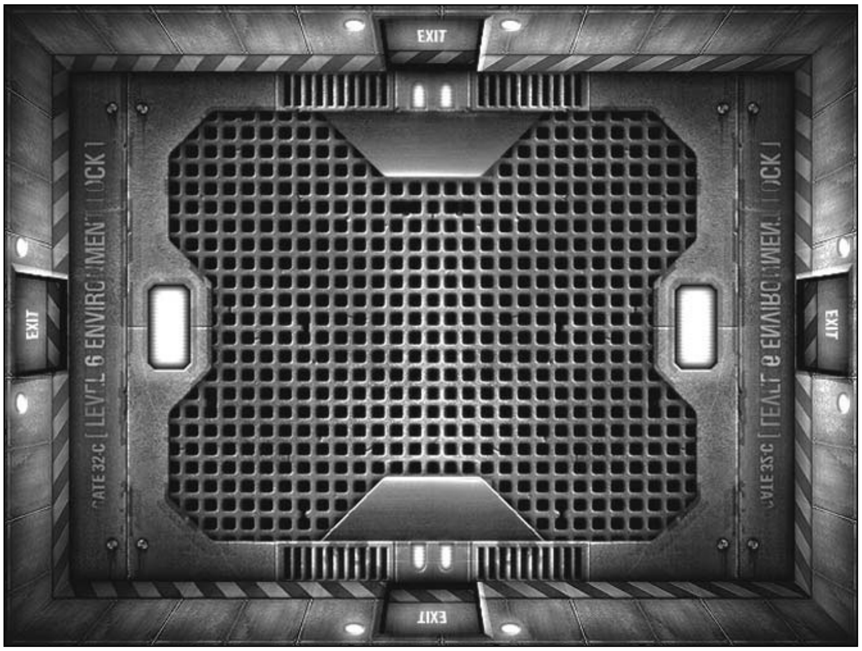

EXIT
EXIT
EXIT
EXIT
GATE 32-C | LEVEL 6 ENVIRONMENT LOCK |
GATE 32-C | LEVEL 6 ENVIRONMENT LOCK |

Figure 16.8

A typical room background.

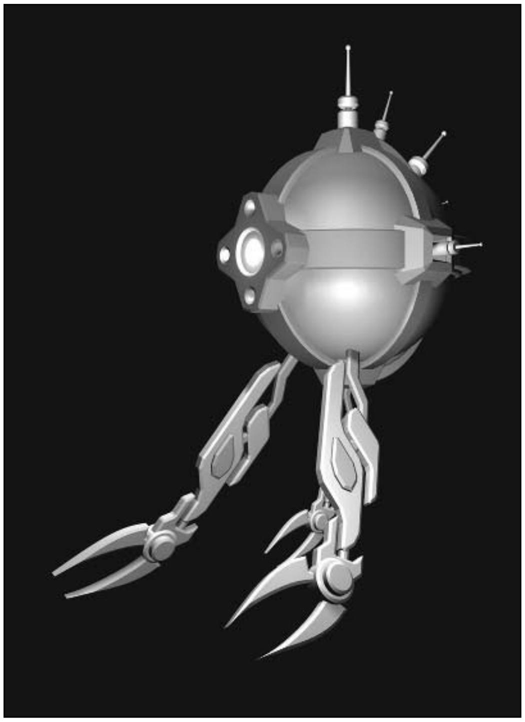


Figure 16.9

The droid model, rendered in 3ds max.

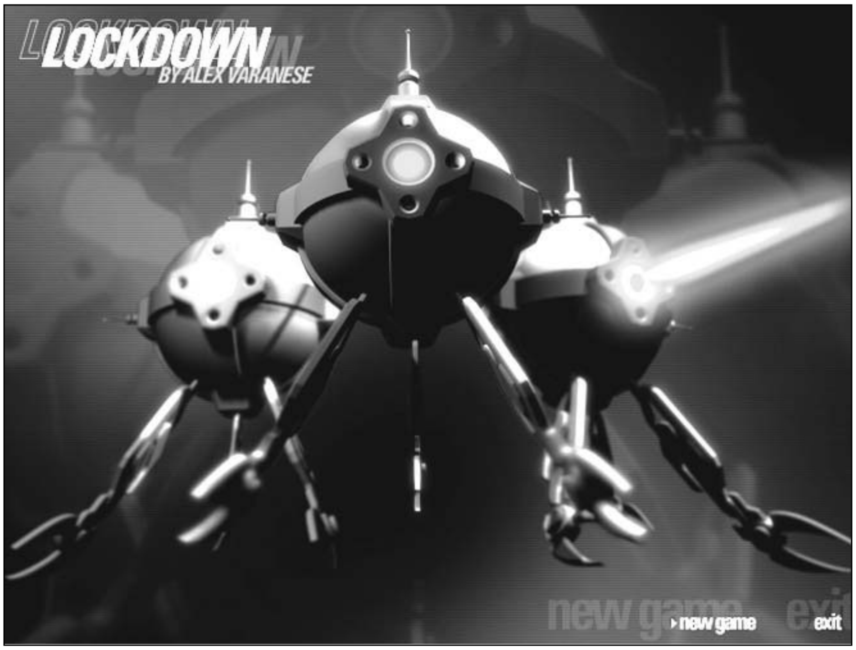


Figure 16.10

The Lockdown title screen.

The Keys

The keys were also rendered in max, and were composed of very basic geometry. Again, however, the aid of a 3D modeler allowed me to convert my simple mesh into a complete animation quite easily. Figure 16.11 is a rendering of a Lockdown key.

The Explosions

Explosions are always tricky when making a game. They usually require too much fluid detail and animation to draw by hand, and volumetric/combustion plug-ins for 3D packages that look even remotely realistic are usually orders of magnitude more expensive than the average home user can afford. I ended up sampling footage of actual explosions from the *Pyromania* CD, a disc containing stock footage of real explosions for use by filmmakers and game developers produced by a company called Visual Concept Entertainment.

Sound

Sound is at the same time one of the most important and overlooked aspects of game development. True immersion and atmosphere simply isn't going to happen without the right ambience and foreground effects, and even though Lockdown is really just a simple vehicle for testing and

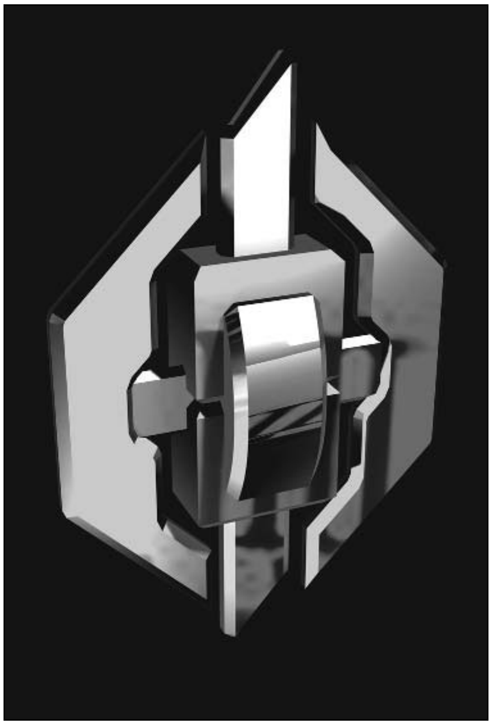


Figure 16.11

The key model.

demonstrating the scripting engine, I figured it'd be worth throwing in a few effects to make things feel more complete.

Lockdown's sound effects can be found in the `Sound/` directory.

Effects

The sound effects are your typical fair—lasers, explosions, and so on and so forth. They originally came from the General 6000 sound collection by Sound Ideas (http://www.sound-ideas.com/), and were further processed using SoundForge and CoolEdit Pro to refine the sounds and make them a bit more uniform and conducive to the game's atmosphere.

Music

There's very little music in the game, but I did throw in a little "theme song" in the beginning. It's a sort of an "Inspector Gadget meets the Atari 2600" sounding number, which I also got from a Sound Ideas stock CD.

For some reason I thought the idea of using the *Friends* theme song instead would have been hysterical. You can thank Andre' Lamothe himself for making sure that didn't happen.

Scripts

The last of the game's major assets are the scripts. Deciding what to script was a somewhat tricky issue, as the final decision can easily lie anywhere on the spectrum between too much and too little. For the sake of simplicity, however, I decided to choose a small and focused domain for the scripts to handle exclusively, rather than pummel the engine (and the reader) with huge amounts of dull and faceless scripts doing menial tasks.

So, scripts are almost entirely focused on the behavior of the enemy droids, which is the most logical approach. This is because details like those of the game engine are more or less static; you know exactly what the engine needs to do, and how it needs to do it. In the case of the droids, however, it's important to have the flexibility and capability to make impulsive changes that scripting provides. This also allows additional droids to be added to the game quite easily at any point in the future.

The Droids

There are three different droid types in the game, so it's understandable that three separate .XSEs are needed. These are `Blue_Droid.xse`, `Grey_Droid.xse`, and `Red_Droid.xse`. Each script controls a different droid type, as is obvious from the names. I'll discuss these scripts in much more detail in the following sections.

Ambience

Although droid behavior is the focus of the scripting in Lockdown, I also threw in one extra script, `Ambient.xse`, to automatically control the ambient effects in the game. As you'll see later, this is a simple script whose only real job is to flicker the lights in the fortress's rooms.

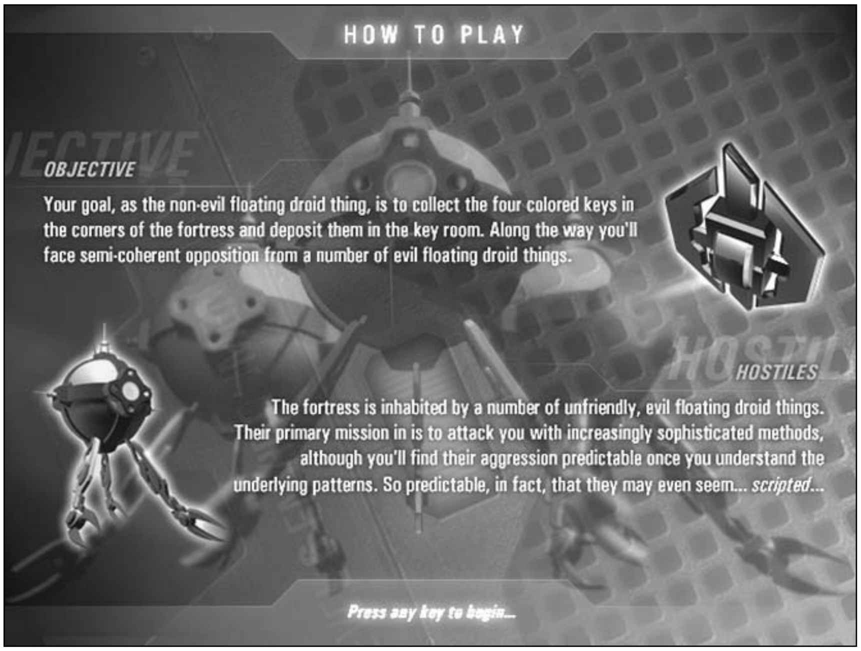
Before getting into the code, check out Figure 16.12, which presents the game's "How to Play" screen.

Phase Three—Planning the Code

So far, Lockdown has been planned with a reasonable level of detail, and you have a good understanding of what sort of assets the game will need. You're now in a position to safely begin planning the game's code.

Game States

As is often the case with modern games, Lockdown was designed as one big state machine. This means that at each iteration of the game's main loop, it can be in any of a number of separate states; be it the title screen, the game-over screen, or the actual game play. The passing of time as well as input from the players provide cues for the game engine to transition to another state, thereby advancing the game. Recall that state machines were also used as the basis for the lexical analyzer in Chapter 13.



HOW TO PLAY

OBJECTIVE

Your goal, as the non-evil floating droid thing, is to collect the four colored keys in the corners of the fortress and deposit them in the key room. Along the way you'll face semi-coherent opposition from a number of evil floating droid things.

HOSTILES

The fortress is inhabited by a number of unfriendly, evil floating droid things. Their primary mission in is to attack you with increasingly sophisticated methods, although you'll find their aggression predictable once you understand the underlying patterns. So predictable, in fact, that they may even seem... *scripted...*

Press any key to begin...

Figure 16.12

The "How to Play" Screen.

Lockdown's State Machine

One of the benefits of the state machine approach to game design is that it allows the entire lifespan of the game, from beginning to end, to be planned out with a single state diagram. This is a great way to quickly and easily get a handle on exactly how things relate to each other before writing any actual code, and was my first step in planning the code layout for Lockdown. My sketch of the game's state machine is presented in Figure 16.13.

What this is basically saying is that the game starts off by initializing itself, and immediately transitions to the title screen state. From here, there are two options—starting a new game or exiting. If the exit option is chosen, the game ends there. Otherwise, the state transitions to the "How to Play" screen, and remains there until a key is pressed. The next state is the "Loading...: screen, which lets the players know that the game's assets are being loaded.

From here, the state transitions to the actual game play, which runs until one of a number of events occurs. The first is pressing the Enter key, which brings up the Zone Map screen that lets the players know where in the fortress they are. The next events that can kill the main game loop are winning or losing the game. In either case, the game transitions to a state that displays a full-screen image either congratulating the player in the event of a win, or, if the player is destroyed, mocking his clearly awful hand-eye coordination and insulting his ethnicity (well, not really, it just says "Game Over"). Both of these states wait for a keypress before transitioning back to the title screen state, where the process begins again.

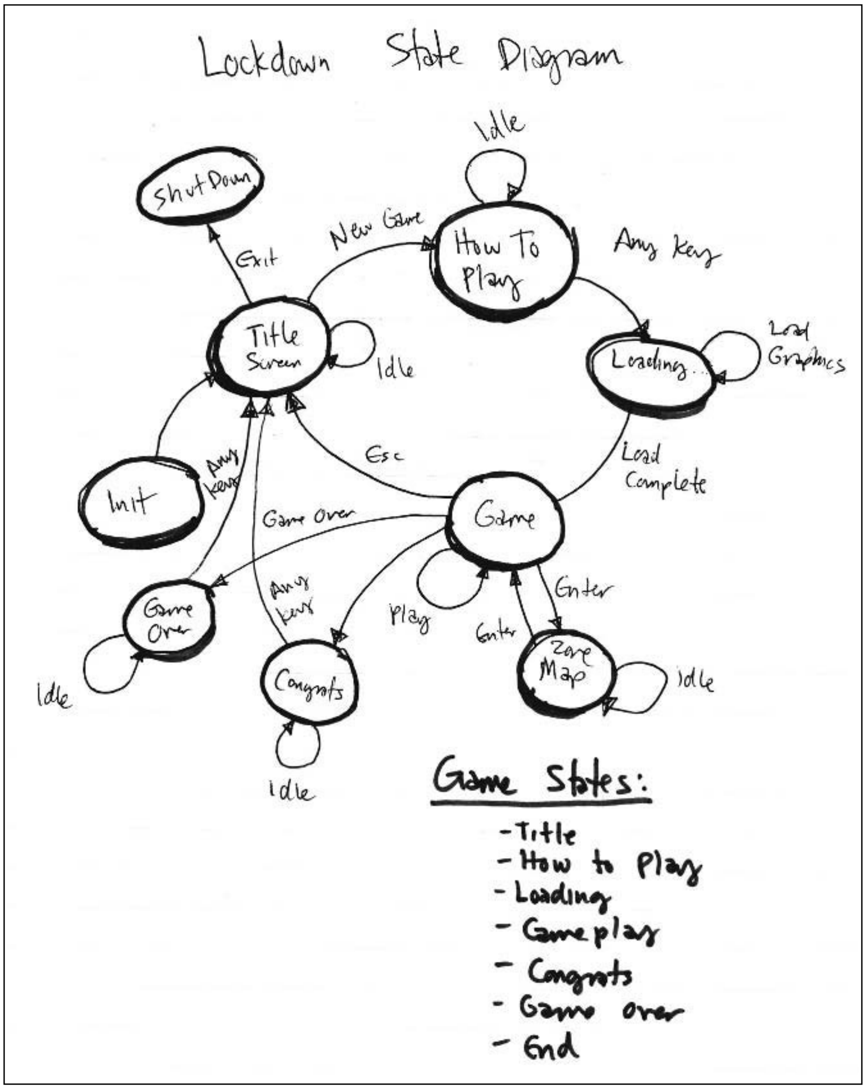


Figure 16.13

The Lockdown state machine.

Scripting Strategy

Because this isn't a book about general game programming, the actual development of Lockdown's engine isn't particularly relevant (or even really all that interesting; it's not exactly a *Halo*-killer). Assuming the engine works, which it does, all you really care about now is using XtremeScript to control the droids.

The scripting strategy is simple; you want to run a single script in the background that controls the environment's ambient effects (in other words, makes the room lights flicker), as well as run any of three droid-controlling behavior scripts. These scripts need to be loaded up front, and, during the execution of the game, run for as long as they're needed.

The remainder of this section focuses on specific areas of the code behind Lockdown, which can be found on the CD in `Programs/Chapter 16/Lockdown/Source/`.

Integrating XtremeScript

Before you can do anything, the XVM needs to be embedded into the Lockdown engine. All this means is including `xvm.h` in the main Lockdown source file and linking `xvm.cpp` with the project. Once inside, you can use `XS_Init ()` and `XS_ShutDown ()` to control the lifespan of the virtual machine, and you're ready to go.

Inside Lockdown's `Init ()` function, the following line of code is added:

```
XS_Init ();
```

And, of course the game's `ShutDown ()` function contains this:

```
XS_ShutDown ();
```

The Host API

The host API used by the scripts you're about to write doesn't need to be particularly extensive, but it does need to be well equipped enough to provide basic information about the location and status of the player and enemy droids, along with the ability to manipulate the droids as well.

Rather than go into much detail on why these functions were chosen now, I'm just going to list them and briefly explain their tasks. It will become clear why they're necessary when you write the scripts that use them in the next section. Furthermore, I won't even go into their implementation; these functions are specifically designed to work with the Lockdown game engine, which means it'd take at least a superficial overview of how the engine works. Because the engine isn't meant to be the focal point of this chapter, I don't want to shift the attention from scripting and will leave them unexplained. Of course, you're free to check them out yourself in the Lockdown source code, which shouldn't be too hard because none of these functions is more than a handful of

NOTE

Even though XtremeScript is a typeless language, I'll be annotating each parameter and function return value with a C-style data type to describe what type of values are expected.

lines anyway. Besides, they should all be self-explanatory to begin with; anyone with even a basic understanding of 2D game programming should feel right at home.

Miscellaneous Functions

```
int GetRandInRange ( int Min, int Max )
```

This function returns a random integer value between `Min` and `Max`, inclusive.

```
void ToggleRoomLights ()
```

Calling this function will toggle the lights in the room, from either light to dark or dark to light. You'll make use of this function in the ambience script.

Enemy Droid Functions

```
void MoveEnemyDroid ( int DroidIndex, int Dir, int Dist )
```

Calling this function will move the specified index in the specified direction with the specified distance.

```
int GetEnemyDroidX ( int DroidIndex )
int GetEnemyDroidY ( int DroidIndex )
```

These functions are used together to get the X, Y location of an enemy droid.

```
int IsEnemyDroidAlive ( int DroidIndex )
```

This function returns `TRUE` if the specified droid is alive, and `FALSE` otherwise.

```
void FireEnemyDroidGun ( int DroidIndex )
```

Calling this function causes the specified droid to fire its laser cannon in whatever direction it happens to be facing.

Player Droid Functions

```
int GetPlayerDroidX ()
int GetPlayerDroidY ()
```

These functions are used to determine the player's X, Y location on-screen.

Figure 16.14 shows Lockdown as the player wipes out some enemies.

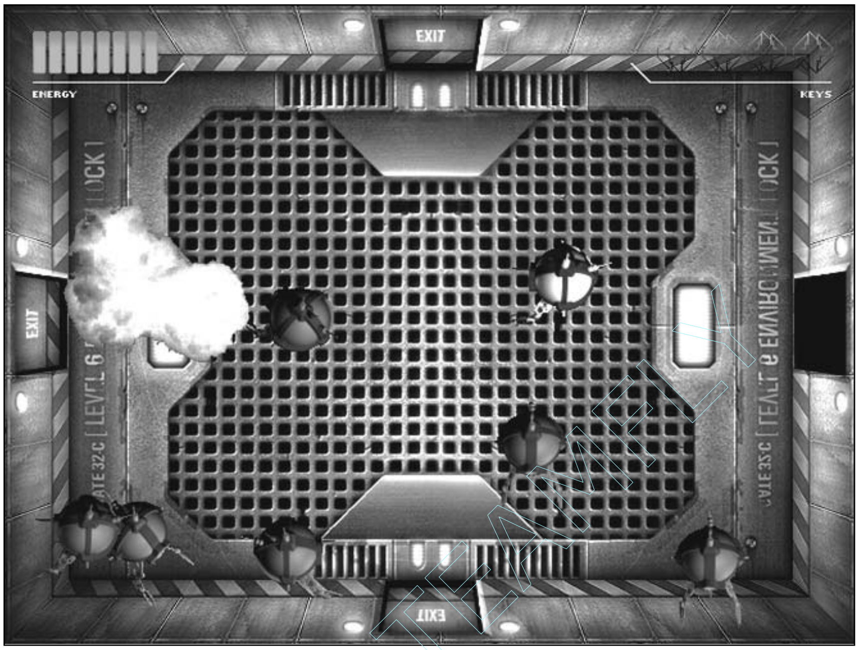


Figure 16.14

The player making short work of blue druids in Lockdown.

Registering the Functions

The Lockdown host API is registered with the XVM in the game's `Init ()` function, right after the call to `XS_Init ()`. As you can see, each of the functions are global, because there's really no practical reason to fence certain functions off to certain scripts:

```
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetRandInRange",
    HAPI_GetRandInRange );

XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "ToggleRoomLights",
    HAPI_ToggleRoomLights );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "MoveEnemyDroid",
    HAPI_MoveEnemyDroid );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetEnemyDroidX",
    HAPI_GetEnemyDroidX );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetEnemyDroidY",
    HAPI_GetEnemyDroidY );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "IsEnemyDroidAlive",
    HAPI_IsEnemyDroidAlive );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "FireEnemyDroidGun",
    HAPI_FireEnemyDroidGun );
```

```
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetPlayerDroidX",
    HAPI_GetPlayerDroidX );
XS_RegisterHostAPIFunc ( XS_GLOBAL_FUNC, "GetPlayerDroidY",
    HAPI_GetPlayerDroidY );
```

Writing the Scripts

Writing the scripts is the fun part, and isn't particularly difficult. You have three scripts to write in total—the ambience script which runs constantly, and three droid behavior scripts that run individually. Let's have a look at each.

The Ambience Script

The ambience script, found in `Ambient.xasm|xse`, is a very small and simple script that randomly flickers the lights in the room. The game engine constantly runs it, allowing the lights to appear as if they're running in the background. The script is so simple that I didn't even feel the need to waste the extra instruction cycles on a high-level script, and instead wrote it directly in XVM assembly.

```
; ---- Directives ------------------------------------------------------------

   SetPriority 20

; ---- Main ------------------------------------------------------------------

    Func _Main
    {
        ; Enter the main loop
        LoopStart:

            ; Get a random number between 0 and 50, inclusive
            Push        0
            Push        50
            CallHost    GetRandInRange

            ; If the number was 1, flicker the lights
            JNE         _RetVal, 1, SkipToggleLights
            CallHost    ToggleRoomLights
        SkipToggleLights:

        Jmp    LoopStart
    }
```

All the script needs is a `_Main ()` function that starts a simple loop. This loop runs infinitely, allowing it to continually execute the game's main loop. At each iteration, the host API function `GetRandInRange ()` is called to get a random number between 0 and 50. If this number is 1, the lights toggle. When this is executed at runtime, the frequency of 1's in this range provides a nice flicker effect.

You'll also notice that the `SetPriority` function is asking for a time slice whose duration is 20. For reasons I'll explain in a later section, this isn't referring to 20 milliseconds, but rather 20 instruction cycles. Again, expect a full explanation of this in a moment; just make a mental note of it for now.

The Blue Droid's Behavior Script

The script that controls the blue droid is much more complicated than the ambience script, so I wrote it in XtremeScript. The blue droid's "AI" is really just a random number generator; it uses the `GetRandInRange ()` function repeatedly to generate new paths, and then incrementally follows them. Let's start by taking a look at the code, which you can find in `Blue_Droid.xss`:

```
// ---- Host API Imports -------------

    host GetRandInRange ();
    host MoveEnemyDroid ();
    host GetEnemyDroidX ();
    host GetEnemyDroidY ();
    host IsEnemyDroidAlive ();

// ---- Main ------------------------

    func _Main ()
    {
        // Droid index counter
        var CurrDroid;
        CurrDroid = 0;

        // Enter the main loop
        while ( true )
        {
            // If the droid is alive, move it
            if ( IsEnemyDroidAlive ( CurrDroid ) )
            {
                // Calculate a new direction, distance and speed
                var Dir;
                var Dist;
                var Speed;

                Dir = GetRandInRange ( 0, 7 );
                Dist = GetRandInRange ( 3, 20 );
                Speed = GetRandInRange ( 5, 12 );

                // Move the droid along the path
                while ( Dist > 0 )
                {
                    MoveEnemyDroid ( CurrDroid, Dir, Speed );
                    Dist -= 1;
                }
            }

            // Move to the next droid
            CurrDroid += 1;
            if ( CurrDroid > 7 )
                CurrDroid = 0;
        }
    }
```

The script starts by importing the required host API functions using the `host` keyword. Within the `_Main ()` function, a loop continually executes that cycles through each droid, picks a random path, and moves it along that path until it reaches its destination. To save time, the script only operates on droids that are alive, a check it makes with the `IsEnemyDroidAlive ()` function. To actually facilitate the physical movement of the droid, the host API function `MoveEnemyDroid ()` is used.

The beauty of this time-slicing system is that it allows you to write individual scripts as if they're the only thing actually executing, even though they're sharing time with the ambience script and the game engine itself.

The Grey Droid's Behavior Script

The grey droid ups the ante a bit by adding the capability to shoot at the players. Although its movement is random, the direction it fires its weapon is based on the player's location. Here's the script, which is available on the CD as `Grey_Droid.xss`:

```
// ---- Host API Imports -------------

    host GetRandInRange ();

    host MoveEnemyDroid ();
    host GetEnemyDroidX ();
    host GetEnemyDroidY ();
    host GetEnemyDroidDir ();
    host IsEnemyDroidAlive ();
    host FireEnemyDroidGun ();

    host GetPlayerDroidX ();
    host GetPlayerDroidY ();
    host GetPlayerDroidDir ();

// ---- Constants ------------

    // Directions

    var NORTH;
    var SOUTH;
    var EAST;
    var WEST;

// ---- Main -----------------

    func _Main ()
    {
        // Initialize our "constants" to values that correspond
        // with Lockdown's internal direction constants
        NORTH = 0;
        EAST = 2;
        SOUTH = 4;
        WEST = 6;

        // Droid index counter
        var CurrDroid;
        CurrDroid = 0;
```

```
// Enter the main loop
while ( true )
{
    // If the current droid is alive, handle its behavior
    if ( IsEnemyDroidAlive ( CurrDroid ) )
    {
        // The current direction, distance and speed
        // of the droid's movement
        var Dir;
        var Dist;
        var Speed;

        // The droid's X, Y location
        var EnemyDroidX;
        var EnemyDroidY;

        // The player's X, Y location
        var PlayerDroidX;
        var PlayerDroidY;

        // Generate a random path to follow
        Dir = GetRandInRange ( 0, 7 );
        Dist = GetRandInRange ( 3, 20 );
        Speed = GetRandInRange ( 5, 12 );

        //  Move the droid along the path
        while ( Dist > 0 )
        {
            // Shoot occasionally
            if ( GetRandInRange ( 0, 8 ) == 1 )
            {
                // Get the enemy's location
                EnemyDroidX = GetEnemyDroidX ( CurrDroid );
                EnemyDroidY = GetEnemyDroidY ( CurrDroid );

                // Get the player's location
                PlayerDroidX = GetPlayerDroidX ();
                PlayerDroidY = GetPlayerDroidY ();
```

```
                    // Use these locations to face the
                    // droid in the proper direction when shooting
                    if ( EnemyDroidX < PlayerDroidX )
                    {
                        Dir = EAST;
                        MoveEnemyDroid ( CurrDroid, Dir, 0 );
                    }
                    else if ( EnemyDroidY < PlayerDroidY )
                    {
                        Dir = SOUTH;
                        MoveEnemyDroid ( CurrDroid, Dir, 0 );
                    }
                    else if ( EnemyDroidX > PlayerDroidX )
                    {
                        Dir = WEST;
                        MoveEnemyDroid ( CurrDroid, Dir, 0 );
                    }
                    else if ( EnemyDroidY < PlayerDroidY )
                    {
                        Dir = NORTH;
                        MoveEnemyDroid ( CurrDroid, Dir, 0 );
                    }

                    // Fire the laser
                    FireEnemyDroidGun ( CurrDroid );
                }

                // Increment the droid's position
                MoveEnemyDroid ( CurrDroid, Dir, Speed );
                Dist -= 1;
            }
        }

        // Move to the next droid
        CurrDroid += 1;
        if ( CurrDroid > 7 )
            CurrDroid = 0;
    }
}
```

For the most part, this script mirrors the functionality of `blue_droid.xss`. The major difference is that now, as the droid moves, it randomly fires at the player. Once again, you use `GetRandInRange ()` to give the droid a 1 in *N* chance to fire at each step. Instead of simply firing the weapon, however, the enemy's and player's location is used to determine which direction it should face before firing, to make it more likely that the player will be struck. `MoveDroid ()` is used to move the droid in this direction, but with a distance of 0—this causes the droid to turn to face the player without actually moving towards her.

Note also that once again, you're simulating constants with globals. The constants refer to the cardinal directions, which makes the direction parameter accepted by `MoveEnemyDroid ()` more readable. These globals are initialized when `_Main ()` starts, and their values correspond to the values used by the Lockdown engine.

The Red Droid's Behavior Script

The last droid to cover is the red droid, whose script provides the most advanced behavior and can be found in `Red_Droid.xse`. The logic here once again builds on the previous droid. While retaining the capability to fire at the player, the red droid can also move towards the player's location, rather than just stumble around randomly. When applied to every droid in the room, this creates a subtle "swarming" effect. Check out the code:

```
// ---- Host API Import --------------

    host GetRandInRange ();

    host MoveEnemyDroid ();
    host GetEnemyDroidX ();
    host GetEnemyDroidY ();
    host GetEnemyDroidDir ();
    host IsEnemyDroidAlive ();
    host FireEnemyDroidGun ();

    host GetPlayerDroidX ();
    host GetPlayerDroidY ();
    host GetPlayerDroidDir ();

// ---- Constants ----------------

    // Directions
    var NORTH;
    var SOUTH;
    var EAST;
    var WEST;

// ---- Functions -------------
    /**************************************
    *
    *    GetPlayerFaceDir ()
    *
    *    Returns the direction in which an enemy droid
    *    should face in order to face the player.
    */

    func GetPlayerFaceDir ( CurrDroid )
    {
        // The specified enemy's location, as well as the player's
        var EnemyDroidX;
        var EnemyDroidY;
        var PlayerDroidX;
        var PlayerDroidY;

        // Get the locations
        EnemyDroidX = GetEnemyDroidX ( CurrDroid );
        EnemyDroidY = GetEnemyDroidY ( CurrDroid );
        PlayerDroidX = GetPlayerDroidX ();
        PlayerDroidY = GetPlayerDroidY ();

        // Perform some simple checks to determine the optimal direction
        if ( EnemyDroidX < PlayerDroidX )
            return EAST;
        else if ( EnemyDroidY < PlayerDroidY )
            return SOUTH;
        else if ( EnemyDroidX > PlayerDroidX )
            return WEST;
        else
            return NORTH;

        // Return north by default
        return NORTH;
    }
```

```
// ---- Main ----------------------------------------------------------------

    func _Main ()
    {
        // Initialize our "constants" to values that correspond
        // with Lockdown's internal direction constants
        NORTH = 0;
        EAST = 2;
        SOUTH = 4;
        WEST = 6;

        // Droid index counter
        var CurrDroid;
        CurrDroid = 0;

        // Enter the main loop
        while ( true )
        {
            // If the droid is active, move it
            if ( IsEnemyDroidAlive ( CurrDroid ) )
            {
                // Calculate a new path in the direction of the player
                var Dir;
                var Dist;
                var Speed;

                Dir = GetPlayerFaceDir ( CurrDroid );
                Dist = GetRandInRange ( 3, 20 );
                Speed = GetRandInRange ( 5, 12 );

                // Move the droid along the path
                while ( Dist > 0 )
                {
                    // Occasionally fire the laser
                    if ( GetRandInRange ( 0, 8 ) == 1 )
                    {
                        // Make sure to face the player when doing so
                        Dir = GetPlayerFaceDir ( CurrDroid );
                        FireEnemyDroidGun ( CurrDroid );
                    }
                    // Increment the droid's positions
                    MoveEnemyDroid ( CurrDroid, Dir, Speed );
                    Dist -= 1;
                }
            }

            // Move to the next droid
            CurrDroid += 1;
            if ( CurrDroid > 7 )
                CurrDroid = 0;
        }
    }
```

This final script runs the gamut of host API functions, importing them all. It also defines a function of its own, `GetPlayerFaceDir ()`. Because the red droid needs to both move and fire in the player's direction, I decided to write a single function that could be called whenever it was necessary to determine which direction the droid should face in order to face the player. The function works by using host API functions to determine both the enemy's and player's location, and uses simple logic to derive a facing direction from those two coordinates.

Figure 16.15 portrays the player taking out the few remaining red druids in a key pedestal room.

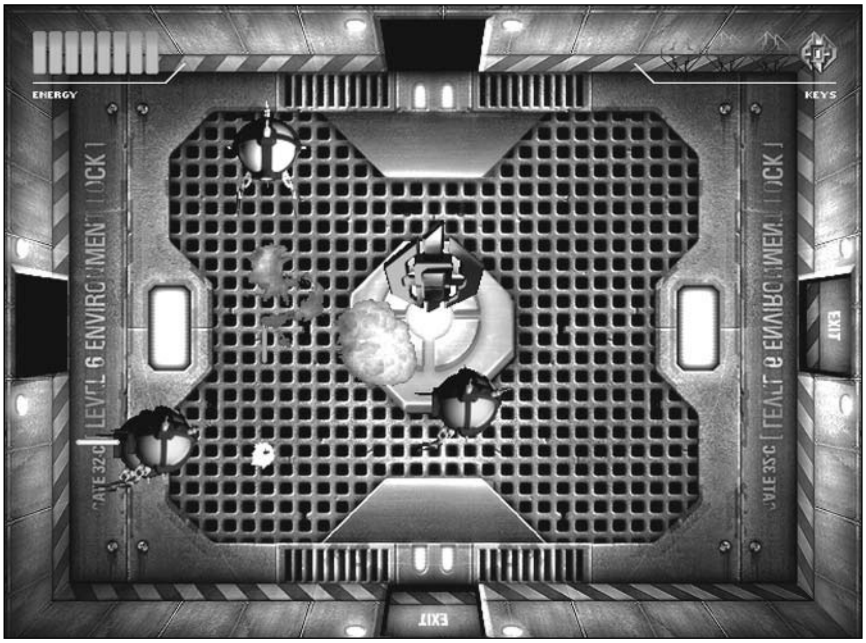


Figure 16.15

Taking out the last red druids in Lockdown.

Within the `_Main ()` function, things look more or less familiar. At each cycle through the loop, the next droid in the list is assigned a path to follow, except you're now using `GetPlayerFaceDir ()` to determine which direction to use. This is how the red droid manages to track the players as they move around the room. Within the movement loop, the frequency of shots fired from the droid's laser cannon is regulated in the same manner as the grey droid; by giving it a 1 in *N* chance.

Compilation

Compiling the scripts is a simple matter of using the XSC compiler, but it's important to note that all three of the droid behavior scripts are compiled with a user-defined priority of 60, like so:

```
XSC Red_Droid.xss -P:60
```

Again, just as was the case with the ambience script, the 60 doesn't refer to milliseconds, but rather to instructions. I'm about to discuss why, but in the meantime, just remember the number 60.

Loading and Running the Scripts

That wraps up the discussion of the scripts, so it's time to load them into the engine. The following code is added to the game's `Init ()` function, just after the call to `XS_Init ()` and the registration of the host API:

```
XS_LoadScript ( "Scripts/Ambient.xse", g_iAmbientThreadIndex,
    XS_THREAD_PRIORITY_USER );
XS_LoadScript ( "Scripts/Blue_Droid.xse", g_iBlueDroidThreadIndex,
    XS_THREAD_PRIORITY_USER );
XS_LoadScript ( "Scripts/Grey_Droid.xse", g_iGreyDroidThreadIndex,
    XS_THREAD_PRIORITY_USER );
XS_LoadScript ( "Scripts/Red_Droid.xse", g_iRedDroidThreadIndex,
    XS_THREAD_PRIORITY_USER );
```

Note that each call to `XS_LoadScript ()` passes the `XS_THREAD_PRIORITY_USER` flag, telling the loader to respect the script-defined priority value rather than overwriting it.

Also, the thread index for each script is saved in a global. These four indexes are globally defined so that any part of the Lockdown engine can refer to the scripts to which they're associated:

```
int g_iAmbientThreadIndex;           // Ambient script thread index
int g_iBlueDroidThreadIndex;         // Blue droid script index
int g_iGreyDroidThreadIndex;         // Grey droid script index
int g_iRedDroidThreadIndex;           // Red droid script index
```

Within the main loop of the game, `XS_RunScripts ()` is called once per frame. This handles any and all running scripts, but the real issue is when and how these scripts should be initially activated.

In the case of the ambience script, you want it running at all times—regardless of what room the player is in. Because of this, the following line appears whenever the game switches into the game play state:

```
XS_StartScript ( g_iAmbientThreadIndex );
```

`XS_StopScript ()` is then called with the same parameter when the game switches back out of the state. The droids are trickier, however, because they depend entirely on the current room. To understand how this works, check out the following excerpt from the Lockdown engine. It's from a function called `InitRoom ()`, which is called whenever the player enters a new room to set everything up. In addition to the function's other tasks, it uses a `switch` block to determine which room type is being entered, and starts and stops the droid scripts as necessary:

```
switch ( iType )
{
    case ROOM_TYPE_NORMAL:
        XS_StartScript ( g_iBlueDroidThreadIndex );
        XS_StopScript ( g_iGreyDroidThreadIndex );
        XS_StopScript ( g_iRedDroidThreadIndex );
        break;

    case ROOM_TYPE_GUARD:
        XS_StopScript ( g_iBlueDroidThreadIndex );
        XS_StartScript ( g_iGreyDroidThreadIndex );
        XS_StopScript ( g_iRedDroidThreadIndex );
        break;

    case ROOM_TYPE_PEDESTAL:
        XS_StopScript ( g_iBlueDroidThreadIndex );
        XS_StopScript ( g_iGreyDroidThreadIndex );
        XS_StartScript ( g_iRedDroidThreadIndex );
        break;
}
```

The `ROOM_TYPE_NORMAL` flag refers to the lowest security rooms; they don't contain keys, and don't border the key pedestal rooms. Because of this, they contain blue (weak) droids. The next room type is `ROOM_TYPE_GUARD`, which also doesn't contain a key, but directly borders a key pedestal room, and therefore requires slightly higher security via the grey droids. The last type is the `ROOM_TYPE_PEDESTAL` room, which houses a key and requires the defense of the red droids.

In conjunction with the constant calling of `XS_RunScripts ()` in the main loop, the logic discussed in this section regulates the activity of the loaded scripts. The ambient script runs at all times during the game play state, and three droid scripts are flipped on and off as the player navigates through the rooms of the fortress.

Figure 16.16 shows the player in the key room, with the red and green panels activated.

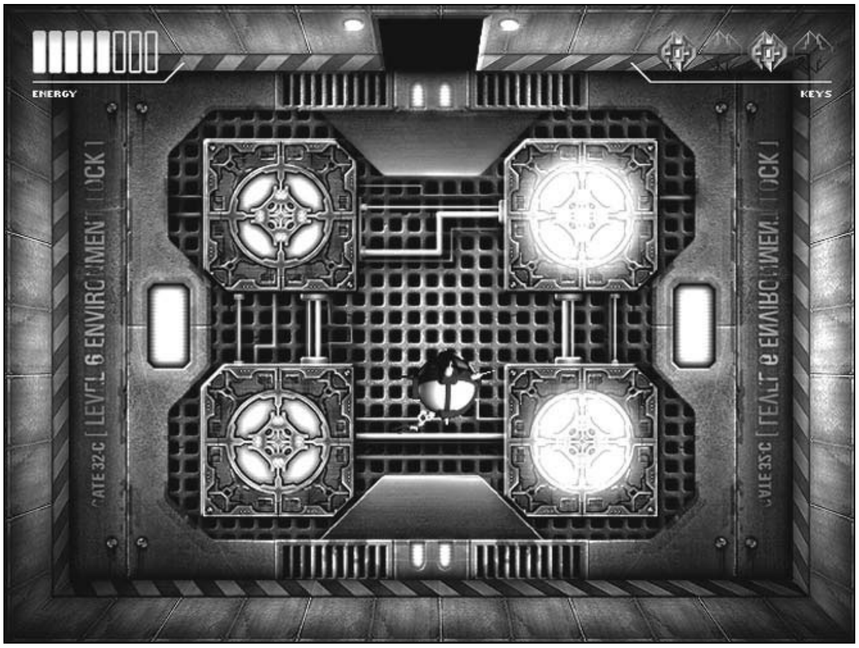


Figure 16.16

The red and green panels of the key room activated in Lockdown.

Speed Issues

The last issue to deal with is the fact that XtremeScript isn't exactly blazingly fast. The system was intentionally designed to be educational and readable above all else, and although this hopefully aided your understanding of what was going on, it certainly takes its toll on performance. The runtime environment has performance issues of its own, but the real culprit here is the code generated by the XtremeScript compiler for evaluating expressions. Flow and control constructs like `if`, `while`, and so on are compiled to lean, reasonable code due to their simplicity, but the bloated expression evaluation code it emits is more than enough to seriously degrade a game's speed.

Although the long-term solution is to tighten up the VM and perform basic optimization on the evaluation expression code emitted by XSC, there are a few tricks you can pull to squeeze some extra speed out of the system as it currently stands.

Minimizing Expressions

As stated, the inherent simplicity of `while` loops, `if` blocks, and other language constructs allow them to be translated to nearly optimal assembly language by nature. Because XSC converts them into little more than a few jumps and labels, the emitted code isn't much different than what you might code by hand. Even functions and function calls are pretty lean—after all, the expression parser always leaves its result on the stack, which is where a parameter needs to be anyway. All the compiler really does is make sure the parameters are pushed on and follow it up with a `Call` or `CallHost` instruction. Again, this is more or less exactly what a human assembly coder would do.

Unfortunately, expression evaluation is where things start to slow down considerably. Although strict and traditional stack usage is probably the easiest and most intuitive way to demonstrate this process, it's hardly the fastest solution. This compiler is great for teaching, but bad for expression evaluating in performance-critical applications. The simple solution to this is to minimize your use of expressions in code that needs to run quickly. For example, an RPG that periodically updates player stats with complex expressions and algorithms can use XtremeScript without a problem, because such updates don't need to occur on a frame-by-frame basis. This is why the scripting system is still great for things like item and weapon definitions.

What it isn't so good for, however, is performing complex operations at each frame. For this reason, it'll help to minimize the complexity of specific expressions in the droids' AI, because their logic is invoked on a per-frame basis. Try splitting up your logic into a number of smaller expressions over time, rather than a single major one. Try calculating values ahead of time, preferably before entering the main loop—this can help preserve the functionality of an algorithm or expression without having to do all of it in the heat of battle.

The XVM's Internal Timer

Another way to speed up the XVM is to alter the way its timing works. Right now, it uses the Win32 API function `GetTickCount ()` to synchronize events like time-slicing to intervals of time based on milliseconds. Although this is certainly a powerful and flexible method for complex games like long-term strategy simulations, the Windows tick counter isn't particularly accurate—only to about 55 milliseconds to be exact.

The problem with this is that a requested time slice of three milliseconds will run for just as long as one set for 55 milliseconds. This is a serious accuracy issue that will accumulate fast when multiple scripts are running at each frame.

Although there are a number of solutions for high-resolution timing, such as the Windows high-performance timers, I decided to go for something simple and straightforward that wouldn't take long to implement and would be very clean and fast. What I decided to do was give the XVM its own tick-counting system, but one that was based on the execution of instructions, rather than the passing of time.

This modification was actually very simple. Remember, the XVM function `GetCurrTime ()` was designed from the beginning as a "black box" that can be implemented with any timing mechanism without disrupting the virtual machine overall. All I had to do was replace the function's body with this:

```
inline int GetCurrTime ()
{
    static unsigned int iCurrTick = 0;

    ++ iCurrTick;
    return iCurrTick;
}
```

Now, every call to `GetCurrTime ()` returns the current tick and increments it. Because you know this function is called after each instruction is executed in `XS_RunScripts ()`, you know it'll always return an accurate and unique tick. This gives you extremely precise control over the time-slicing of your threads, allowing you to coordinate their execution on an instruction level.

The one issue here is that it does have an effect on the values of a thread's priority level. For example, the `XS_THREAD_PRIORITY_*` constants are no longer meaningful in the same way, and must be rewritten to compensate for the new timing mechanism. Furthermore, any script with a user-defined time slice must be recompiled or reassembled, because the requested value is no longer in milliseconds, but in ticks. This is why I used numbers like 20 and 60 when defining the time slices of Lockdown's scripts.

How to Play Lockdown

To finish things up, I'd just like to briefly cover how Lockdown is actually played, so you can play around with the game on your own. Although I've covered individual aspects of the game's control scheme throughout this chapter, there hasn't been an explicit discussion of how exactly a player plays the game.

Controls

Lockdown's controls are simple. The arrow keys move the player droid around within the room, and by holding down two keys at once, the player can move diagonally. Pressing Space fires the droid's laser cannon, although this only works while facing one of the cardinal directions (in other words, the player can't shoot while facing or moving diagonally). At any time, Escape can be pressed to exit the game and return to the title screen.

Interacting with Objects

There are three major objects the player interacts with throughout the game, aside from the enemy droids. These are the keys, the doors, and the key panels. All of these objects can be manipulated with nothing more than the arrow keys; doors open automatically as the players approach them, keys can be collected simply by maneuvering the player droid into them, and the key panels are activated by passing over them.

The Zone Map

At any time during the game, the players can press Enter to invoke the Zone Map, which lets the players know where within the fortress they currently are. Their position is marked with a blinking green cursor, in the shape of four arrowheads pointing towards their center. Pressing Enter again will return the player to the game.

Battle

All rooms aside from the key rooms are inhabited by hostile enemy droids. What the droids lack in strategic intelligence, they make up for in numbers and dedication. Each room starts off with eight droids, all of which will attack the player until it's destroyed. Aside from avoiding them altogether (which isn't easy), the player's only option is to fight back. He needs to aim the laser cannon at the nearest droid and barrage it with shots until it explodes.

TIP

Don't kill any more of your enemies than you have to. The outcome of the game depends on whether or not you get the keys, not on how many droids you can send down the garbage chute. Especially in the case of the key pedestal rooms, do what you need and get out before they have a chance to do significant damage.

Completing the Objective

Ultimately, the player's main goal is to collect the keys and deposit them in the key room. My personal strategy for doing this is, starting from the first room, to move through the fortress counter-clockwise (although this order is arbitrary). I move to the southwest corner, grab the yellow key, move to the northwest corner, grab the blue key, and then head east and make a stop in the key panel room. This allows me to initially activate the yellow and blue key panels. Also, because the key room is uninhabited, I use this stop as a chance to let my energy recharge without being disturbed.

I then dive back into the fray, and head to the northeast corner where I pick up the red key. From there I move south until I hit the southwest corner and grab the green key. This completes my inventory, so I head back to the key room and drop them off for the win.

Summary

Congratulations! You have escaped Lockdown! Or so says the game. More importantly, however, you've reached the finish line as a scripting master and now understand everything that goes into both the development of a scripting system, as well as its applications in a complete game.

What you've accomplished here is no small task. From the development of your own custom language, to its complete implementation, to its application to a game, you've (hopefully) worked your way through hundreds of pages and thousands of lines of code. Sure, the virtual machine could use some extra performance, and the compiler is in desperate need of at least basic optimization, but the framework is there, and nothing short of complete. You now know *everything* you'll need to know to progress into the highest echelons of scripting, like garbage collected runtime environments, advanced high-level language features, and optimizing compilers.

Fortunately, the next chapter offers plenty of suggestions to consider when advancing your newfound mastery of scripting. Now that you understand how an assembler works, you can try devising a new instruction set or adding new syntactic features to the assembly language. Now that you can build a virtual machine, you can learn about how high-performance runtime environments are designed and target the scripting of a truly bleeding-edge game like an advanced FPS or racing game. And of course, now that you've worked your way through the design of a complete compiler, you've got the prerequisite understanding to pursue new parsing methods, more complex source languages, and of course, optimization. Chapter 17 covers all of this in more depth, so as the final step of your quest, I suggest you check it out.

On the CD

This chapter focused on the development of Lockdown, an example game that puts the XtremeScript system to actual use. The complete Lockdown game can be found in both source and executable form in `Programs/Chapter 16/Lockdown/`. You'll also find the slightly modified version of the XVM the game used in this folder, so be sure to check that out as well (remember, I changed its timing method to accommodate higher-performance requirements).

Challenges

- *Intermediate:* Change the scripts of one of the droids to include all three behavior types. For example, modify `Red_Droid.xse` so each of the on-screen droids behaves with one of the three existing attack methods, making them seem more random and lifelike.
- *Intermediate:* Modify the behavior of the existing droids. For example, give the blue droids the capability to follow you, to make up for their non-functioning laser cannon. Or make the red droids even more devious by moving them faster or increasing the frequency by which they fire their weapon.
- *Game Related:* It's technically not related to scripting or game enhancements, but as a refreshingly non-technical challenge, try beating Lockdown *without* using your weapon. Your only strategy without the laser cannon is to avoid the enemy droids entirely, which can be tricky—they have a tendency to leap across the room when you least expect it. Remember, stop in a safe place whenever you can to let your energy recharge. This won't be as easy as usual, because you can't clear a room out without your gun, so your best bet is the central key room.

CHAPTER 17

Where to Go From Here

"Now that you've found Robert Porter, take good care of him."
——*Prot,* K-Pax

Well, well, well. Look at you, Mr. Fancypants. You started with nothing, and after 16 chapters of theory, design explanations, implementation details, and more exposure to my pompous and self-serving sense of humor than anyone should have to endure, you have walked away with a feature-rich, high-level, custom-designed-and-implemented scripting system that's ready to be dropped into your next game project. You now have the ability to describe virtually any action or behavior to the entities of your games, with *total* flexibility——no recompiling the entire engine just to change a few lines of dialogue or tweak the range of your plasma rifle. Now, with a few lines of C-style code, a single pass through a custom-built compiler, and a snap of your fingers for dramatic effect, you can make anything happen in your game universe.

This chapter, although certainly not "required reading", will be a nice and easy way to round out the scripting education this book aims to provide with some brief reflection and insight on where to go from here. I'm going to wrap things up by covering

- How to expand your knowledge of the topics covered throughout the course of the book.
- Advanced subjects that apply directly or indirectly to game scripting for your consideration.
- How to reverse engineer a Furby for the purpose of committing unspeakable atrocities.

So What Now?

Throughout this book, you've learned the details behind designing high- and low-level languages, the virtual machines they run on, and the compilers and assemblers they're translated with. You've learned how and why runtime environments are designed the way they are, and how the formerly mysterious internals of a high-level compiler actually work. With the knowledge presented here, you should be capable of writing your own compilers, assemblers, and embeddable virtual machines. Of course, what you do with this knowledge is up to you, as there are a number of paths to choose.

- Use an existing scripting system like Lua, Python, or Tcl, but with a much more intimate understanding of how it's working on the inside than the other kids on your block. This may be the best option for professionals working with a full team, or those under a tight schedule. You may be using someone else's software, but you'll understand how it was designed and implemented much more clearly than before.

- Use the XtremeScript system developed over the course of the book and included on the CD, with a 100 percent understanding of how it's all working. You're of course free, if not encouraged, to make changes wherever you see fit, or use it as-is, right out of the box.
- Modify XtremeScript to work with a language of your own design, geared towards your own purposes.
- Put your Jedi skills to the ultimate test and use the techniques you've learned to build your own scripting system from the ground up (for fun and profit!).
- Forget scripting, forget game development, sell your computer on eBay and start a hot new boy band. They make way more money anyway.

Aside from maybe that last one, all of these paths are worthwhile pursuits. Regardless of how involved you were in the creation of whatever scripting system you go with, however, you'll always be able to capitalize on an in-depth understanding of how these things work. In short, if you've read, understood, and (ideally) implemented everything in this book, you've truly attained scripting mastery. Congratulations! You could totally hang out with me now!

TIP

Feeling a bit of that "post-book depression"? Wish there was still more hot, steamy scripting action on the horizon? Well don't feel bad, there's still an entire *index* to explore. We used real small print and everything!

Of course, no book under 50,000 pages will be capable of teaching you *everything*, and if you've made it this far, you're probably the inquisitive type and would like to know where to go from here to expand your abilities and understanding. Fortunately for you, there's still an entire universe to explore—*literally*.

Expanding Your Knowledge

First and foremost, let's talk about the general path you can take from here to become more familiar with the topics covered throughout the book. I think it's safe to say that compiler theory took center stage when compared to everything else, but the concepts discussed behind virtual machines are, at least in certain ways, equally complex.

Compiler Theory

Let's start with compiler theory. Because compilers are some of the oldest and most complex pieces of software in existence, they've been in constant development for decades, which is really just another way of saying there's a *lot* to learn. With the emergence of highly object-oriented languages and distributed computing, the load that bears down on compilers, linkers, and loaders is a considerable one.

To get you started, though, let's discuss some places to immediately go from here. The following topics will be, more or less, listed in order of increasing complexity, so try and pursue them in order.

More Advanced Parsing Methods

Like I've mentioned numerous times, this book has focused on recursive descent parsing because it's among the most natural and intuitive ways to parse code. However, bottom-up parsing, specifically shift/reduce, is far and away the chosen method of the compiler industry at large. Because of this, regardless of how you ultimately choose to parse code in your present and future projects, it's always a good idea to understand both top-down and bottom-up methods.

Most compiler texts focus heavily on bottom-up methods, so you shouldn't have trouble finding information on the topic. However, remember that everything has its ups and downs. Some of the particular disadvantages of bottom-up parsing include:

- Added complexity in the overall algorithm tends to make things harder in general to get working.
- Parsers sophisticated enough to handle full-scale programming languages are, for the most part, far too detailed to be written by hand, and therefore must be generated by an external utility like *yacc* (for UNIX/Linux users) or *Bison* (for Windows users).

Of course, because of the second reason mentioned, you won't have any trouble finding such parser generators. In fact, this disadvantage can also be seen as a huge up side, because it means that once you understand the language of a program like *yacc* (generally BNF or derivative thereof), you can get a parser up and running in minutes by simply hooking up the source code it generates to your compiler framework. To be honest, you don't even have to understand how shift/reduce works in the first place to get a parser generator's output to work. Of course, I'd highly recommend you do——it's always good, if not invaluable, to understand exactly how something works before using it.

The good thing about developing a compiler in a modular fashion like you have in this book, however, is that individual modules can be swapped in and out easily. Reworking your compiler to parse code with the shift reduce algorithm is confined entirely to replacing the parser module. No other major aspect of the program should need to change, as shown in Figure 17.1.

Object-Orientation

In addition to simply being another approach to language design, object-orientation often requires you to rethink the very structure of the compiler as well. Remember, objects aren't simply another feature in the language's bullet-list—they're an entirely new paradigm that shouldn't be taken lightly.

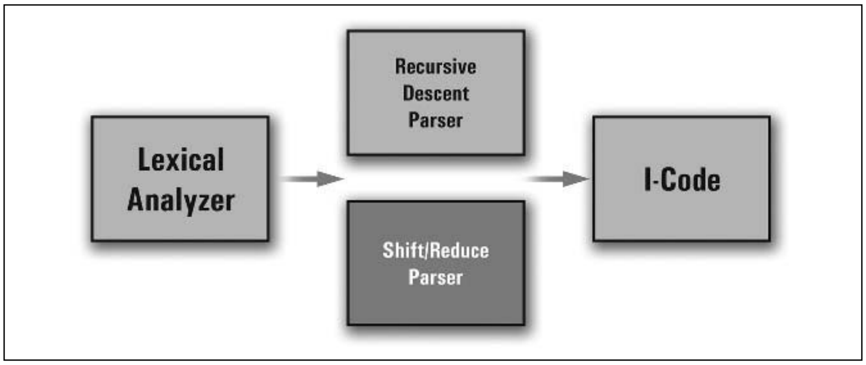


Figure 17.1

Swapping out the parser module and replacing it.

Yes, "OOP" has been quite a hot buzzword lately and will probably remain so for a while. But like all buzzwords, the subject should be approached with great caution. Do you really *need* objects to make your language work the way you want it to? Is it necessary, or are you just doing it to impress your message board buddies? A strong argument can be made both for and against the decision to include objects in your language's implementation.

On the one hand, objects are a highly intuitive and flexible way to represent game entities, so if your game engine itself is highly object-oriented, you might find it convenient, if not necessary, to do the same in your scripting language. Of course, objects and their associated design patterns are also orders of magnitude more complex than straight procedural programming, at least when used to their fullest extent, which means you're opening the door to all sorts of performance overhead and stability issues. Although object-oriented programming does have the potential to create extremely robust , error-resistant, high-performance programs in the hands of a seasoned pro, newbies and intermediate users are ironically capable of wreaking true havoc with sloppy or haphazard use objects.

My overall advice is to simply go with the facts and avoid hype. Decisions made based on what seems trendy at the moment almost invariably end badly–just look at Battlefield Earth–so make sure the design of your language puts efficiency and pragmatics above looking cool.

Optimization

Optimizing code is a complex black art that only the highest echelon of compiler writers can truly claim mastery of. It's a math-heavy field that requires a lot of studying and unfortunately can't be wrapped up in a nice tidy "silver bullet" algorithm that solves everything.

Of course, regardless of the complexity involved, optimization is one of the cornerstones of modern compiler construction, and certainly wouldn't hurt in your case, given the already significant overhead of virtual machine-based scripting. On the other hand, however, it's important to remember that performance overhead is the result of a number of factors, not just one. For example, while the quality of the compiler's generated code does indeed play a large role, the simple fact that scripts run in a virtual environment rather than directly on the native processor takes a significant toll as well. Here are some facts to keep in mind:

- Many scripts in full-scale game projects aren't particularly complex to begin with– like ambient background logic, for example– which means that even a highly optimizing compiler like Visual C++ wouldn't have a whole lot to work with in many cases.
- The only real culprit in our compiler's generated code is expression evaluation. Loops, conditional logic, and function calls are pretty lean by their very nature, which means the brunt of your optimization effort should be focused on expressions.

Artificial Intelligence

Something that may or may not surprise you is AI's role in optimization. If you think about it, the ability to perform large-scale optimizations on code is a very human ability; it requires extremely sophisticated pattern recognition, and a large and somewhat organic knowledge base of previous situations and general techniques. It almost goes without saying that the future of optimizing compilers lies in increasingly sophisticated AI that, rather than attempting to replace the human approach to optimization, will reproduce it instead. Fuzzy logic, genetic algorithms, and code evolution will be commonly used techniques within the next 5-10 years.

Runtime Environments

The XtremeScript virtual machine is a powerful and flexible runtime environment, with direct support for priority-based multithreading, a rich set of integration features, and other such details. It's still the tip of the iceberg, however, so here are some initial targets to set your sights on if you choose to further your study of runtime environments.

The Java Virtual Machine

The JVM is an extremely high-end virtual machine that's been in development for years as the Java language has evolved. Fortunately for you, there's also been a wealth of information published on it, in the form of white papers and books. Studying the internals of the JVM is a great way to further your overall understanding of VM architecture, and is a good way to drum up ideas for your own runtime environments. One aspect of real-world virtual machines I strongly suggest you explore is *garbage collection*, a method for automatically freeing dynamically allocated memory blocks so the programmer doesn't have to worry about them. Of course, since XtremeScript doesn't support dynamic allocation in the first place, the subject had little bearing up till this point.

One thing to keep in mind, however, is that the JVM is designed to mimic a far lower-level of processing than the XVM. In the context of game scripting, speed and relative simplicity are far more important than low-level control in most circumstances, so there are certain aspects of the system that you should recognize as inappropriate in the context of game scripting. A good general rule of thumb is that the higher-level the feature, the most applicable it is to your goals. Remember, unless you have a specific reason to do so, it's generally a good idea to keep your VM as high-level as possible, without encroaching on flexibility of course. As always, the more you can implement in C, the faster your results will be (that was an unfortunate rhyme).

Alternative Operating Systems

The PC gaming world currently revolves around Windows to be sure, but there are definitely other operating systems out there to consider. Namely, the Mac and Linux platforms are slowly picking up speed and may prove to be forces to be reckoned with in the future. Fortunately, virtual machines and cross-platform interoperability almost go hand in hand (after all, that's the principal Java was founded on).

What this means is that once a game is finished, its *entire* script-oriented aspect can be ported to other platforms by simply porting the VM. The scripts themselves, because they run on a purely virtual platform, never have to know about or interact with the underlying operating system (or even physical hardware), as shown in Figure 17.2. Understanding more about alternative operating systems opens up the possibility of porting your VM elsewhere, paving the way for full-on ports of your game.

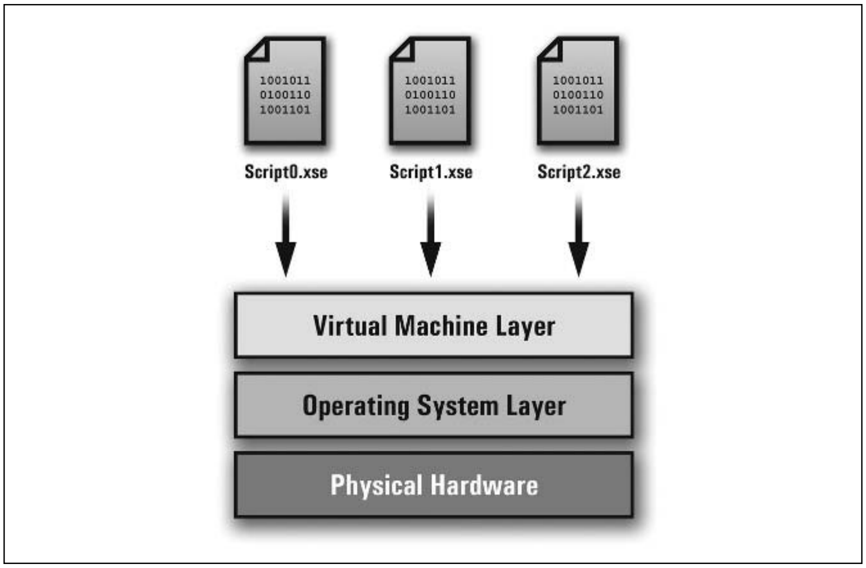


Figure 17.2

Once the VM is ported, scripts can run on any underlying hardware and operating system.

Operating System Theory

Aside from familiarizing yourself with the details of alternative operating systems for the purpose of porting, an understanding of general operating system theory can be invaluable when designing or redesigning your scripting system's runtime environment. After all, virtual machines are very closely related to operating systems, both in terms of architecture and overall purpose. Studying the low-level details of how operating systems are designed and implemented will provide an insight into how to structure your virtual machine in the ideal manner.

Advanced Topics and Ideas

Now it's time for some real fun. In addition to the general suggestions listed previously as places to go from here, I want to take some time and cover some specific topics and ideas that will hopefully spark your interest in further study and development.

The Assembler and Runtime Environment

Our assembler is reasonably sophisticated and more or less demonstrates everything a virtual bytecode assembler is responsible for, but there are plenty of ways that both XVM assembly language and XASM can be enhanced or changed.

A Lower-Level Assembler

Although high-level assemblers that directly support symbolic variables, arrays, functions and other such constructs are commonplace nowadays, this wasn't always the case. Furthermore, even today's assemblers for hardware platforms like the 80x86 are still *considerably* less abstracted and high-level than XASM.

For example, not only does the assembler directly support functions and function calls, but such a feature would be impossible to implement without XASM's specific syntax for doing so. Given the general inability to access the stack outside of the standard push-and-pop interface, a pure assembly script would have no way to construct and destruct stack frames on its own. Furthermore, internal values like the instruction pointer and the contents of the function table are completely hidden from the script, regardless of whether it was written in pure assembly, which results in additional limitations.

A lower-level compiler would not hide as many (or any) of these things, and instead give its assembly language less restricted access to more of the runtime environment's internal data. Of course, even then it's important to enforce some sort of security to prevent malicious or badly coded scripts from going nuts and blowing everything up. What follows are some ideas to consider if you decide to build an assembler with lower-level access in mind.

Random Access to the Stack

This can be as easy as defining a built-in array, perhaps called `_Stack []`, wherein each element maps directly to its corresponding stack index. This would allow any part of the stack to be written to and read from by the script itself at any time, allowing for greater flexibility. For one thing, parameters would be accessible without the `Param` directive. I actually considered doing this for the book's implementation of the XVM, but decided against it at the last minute for the purpose of just keeping things simple. Check out Figure 17.3.

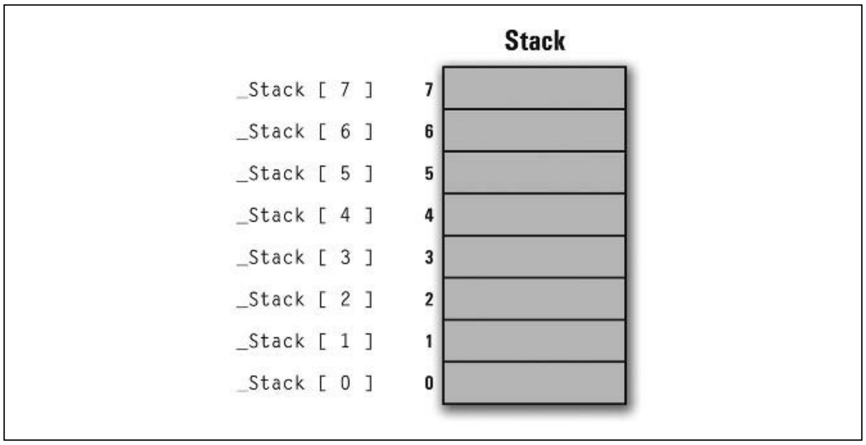


Figure 17.3

Accessing the stack randomly via a `_Stack []` array.

Stack Registers

Currently, the stack can be accessed relative to the bottom by using positive indexes, and relative to the top of current stack frame using negative ones. A lower-level assembler might instead only accept positive indexes, and provide registers to the top of the stack, and perhaps the top of the current stack frame as well. Scripts could then directly refer to these registers when accessing local data and parameters.

Of course, there's a lot to be said for high-level assemblers and runtime environments, as you'll see in the next section.

A Lower-Level Virtual Machine

Especially when compared to many existing VMs like the Java Virtual Machine, the XVM is an extremely high-level runtime environment. Its strongly typeless nature, combined with its highly specialized memory architecture, limits some of the lower-level tasks and capabilities often associated with assembly language programming. Check out some of these ideas for developing a lower-level VM.

Unified Memory

Currently, the XVM *enforces* separate regions of memory for a script's code and stack. Most hardware machines, as well as many virtual ones, take the opposite approach and instead provide a single, contiguous region of memory for a program's code and data. In such implementations, a particular subsection of this memory is reserved for code, called the *code segment*, whereas the stack is fenced off in an area called the *stack segment*. Although these two segments are indeed kept separate by convention and through some help from the assembler, they're by no means inaccessible from each other. For example, the code segment can be written to in order to change the behavior of a program at runtime, a technique known as *self-modifying code*. Overall, a unified memory system allows for greater flexibility when attempting to use esoteric techniques such as self-modifying code, loading machine code from the disk into the data segment for dynamic linking, and other such techniques. Check out Figure 17.4.

VM-Based Strings

The current string implementation occupies only one element of the virtual machine's memory because all the VM specifically needs to track is the string pointer. The actual string data always

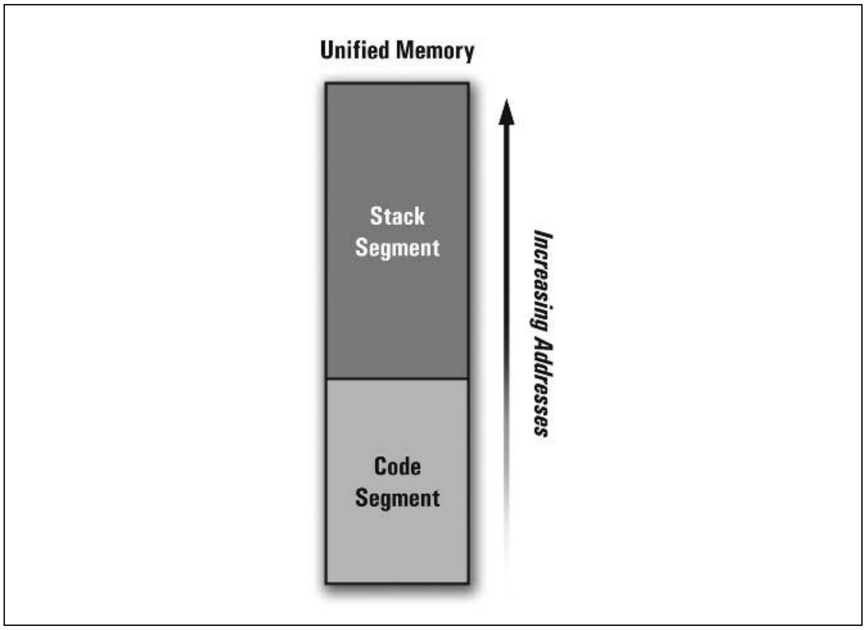


Figure 17.4

Unified memory keeps everything within the same address space.

resides in the host application's memory, making individual characters inaccessible unless `GetChar` and `SetChar` are used. A lower-level approach would be to give each element in memory the capability to hold a single character, rather than an entire string, so that contiguous regions of memory would be used to store strings character-by-character. This approach gives scripts greater flexibility when dealing with string data and allows for more elaborate and intricate string operations to be performed without specifically writing instructions to handle them.

High-Level or Low-Level VM?

So which is it? A high-level or low-level VM? The way I see it, high-level is almost always the way to go. I really only mentioned the low-level approaches to help you understand that virtual machines can be approached in a number of ways. The JVM, for example, must appeal to a huge range of software applications and provide low-level system access whenever necessary—especially in the case of higher-end software like Java-based Web servers, database drivers, and other business applications.

Scripting, on the other hand, is all about speed and simplicity (for the most part). Because of this, it's generally a good idea to keep things as abstracted as possible to ensure that the real underpinnings and performance critical sections of the system are implemented in C. Regardless, it's good to keep the possibilities in mind. Sometimes a hybrid is in order—a mostly abstract VM with some specific low-level facilities exposed. The rule of thumb is to always make a laundry list of your must-have features, and design a system that functions on as high a level as possible without compromising the list.

Dynamic Memory Allocation

Dynamic memory allocation can become important when scripts need to manage large amounts of data that will vary wildly in size from one instance to the next. In these cases, static arrays that hold the maximum number of elements needed can end up being a waste. Furthermore, the capability to allocate and free arbitrary chunks of data at runtime opens up the possibility of implementing high-level data structures like linked lists, trees, and hash tables, just as you would in C.

To support dynamic memory, the system really just needs to wrap `malloc ()` and `free ()` (or `new` and `delete` if you're a C++ user) in host API functions or perhaps new XVM instructions. The only real consideration to keep in mind is the ability to abuse this feature, because memory is always a crucial commodity that a malicious script could intentionally try to hoard from other processes. Of course, the real issue with dynamic memory allocation is that it almost requires that pointers be introduced into the language—something which I'll talk about later in this chapter.

The Compiler and High-Level Language

The XtremeScript compiler is undoubtedly powerful, and definitely well suited for the task of game scripting. Of course, there are countless ways to improve it and enhance its features, so let's talk about a few of them. You may find that attempting to implement some of the following suggestions will help you advance in your understanding of scripting and compiler theory in general far more than anything else, so take them seriously.

Language Enhancements

Right off the bat, there are probably a number of things you'd like your high-level language (or a modified version of XtremeScript) to support. Some of these are simple syntax additions, some are new code and data structures, and some may be entirely new paradigms.

switch

One commonly used feature of C/C++ that's absent from the implementation of XtremeScript is the `switch` block, which allows a single value to be tested against a number of conditions. Here's an example:

```
switch ( X )
{
    case 0:
        // X equals 0
        break;
    case 1:
        // X equals 1
        break;
    case 2:
        // X equals 2
       break;
    default:
       // X is none of the above
       break;
}
```

The actual implementation of this structure is rather simple. The compiler simply has to generate a unique label for each `case`, followed by a label at the very bottom of the structure's output that can be unconditionally jumped to in the case of `break` statements. In between each label and the jump to the end of the structure lies the code that implements each `case`. These blocks of code are invoked using conditional jumps based on the specified variable and each individual `case` value. Here's the possible assembly output for the previous code:

```
; Comparisons/jumps
JE          X, 0, Case0
JE          X, 1, Case1
JE          X, 2, Case2
Jmp         _Default:

; Case implementations
_Case0:
    ; X equals 0
    Jmp         _Break
_Case1:
    ; X equals 1
    Jmp         _Break
_Case2:
    ; X equals 2
    Jmp         _Break

; Default case
_Default:
     ; X is none of the above

; End of structure
_Break:
```

The code begins by comparing `X` to each specified `case` value and jumping to the proper handler. If none of the comparisons evaluates to true, the code vectors to a `default` case handler, which might not be specified by the high-level script. Each `case` handler starts with a label in the form of `_Case*`, where `*` is the value that `X` must equal in order to invoke the block. The code then immediately follows (represented in this example with comments), and the `break` statement is implemented with an unconditional jump to the `_Break` label. Of course, C's `switch` allows each `case` to "fall through" to the one below it by omitting the `break`, which can be implemented by simply suppressing the output of the `Jmp _Break` line in any `case` that doesn't end with `break`.

Structures and Other Forms of Aggregate Data

Structures and aggregate data are some of the major cornerstones of programming, and definitely have their applications in scripting. Internally, structures are really quite similar to arrays, which means you shouldn't have too much trouble implementing them if you take it slow and keep your thoughts organized. Imagine, for example, that the `struct` keyword was added to XtremeScript, like this:

```
struct MyStruct
{
    var X;
    var Y;
    var Z [ 16 ];
}
```

This structure can really be seen as an 18-element array, wherein `X` and `Y` are elements 0 and 1, and `Z [ 0 ]` through `Z [ 15 ]` are elements 3 through 17. Figure 17.5 presents an example of a structure and its representation on the stack. The only syntactic difference is that instead of using array index notation, like this:

```
MyStruct Q;
Q [ 0 ];          // X
Q [ 1 ];          // Y
Q [ 5 ];          // Z [ 2 ]
```

Elements are referred to by name (as well as an optional array index, in the case of `Z []`), like this:

```
MyStruct Q;
Q.X;
Q.Y;
Q [ 2 ];
```

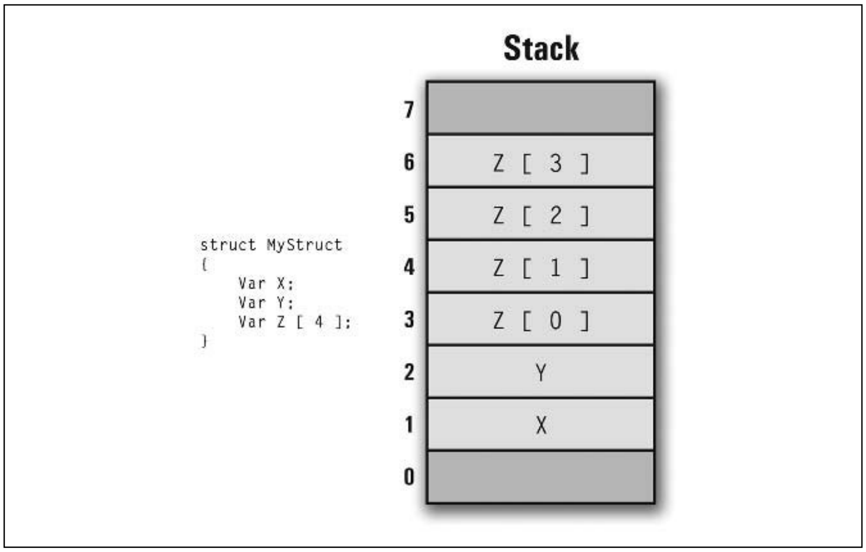


Figure 17.5

A structure represented on the stack.

Implementing structures up to this point is rather easy, because it really is just a reworked version of the already existing array feature. The real issues arise when you allow structures to contain references to other structures, and arrays of structures to be declared. Imagine the following scenario:

```
struct StructX
{
    var Elmnt0;
    var Elmnt1;
    var Elmnt2;
}

struct StructY
{
    var Elmnt0;
    var Elmnt1;
    StructX Elmnt2;
}

struct StructZ
{
    var Elmnt 0;
    StructX Elmnt1;
    StructY Elmnt2 [ 8 ];
}

StructX MyX;
StructY MyY [ 16 ];
StructZ MyZ [ 32 ];
```

As you can imagine, there's a *lot* more going on here than there was in the previous example. Structures are nested within other structures, arrays are defined with structure elements, and so on. It's now possible to encounter scripts with code like this:

```
MyY[MyX.Elmnt1].Elmnt2.Elmnt0 = MyZ[4].Elmnt2[MyZ[4].Elmnt0];
```

Of course, this all looks a lot harder than it actually is. The most important key to remember when implementing structures is *recursion*. When the parser encounters a structure reference, it needs only call a `Parse* ()` function capable of parsing structure field references, which in turn may call itself. As long as this function can also parse array elements, any level of structure nesting can be supported.

Pointers and References

Currently, the only method of indirection supported by XtremeScript is the use of variables and arrays to reference literal values. Pointers and references, however, add an additional level of indirection wherein variables can point to other variables.

As an example, consider the following pointer syntax for XtremeScript:

```
var MyVar;              // Declare a variable
var * pMyVar;           // Declare a pointer

pMyVar = & MyVar;       // Point pMyVar at MyVar
* pMyVar = "Hello!";    // Assign a value to MyVar through the pointer
```

This example introduces two new operators, the pointer dereference operator `*` and the address-returning operator `&`, both of which behave like their C counterparts. Internally, the addition of pointers really isn't that difficult. Currently, the runtime environment's `Value` structure allows operands within the instruction stream to reference values on the stack using the `iStackIndex` field. By allowing stack values to use this field as well, they can reference other stack values, and effectively become pointers. This is expressed visually in Figure 17.6.

The only other real issue is expressing this new functionality using the syntax of the assembler. Due to the high-level nature of the assembler, there are two ways to approach this problem.

The first is to simply add pointer-specific syntax to the assembler as well. The `&` operator can be translated to assembly with the addition of a new instruction, like so:

```
LEA        X, Y                    ; Put the address of Y into X
```

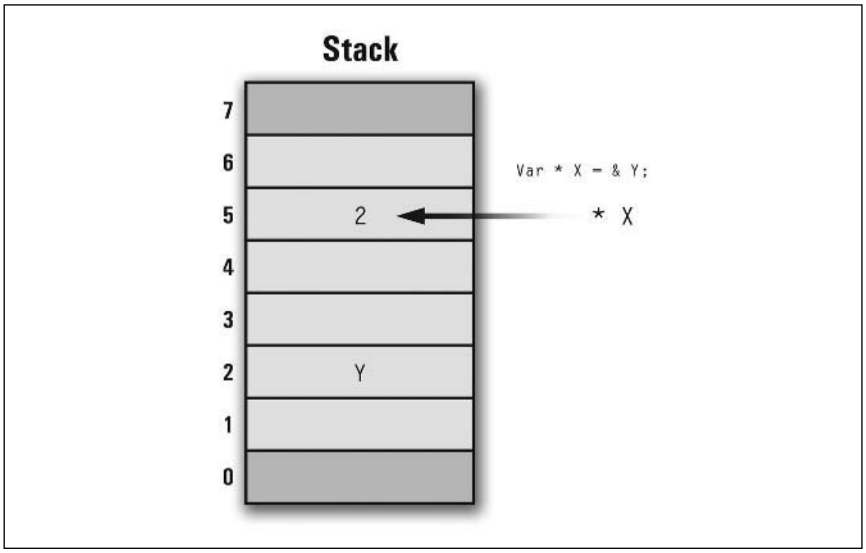


Figure 17.6

Pointers allow further indirection among variables.

I took the mnemonic from the 80x86's `LEA` instruction, an acronym that stands for **L**oad **E**ffective **A**ddress. This instruction is used to determine the address of the specified identifier, and is more or less analogous to what you're doing here.

Once an XASM variable has been assigned the stack index of another, you need a way to tell instructions like `Mov` and `Add` that you're passing a pointer to another variable, not a literal value. For example, even though `X` was assigned `Y`'s stack index, the following instruction would simply add that address to the variable `Z`:

```
Add        Z, X
```

What you actually want to do is add the value *pointed to* by `X`, which is the value of `Y`. You can borrow some more 80x86 syntax to tell an instruction when the value of the specified variable should be interpreted as a reference to a stack address:

```
Add        Z, [ X ]
```

The `[]` notation tells the instruction that the value of `X` is the index into the stack where the real value resides. It's no coincidence that this syntax looks so much like array notation; because the stack is a contiguous block of memory accessed with integer indexes, it really is just one big array. It's like the `_Stack []` array I suggested earlier, just without the `_Stack` identifier.

Basic Object-Orientation

Lastly, if you're really feeling brave, you can take the `struct` idea to another level by adding the capability to embed functions within them. Here's an example:

```
class MyClass         // Define a class
{
    var MyProperty;         // Define a property
    func MyMethod ();       // Declare a method
}

func MyClass::MyMethod ()   // Define the method
{
    MyProperty = 3.14159;   // Set the property
}

MyClass MyObject;             // Create an object of the class
MyObject.MyProperty = 0;     // Set the property
MyObject.MyMethod ();           // Call the method
```

The function is first declared to be within the scope's class, and is then defined later in the script using the :: scope resolution operator from C++. Notice also that `MyProperty` isn't defined within the scope of the function, but is referenced anyway. This is because class methods and proprieties share the same scope.

Remember, aside from the addition of functions, a `class` is implemented just like a `struct`, so you can get a basic idea of how they work from the previous section on structures. Methods are really quite an easy addition; they can be represented internally just like any other function, with the only difference being the syntax by which they're called. Remember, even though a class may have many instances at runtime, its methods have to exist in only one place.

NOTE

In case you're a straight-C programmer and are confused by all this C++ crazy talk, here's some quick info: *properties* are a class's variables, *methods* are a class's functions, and the *scope resolution operator* is used to bind a function definition to a specific class (hence, to resolve its scope).

Additional Object-Oriented Functionality

Once you have a basic OO framework up and running, you can add many common OOP features rather easily. Access modifiers, such as `public`, `private`, and `protected`, can be resolved entirely at compile-time, because all they really do is limit the way a class's members are referenced within the script. Composition, single inheritance, and friend classes aren't too terribly difficult either, because all they really do is increase the number of members that a given object can reference at any given time.

The real issues arise when virtual functions and dynamic casts come into play. They have an effect on an object's runtime behavior. Such additions often affect the entire scripting system, from the compiler all the way down to the runtime environment.

In general, I strongly suggest you attempt to add basic OO to your language with single inheritance. If you can manage structures, you can definitely implement this much without too much headache, and you'll have a very intelligent method of organization to work with in your future scripting projects. Especially for users of object-oriented game engines, a scripting system with basic support for classes and objects can be quite helpful. Above all else, however, it's a great learning experience.

Directly Compiling to Executables

The XtremeScript system begins on one end with the XtremeScript compiler, and ends on the other with the XVM. In between, XASM facilitates the translation of the assembly language output generated by the compiler to a binary executable ready to run in the virtual machine. This approach boasts many advantages, such as:

- A simplified compiler that can directly leverage the features of the assembler in the code it outputs.
- Minimized redundancy; because the assembler is already translating assembly to executable files, there's no need to bend over backwards to make the compiler do the same thing.
- The ability to directly hand-tune, optimize, or otherwise modify the output of the compiler, because it's entirely human readable.
- A clearer translation from high-level code to executable bytecode; by manually adding an intermediate assembly step, the process is easier to grasp. This is particularly advantageous in the case of a book.

However, most modern compilers don't work this way, and instead directly output machine code. This is definitely a more compact approach, and ultimately means faster compile-times because there are no temporary files to generate or intermediate steps to perform. The only real difference is that instead of translating I-code to XVM assembly, it's converted to XVM bytecode. Because bytecode instructions have a one-to-one mapping with instruction mnemonics, this is a pretty easy change. The complexity lies in reproducing the assembler's other features, such as managing the stack layout of a script's local and global variables, building a function table that can be used at runtime, and properly formatting an XVM executable. However, reworking the compiler to directly output .XSE's doesn't require anything you didn't learn during the development of XASM.

In addition, a compiler that can directly generate executable code can be used in a number of other applications, which I'll discuss now.

An Embeddable Compiler Module

If you remember back to the discussion of Lua, you'll remember that compiling source code was optional. You could either pass it through `Lua` to get a compiled version that would be loaded and run more quickly, or the Lua application could directly load source code, which would be compiled at runtime in memory. And don't forget the handy `lua` interpreter, which directly interpreted and executed source code as it was typed into the console.

All of these capabilities are made possible with a compiler that is embeddable as a self-contained module, much like the virtual machine. When the compiler is implemented in this way, and defined with a single input (a source file) and a single output (the in-memory representation of a compiled script), it can be dropped into any program and immediately put to use.

The advantages of this approach should be obvious:

- **Eased development process.** By making the standalone compiler optional, the constant tweaking and updating that will invariably be a large part of game scripting can be eased by eliminating the intermediate compile step. Scripts can be immediately loaded by the game engine, which tends to be much faster and easier when repetitive modifications are being made.
- **User development tools.** Scripting isn't just a tool for developers—it's a great way to give users (players) more control and input over the game. In addition to mod authors, even more casual players stand to gain from a simple scripting language that allows them to exert more complex control. Imagine a real-time strategy game that let players write entire scripts to control the deployment and behavior of units, allowing self-reliant, autonomous CPU players to run in parallel with the human player in the pursuit of a common goal. Users won't want to deal with a compiler, and may be put off by the additional complexity they associate with it. Allowing them to directly run human-readable source is a much more intuitive alternative.
- **Standalone interactive interpreters.** Just like the interpreters that came with Lua, Python, and Tcl, an XtremeScript interactive interpreter could be built that would allow individual lines of code or small script fragments to be immediately tested without a separate virtual machine, host application, or extraneous source and executable files.

Remember, the key to a good embeddable compiler is a strongly defined interface, as shown in Figure 17.7. The host application should be able to load and compile a source file with a single function, by providing a source filename and a pointer to a `Script` structure that will be filled with the fully compiled results. This way, one function call is all it takes to get the job done. Such a clean and simple interface will allow you to immediately put the system to use.

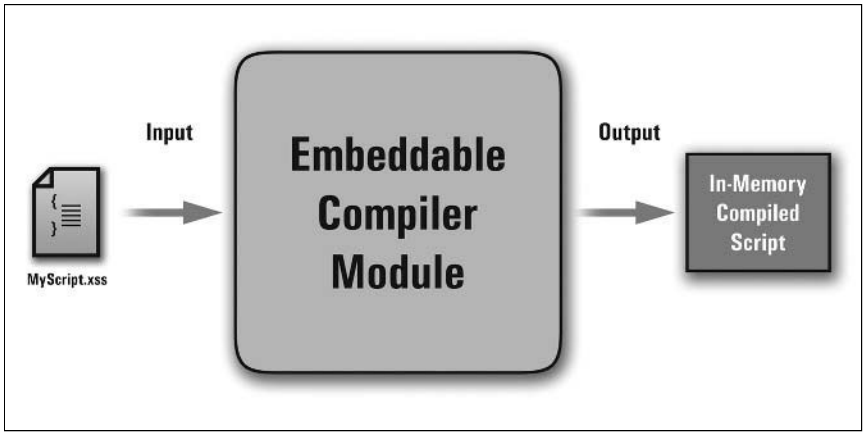


Figure 17.7

An ideal interface for an embeddable compiler module.

Summary

Well, this has been quite a little journey, eh? If you're anything like I was, you probably thought the idea of building a high-level compiler and suitable runtime environment was impossible for mere mortals, and yet here you are—as long as you've followed along all this way, you too have ascended to the rank of scripting master. Sure, you've still got a lot to learn—recursive descent parsing is somewhat elementary approach, and the language you're dealing with isn't the most sophisticated one in the world. But of course, just as there are varying degrees of black belts, there are many levels of mastery.

The bottom line is that you've hopefully learned exactly how game scripting works, from the design of a high-level language to its final execution in an embeddable, virtual environment. You now have complete external control over the games you make, and have learned the fundamentals behind all sorts of high-level language processing and translation. In addition to compilers, you should be able to apply your newfound skills to interpreted, user-end scripting languages, the processing of player-inputted dialogue for complex RPGs or text adventures, and a multitude of other tasks.

Furthermore, now that you know the basics, you're free to go nuts and take everything to the next level—I encourage you to add some of the additional features discussed in this closing chapter, as well as any other ideas you have. Remember—your creativity is the only real barometer for what a language should consist of—everything from its syntax to its major constructs and features are up to you now. Go by the examples set by other languages when you feel you stand to gain from it, and let your imagination run wild when you don't.

To put it simply, game scripting is a complex task, but one that's becoming more and more of a necessity in the world of game development. As games become more cinematic and complex, it becomes increasingly important to isolate these artistically driven aspects of a game's functionality, just as art, sound, and other data have been for years. But as has always been the case, truly memorable games are not driven by technology or mile-long feature lists—they're driven by genuine creative vision that *utilizes* technology, rather than hides behind it. Scripting isn't a magic wand that will make your game better—it simply provides a far greater structure within which an already good game will thrive.

In closing, my final word of advice is to use scripting for what it is. Choose an existing scripting system, build your own, or even use the one I've provided on the CD. No matter what option you ultimately go with, though, take advantage of it to its fullest extent. Scripting gives you the freedom to bring the interactivity and immersion of your game world to a new level—where characters live and breathe, where every object has function to match its form, and where the events that will ultimately carry players to the game's conclusion are described and presented with the utmost of clarity. A game's greatest asset is its suspension of disbelief—its ability to remove the players from reality and drop them head-first into a self-contained world—and *this* is what scripting is all about.

So, that's that. I hope you've learned as much from this book as I attempted to explain. When I first set out to solve the mystery of high-level scripting, my only options were esoteric and rather dull textbooks intended for college courses. What I wanted was a book that spoke to a person like *me*—a game developer who just wanted a powerful way to control his game and the entities therein—and that was the motivation for this book's approach. I certainly hope this has saved you some headaches by allowing you to bypass this decidedly inconvenient route, and I hope you enjoyed it!

Good luck!

—Alex Varanese
`alex@amvbooks.com`

APPENDIX A

What's on the CD

The included CD-ROM contains a number of supplemental materials to enhance your experience with the book. They're organized into the simple directory structure listed here:

- `Articles/` - A small collection of articles that discuss aspects of scripting not directly covered in the book.
- `Programs/` - Contains the entire set of code and executable demos for the book's chapters. This folder is broken down into subfolders for each chapter. For example, Chapter 12's code and executables can be found in `Programs/Chapter 12/`. Within each chapter folder you'll find a Read Me! file that briefly introduces the programs and provides instructions on how to compile them.
- `Software/` - A number of programs that I think pertain to scripting in some way. Examples of included programs are Flex and Bison, as well as text and hex editors and parser generators.
- `Scripting Systems/` - This folder contains scripting systems for you to use in your games and programs.
- `XtremeScript/` - Over the course of the book, we develop the XtremeScript scripting system. Rather than let you hunt through the program demos to find the completed version, I've collected everything– the compiler, assembler, virtual machine and stand-alone VM console, and put them in one place.

Each folder contains a Read Me!.txt file with important information about the folder's contents, and any instructions for compiling or installing it. It's important that you read them, but if you still find that you are having trouble with something, don't hesitate to email me about it at

alex@amvbooks.com

I'm always available to help out with book-related issues.

The CD-ROM Interface

Also included on the CD-ROM is a graphical, HTML-based interface you can use to easily browse the disc's contents. Since the interface is web-based, you'll need a 4.0 browser to view it. I recommend Microsoft Internet Explorer.

Installation

Installation is simple; some programs included have their own executable installers or self-extracting archives, while the rest of the content—namely the program demos and code—are "installed" by simply dragging them from the CD to your hard drive. The GUI should run automatically on its own, but if if it doesn't, just use a program like Windows Explorer or the My Computer icon on your desktop to navigate your way to the contents and manually install or drag whatever you need.

DirectX SDK

Lastly, you'll need the DirectX SDK to view the book's graphical demos. If you don't already have it, or don't have the most recent version (8.1 at the time of this writing), you can install it on your system directly through the CD-ROM GUI, or run the executable installer found in the CD's `DirectX/` folder.

CAUTION

Files copied from a CD-ROM are often tagged with an "Archive" or "Read-Only" flag. This flag is initially set because a CD-ROM's contents can't be rewritten, but once you've dragged a copy onto your hard drive, this limitation no longer applies. However, your file system or shell will often leave this flag set, so make sure to change it manually yourself. Forgetting to do so will make the program demos' source code read-only, for example. To do this on a Windows machine, select all of the folders and/or files you've dragged from the CD, right-click them to bring up their collective Properties dialog box, and uncheck both the "Archive" and "Read-Only" check boxes. Press Apply and you should be good to .

INDEX

A

abstraction layer, 174–179
ActiveStateTcl, 288
AI (artificial intelligence), 57
    compilers, 1184
    enemies, 57–60
allocating memory directly, 1189
analysis
    parsing, 985–987
    semantic (compiling), 764
APIs, 20
    hosts. *See* host APIs
    SDKs, 24
applications. *See* host applications
architecture. *See also* structure
    hardware, targeting, 780–781
    modular, 31
    procedural scripting systems, 156–157
    XVM, 569–570, 582–589
arithmetic (XVM Assembly), 400–401
arrays
    associative (Lua), 193–197
    flags, 33–34, 38
    multithreading, 667–677
    parsing, 1017–1020
    Tcl, 301–303
    XtremeScript, 353–354
artificial intelligence. *See* AI
assemblers. *See* XASM
assembling, 753
    function calls, 423–428
    instructions, 414–416
    jumps, 423–428
    literals, 422–423
    lower-level, 1186–1189
    operands, 420–422
    procedural scripting systems, 167–168
    strings, 422–423
    variables, 416–420
    XSE executable, 558–563
assembly languages, 17
    CISC, 386–388
    conditional logic, 377–380
    defined, 370–371
    expressions, 340–344, 373–375
    Fibonacci Sequence, 344–346
    functions, 344–346, 389–392
    instructions, 337–344, 372
        orthogonal, 388–389
    iterating, 375–383
    jump instructions, 375–383
    libraries, 344–346
    loops, 375–383
    macro assemblers, 374
    mnemonics, 383–385
    OOP, 346–349
    opcodes, 383, 384, 385
    operands, 337–344, 372–373
    operators, 340–344
    parameters, 392–395
    recursion, 344–346
    registers, 389
    RISC, 386–388
    scope, 344–346, 395–397
    stacks, 389–397
    values, 392–395
    variables, 395–397

assembly languages (continued)
- XVM Assembly
  - arithmetic, 400–401
  - bitwise, 401
  - comments, 407
  - conditional logic, 402–403
  - defined, 397–399
  - directives, 404–407
  - escape sequences, 407
  - functions, 403–406
  - instructions, 399–404
  - memory, 399–400
  - overview, 408
  - stacks, 403–405
  - strings, 402

assignment statements, parsing, 1065–1072
associative arrays
- Lua, 193–197
- Tcl, 301–303

asynchronous script function calls, 719–728
atomic operations (multithreading), 661–664

B

back end compiling, 768
- code emitter module, 863
- XASM, 863

Backus-Naur Form (BFN), 988–989
binary format (CBS), 137–146
binary operations (XVM), 638–639
bitwise XVM Assembly, 401
BlitFrame function, 239
BlitSprite function, 239
BNF (Backus-Naur Form), 988–989
Boolean constants, 115
bouncing sprite demo, 181–184
- Lua, 228–241
- Python, 277–286
- Tcl, 322–329

branching (parsing), 1092–1099
brute force lexers, 789
building XVM, 589
bytecode, 753

C

C
- functions
  - exporting, 271–276
  - Lua, 215–219
  - Tcl commands, 316–320
- integrating
  - Lua, 205
  - Python, 263
  - Tcl, 312

calling
- commands (Tcl), 315–316
- functions
  - asynchronous script, 719–728
  - expression parser, 1051–1053
  - host API, 699–711
  - host applications, 686–694
  - parsing, 1073–1079
  - Python, 268–271
  - script, 711–728
  - synchronous script, 713–719
  - XVM, 578–581

CallLuaFunc function, 236
cascading errors, 930–932
case-sensitivity, Tcl, 291
CBS (command-based scripting), 64–65
- advanced, 114
- binary format, 137–146
- Boolean constants, 115
- CD, 111
- code blocks, 128–131
- commands, 68
  - extracting, 81–87
  - handlers, 87–90
- compiler overview, 140–142
- compiling errors, 139
- concurrent execution, 109–110
- conditional logic, 125–128
- constants
  - executing, 124–125
  - loading, 124–125

preprocessing, 120–124
data types, 115–125
designing, 74
domains, 68
engines
functionality, 69–71
high-level control, 65–67
events, 69
hierarchy, 135–137
executing, 78–81
floating points, 115–116
game flags, 125–128
game intro, 90
implementing, 93–94
language, 91–92
script, 92–93
hacking, 139–140
implementing, 74–90
interfaces, 75–78
internal constant lists, 117–120
logic (iterative), 131–133
loops, 68
nesting, 133–135
parameters, 144–146
extracting, 81–87
preprocessing, 143–150
RPGs
characters, 95–108
implementing, 101–105
language, 95–97
loops, 105–108
motion, 97–99
scripts, 99–100
scripts
executing, 71–74, 142–143
loading, 71–78
looping, 73–74
speed, 137–139
symbolic constants, 116–117
tiles, 69–70
writing, 75

CD
CBS, 111
compiler, 981
lexers, 855–856
Lockdown, 1177
parsing, 1134–1135
scripting systems, 334
XASM, 564
XVM, 649, 746
characters
lexing, 785–786
NPCs, 34–41
RPGs
CBS, 95–108
implementing, 101–105
language, 95–97
loops, 105–108
moving, 97–99
scripts, 99–100
CISC (Complete Instruction Set Computing), 386–388
code
blocks
CBS, 128–131
parsing, 1001–1007
XtremeScript, 358
bytecode, 753
compiled, 24–26
engines
compile time, 6–13
runtime, 10–12
expression parser, 1037–1048
high-level
procedural scripting systems, 157–158
XtremeScript, 162–166
I-code. *See* I-code
interpreted, 24–26
linking, 779–780
loading, 779–780
Lockdown, 1155–1157

code (continued)
    low-level
        procedural scripting systems, 158–159
        XtremeScript, 167–168
    machine, 17, 753
    opcodes, 17
    relocatable, 779–780
    source
        compilers, 863–864, 919–922
        I-code, 940–942
        XASM, 470–471
code-emitter module
    compiler, 863, 950–969
    directives, 953–955
    format, 950–951
    functions, 958–966
    headers, 952–953
    parsing functions, 1026–1028
    symbols, 955–958
    XVM Assembly files, 966–969
coercion, data types (Lua), 192
command-based scripting. *See* CBS
command-based scripting systems, 22–23
command-line compilers, 874–879
commands
    case-sensitivity, 291
    CBS, 68
        extracting, 81–87
        handlers, 87–90
    Tcl, 290–292
        C functions, 316–320
        calling, 315–316
comments
    Lua, 188
    Python, 244
    Tcl, 297, 298
    XtremeScript, 362
    XVM Assembly, 407, 442
communication, inter-script, 59
compile time code, 6–13
compiled code, 24–26
compilers. *See also* compiling
    AI, 1184
    assembling, 753
    back end
        code emitter module, 863
        XASM, 863
    bytecode, 753
    CBS overview, 140–142
    CD, 981
    code
        linking, 779–780
        loading, 779–780
        relocatable, 779–780
    compiling, 753, 769
    data structures
        linked lists, 880–888
        stacks, 888–890
    demos, 1099–1134
    encapsulation, 866–867
    error handling, 928–932
    front end, 859
        lexer module, 861
        loader module, 860
        parser module, 862
        preprocessor module, 861
    functions, 865, 910–915, 922–927
    global variables, 890–891
    hard-coding, 975–981
    hardware architectures, targeting, 780–781
    headers, 864
    high-level languages, 753, 1190–1198
    I-code, 866
        instructions, 933–938
        interface, 942–949
        jump targets, 938–940
        source code, 940–942
    I-code module, 862
    initializing, 891–892
    initiating, 972

interface, 870
command-line options, 874–879
filenames, 871–874
logos, 870–871
interfaces, 866–867
life-span, 867–870
low-level languages, 753
luac, 185–186
machine code, 753
modules
code-emitter, 950–969
code-emitter. *See* code-emitter module
I-code, 932–949
lexer, 916–928
loader, 895–897
overview, 893–895
parser, 928
parser. *See* parsing
preprocessor, 897–904
OOP, 1182–1183
optimizing, 771–772, 1183–1184
parsing, 1182
platforms, retargeting, 778–779
preprocessing, 773
files, 773–775
macros, 776–777
printing statistics, 972–975
shutdown, 892–893
source code, 863–864, 919–922
strategy, 858–859
strings, 866, 915
symbols, 864–865, 905–910
theory, 752–753
tokens, 916–919
XSE executables, 969–971
XtremeScript. *See* XtremeScript
compiling. *See also* compilers
back end, 768
CBS errors, 139
compilers, 769
front end, 768
I-code, 765
lexers, 757–760
lexing, 755–757
parsing, 760–764
passes, number of, 766–767
platfroms, 768
procedural scripting systems, 162–166
projects
Lua, 206–207
Python, 263–265
Tcl, 312–313
semantic analysis, 764
steps, 753–755
Tcl, 290
tokenizing, 755–757
Complete Instruction Set Computing (CISC), 386–388
concurrent execution, 109–110
multithreading, 659–666
conditional logic
assembly languages, 377–380
CBS, 125–128
Lua, 200–201
parsing, 1092–1099
Python, 256–258
Tcl, 306–308
XtremeScript, 358–360
XVM, 640–641
XVM Assembly, 402–403
conditional statements, parsing, 1092–1099
constants
Boolean, 115
CBS
executing, 124–125
loading, 124–125
preprocessing, 120–124
internal lists, 117–120
Lua, 215
public, 696
symbolic, 116–117
content, 15

context switches, 655
  multithreading,679–682
cooperative multitasking, 654–658
core (Tcl), 290
counting references (Python), 266
critical sections, multithreading, 663–664

D

data structures
  compilers
    linked lists, 880–888
    stacks, 888–890
  Lua, 241
  XtremeScript, 351–354
data types
  CBS, 115–125
  Lua, 191–193
    coercion, 192
  Python, 246
debug libraries (Python), 264–265
declarations, parsing, 1008
  arrays, 1017–1020
  code-emitter module, 1026–1028
  functions, 1008–1017
  host API functions, 1020–1026
  variables, 1017–1020
define keyword, 361
delimiters, lexers, 822–826
demos
  bouncing sprite, 181–184
    Lua, 228–241
    Python, 277–286
    Tcl, 322–329
  compiler, 1099–1134
  lexers, 849–855
design
  CBS, 74
  XtremeScript, 349–350
dictionaries, modules, 269–270
directives
  code-emitter module, 953–955
  Func, 432–434
  Param, 436–438
  SetStackSize, 431–432
  Var, 434–436
  XASM parsing, 529–541
  XVM Assembly, 404–407, 431–439
directories
  Python, 243
  Tcl, 288–289
displaying lexer results, 809–811
domains (CBS), 68
dynamically linked module scripting systems,
    23–24

E

elseif statements (Lua), 200–201
embeddable scripting systems, 179
embedding (XVM host applications), 741
encapsulation, compilers, 866–867
enemies
  FPSs, 57–60
  RPGs, 45–50
engines
  code
    compile time, 6–13
    runtime, 10–12
  defined, 15
  functionality, 69–71
  high-level control, 65–67
entry points, 18
  XVM, 576
error handling
  cascading errors, 930–932
  CBS, 139
  compilers, 928–932
  lexers, 797
  Lua, 209
  multithreading, 672–673
  XASM, 525–527
escape sequences
  Lua, 197–198
  XtremeScript, 362
  XVM Assembly, 407

events
 CBS, 69
 FPSs, 52
 hierarchy, 135–137
exceptions
 Python, 286
 Tcl, 330
executables
 XASM, 444–455
 XSE
  assembling, 558–563
  compilers, 969–971
  functions, 556–558, 601–603
  header, 552–553, 594–595
  host APIs, 557–558, 602–603
  host applications, 731–732
  instructions, 553–555, 595–599
  strings, 555–556, 599–601
executing
 CBS, 78–81
 concurrent multithreading, 659–666
 concurrently, 109–110
 constants, 124–125
 scripts
  CBS, 71–74, 142–143
  Lua, 219–221
  XVM, 576–577
 XVM, 627–628
  binary operations, 638–639
  conditional logic, 640–641
  functions, 642–645
  instruction pointers, 634–636
  instructions, 628–647
  operands, 636
  pauses, 633–634, 646
  terminating, 646–648
exporting functions (Python), 271–276
expression parser
 coding, 1037–1048
 factors, 1048–1051
 function calls, 1051–1053
 operators, 1053–1058
 overview, 1033–1036
 values, 1058
expressions
 assembly languages, 340–344, 373–375
 Lua, 198–200
 parsing, 1028–1033
 Python, 254–256
 Tcl, 303–306
 XtremeScript, 354–358
extensions
 Lua, 241
 Tcl, 290, 330
extracting (CBS), 81–87

F

factors (expression parser), 1048–1051
Fibonacci Sequence, 344–346
files
 code-emitter module, 966–969
 external functionality, 14–15
 lexers, 793–795
 compilers
  filenames, 871–874
  preprocessing, 773–775
first-person shooters. *See* FPSs
flags
 arrays, 33–34, 38
 CBS, 125–128
floating points, 115–116
for loops, parsing, 1092
format, code-emitter module, 950–951
FPSs (first-person shooters), 50
 enemies, 57–60
 events, 52
 inter-script communication, 59
 objects, 51–57
 puzzles, 51–57
 switches, 51–57

framework, XASM, 469, 494–495
    functions, 479–482
    headers, 473
    host API, 487
    instructions, 471–473, 487–494
    interface, 470
    labels, 485–487
    linked lists, 474–477
    source code, 470–471
    strings, 477–479
    symbols, 482–485
front end compiling, 768, 859
    lexer module, 861
    loader module, 860
    parser module, 862
    preprocessor module, 861
Func directive, 432–434
functionality
    engines (CBS), 69–71
    external files, 14–15
functions
    assembly languages, 344–346, 389–392
    BlitFrame, 239
    BlitSprite, 239
    C
        Lua, 215–219
        Tcl commands, 316–320
    calling
        assembling, 423–428
        asynchronous script, 719–728
        expression parser, 1051–1053
        host API, 699–711
        host applications, 686–689
        parsing, 1073–1079
        Python, 268–271
        script, 711–728
        synchronous script, 713–719
        XVM, 578–581
    CallLuaFunc, 236
    code-emitter module, 958–966
    compilers, 865, 910–915, 922–927
    GetCommand, 82–84
    GetCurrLexeme, 497
    GetCurrTime, 634
    GetIntParam, 84–85
    GetLookAheadChar, 497–498
    GetNextToken, 496
    GetStringParam, 85–87
    global, 223
    HandleFrame, 238–240
    inline, 361
    instructions, 373
    inter-language, 180
    intra-language, 180
    Lua, 203–205
        importing, 221–226
    MultiplyString, 222–223
    parsing, 1008–1017
        host APIs, 1020–1026
    Print, 192
    PrintStringList, 219–221
    public, 694–695
    Python, 261–263
        exporting, 271–276
        list, 286
    ResetLexer, 498
    script control, 697–699
    SkipToNextLine, 498
    Tcl, 310–312, 316–320
    XASM, 479–482
        parsing, 531–534
        XVM Assembly input, 432–434, 440–442
        XVM Assembly output, 453–455
    XSE executable, 556–558, 601–603
    XtremeScript, 360–361
    XVM, 587–588, 601–603
        executing, 642–645
        structure interfaces, 621–623
    XVM Assembly, 403–406
        input, 432–434, 440–442
        output, 453–455

G

games
  content, 15
  engines, 15
  flags (CBS), 125–128
  intro sequence, 90
    implementing, 93–94
    language, 91–92
    script, 92–93
  Lockdown
    code, 1155–1157
    graphics, 1151–1153
    host API, 1158–1161
    logic, 1142–1150
    playing, 1175–1177
    premise, 1140–1141
    scripts, 1161–1173
    sound, 1153–1154
    speed, 1173–1175
    state, 1155–1157
    storyboards, 1142–1150
    XtremeScript, 1158
  logic, modular, 31
GetCommand function, 82–84
GetCurrLexeme function, 497
GetCurrTime function, 634
GetIntParam function, 84–85
GetLookAheadChar function, 497–498
GetNextToken function, 496
GetStringParam function, 85–87
global data tables (XVM), 571–572
global functions, 223
global variables, 226–228
  compilers, 890–891
  Tcl, 320–322
  tracking, 689–694
graphics (Lockdown), 1151–1153

H

hacking (CBS), 139–140
HandleFrame function, 238–240
handlers, command, 87–90
hard coding, 6–13
  compiler, 975–981
hardware architectures, targeting, 780–781
hash tables (Tcl), 301–303
headers
  code-emitter module, 952–953
  compilers, 864
  XASM, 473
    XVM Assembly output, 445–447
  XSE executable, 552–553, 594–595
  XVM, 583, 594–595
hierarchy, events, 135–137
high-level code
  procedural scripting systems, 157–158
  XtremeScript, 162–166
high-level engine control, 65–67
high-level languages, 753, 1190–1198
host APIs
  function calls, 699–711
  Lockdown, 1158–1161
  Lua, 229–230
  parsing, 1062–1064
  parsing functions, 1020–1026
  Python, 273–278
  Tcl, 323
  XASM, 440–441, 454–455, 487
  XSE executable, 557–558, 602–603
  XVM, 587–588, 602–603
    host applications, 742
    structure interfaces, 621–624
host applications, 18–20
  Lua, 230–234
  parsing, 1058–1062
  Python, 278–281
  Tcl, 323–325

host applications (continued)
XVM, 573–574, 682
asynchronous script function calls, 719–728
calling functions, 686–689
control functions, 697–699
embedding, 741
host API function calls, 699–711
host APIs, 742
integration interface, 686–728
multithreading, 728–739
native threads, 684
output, 745–746
priorities, 730–731, 734–735
public interface, 694–696
running scripts, 683–685
script function calls, 711–728
scripts, 739–745
synchronous script function calls, 713–719
time slicing, 684, 730–731
tracking global variables, 689–694
updating, 735–739
XASM, 733
XSE executables, 731–732

I

I-code
compilers, 866
compiling, 765
instructions, 933–938
interface, 942–949
jump targets, 938–940
source code, 940–942
I-code module, 862
compiler, 932–949
identifiers
Lua, 188
XASM, 438–439
if statements
Lua, 200–201
parsing, 1092–1099
implementing
CBS, 74–90
game intro, 93–94
lexers, 757–760
RPG characters, 101–105
scripting systems, 179–181
XASM, 455–456
importing functions (Lua), 221–226
indentifiers (lexers), 811–822
initializing
compilers, 891–892
lexers, 800–802
Lua, 207–208
multithreading, 674
Python, 265
Tcl, 313
XASM parsing, 528–529
XVM, 624–627
initiating compilers, 972
inline functions, 361
input (XASM)
comments, 442
directives, 431–439
functions, 432–434, 440–442
host API, 440–441
identifiers, 438–439
instructions, 439–440
line labels, 440
overview, 430–431
parameters, 436–438
scripts, 442–444
stacks, 431–432
variables, 434–436
instructions
assembling, 414–416
assembly languages, 337–344, 372
jump, 375–383
orthogonal, 388–389
functions, 373
I-code, 933–938
mnemonics, 17

XASM, 471–473, 487–494
parsing, 543–551
XVM Assembly input, 439–440
XVM Assembly output, 447–451
XSE executable, 553–555, 595–599
XVM, 571, 584–585, 595–599
executing, 628–633, 637–647
pointers, 634–636
structure interfaces, 604–616, 622–623
XVM Assembly, 399–404
input, 439–440
output, 447–451
integration
abstraction layer, 174–179
C
Lua, 205
Python, 263
Tcl, 312
interfaces, 174–179, 686–728
scripting systems, 174–179
interactive interpreters
lua, 186–187
Python, 243–244
Tcl, 289
interfaces
CBS, 75–78
compiler, 870
command-line options, 874–879
filenames, 871–874
logos, 870–871
compilers, 866–867
I-code, 942–949
integration, 174–179, 686–728
scripting systems, 174–179
structure interfaces (XVM), 603–604
functions, 621–623
host APIs, 621–624
instructions, 604–616, 622–623
stacks, 616–623
XASM, 470
XVM
integration interface, 686–728
public interface, 694–696
inter-language functions, 180
Intermediate code. *See* I-code
internal constant lists, 117–120
interpreted code, 24–26
interpreters
defined, 24
interactive
lua, 186–187
Python, 243–244
Tcl, 289
inter-script communication, 59
intra-language functions, 180
items (RPGs), 41–45
iterating. *See also* loops
assembly languages, 375–383
logic, 131–133
Lua, 201–203
Python, 258–261
Tcl, 308–310
XtremeScript, 358–360

J–K

Java, 27
jumps
assembling, 423–428
instructions, 375–383
targets (I-code), 938–940
JVM, 1184–1185
keyword, define, 361

L

labels (XASM), 485–487
language
game intro, 91–92
RPG characters, 95–97

languages
  assembly. *See* assembly languages
  high-level, 753
  inter-language functions, 180
  intra-language functions, 180
  low-level, 753
  procedural scripting, 336–337
  XtremeScript. *See* XtremeScript
layer, abstraction, 174–179
lexemes, 785–786
lexer module
  compiler, 916–928
  compilers, 861
lexers. *See also* lexing
  CD, 855–856
  delimiters, 822–826
  demo, 849–855
  error handling, 797
  identifiers, 811–822
  implementing, 757–760
  numeric, 797–798
    displaying reults, 809–811
    initializing, 800–802
    loops, 802–809
    state, 800
    state diagrams, 799
    strategy, 798–799
    tokens, 800
  operators, 831–849
  reserved words, 811–822
  results, 795–796
  states, 812–813
  strings, 827–831
  text files, 793–795
  tokens, 812–813
  upgrading, 814–818
  XASM, 495–524
  XtremeScript, 811–822
lexing. *See also* lexers
  characters, 785–786
  compiling, 755–757
  lexemes, 785–786
  methods, 787–792
  overview, 784
  tokenization, 787
  utilities, 788
  writing, 788
    brute force, 789
    semi-state machines, 789–790
    state machines, 791–792
  XASM, 456–462
libraries
  assembly languages, 344–346
  Lua, 185, 241
  Python debug, 264–265
lifecycles (XVM), 574
life-span (compilers), 867–870
line labels (XASM)
  parsing, 542–543
  XVM Assembly input, 440
linked lists
  compilers, 880–888
  multithreading, 667–672
  XASM, 474–477
linkers, 18
linking code, 779–780
list functions (Python), 286–287
lists
  internal constants, 117–120
  linked
    compilers, 880–888
    multithreading, 667–672
    XASM, 474–477
  Python, 251–254
  Tcl, 330
literals, assembling, 422–423
loader module
  compiler, 895–897
  compilers, 860
loaders, 11, 18
loading
  code, 779–780
  constants, 124–125
  scripts

CBS, 71–78
Lua, 208–209
Python, 266–268
Tcl, 314–315
XVM, 574–575
local variables (assembly languages), 395–397
Lockdown
CD, 1177
code, 1155–1157
graphics, 1151–1153
host API, 1158–1161
logic, 1142–1150
playing, 1175–1177
premise, 1140–1141
scripts, 1161–1173
sound, 1153–1154
speed, 1173–1175
state, 1155–1157
storyboards, 1142–1150
XtremeScript, 1158
logic
conditional
assembly languages, 377–380
CBS, 125–128
Lua, 200–201
parsing, 1092–1099
Python, 256–258
Tcl, 306–308
XtremeScript, 358–360
XVM, 640–641
XVM Assembly, 402–403
games (modular), 31
iterative, 131–133
Lockdown, 1142–1150
logos (compilers), 870–871
loops. *See also* iterating
assembly languages, 375–383
CBS, 68, 73–74
for (parsing), 1092
lexers, 802–809
Lua, 201–203
Python, 258–261
RPG characters, 105–108
Tcl, 308–310
while (parsing), 1079–1091
XtremeScript, 358–360
lower-level assembly, 1186–1189
low-level code
procedural scripting systems, 158–159
XtremeScript, 167–168
low-level languages, 753
Lua, 27, 185–187
associative arrays, 193–197
bouncing sprite demo, 228–241
C
functions, 215–219
integrating, 205
comments, 188
conditional logic, 200–201
constants, 215
data types, 191–193
coercion, 192
error codes, 209
escape sequences, 197–198
expressions, 198–200
extending, 241
functions, 203–205
importing, 221–226
host API, 229–230
host applications, 230–234
identifiers, 188
initializing, 207–208
iterating, 201–203
language, 187–205
libraries, 185, 241
loops, 201–203
OOP, 241
operators, 198–200
projects, compiling, 206–207
scripts, 234–241
executing, 219–221
loading, 208–209
semicolons, 189
stacks, 209–215

Lua (continued)
statements, 189
states, 207–208
strings, 193–198
tables, 193–197
tag methods, 241
variables, 188–191
global, 226–228
Web sites, 242
Lua data structures, 241
lua interactive interpreter, 186–187
luac compiler, 185–186

M

machine code, 17, 753
macro assemblers, 374
macros, preprocessing, 776–777
managing memory (XASM), 429–430
memory
direct allocation, 1189
XASM, managing, 429–430
XVM Assembly, 399–400
methods
lexing, 787–792
Lua, 241
mnemonics, 17
assembly languages, 383–385
mods, 24
modular architecture, 31
modular game logic, 31
modules
compilers
code-emitter. *See* code-emitter module
I-code, 932–949
lexer, 861, 916–928
loader, 860, 895–897
overview, 893–895
parser. *See* parsing
preprocessor, 861, 897–904
dictionaries (Python), 269–270
I-code, 862
moving RPG characters, 97–99
multi-pass compiling, 766–767
MultiplyString function, 222–223
multitasking, 658
cooperative, 654–658
preemptive, 654–658
multithreading (XVM), 573
arrays, 667–677
atomic operations, 661–663
concurrent execution, 659–666
context switch, 655
context switches, 679–682
cooperative, 654–658
critical sections, 663–664
error handling, 672–673
host applications, 728–739
initializing, 674
linked lists, 667–672
multitasking, 658
mutexes, 664–665
overview, 653–654
preemptive, 654–658
race conditions, 659–661
scripts, 667–677
semaphores, 665–666
threads, 677–682
tracking, 678–679
mutexes (multithreading), 664–665

N

native threads, 684
nesting (CBS), 133–135
NPCs (non-player characters), 34–41
numeric lexers, 797–798
displaying reults, 809–811
initializing, 800–802
loops, 802–809
state, 800
state diagrams, 799
strategy, 798–799
tokens, 800

O

object-oriented scripting systems, 21–22
objects. *See also* OOP
  FPSs, 51–57
  Python, 265–276
  RPGs, 41–45
OOP. *See also* objects
  assembly languages, 346–349
  compilers, 1182–1183
  Lua, 241
  Python, 286
opcodes, 17
  assembly languages, 383–385
operands
  assembling, 420–422
  assembly languages, 337–344, 372–373
  parameters, 373
  XASM (XVM Assembly output), 449–451
  XVM, executing, 636
operations, binary, 638–639
operators
  assembly languages, 340–344
  expression parser, 1053–1058
  lexers, 831–849
  Lua, 198–200
  precedence, 363–364
  Python, 254–256
  Tcl, 303–306
  XtremeScript, 354–358, 363–364
optimizing compilers, 771–772, 1183–1184
options (compilers), 874–879
orthogonal instructions, 388–389
OSs, 1185–1186
output
  XASM
    executables, 444–455
    functions, 453–455
    headers, 445–447
    host API, 454–455
    instructions, 447–451
    operands, 449–451
    strings, 451–452
  XVM host applications, 745–746

P

packages, Python, 286
Param directive, 436–438
parameters
  assembly languages, 392–395
  CBS, 144–146
    extracting, 81–87
  operands, 373
  passing (Python), 270
  XASM
    parsing, 540–541
    XVM Assembly input, 436–438
parser module. *See* parsing
parsing
  analysis, 985–987
  assignment statements, 1065–1072
  branching, 1092–1099
  CD, 1134–1135
  code blocks, 1001–1007
  compilers, 928, 1182
  compiling, 760–764, 862
  declarations, 1008
    arrays, 1017–1020
    code-emitter module, 1026–1028
    functions, 1008–1017
    host API functions, 1020–1026
    variables, 1017–1020
  expression parser
    coding, 1037–1048
    factors, 1048–1051
    function calls, 1051–1053
    operators, 1053–1058
    overview, 1033–1036
    values, 1058
  expressions, 1028–1033
  function calls, 1073–1079
  host APIs, 1062–1064

parsing (continued)
- host applications, 1058–1062
- loops
  - for, 1092
  - while, 1079–1091
- overview, 984–985
- recursive descent, 994–996
- scope, 996–997
- statements, 1001–1007
  - conditional, 1092–1099
  - if, 1092–1099
- strategy, 1000–1001
- syntax diagrams, 987–988
- tokens, 997–1000
- trees, 989–993
- XASM, 456–462, 527–528
  - directives, 529–541
  - functions, 531–534
  - initializing, 528–529
  - instructions, 543–551
  - line labels, 542–543
  - parameters, 540–541
  - stacks, 530–531
  - variables, 535–540

passes, compiling, 766–767
passing parameters (Python), 270
pattern matching (Tcl), 330
pauses (XVM), 633–646
platforms
- compiling, 768
- retargeting, 778–779

playing Lockdown, 1175–1177
pointers
- instructions (XVM), 634–636
- XtremeScript, 350

precedence
- operators, 363–364
- XtremeScript, 357–358, 363–364

preemptive multitasking, 654–658
premise, Lockdown, 1140–1141
preprocessing
- CBS, 143–150
- compilers, 773
  - files, 773–775
  - macros, 776–777
- constants, 120–124
- XtremeScript, 362–363

preprocessor module
- compiler, 897–904
- compilers, 861

Print function, 192
printing compiler statistics, 972–975
PrintStringList function, 219–221
priorities (XVM), 730–735
procedural scripting languages, 336–337
procedural scripting systems, 21–22
- architecture, 156–157
- assembling, 167–168
- code
  - high-level, 157–158
  - low-level, 158–159
- compiling, 162–166
- VMs, 159–161, 168–171
- XtremeScript, 161, 168–171
  - high-level code, 162–166
  - low-level code, 167–168

programming overview, 16–18
projects, compiling
- Lua, 206–207
- Python, 263–265
- Tcl, 312–313

public constants, 696
public functions, 694–695
public interface, 694–696
puzzles (FPSs), 51–57
Python
- bouncing sprite demo, 277–286
- C, integrating, 263
- comments, 244
- concepts, 184
- conditional logic, 256–258
- data types, 246

debug library, 264–265
directories, 243
exceptions, 286
expressions, 254–256
functions, 261–263
calling, 268–271
exporting, 271–276
list, 286
host APIs, 273–278
host applications, 278–281
initializing, 265
interactive interpreter, 243–244
iterating, 258–261
lists, 251–254
loops, 258–261
module dictionaries, 269–270
objects, 265–276
OOP, 286
operators, 254–256
overview, 242
packages, 286
parameters, 270
projects, compiling, 263–265
reference counting, 266
scripts, 281–286
loading, 266–268
strings, 247–251
variables, 244–246
Web sites, 286–287

R

race conditions (multithreading), 659–661
reading files (lexers), 793–795
recursion
assembly languages, 344–346
Tcl, 292–297
recursive descent parsing, 994–996
Reduced Instruction Set Computing (RISC), 386–388
references, counting, 266
registers, assembly languages, 389
relocatable code, 779–780
reserved words
lexers, 811–822
XtremeScript, 363–364
ResetLexer function, 498
results, lexers, 795–796
retargeting platforms (compilers), 778–779
RISC (Reduced Instruction Set Computing), 386–388
RPGs
array flags, 33–34, 38
characters
CBS, 95–108
implementing, 101–105
language, 95–97
loops, 105–108
moving, 97–99
scripts, 99–100
enemies, 45–50
items, 41–45
NPCs, 34–41
objects, 41–45
stories, 32–34
weapons, 41–45
Ruby, 26
running scripts
SVM, 683–685
Tcl, 314–315
runtime
code, 10–12
environments (XVM), 568–569

S

scope
assembly languages, 344–346, 395–397
parsing, 996–997
script function calls, 711–728
asynchronous, 719–728
synchronous, 713–719

scripting
  CBS. *See* CBS
  defined, 14–15
  languages. *See* languages
  overview, 5–6, 15–20
  purpose, 30–32
  systems, 20–27
    abstraction layer, 174–179
    CD, 334
    code, 24–26
    command-based, 22–23
    dynamically linked modules, 23–24
    embeddable, 179
    implementing, 179–181
    integration, 174–179
    interfaces, 174–179
    interpreters, 24
    Java, 27
    Lua. *See* Lua
    object-oriented, 21–22
    procedural. *See* procedural scripting systems
    Python. *See* Python
    Ruby, 26
    selecting, 331–333
    Tcl. *See* Tcl
scripts
  CBS
    executing, 71–74, 142–143
    loading, 71–78
    looping, 73–74
  control functions, 697–699
  game intro, 92–93
  inter-script communication, 59
  Lockdown, 1161–1173
  Lua, 234–241
    executing, 219–221
    loading, 208–209
  multithreading, 667–677
  Python, 281–286
    loading, 266–268
  RPG characters, 99–100
  Tcl, 325–329
    loading, 314–315
    running, 314–315
  XASM (XVM Assembly input), 442–444
  XVM
    entry point, 576
    executing, 576–577
    host applications, 739–745
    loading, 574–575
    running, 683–685
SDKs, 24
selecting scripting systems, 331–333
semantic analysis, 985
  compiling, 764
semaphores (multithreading), 665–666
semicolons (Lua), 189
semi-state machines (lexers), 789–790
SetStackSize directive, 431–432
shutdown, compilers, 892–893
single-pass compiling, 766–767
sites
  Lua, 242
  Python, 286–287
  Tcl, 330–331
SkipToNextLine function, 498
slicing time, 684
  XVM host applications, 730–731
sound (Lockdown), 1153–1154
source code
  compilers, 863–864, 919–922
  I-code, 940–942
  XASM, 470–471
speed
  CBS, 137–139
  Lockdown, 1173–1175
sprite demo, 181–184
  Lua, 228–241
  Python, 277–286
  Tcl, 322–329

stacks
  assembly languages, 389–397
  compilers, 888–890
  Lua, 209–215
  XASM
    parsing, 530–531
    XVM Assembly input, 431–432
  XVM, 571, 585–586
    structure interfaces, 616–623
  XVM Assembly, 403–405, 431–432
state diagrams (lexers), 799
state machines (lexers), 791–792
statements
  Lua, 189
    elseif, 200–201
    if, 200–201
    while, 201–203
  parsing, 1001–1007
    assignment, 1065–1072
    conditional, 1092–1099
    if, 1092–1099
states
  lexers, 800, 812–813
  Lockdown, 1155–1157
  Lua, 207–208
statistics, compiler, 972–975
steps, compiling, 753–755
stories, RPGs, 32–34
storing files
  external, 14–15
  lexers, 793–795
storyboards (Lockdown), 1142–1150
strategy
  compilers, 85–859
  lexers, 798–799
  parsing, 1000–1001
strings
  assembling, 422–423
  compiler, 915
  compilers, 866
  lexers, 827–831
  Lua, 193–198
  Python, 247–251
  Tcl, 330
  XASM, 462–469, 477–479
    XVM Assembly output, 451–452
  XSE executable, 555–556
    XVM, 599–601
  XtremeScript, 352–353
  XVM Assembly, 402, 451–452
structure. *See* architecture
structure interfaces (XVM), 603–604
  functions, 621–623
  host APIs, 621–624
  instructions, 604–616, 622–623
  stacks, 616–623
substitution (Tcl), 292–297
switches (FPSs), 51–57
symbolic constants, 116–117
symbols
  code-emitter module, 955–958
  compiler, 905–910
  compilers, 86–865
  XASM, 482–485
synchronous script function calls, 713–719
syntactic analysis, 985
syntax (BNF), 988–989
syntax diagrams
  parsing, 987–988
  XtremeScript, 1100
systems, scripting. *See* scripting, systems

T

tables (Lua), 193–197
tag methods (Lua), 241
targets
  hardware architectures, 780–781
  jump (I-code), 938–940

Tcl
- ActiveStateTcl, 288
- arrays, 301–303
- bouncing sprite demo, 322–329
- C, integrating, 312
- case-sensitivity, 291
- commands, 290–292
  - C functions, 316–320
  - calling, 315–316
- comments, 297–298
- compiling, 290
- concepts, 184
- conditional logic, 306–308
- core, 290
- directories, 288–289
- exception handling, 330
- expressions, 303–306
- extensions, 290, 330
- functions, 310–312, 316
- global variables, 320–322
- hash tables, 301–303
- host API, 323
- host applications, 323–325
- initializing, 313
- interactive interpreter, 289
- iterating, 308–310
- lists, 330
- loops, 308–310
- operators, 303–306
- overview, 287
- pattern matching, 330
- projects, compiling, 312–313
- recursion, 292–297
- scripts, 325–329
  - loading, 314–315
  - running, 314–315
- strings, 330
- substitution, 292–297
- Tk, 330
- variables, 298–301
- Web sites, 330–331

terminating XVM, 581, 646–648

theory, compilers, 752–753

threads
- multithreading, 677–682
  - arrays, 667–677
  - atomic operations, 661–663
  - concurrent execution, 659–666
  - context switch, 655
  - context switches, 679–682
  - cooperative, 654–658
  - critical sections, 663–664
  - error handling, 672–673
  - host applications, 728–739
  - initializing, 674
  - linked lists, 667–672
  - multitasking, 658
  - mutexes, 664–665
  - overview, 653–654
  - preemptive, 654–658
  - race conditions, 659–661
  - scripts, 667–677
  - semaphores, 665–666
  - threads, 677–682
  - tracking, 678–679
- native, 684

tiles (CBS), 69–70

time slicing, 684
- XVM host applications, 730–731

Tk, 330

tokenization

tokenizer (XASM), 495–524

tokenizing
- compiling, 755–757
- lexing, 787

tokens
- compilers, 916–919
- lexers, 800, 812–813
- parsing, 997–1000

tracking
- global variables, 689–694
- threads, 678–679

trees (parsing), 989–993

U

updating XVM host applications, 735–739
upgrading lexers, 814–818
utilities, lexing, 788

V

values
  assembly languages, 392–395
  expression parser, 1058
  XVM, 583–584
Var directive, 434–436
variables
  assembling, 416–420
  assembly languages, 395–397
  global, 226–228
    compilers, 890–891
    Tcl, 320–322
    tracking, 689–694
  Lua, 188–191
  parsing, 535–540, 1017–1020
  Python, 244–246
  Tcl, 298–301
  XASM
    parsing, 535–540
    XVM Assembly input, 434–436
  XtremeScript, 351–352
VMs (virtual machines), 18–20, 159
  procedural scripting systems, 159–161, 168–169, 170–171
  XVM. *See* XVM

W

weapons (RPGs), 41–45
Web sites
  Lua, 242
  Python, 286–287
  Tcl, 330–331
while loops, parsing, 1079–1091
while statements (Lua), 201–203
words, reserved (XtremeScript), 363–364
writing
  CBS, 75
  lexing, 788
    brute force, 789
    semi-state machines, 789–790
    state machines, 791–792

X

XASM
  assembling
    function calls, 423–428
    instructions, 414–416
    jumps, 423–428
    literals, 422–423
    operands, 420–422
    strings, 422–423
    variables, 416–420
  CD, 564
  compilers, 863
  error handling, 525–527
  framework, 469, 494–495
    functions, 479–482
    headers, 473
    host API, 487
    instructions, 471–473, 487–494
    interface, 470
    labels, 485–487
    linked lists, 474–477
    source code, 470–471
    strings, 477–479
    symbols, 482–485
  implementing, 455–456
  lexer, 495–524
  lexing, 456–462
  lower-level assembly, 1186–1189
  memory
    allocation, 1189
    managing, 429–430
  overview, 412–414, 428

XASM (continued)
parsing, 456–462, 527–528
directives, 529–541
functions, 531–534
initializing, 528–529
instructions, 543–551
line labels, 542–543
parameters, 540–541
stacks, 530–531
variables, 535–540
strings, 462–469
tokenizer, 495–524
XSE executable
assembling, 558–563
functions, 556–558
header, 552–553
host APIs, 557–558
instructions, 553–555
strings, 555–556
XtremeScript, 769–770
XVM host applications, 733
XVM Assembly input
comments, 442
directives, 431–439
functions, 432–434, 440–442
host API, 440
identifiers, 438–439
instructions, 439–440
line labels, 440
overview, 430–431
parameters, 436–438
scripts, 442–444
stacks, 431–432
variables, 434–436
XVM Assembly output
executables, 444–455
functions, 453–455
headers, 445–447
host API, 454–455
instructions, 447–451
operands, 449–451
strings, 451–452

XSE executable
assembling, 558–563
compilers, 969–971
functions, 556–558, 601–603
header, 552–553, 594–595
host APIs, 557–558, 602–603
host applications, 731–732
instructions, 553–555, 595–599
strings, 555–556, 599–601
XtremeScript
arrays, 353–354
code blocks, 358
comments, 362
conditional logic, 358–360
data structures, 351–354
design, 349–350
escape sequences, 362
expressions, 354–358
functions, 360–361
inline, 361
iterating, 358–360
lexers, 811–822
Lockdown, 1158
loops, 358–360
operators, 354–358
precedence, 363–364
pointers, 350
precedence, 357–358
preprocessor, 362–363
procedural scripting systems, 161, 168–171
high-level code, 162–166
low-level code, 167–168
reserved words, 363–364
strings, 352–353
syntax diagram, 1100
variables, 351–352
XASM, 769–770
XtremeScript Assembler. *See* XASM
XVM
architecture, 569–570
building, 589
CD, 649, 746

executing, 627–628
  binary operations, 638–639
  conditional logic, 640–641
  functions, 642–645
  instruction pointers, 634–636
  instructions, 628–633, 637–647
  operands, 636
  pauses, 633–634, 646
  terminating, 646–648
functions, 587–588
  calling, 578–581
global data tables, 571–572
headers, 583
host APIs, 587–588
host applications, 573–574, 682
  asynchronous script function calls, 719–728
  calling functions, 686–689
  control functions, 697–699
  embedding, 741
  host API function calls, 699–711
  host APIs, 742
  integration interface, 686–728
  multithreading, 728–739
  native threads, 684
  output, 745–746
  priorities, 730–735
  public interface, 694–696
  running scripts, 683–685
  script function calls, 711–728
  scripts, 739–745
  synchronous script function calls, 713–719
  time slicing, 684, 730–731
  tracking global variables, 689–694
  updating, 735–739
  XASM, 733
  XSE executables, 731–732
initializing, 624–627
instructions, 571, 584–585
lifecycle, 574
multithreading, 573
  arrays, 667–677
  atomic operations, 661–663
  concurrent execution, 659–666
  context switch, 655
  context switches, 679–682
  cooperative, 654–658
  critical sections, 663–664
  error handling, 672–673
  initializing, 674
  linked lists, 667–672
  multitasking, 658
  mutexes, 664–665
  overview, 653–654
  preemptive, 654–658
  race conditions, 659–661
  scripts, 667–677
  semaphores, 665–666
  threads, 677–682
  tracking, 678–679
overview, 568–569
runtime environments, 568–569
scripts
  entry point, 576
  executing, 576–577
  loading, 574–575
stacks, 571, 585–586
structure, 582–589
structure interfaces, 603–604
  functions, 621–623
  host APIs, 621–624
  instructions, 604–616, 622–623
  stacks, 616–623
terminating, 581
values, 583–584
XSE executable
  functions, 601–603
  header, 594–595
  host APIs, 602–603
  instructions, 595–599
  overview, 590–593
  strings, 599–601

XVM Assembly
  arithmetic, 400–401
  bitwise, 401
  code-emitter module, 966–969
  comments, 407
  conditional logic, 402–403
  defined, 397–399
  directives, 404–407
  escape sequences, 407
  functions, 403–406
  instructions, 399–404
  memory, 399–400
  overview, 408
  stacks, 403–405
  strings, 402
  XASM input
    comments, 442
    directives, 431–439
    functions, 432–434, 440–442
    host API, 440–441
    identifiers, 438–439
    instructions, 439–440
    line labels, 440
    overview, 430–431
    parameters, 436–438
    scripts, 442–444
    stacks, 431–432
    variables, 434–436
  XASM output
    executables, 444–455
    functions, 453–455
    headers, 445–447
    host API, 454–455
    instructions, 447–451
    operands, 449–451
    strings, 451–452

Gamedev.net

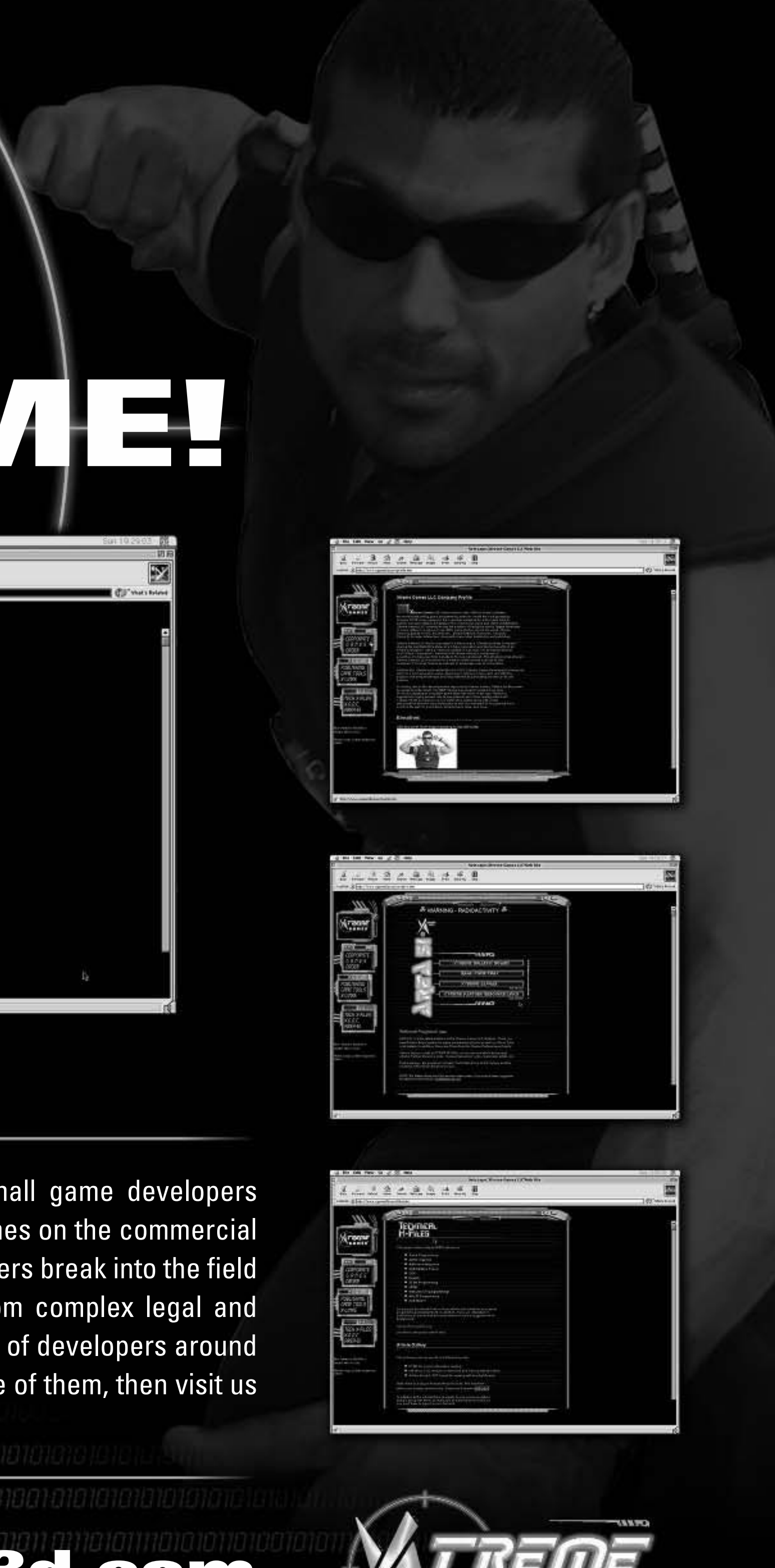
The most comprehensive game development resource

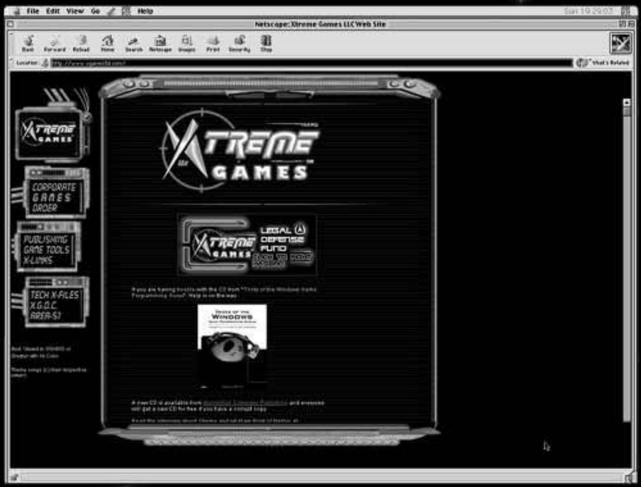
The latest news in game development
The most active forums and chatrooms anywhere, with insights and tips from experienced game developers
Links to thousands of additional game development resources
Thorough book and product reviews
Over 1000 game development articles!
Game design
Graphics
DirectX
OpenGL
AI
Art
Music
Physics
Source Code
Sound
Assembly
And More!

Gamedev.net

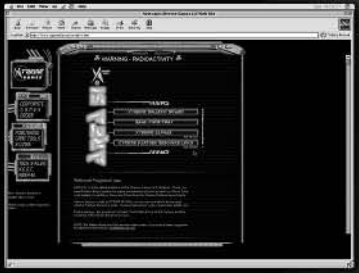
TAKE YOUR GAME TO THE

XTREME!





Netscape: Xtreme Games LLC Web Site
XTREME GAMES
CORPORATE GAMES ORDER
PUBLISHING GAME TOOLS X-LINKS
TECH X-FILES X.G.D.C AREA-51
LEGAL DEFENSE FUND



Xtreme Games LLC was founded to help small game developers around the world create and publish their games on the commercial market. Xtreme Games helps younger developers break into the field of game programming by insulating them from complex legal and business issues. Xtreme Games has hundreds of developers around the world, if you're interested in becoming one of them, then visit us at **www.xgames3d.com.**

www.xgames3d.com

XTREME LLC GAMES