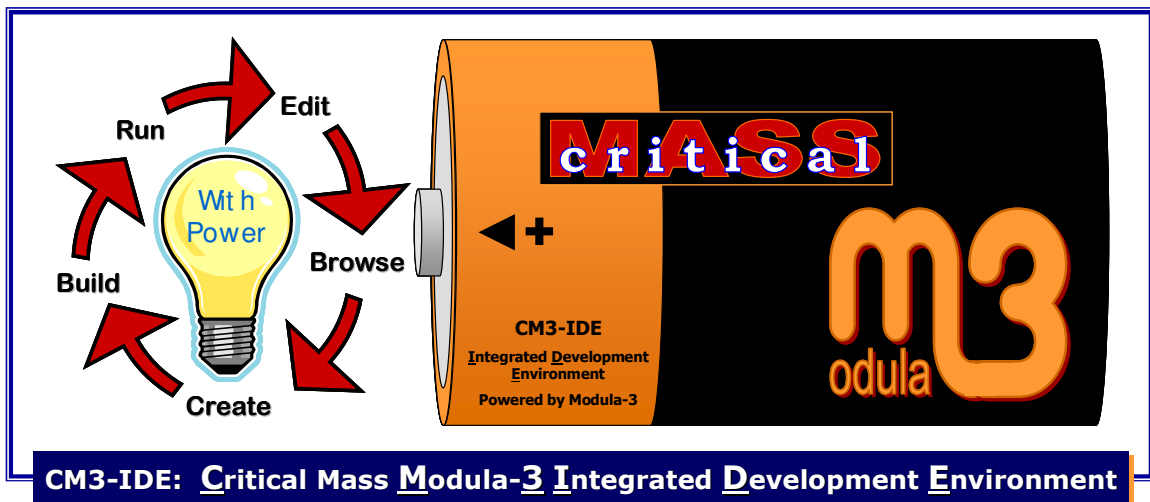




CRITICAL MASS MODULA-3 (CM3)

Integrated Development Environment (IDE)



CM3-IDE User Guide

This page left blank intentionally.

CRITICAL MASS MODULA-3 INTEGRATED DEVELOPMENT ENVIRONMENT

CM3-IDE

User Guide

elego Software Solutions GmbH
Gustav-Meyer-Allee 25 / Building 12
13355 Berlin

June 2008

CM3-IDE was originally developed as Reactor by Bill Kalsow and Farshad Nayeri at Critical Mass, Inc., now named IGEN Corporation. The software was later open-sourced through the tireless efforts of Randy Coleburn and Olaf Wagner.

elego Software Solutions ‘inherited’ the complete sources for the Critical Mass Modula-3 compiler and development system from Critical Mass, Inc., in 2000, and has since made several releases of the system in source and binary form. In March 2002 elego Software Solutions also took over the repository of the other active Modula-3 distribution PM3, till then maintained at the Ecole Polytechnique at Montreal.

So currently elego Software Solutions is hosting the complete CVS source code repositories and providing several possibilities to download Modula-3 sources or installation archives. You must decide if you want sources, CVS repositories (RCS files), or installation archives, and you can get them in several ways: using (anonymous) CVS, CVSup, HTTP, or FTP.

Everybody who wants to work on the CM3 or PM3 sources directly can get write access if he/she provides an ssh key (protocol version 2 (DSA) preferred). Send email to m3-support@elego.de if you are interested, and have a look at the CM3 configuration management rules available on the web site: <http://modula3.elegosoft.com/cm3/>

This manual is derived from the original Reactor User Guide published by Critical Mass, Inc.
Copyright © 1996 Critical Mass, Inc. All Rights Reserved.

This manual, as well as the software described in it, is furnished under license and may not be used or copied in accordance with the terms of such license.

Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Critical Mass, Inc.

The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Critical Mass, Inc., or by elego Software Solutions. Critical Mass, Inc. and/or elego Software Solutions assumes no responsibility or liability for any errors or inaccuracies that may appear in this manual.

Critical Mass, the Critical Mass logo, Reactor, the Reactor logo, CM3, and Critical Mass Modula-3 are trademarks of Critical Mass, Inc.

Alpha AXP and Digital Unix are trademarks of Digital Equipment Corporation. OSF/I is a registered trademark of Open Software Foundation, Inc. Unix is a registered trademark of UNIX System Laboratories, Inc. SPARC, SunOS, and Solaris are trademarks of Sun Microsystems. Microsoft is a registered trademark of Microsoft Corporation. Windows and Win32 are trademarks of Microsoft Corporation. IBM and AIX are registered trademarks of International Business Machines Corporation. HP, HP Precision Architecture, and HP/UX are trademarks of Hewlett Packard. PostScript, and Acrobat are registered trademarks of Adobe Systems Incorporated.

Table of Contents

0.	Introduction	1
0.1	About This Manual	1
0.1.1	Welcome to CM3-IDE!	1
0.1.2	What's Inside	1
0.1.3	Typographic Conventions	2
0.1.4	Before You Begin	2
0.1.5	Keeping in Touch	2
0.2	About CM3-IDE	3
0.2.1	CM3-IDE's Development Environment	3
0.2.2	Systems Development with CM3-IDE	4
0.2.3	Programming in CM3-IDE	4
1.	Learning the Basics	5
1.1	Starting CM3-IDE	6
1.2	A Quick Walkthrough	7
1.3	Creating a Package From Scratch	15
1.3.1	List of All Packages	15
1.3.2	Creating New Packages	17
1.3.3	Procedure Calls in CM3-IDE: What is "IO.Put?"	20
1.3.4	The IO Interface	20
1.3.5	CM3-IDE Makefiles (m3makefile)	24
1.4	Summary	27
2.	The CM3-IDE Environment	29
2.1	Common Tools, Icons, and Visual Elements	29
2.1.1	Quick Access Icons	30
2.1.2	Action Buttons	31
2.1.3	The Find Type-in	32
2.2	CM3-IDE Start Screen	32
2.2.1	Start Screen: System	34

2.2.2	Start Screen: Language	34
2.2.3	Start Screen: Help	34
2.2.4	Customizing the CM3-IDE Start Screen	35
2.3	Summary Screens	36
2.3.1	Package Summary	36
2.3.2	Library Summary	37
2.3.3	Program Summary	39
2.3.4	Interface Summary	40
2.3.5	Module Summary	41
2.4	CM3-IDE's Web Namespace	42
2.4.1	CM3-IDE URLs	43
2.4.2	Example CM3-IDE URLs	43
2.4.3	Regular Expressions in CM3-IDE URLs	44
2.5	Summary	45
3.	Building and Sharing Packages	47
3.1	Building Packages	48
3.2	Directory Structure of a Package	49
3.3	CM3-IDE makefiles	50
3.3.1	Basic Makefile Commands	51
3.3.2	Additional Makefile Commands	53
3.4	Managing Multiple Packages	53
3.5	Shipping Packages	54
3.6	Package Roots	55
3.6.1	Example: Creating a New Package Root	56
3.7	Sharing Packages	57
3.7.1	Example: Adam's and Eve's Joint Project	58
3.8	Builder Options	62
3.9	Summary	64
4.	Customizing CM3-IDE	65
4.1	CM3-IDE Configuration Screen	66
4.1.1	Navigating to the Configuration Page	66
4.1.2	Saving Configuration Changes	66
4.1.3	Display Settings	66
4.1.4	Package Roots Settings	68
4.1.5	Communication Settings	69
4.1.6	Miscellaneous Settings	69

4.1.7	Helper Procedures	70
4.2	Summary	72
5.	Beyond The Basics	73
5.1	Exceptions: Error Handling in CM3-IDE	74
5.1.1	How Exceptions Work	74
5.1.2	Declaring Exceptions	74
5.1.3	Triggering Exceptions: RAISE Statement	75
5.1.4	Handling Exceptions: TRY-EXCEPT Statement	75
5.1.5	Cleaning up: TRY-FINALLY Statement	76
5.1.6	Trapping All Exits from a Block of Code	77
5.1.7	An Example of Exception Handling	77
5.1.8	Programming without Exceptions	77
5.1.9	Making Programs Robust with Exceptions	79
5.2	Object Types: Object-oriented Programming	81
5.2.1	Programming with Objects: A Complete Example	83
5.3	Threads: Managing Concurrent Activities	86
5.4	Opaque Types: Information Hiding And Encapsulation	88
5.4.1	Fully Opaque Types	89
5.4.2	Clients of an Opaque Type	91
5.4.3	Partially Opaque Types: Revealing Types in Moderation	92
5.4.4	Subtyping Partially Opaque Type	94
5.4.5	Clients of a Partially Opaque Type	96
5.5	Generics: Resuable Data Structures and Algorithms	97
5.5.1	Using Generics	98
5.5.2	A Generic Example: List	98
5.5.3	Parameter to a Generic: Atom	99
5.5.4	Instantiating a Generic: AtomList	99
5.5.5	Using Instances of Generics	99
5.5.6	Instantiating Generics for User-Defined Types	101
5.5.7	Instantiating Generics in a Makefile	102
5.6	Unsafe Constructs: System Programming in CM3-IDE	102
5.6.1	Unsafe Coding Example	103
5.7	Summary	105
6.	Development Recipes	107
6.1	Robust Distributed Applications: Network Objects	108
6.1.1	The Common Interface	108

6.1.2	A Network Object Server	109
6.1.3	A Network Object Client	113
6.2	Client/Server Computing: Safe TCP/IP Interfaces	115
6.2.1	A TCP/IP Client: Finger	115
6.2.2	A TCP/IP Server: HTTPD	117
6.3	Taking Persistent Snapshots of Objects: Pickles	120
6.4	Quick Comparison of Large Data: Fingerprints	122
6.5	Portable Operating System Interfaces	124
6.6	Dynamic Web Applications: the Web Server Toolkit	135
6.7	Interacting with C Programs	137
6.7.1	Calling C: A Unix Example	137
6.7.2	Calling C: A Win32 Example	139
6.7.3	Calling Modula-3 from C	140
6.8	Summary	142
7.	CM3-IDE Interface Index	143
7.1	Data Types, Data Structures, and Algorithms	144
7.1.1	Basic Data Types	144
7.1.2	Collections, Lists, Tables, Sets	145
7.1.3	Linked Lists	145
7.1.4	Sorted Linked Lists	145
7.1.5	Property lists	145
Tables		146
7.1.6	Sorted tables	146
7.1.7	Sequences	146
7.1.8	Priority queues	147
7.1.9	Sets	147
7.1.10	Sorting Lists, Tables, and Arrays	147
7.2	Standard Libraries	148
7.2.1	Math, Geometry, Statistics, Random numbers	148
7.2.2	Floating point	148
7.2.3	Environment, Command line parameters	148
7.2.4	I/O streams, Reading and Writing, Files	148
7.2.5	Formatting, I/O Conversion	149
7.2.6	Threads	149
7.3	Systems Development	150
7.3.1	Distributed and Client/Server Development	150
7.3.2	Databases and Persistence	150

7.3.3	Operating System, Files, Processes, Time	151
7.3.4	Interoperability with C	153
7.3.5	Low-level Run-time Interfaces	153
7.4	Miscellaneous	153
8.	Further Information	155
8.1	Books	156
8.1.1	System Programming with Modula-3	156
8.1.2	Modula-3	156
8.1.3	Algorithms in Modula-3	156
8.1.4	Programming with Modula-3: An Introduction to Programming with Style	157
8.2	Technical Documentation	157
8.2.1	Reactor White Paper	157
8.2.2	Some Useful Modula-3 Interfaces	157
8.2.3	Network Objects	158
8.2.4	Trestle Reference Manual	158
8.2.5	VBToolkit Reference Manual: A toolkit for Trestle	158
8.2.6	Obliq-3D Tutorial and Reference Manual	158
8.3	Introductory Programming Articles	158
8.3.1	Modula-3 Reference and Tutorial	158
8.3.2	Net Balance: A Network Objects Example	159
8.3.3	Building Distributed OO Applications: Modula-3 Objects at Work	159
8.3.4	Partial Revelation and Modula-3	159
8.3.5	Initialization of Object Types	159
8.3.6	Trestle Tutorial	159
8.3.7	Trestle by Example	160
8.4	Systems Built Using Modula-3	160
8.4.1	The Juno-2 Constraint-Based Drawing Editor	160
8.4.2	Zeus: A System for Algorithm Animation and Multi-View Editing	160
8.4.3	Writing an Operating System with Modula-3	160
8.4.4	The Whole Program Optimizer	160
8.5	Parallel Programming	161
8.5.1	An Introduction to Programming with Threads	161
8.5.2	Synchronization Primitives for a Multiprocessor: A Formal Specification	161
8.6	Garbage Collection	161
8.6.1	Compacting Garbage Collection with Ambiguous Roots	161
8.6.2	Distributed Garbage Collection for Network Objects	162

8.6.3	Portable, Mostly-Concurrent, Mostly-Copying Garbage Collection for Multi-Processors	162
8.7	Comparisons to Other Languages	162
8.7.1	A Comparison of Modula-3 and Oberon-2	162
8.7.2	A Comparison of Object-Oriented Programming in Four Modern Languages	162
8.8	Summary	163

Figures

Figure 1. The Start Screen.....	6
Figure 2. Examples Area.....	7
Figure 3. A Package Summary	8
Figure 4. A Module Summary.....	10
Figure 5. A Schematic of the IO Interface.....	12
Figure 6. Building a package.....	13
Figure 7. A Package Summary containing a Built Program.....	13
Figure 8. A Program Summary with a Run Button	14
Figure 9. Running "Hello World".....	15
Figure 10. Packages Page: Listing of All Packages in CM3-IDE.....	16
Figure 11. Top of the Packages Page.....	17
Figure 12. Package Creation Dialog.....	18
Figure 13. The top of the Module Summary Page.....	19
Figure 14. A Procedure Call Crossing Module Boundaries.....	20
Figure 15. The top of the IO Interface.....	21
Figure 16. Some Procedures Defined in the IO Interface.....	22
Figure 17. The top of a Program Summary page.....	26
Figure 18. Quick Access Icons for <code>/proj/hello/src/hello.m3</code>	30
Figure 19. CM3-IDE's Start Screen	33
Figure 20. A Package Summary	36
Figure 21. A Library Summary.....	38
Figure 22. Top Portion of a Program Summary	39
Figure 23. An Interface Summary	40
Figure 24. Portions of a Module Summary.....	41
Figure 25. CM3-IDE's Build Button	48
Figure 26. CM3-IDE Package Directory Structure	49
Figure 27. Package Roots Section of the Configuration Screen	56
Figure 28. The "Package Roots" section of Eve's Configuration Page.....	59
Figure 29. Save Changes and Display Settings	66
Figure 30. Package Root and Communication Configuration	68
Figure 31. Miscellaneous Settings and Helper Procedures	70

This page left blank
intentionally.

0. Introduction

0.1 About This Manual

0.1.1 Welcome to CM3-IDE!

If you are reading this manual because you would like to learn how to use CM3-IDE, you've come to the right place.

This manual will teach you what you need to know to use CM3-IDE for your development tasks. To make the most of this manual and of CM3-IDE, you should try out the hands-on tutorials using CM3-IDE's development environment.

0.1.2 What's Inside

Chapter 1, **Learning the Basics** on page 5 introduces you to the basic concepts of CM3-IDE: packages, modules, interfaces, importing, and exporting, by walking through two hands-on tutorials.

Chapter 2, **The CM3-IDE Environment** on page 29 tours the commonly used screens in the CM3-IDE development environment.

Chapter 3, **Building and Sharing Packages** on page 47 explains how to build packages, and how to share them with others developers in your team. By the end of Chapter 3, you should have a solid understanding of the CM3-IDE development environment.

Chapter 4, **Customizing CM3-IDE** on page 65 describes CM3-IDE's Configuration page. Use this information to tailor CM3-IDE's behavior to fit your individual or team development needs.

Chapter 5, **Beyond the Basics** on page 73 introduces the more advanced language features, such as object types, threads, exception handling, generics, and provisions for unsafe code.

Chapter 6, **Development Recipes** on page 107 includes recipes for building some simple, but real applications: client/server computing, distributed computing, building



INTRODUCTION

dynamic web applications, integrating legacy code, and using operating system interfaces.

Chapter 7, **CM3-IDE Interface Index** on page 143 outlines the most common of the hundreds of interfaces in CM3-IDE.

Chapter 8, **Further Information** on page **Error! Bookmark not defined.** cites other sources of information about CM3-IDE.

0.1.3 Typographic Conventions

In this manual, the body text is typed in a serif typeface. References to code are typeset a sans serif code typeface, for example:

```
FOR i := 1 TO 10 DO IO.PutInt(i) END;
```

Step

Performing some tasks may involve multiple steps. To help continuity of the text in this manual, each step is marked on the left hand-side with a “STEP” icon.

0.1.4 Before You Begin

To achieve the most from reading this user guide, you should:

- Know the basics of programming. Reading this manual does not require extensive programming experience, however, you are expected to know the basics.
- Learn how to operate your web browser and text editor. CM3-IDE allows you to use a browser and editor of your choice; hence you must know how to use them before you can use CM3-IDE effectively.

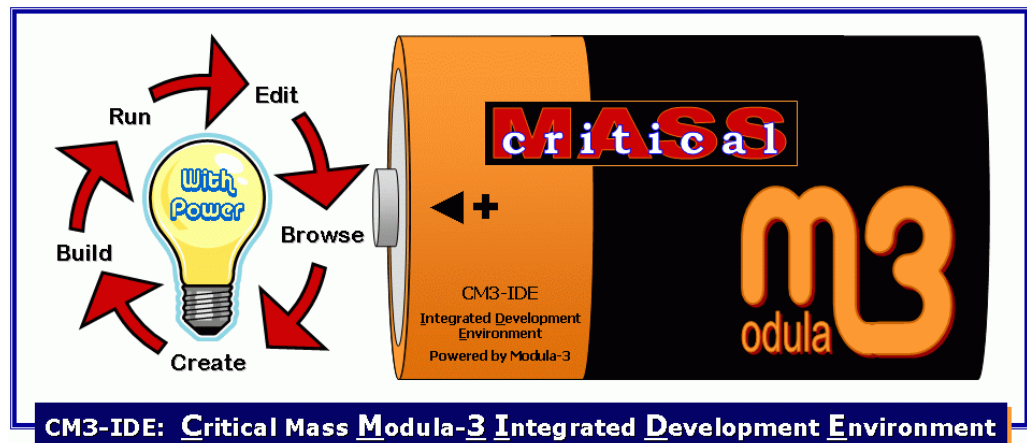
If CM3-IDE’s default browser and editor don’t match your preferences, change the CM3-IDE settings as described in Chapter 4, **Customizing CM3-IDE** on page 65.

- Install CM3-IDE on your system. For instructions, see the *Installation Guide* distributed with your copy of CM3-IDE.

0.1.5 Keeping in Touch

Our web address is <http://modula3.elegosoft.com/cm3/>. Visit us there for updates and information about CM3-IDE. Good luck. We hope you enjoy using CM3-IDE!

0.2 About CM3-IDE



CM3-IDE is a complete environment for the development of robust, multi-platform, client/server and distributed applications.

CM3-IDE provides built-in support for modern systems programming tasks through the use of features such as multi-threading, automatic garbage collection, exceptions, and separation of interfaces from implementations.

CM3-IDE's strength lies in its support for the development of high-performance back-end servers and middle layers in a multi-tier architecture. You can also use CM3-IDE to build user interfaces.

0.2.1 CM3-IDE's Development Environment

CM3-IDE development environment is a dynamic, custom web server. To navigate within the environment, you use a standard web browser such as Netscape Navigator or the Internet Explorer.

Each file in your project and every command in CM3-IDE maps to a location in the CM3-IDE's web namespace. CM3-IDE continually tracks changes in your system and uses the information to build and maintain hypertext links to program components.

This unique design has many advantages, among them:

- The web is an intuitive metaphor to most users, hence it is easy to begin using CM3-IDE.
- CM3-IDE's feel and function remain the same on all platforms.
- Within your projects, you can embed references to external documentation, point out relationships with other projects in your system, or create links to a

README file. You can embed a reference to a CM3-IDE project in an e-mail message.

- CM3-IDE's web-based metaphor allows you to quickly access information about program components, types, and their relationships.

0.2.2 Systems Development with CM3-IDE

CM3-IDE's high-performance compiler generates native code from the same source code whether you run on Windows or Unix. The smart but simple-to-use builder keeps track of dependencies between various program elements automatically, whether or not you use makefiles. Customizing CM3-IDE is straightforward, also.

The CM3-IDE installation includes a collection of portable, well-documented, and thread-friendly libraries, giving you access to thousands of calls in several hundred interfaces.

CM3-IDE's simple repository system allows you and your co-workers to share released code.

0.2.3 Programming in CM3-IDE

CM3-IDE's primary programming language is Modula-3.

Modula-3's Pascal-like syntax and concise definition make it easy to learn and use; yet Modula-3 is an extremely powerful and versatile development tool. Modula-3 has been used extensively for building robust, distributed programs for over a decade.

A language reference, tutorial, and many examples are available online as part of the CM3-IDE environment.

1. Learning the Basics

Read this chapter if you have never used CM3-IDE before.

This chapter introduces you to CM3-IDE’s web-based environment, and to five basic concepts that are central to CM3-IDE: *packages*, *modules*, *interfaces*, *importing*, and *exporting*. By the end of this chapter, you’ll be familiar with these concepts and with some of the screens that you’ll use often.

If you haven’t installed CM3-IDE yet, follow the instructions in the *CM3-IDE Installation Guide* to get started. The rest of this chapter also assumes that you know how to use your web browser and your text editor well.

The chapter is divided into three parts:

Starting CM3-IDE on page 6 outlines how you start the CM3-IDE development environment.

A Quick Walkthrough on page 7 uses the old standby, the “hello world” program to tour the browse and build features of CM3-IDE. In this first tutorial, you’ll learn how to:

- create a new package from an existing example
- build a simple “hello world” program
- run the program from within CM3-IDE
- explore how CM3-IDE automatically updates its virtual namespace to keep up with changes to your system

The second tutorial, **Creating a Package from Scratch** on page 15 covers some of the same concepts as the first, but in greater depth. This time, you:

- create your own package
- open and run your text editor from within CM3-IDE
- edit and compile sources and makefiles
- browse one of CM3-IDE’s library packages.



1.1 Starting CM3-IDE

You can start CM3-IDE by typing CM3-IDE at the command-prompt, assuming the CM3-IDE program is in your executable path. If you haven't installed CM3-IDE yet, or you are unable to locate the executable program for CM3-IDE, see the *CM3-IDE Installation Guide*.

Once started, CM3-IDE automatically spawns your web browser and points it to CM3-IDE's start screen, as in Figure 1.

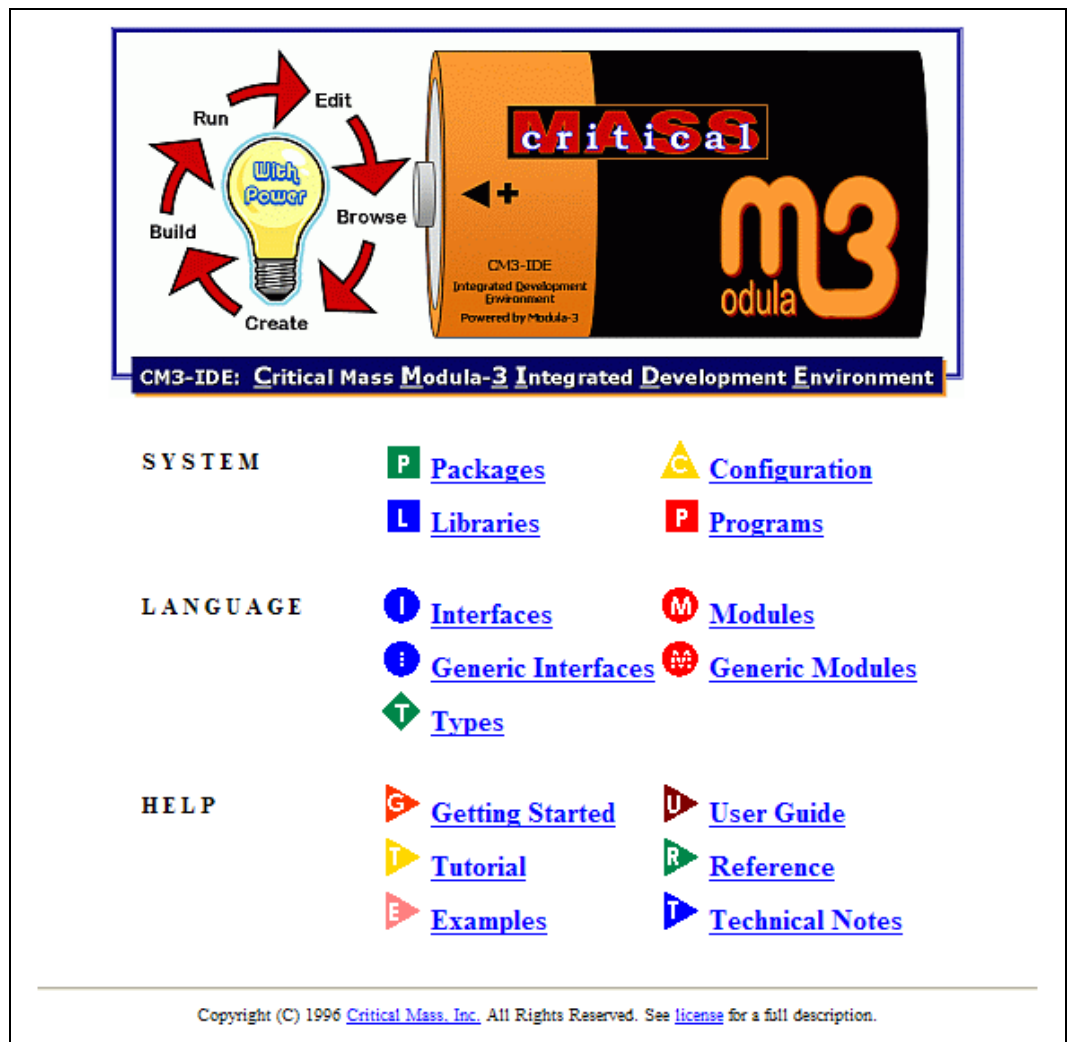



Figure 1. The Start Screen

At the top of the start page is the CM3-IDE logo, and below that a set of icons that represent elements of the CM3-IDE environment. They are divided into three groups: System, Language, and Help.

1.2 A Quick Walkthrough

Having started CM3-IDE, you will see CM3-IDE's start page in your browser's window (Figure 1). Here we quickly walk through the building of a Hello World example program.

Step

The Start Screen. From the start screen, follow the link to  Examples. (It's in the Help category, toward the bottom of the screen.)

Step

Click on the item named “Hello World” (Figure 2). CM3-IDE will create a new example program named “**hello**”, and will take you to the package summary for the hello package.



Figure 2. Examples Area

Packages

Using CM3-IDE, you divide your programming projects into *packages*. A package is a unit of ownership in the CM3-IDE system. A project consists of one or more packages. For large projects, different people may “own” different packages.

A CM3-IDE package comprises:

- zero or more modules
- zero or more interfaces
- a makefile (called “**m3makefile**”)

The makefile tells the compiler how to put everything together. For simple programs, you may get away without having a makefile.

Each package has its own directory on your system, where all its source files are stored together.

Package Summary. A package summary page outlines different elements that comprise a package (Figure 3). You can follow the links on this page to view any of its components.

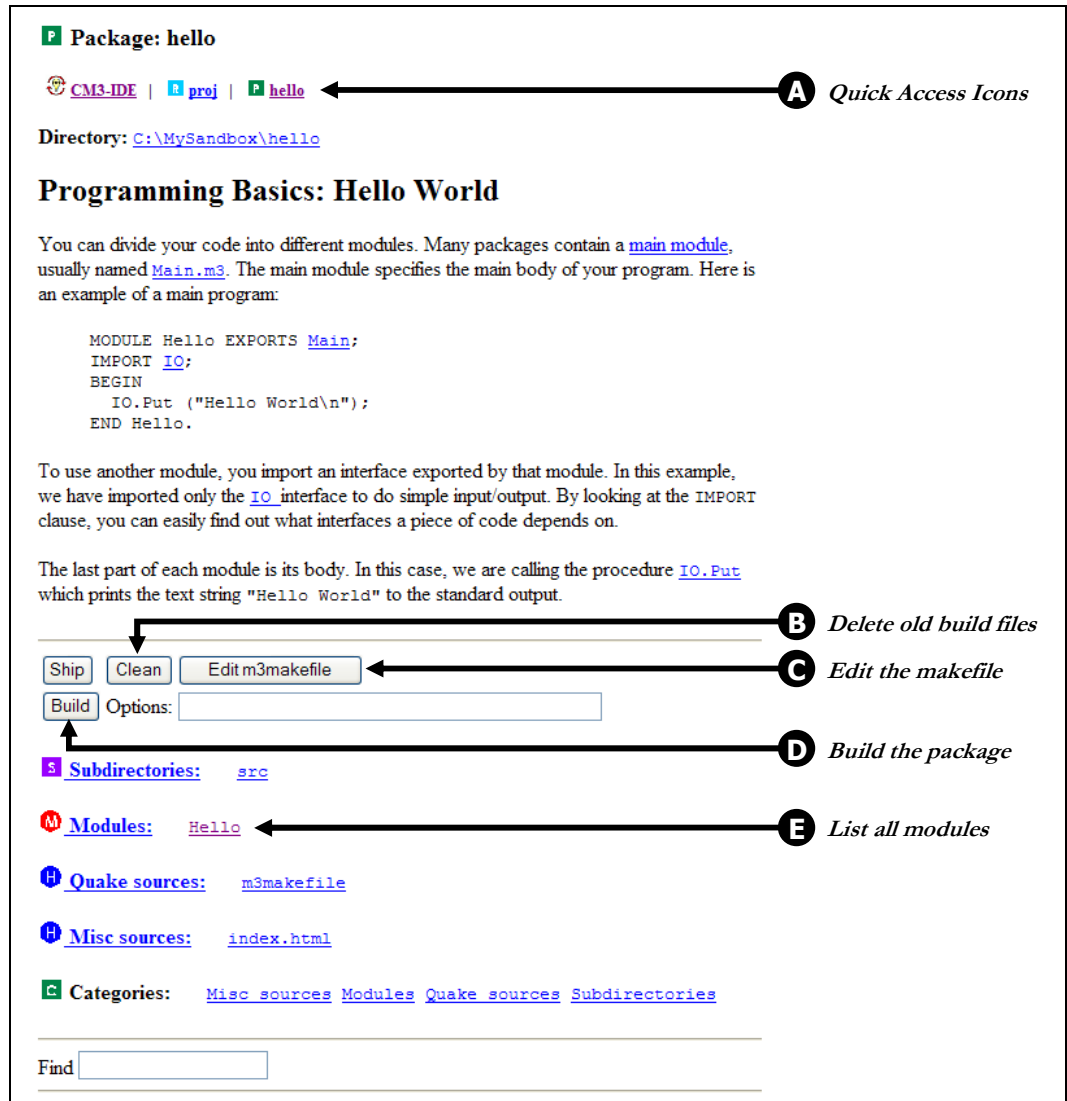

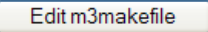




Figure 3. A Package Summary

At the top of the page, you'll see a row of Quick Access Icons. Below that, a button labeled Build, and below that, package components, such as Subdirectories and Modules.

- A** Use the **Quick Access Icons** to navigate to other locations in the CM3-IDE Environment. For more information, see **The CM3-IDE Environment** on page 29.


LEARNING THE BASICS


- B**  The **Clean** button tells CM3-IDE to delete files from previous builds. Clean does not remove sources of your program.
- C**  The **Edit m3makefile** button starts your text editor and opens the makefile for this package.
- D**  The **Build** button activates CM3-IDE's builder and uses the instructions in your makefile to build the package. If there is no makefile, CM3-IDE's builder will scan your package's directory tree and attempts to build a program based on that information.
- E** The **modules** available in this package are listed under the  Modules heading. This page has only one entry—it's called Hello.






Modules

A *module* is a named collection of declarations, including constants, types, variables, procedures, and their associated bodies.

Step

Module Summary. Next, follow the link Hello under the heading  Modules to go to the summary page for the Hello module. Your browser will display a page titled “Module: Hello” (Figure 4).

 **Module: Hello**

 [CM3-IDE](#) |
  [proj](#) |
  [hello](#) |
  [src](#) |
  [Hello](#)

Path: [C:\MySandbox\hello\src\Hello.m3](#) **Last modified:** Jan 26 04:06

Ship Clean Edit m3makefile Edit source

Build Options:

MODULE Hello EXPORTS [Main](#);

Each module must have a name, which is declared in the MODULE statement. By convention, the main module for an executable program exports the interface Main, as does the Hello module here.

Each module can also import interfaces exported by other modules. This is how you reuse code from libraries or your own modules. Here, we have imported interface IO which is a simple input/output interface.

From the browser, you can learn what the imported interfaces do by following the link associated with their name.

IMPORT [IO](#);

The main body of a module or the initialization section includes statements that are executed at the beginning of the program. In this case, we are the main module, and all we do is print Hello World! on standard output.

```

BEGIN
    IO.Put ("Hello World!\n");
END Hello.
  
```

Don't forget to include the module name in the last END in your program.

Figure 4. A Module Summary

Viewing Code of a Module. On this page, you can view the code for `Hello.m3`, the file containing the Hello module.

CM3-IDE is case-sensitive.

```
MODULE Hello EXPORTS Main;
IMPORT IO;
BEGIN
  IO.Put ("Hello world\n");
END Hello.
```

Note that CM3-IDE is case-sensitive. Keywords are always in upper-case.

Module Statement. The first line reads:

```
MODULE Hello EXPORTS Main;
```

This is the *module statement*. Each module must have a name; in this case the name is **Hello**. By including “**EXPORTS Main**” in the module statement, this module is considered to be the main module, i.e., the module containing the main body of the program.

Main Module of a Program

Every program must have a single main module, specifying its main body. The main module for your program exports the **Main** interface. This can be done either by naming your main module **Main**, or by including **EXPORTS Main** in the module statement for the main module.

Import Statement. The next line reads:

```
IMPORT IO;
```

This is an *import statement*. To use items defined in another module, you *import* an interface *exported* by that module. You do that by listing it here, in the import statements for your module. In this example, we have imported only the IO interface for doing simple output (Figure 5). By looking at the import statements for a module, you can easily find out what interfaces it depends on.

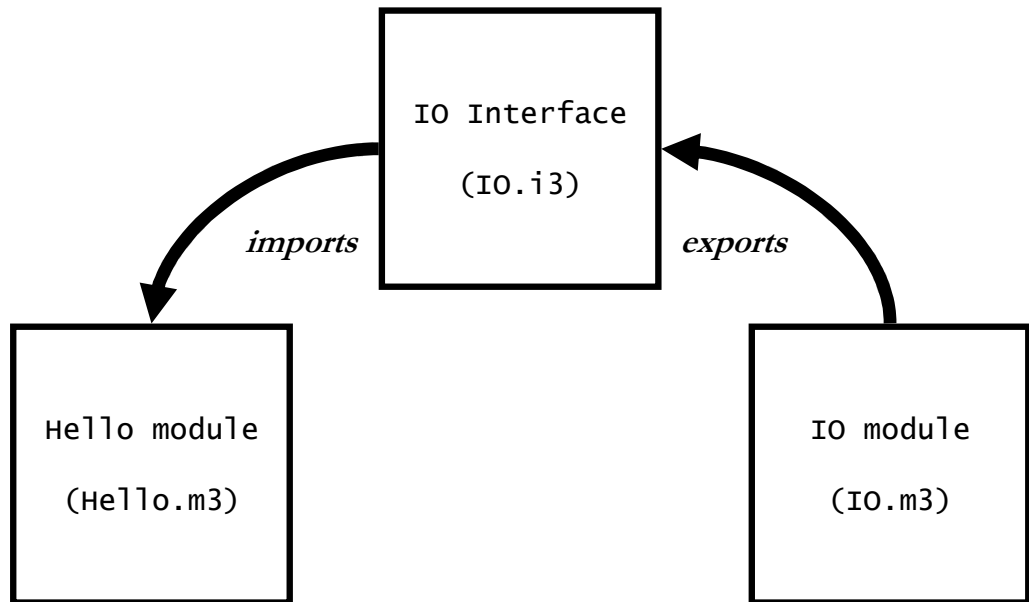


Figure 5. A Schematic of the IO Interface

The module body simply calls the procedure in the **IO** interface, denoted by **IO.Put**. **IO.Put** prints out the text string “Hello world”, followed by a new line (`\n`) to the standard output.

```

BEGIN
    IO.Put (“Hello world\n”);
END Hello.
  
```

Interfaces

An *interface* defines what parts of a module are visible to its clients. An interface can include declarations for types, procedures, constants, and variables.

Usually, the name for an interface matches that of the module that exports it; for example, the **IO** module exports the **IO** interface. (This does not have to be the case at all times.)

A useful way to think about an interface is as a window into the module that exports it.

Building a Package. This next step will produce an executable program by building the sources for the hello package.

Step

Click the Build button from the module summary. This will start the builder, taking you to a Build Results page (Figure 6). From this page, you can view the output from your build. Errors will appear here as hypertext links to the line of code that generated them. With this example, you should not see any errors. If you do, retrace your steps up to this point.



Figure 6. Building a package.

Step

From Building to Running. Once built, follow the [hello](#) link in the Quick Access Icons on top of the Build Results page. This returns you to the package summary for the hello package. (You can also use the “Back” button on your browser.)

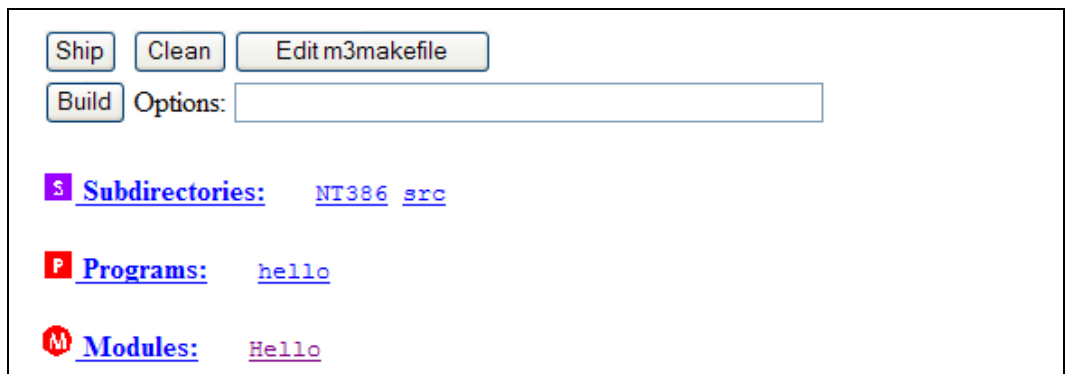


Figure 7. A Package Summary containing a Built Program

Step

Program Summary. If you look at the bottom of the page, you'll notice a change: You should see a new category labeled **P** Programs. Next to the icon you'll see the word "hello." This is the program you just built. Click on the word "hello" to navigate to the Hello program summary.

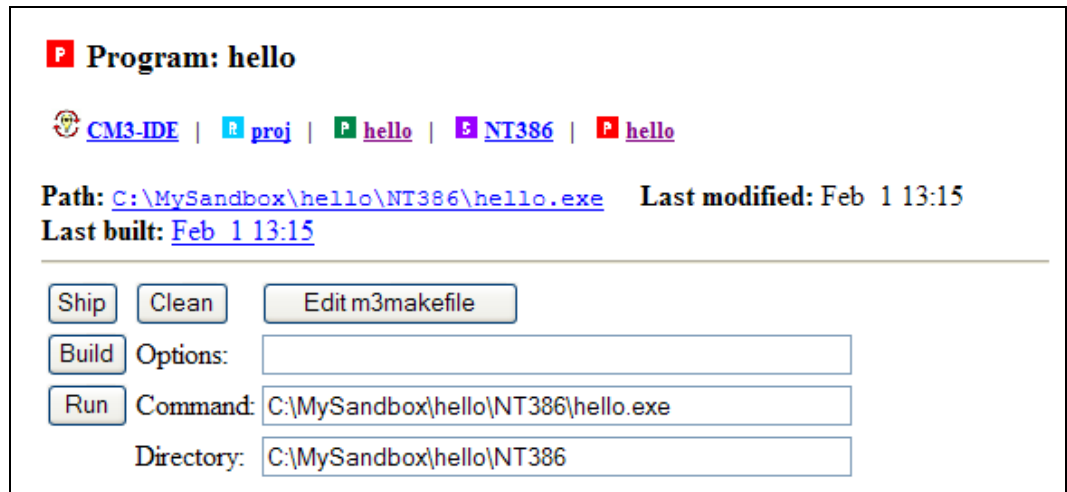


Figure 8. A Program Summary with a Run Button

Missing Program Icons in a Package Summary. If you don't see a program icon in your package summary, you may have to reload the page or click on the Rescan button (when available) to update CM3-IDE's browser view with the package contents on your file system.

If you still can't see a program icon, you probably did not build the package properly. (Perhaps you encountered a compilation or installation error.) Retrace your steps up to this point, making sure you followed the instructions correctly, and compare their results with the user guide.

Running a Program. Similar to other pages for a package, from the program summary page, you can navigate to the package top, or to any package components, or rebuild your program.

More importantly, you can run your program from this screen. (See Figure 8.) The Run button is directly underneath the Build button. Next to the Run button is a type-in field where you can enter the text as you would on a command line. (CM3-IDE should have already done this for you.) Beneath that is a text box containing the path to the directory in which your package resides.

Step

Click the Run button now. Your program will run, and your browser will display the result of the execution of the program. In this case, you should see the text "Hello world" appear in the program results page. (See Figure 9.)

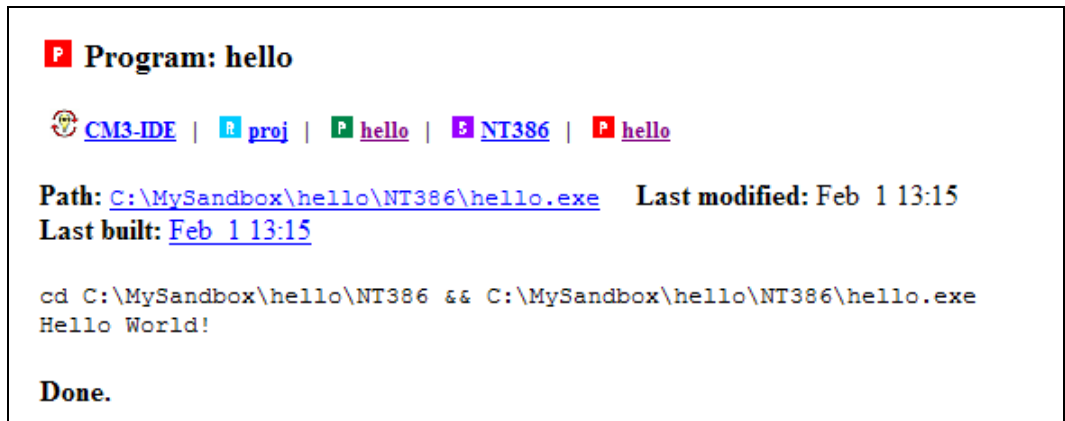


Figure 9. Running "Hello World".

You've just built and run your first CM3-IDE program.

You may use the CM3-IDE icon at the top left of the page you are on to return to the CM3-IDE start screen.

1.3 Creating a Package From Scratch

In the first tutorial you used a ready-made package that comes with CM3-IDE. All you had to do was navigate to it, build and run it. This time, you'll create a new package, open your text editor from CM3-IDE, add some code, and take a look at a very basic makefile.

1.3.1 List of All Packages

Step

From the CM3-IDE Start Page, follow the link to Packages to see a list of all available packages that are currently available within CM3-IDE. (See Figure 10.)

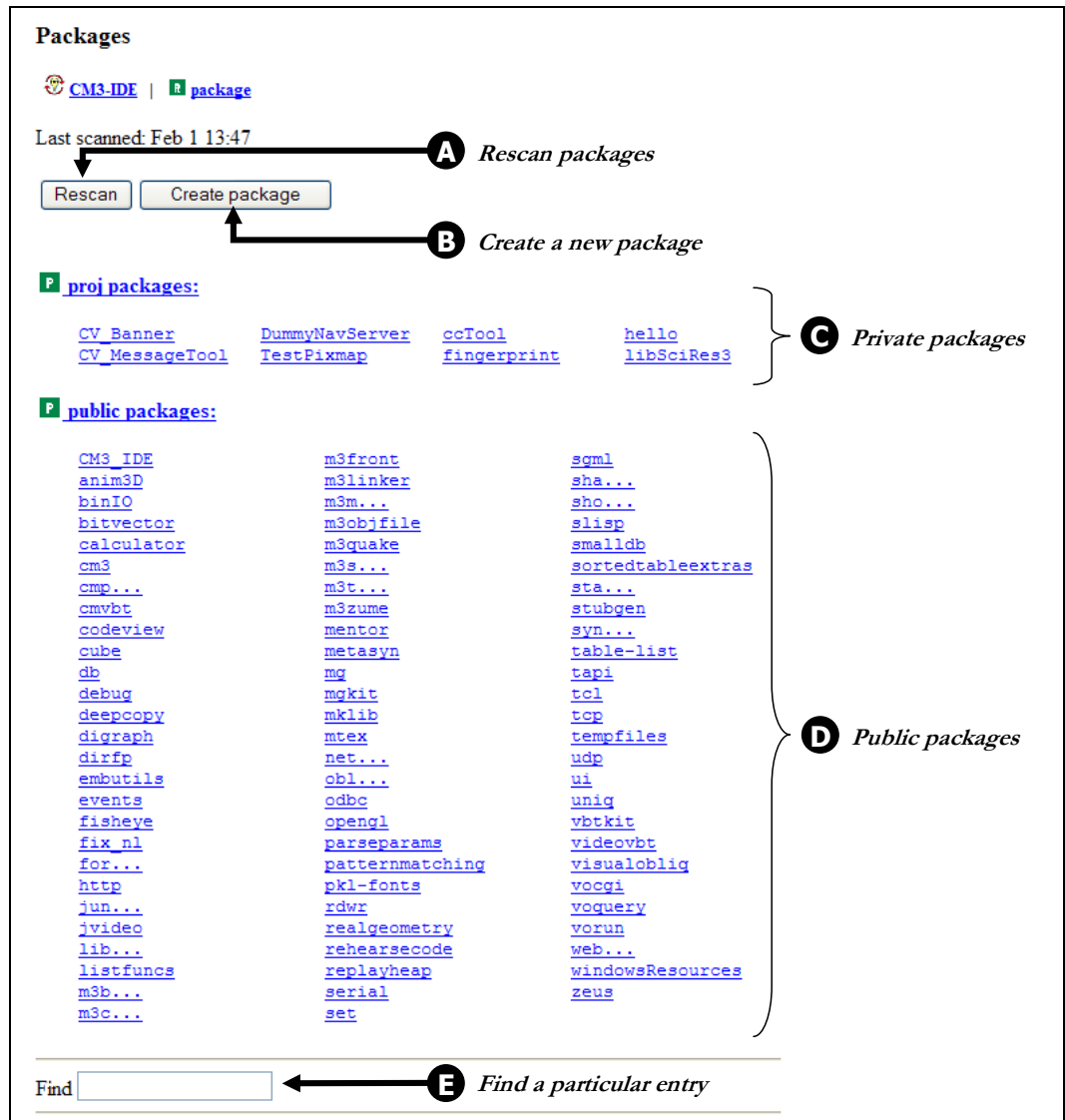


Figure 10. Packages Page: Listing of All Packages in CM3-IDE

Each (highlighted) package name represents a link to the summary page of a particular package. Some of the functions available on this page are:

- A** Rescan tells CM3-IDE to update its database from the files in your filesystem.
- B** Create New Package takes you to the New Package dialog, where you can create a new package.
- C** Your private packages are normally filed under “proj.”

- D** Public packages are normally filed under “public.”
- E** The Find type-in field instructs CM3-IDE to only list entries that match a particular regular expression, such as “m*”.

1.3.2 Creating New Packages

Step

Navigating to the Create Package dialog. Near the top of the page, you should see two buttons. (See Figure 11.) Click the right button, “Create package”. CM3-IDE will take you to the Create Package dialog.

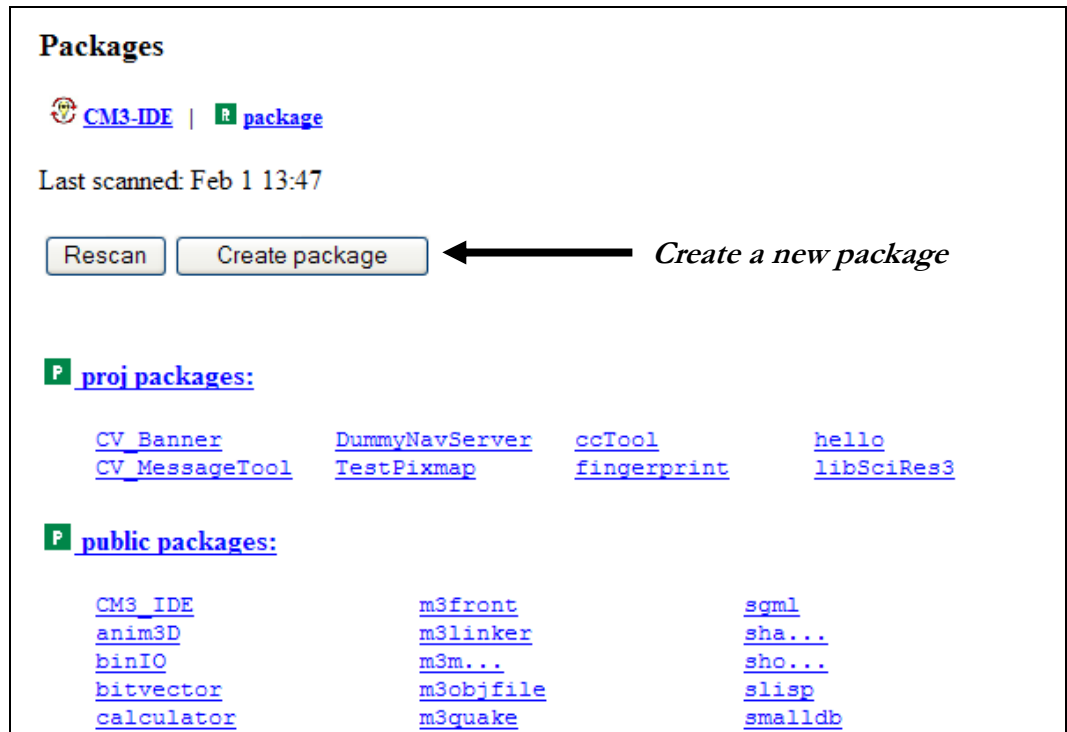


Figure 11. Top of the Packages Page

The Create Package dialog. Before CM3-IDE can create a package, you need to specify some information about the package to be created (Figure 12).

Create package A Click here to return to start page

CM3-IDE | form | new-pkg

(Specify the new package's root, name and kind)


Package root to use
☒ proj B Instruct CM3-IDE which package root to use


Name of the package
 C Name the package

What kind of a package
☒ **P** Program D Assign package type
☐ **L** Library

Create New Package

Figure 12. Package Creation Dialog

A To return to CM3-IDE’s start page, follow the  CM3-IDE link.

B Package roots are used by CM3-IDE to organize your packages. Before CM3-IDE can create any package, it needs to know what root that package will reside in. The package root “**proj**”—where your private packages reside—is a good place for this example package. Choose **proj**. 

Step

C Enter the name of your package here. Under “Name of the Package”, enter “MyPackage”.

Step


D When you create a new package, you’re given the option of creating a library and program. You are not locked into your decision here. Under “What Kind of Package,” select **P** Program.

Step

Click on the “Create new package” button at the bottom of the screen. CM3-IDE will create your package and point your browser to the package summary for the new package.

Step

Package Summary. You have just created the package **MyPackage**, but it doesn't do anything yet. You will need to write some code.

Step Look for the  Modules icon on the part of the page where the program elements are summarized. You'll see that there is only one module listed there, named "MyPackage". Click on it.

Module Summary. You may remember this page from our previous example. A module summary contains the code and relevant information regarding a module in a package. About half-way down the page you should see the module's code:

```
MODULE MyPackage EXPORTS Main;
BEGIN
END MyPackage.
```

Every CM3-IDE program must have a single main module, so when you tell CM3-IDE to create a program, CM3-IDE starts you off with an empty main module. Notice that there is no **IMPORT** statement here, and there is nothing between the keywords **BEGIN** and **END**. You will need to supply these.

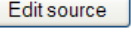
Step **Coding a Module.** Near the top of this page, find the row of action buttons. Click on . CM3-IDE should start your text editor and open the file **MyPackage.m3**.



Figure 13. The top of the Module Summary Page

Step In your text editor, add the following line between the line containing the word **MODULE** and the line containing the word **BEGIN**:

```
VAR name: TEXT; (* a string variable called "name" *)
```

This line uses the keyword **VAR** to declare a variable called "name", to be a string. The line ends with a semi-colon. Comments in CM3-IDE begin with "(*" and end with "*)"

Step

Add the following lines between the line containing “BEGIN” and the line containing “END”:

```
IO.Put("Enter the name of your nemesis: ");
name := IO.GetLine();
IO.Put(name & " is a stupidhead.\n");
```

The first line calls the procedure **IO.Put** passing the string “Enter the name of your nemesis:”.

The second line calls the procedure **IO.GetLine** and puts the string returned from that procedure into the **TEXT** variable you declared above, **name**.

The third line calls **IO.Put** again, passing it the string in the parentheses that follow.

1.3.3 Procedure Calls in CM3-IDE: What is “IO.Put?”

The expression **IO.Put** refers to a procedure **Put** in an interface called **IO**.

IO.Getline refers to the **Getline** procedure in the interface **IO**. (See Figure 14.)

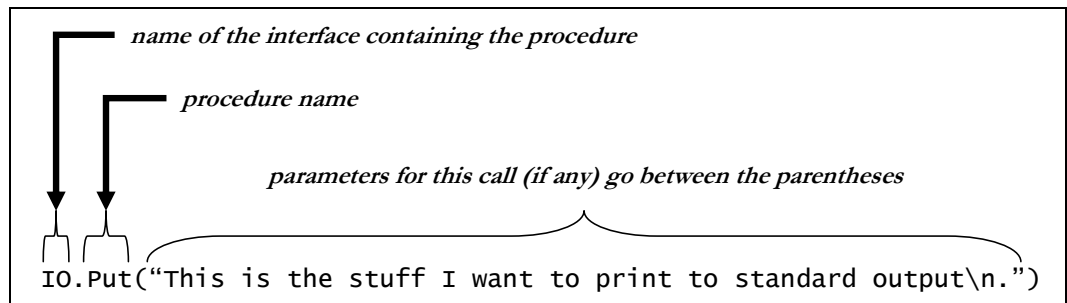


Figure 14. A Procedure Call Crossing Module Boundaries

The identifier to the left of the “.” is the name of the interface in which the procedure is declared. The one to the right of the “.” is the name of the procedure. Parameters for this call go inside the parentheses that follows the procedure’s name.

Interfaces in CM3-IDE are used to bundle relevant procedures, types, and constants in one syntactic unit. For example, the **IO** interface includes all the procedures you may need for simple input/output.

1.3.4 The IO Interface

Some of the procedures in the **IO** interface will be used in the package you are building. Before using a procedure, you may want to make sure you are calling the correct procedure by reviewing the interface where it is defined. The next few steps show how you can explore the **IO** interface from your current package.


Step







Navigating to the IO Interface. Leave your text editor open and return to your web browser. Your browser should still show the module summary for **MyPackage**.

Step

Click on the  CM3-IDE icon at the top of the page to return to the start screen.

Step

On the start screen, find the  Interfaces icon and click on it. to navigate to a list of all available interfaces. Find and click on the word “**IO**.” (Depending on your CM3-IDE display settings, you may have to click on an “**I . . .**” entry first.) This should bring you to the “**IO**” interface (Figure 15).


[CM3-IDE](#) |
  [public](#) |
  [libm3](#) |
  [src](#) |
  [rw](#) |
  [IO](#)

Path: [c:\cm3\pkg\libm3\src\rw\IO.i3](#) **Last modified:** Jan 24 14:44
Exported by: [IO.m3](#) **Imported by:** [65 units](#)

The **IO** interface provides textual input and output for simple programs. For more detailed control, use the interfaces **Rd**, **Wr**, **Stdio**, **FileRd**, **FileWr**, **Fmt**, and **Lex**.

The input procedures take arguments of type **Rd.T** that specify which input stream to use. If this argument is defaulted, standard input (**Stdio.stdin**) is used. Similarly, if an argument of type **Wr.T** to an output procedure is defaulted, **Stdio.stdout** is used.

```

INTERFACE IO;

IMPORT Rd, Wr;

PROCEDURE Put (txt: TEXT; wr: Wr.T := NIL);

```

Output txt to wr and flush wr.

Figure 15. The top of the IO Interface

Look at the top of interface **IO**. Beneath the Quick Access Icons, you’ll see information about the path, last modified date, and import and export lists. You can use the links under “Imported by” to find out what units use this interface, or the links under “Exported by” to see where the procedures declared here are defined.

Further down the page is a list of elements declared in the **IO** interface (Figure 16). The procedure **Put** is at the top of the list, and **GetLine** is fifth from the top; these are the procedures used in **MyPackage**. Notice that the procedure names here are highlighted.

```

INTERFACE IO;

IMPORT Rd, Wr;

PROCEDURE Put (txt: TEXT; wr: Wr.T := NIL);

    Output txt to wr and flush wr.

PROCEDURE PutChar (ch: CHAR; wr: Wr.T := NIL);

    Output ch to wr and flush wr.

PROCEDURE PutWideChar (ch: WIDECHAR; wr: Wr.T := NIL);

    Output ch to wr and flush wr.

PROCEDURE PutInt (n: INTEGER; wr: Wr.T := NIL);

    Output Fmt.Int (n) to wr and flush wr.

PROCEDURE PutReal (r: REAL; wr: Wr.T := NIL);

    Output Fmt.Real (r) to wr and flush wr.

PROCEDURE EOF (rd: Rd.T := NIL): BOOLEAN;

    Return TRUE iff rd is at end-of-file.

EXCEPTION Error;

The exception Error is raised whenever a Get procedure encounters syntactically invalid input,
including unexpected end-of-file.

PROCEDURE GetLine (rd: Rd.T := NIL): TEXT RAISES {Error};

    Read a line of text from rd and return it.

A line of text is either zero or more characters terminated by a line break, or one or more characters
terminated by an end-of-file. In the former case, GetLine consumes the line break but does not
include it in the returned value. A line break is either {\tt "\n"} or {\tt "\r\n"}.

```

Figure 16. Some Procedures Defined in the IO Interface

Step

Click on the name of the procedure **GetLine**. Your browser will display the module summary page of the **IO** module, which contains the code for this procedure.

Now, you know what **IO.GetLine** and **IO.Put** do.

Wrapping up the code. Return to your text editor. Insert the cursor immediately below the line:

```
MODULE MyPackage EXPORTS Main;
```

Step

Type:

```
IMPORT IO;
```

The file in your text editor should now read as follows:

```
MODULE MyPackage EXPORTS Main;
IMPORT IO;
VAR name: TEXT; (* string variable called "name" *)
BEGIN
  IO.Put("Enter the name of your nemesis: ");
  name := IO.GetLine();
  IO.Put(name & " is a stupidhead.\n");
END MyPackage.
```

What did you just do? You have imported the **IO** interface in your module, so that you can access the two procedures declared inside it: **IO.Put** and **IO.GetLine**.

Modules and Interfaces: Importing and Exporting

You can control how modules and interfaces interact through **IMPORT** and **EXPORT** statements.

A *module* defines a collection of program elements. These elements could be constants, types, variables, or procedures. The module *exports* an interface to make some of its component elements available to *clients*.

An *interface* is a list of the elements to be made available. Any file that **IMPORTs** an interface is said to be a *client* of the interface. The **MyPackage** module used in this example is a client of the interface **IO**.

You can only access the procedures contained in a module by importing an interface that was exported by that module. To use **IO.Put** and **IO.GetLine**, you needed to import the **IO** interface. You did that by inserting "**IMPORT IO**" right after the module declaration.

Back to the Package. Now that you have reviewed the **IO** interface, it is time to build your program.

Step

Save your file, **MyPackage.m3**, and quit your text editor.

Step

In your web browser, find “Build Package: MyPackage” in your browser’s history (usually listed under the “Go” menu.) Or just hit the “Back” button of your browser enough times to get back to the **MyPackage** summary.

In the next few steps, we will quickly review the makefile for this project. Then we will build and run the program.

**A CM3-IDE
makefile is named
m3makefile.**

1.3.5 CM3-IDE Makefiles (m3makefile)

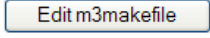
A CM3-IDE makefile is named **m3makefile**. A makefile is a text file containing instructions that tell CM3-IDE’s builder how to build a program or a library. An instruction is followed by one or more arguments in parentheses, similar to a procedure call in a programming language. Indeed, to build your package, CM3-IDE interprets your **m3makefile** as a little program.

Each instruction may specify a library, interface, or module to be included as part of the build. Comments in makefiles start with % and continue to the end of line.

For simple programs you can omit the makefile, and the builder will automatically find your modules and interfaces and their dependencies. However, creating makefiles for CM3-IDE packages is a good idea in general, especially since they are easy to create.

The button on the far right of the row of buttons near the top of the “a package summary” page is labeled “Edit m3makefile”:

Step

Click on . CM3-IDE will start your text editor, and open the file “m3makefile”. Here is what you should see in your text editor:

```
% Makefile for MyPackage
import("libm3")
implementation("MyPackage")
program("MyPackage")
```

When you create a new package, CM3-IDE automatically creates a basic makefile for you. As your package grows and becomes more complex, it is up to you to make sure your makefile is up-to-date, though doing so is straightforward.

Let’s take a look at the makefile for this package, line by line.

The first line:

```
% Makefile for My Package
```

is a comment. The rest of the line after % is ignored by CM3-IDE.


The second line:

```
import("libm3")
```

is a *makefile import command*. The **import** command tells the compiler that the program uses routines in the standard library, **libm3**. That’s the library that contains the **IO** interface and module.

Libraries

In CM3-IDE, a *library* is a package whose code may be reused as part of another library or an executable program. To use functionality of a library, you must import it in your makefile.

To learn more about libraries see Chapter 3, **Building And Sharing Packages** on page 47. To see a list of available libraries in CM3-IDE, click the  Libraries icon on the start screen.

Most makefiles include one or two import commands. If you use routines from other libraries, you must include other import commands that tell the compiler which libraries to include.

The third line:

```
implementation("MyPackage")
```

marks the program **MyPackage.m3** as a module implementation for your package. In your makefile, there must be one **implementation** command for each **“.m3”** file in your program. In this case, there was only one such file: **MyPackage.m3**.

Finally, the last line:


```
program("stupidhead")
```

tells the compiler to name the resulting executable file **“stupidhead”**. On Windows, executables have an **“exe”** extension.

Step

Quit your text editor, and, if you’ve modified your makefile, make sure you don’t save the changes. The makefile is fine as it is.



Step

Click the  button in the package summary for **MyPackage**. CM3-IDE will build your program, and point you to the Build Results page for **MyPackage** which will show any compiler error messages (in this case, you should not have gotten any) and warnings (which you may ignore for the moment.)

Your program is now ready to run. This program, however, is a bit more interactive than the one in this chapter’s first tutorial. You will need to run this one from a

command-line prompt. Once CM3-IDE has created an executable, you can run it directly from the operating system. This is what we'll do with this program.

Step

Click the  MyPackage icon in the Quick Access Icons area, or on the Back button of your browser to return to the package summary for **MyPackage**. You should see a link to the **stupidhead** program next to the  Programs category. If you don't, click the Reload button of your web browser.

Step

Click on the name of your new program to go to its program summary. At the top of that page, immediately beneath the Quick Access icons, you can read the location of the new program in your file system, right after "Path:".

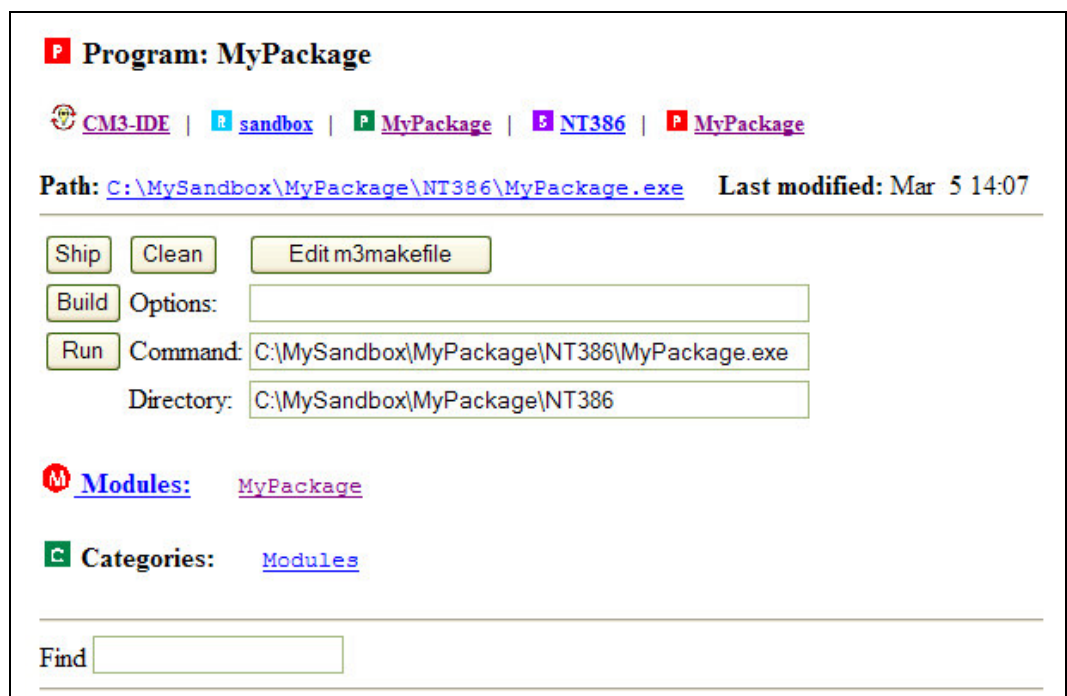


Figure 17. The top of a Program Summary page

Step

At the command-line, change your working directory to the one containing the executable. Type "**MyPackage**" at the shell prompt to run the program. Here is what you should see in your system window:

Enter the name of your nemesis:

Step

Do what it says; type the name of your nemesis here. If you enter "**My boss**" the program will write:

My boss is a stupidhead.

at the shell window and exit. What an intelligent and well-conceived program!

1.4 Summary

In CM3-IDE, projects are divided into *packages*. A project can consist of one or more packages. A CM3-IDE package comprises one or more modules and interfaces, along with a makefile that tells the compiler how to put everything together. Unlike their ancestors, CM3-IDE makefiles don't need dependency definitions.

Modules and *interfaces* are the building blocks of a CM3-IDE package. A module is a named collection of declarations, including constants, types, variables, and procedures. An interface can be thought of as a window into a module's functionality. To use another module, you *import* an interface that was *exported* by that module.

Both interfaces and modules may use *import statements*. By looking at the import statements for a module, you can easily discover its dependencies on other interfaces.

The basic form of a module and an interface is:

MODULE module-name;	INTERFACE interface-name;
IMPORT intf-1, intf-2,...;	IMPORT intf-1, intf-2,...;
Declarations ;	
BEGIN	Declarations ;
Statements ;	
END module-name.	END interface-name.

Statements terminate with a semicolon (“;”). Comments begin with “(“ and end with “*)”.

CM3-IDE makefiles, usually named **m3makefile**, define the steps for building a package. Here is a simple makefile:

```
% Makefile for a simple package
import("libm3")
implementation("module_name")
program("program_name") or library("lib_name")
```

Comments in makefiles start with “%” and continue to the end of the line. The call “**program**” at the end of a makefile marks that this package should be built as an executable program; the call “**library**” means this package should be built as a reusable library.

CM3-IDE is case-sensitive.

All keywords in CM3-IDE are capitalized.

This page left blank
intentionally.

Read this chapter for an overview of CM3-IDE's common user interface elements.

You also can learn about CM3-IDE's web namespace here.

2. The CM3-IDE Environment

This chapter describes CM3-IDE's common screens, tools, and icons. You can use this chapter as a reference for finding information about elements of CM3-IDE's development environment.

If you haven't already, you may consider reviewing the previous chapter to learn the basics of the CM3-IDE environment.

There are four sections in this chapter:

Common Tools, Icons, and Visual Elements on page 29 describes the common tools and icons within the CM3-IDE environment.

CM3-IDE Start Screen on page 32 describes CM3-IDE's top-level screen in detail.

Summary Screens on page 36 describes screens that are most useful for your development tasks with CM3-IDE. Each screen accompanies a summary of the information and links available from that page.

CM3-IDE's Web Namespace on page 42 provides more in-depth information about CM3-IDE's web namespace.

2.1 Common Tools, Icons, and Visual Elements

CM3-IDE's design defines a consistent environment for navigating, building, and sharing your programs. Common icons, tool buttons, and visual elements reinforce the relationship between different elements, enhancing your ability to find, filter, or act upon information presented within the CM3-IDE development environment.

Here we describe three important elements common to most CM3-IDE pages: Quick Access Icons, Action buttons, and the Find Type-in.



2.1.1 Quick Access Icons



From the CM3-IDE start screen, click on any icon. Wherever you wind up, you will find a row of small icons that cross the top of the page, just under the title for the page. This row of icons is called the quick access icons. The left-most icon (CM3-IDE) always returns you to the start screen. They can bring you to the top of a package, the current subdirectory, or to program, module, or interface summaries of your package components. They can speed your navigation by allowing you to quickly move from one part of the CM3-IDE environment to another.

Together, these icons present an active, visual placeholder for your location in the CM3-IDE web namespace. By using them, you can eliminate most of the Back and Forward activities common to web navigation of complex namespaces.

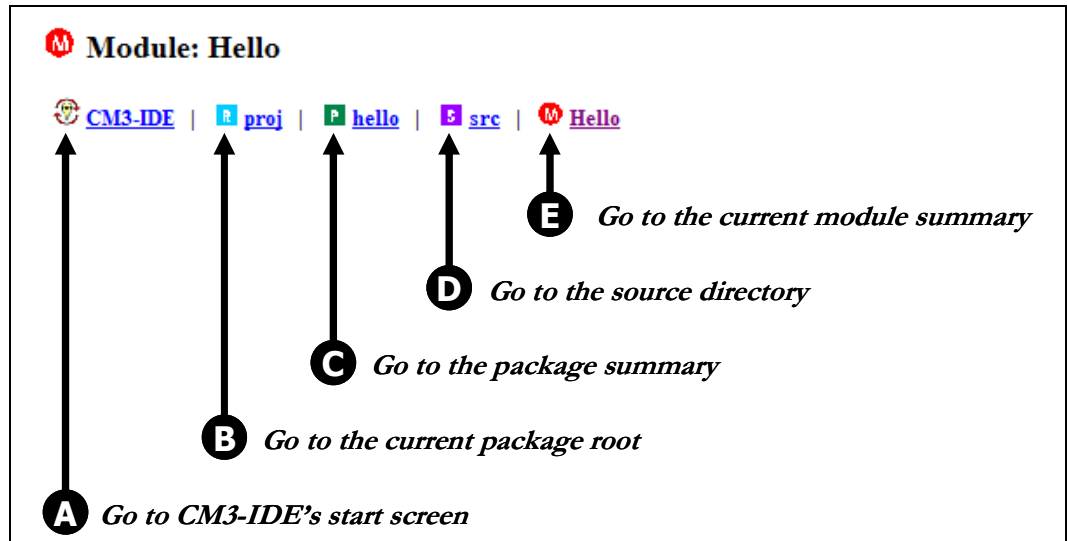



Figure 18. Quick Access Icons for /proj/hello/src/Hello.m3

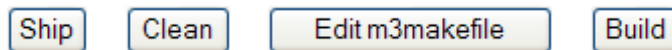
Figure 18 displays a sample row of quick access icons. The icons are arranged according to their level in CM3-IDE's web namespace hierarchy:

- A** the left-most icon (the one with the CM3-IDE logo ) will take you all the way back to the start screen
- B** takes you to the package root containing the current package
- C** takes you to the package summary for the current package

- D** takes you to the sources for your current package. CM3-IDE uses the “src” subdirectory of a package to keep the sources for your program.
- E** the last icon in the path points to the current page, which happens to be a module summary in this example. You can click on the last icon to reload the current page.

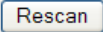
The quick access icons work well in conjunction with the history mechanism of your web browser. If you navigate using the quick access icons, your web browser will not lose the starting point in your navigation. Quick access icons allow you to move to a page you’ve already visited without causing your browser to lose track of your current page in its history.


2.1.2 Action Buttons





In many ways, working with CM3-IDE is just browsing a web server—much of CM3-IDE’s functionality allows fluid navigation within your programming workspace, for example, to jump from a package summary page to the summary page for a library it builds. Each icon or link on a CM3-IDE screen is usually a reference to another element within CM3-IDE.

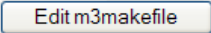
However, CM3-IDE is more than just a fancy web server. Many CM3-IDE screens allow the invocation of *actions*. For example, a module summary screen includes an action to invoke an editor on the code for the module. Such actions are represented by *action buttons*. The set of available actions varies from page to page, and so does the set of action buttons. Here we describe some of the common action buttons:

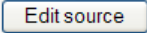
 Forces CM3-IDE to scan the filesystem to update its state for the current page.


 Builds the currently selected package and shows you the result of the build. Use the text field to the right of the Build button to enter options for the builder. For more information on building, see **Building and Sharing Packages** on page 47.

 Erases derived files from the current package. Clean will not erase source programs. For more information on building, see **Building and Sharing Packages** on page 47.

 Releases the current package as a public package, making its contents available for importing, browsing, or executing by other developers in your team. For more on shipping, see **Building and Sharing Packages** on page 47.

 Instructs your text editor to open the makefile used to build the current program.

 Instructs your text editor to open the source file corresponding to the current page.

 Command:
 Directory:

Instructs CM3-IDE to run the specified command in the specified directory. CM3-IDE normally fills in the command and directory fields, but you may change them.

2.1.3 The Find Type-in

Find

Wherever CM3-IDE displays a dynamic list of choices (for example, the list of all the packages in your system, or the list of all the modules in a particular package), it also presents the *Find type-in*. It allows you to filter the set of available entries in the current screen to a smaller set, based on the information specified in its type-in field.

To use the find type-in, simply type in the text you are searching for and press return.

The find type-in can accelerate your navigation within CM3-IDE in several ways:

- instead of visually searching a screen for a particular element, you may type the element name, for example “**Main.m3**”, in the find type-in.
- instead of searching within a multi-page list of entries by navigating, you can type the name of the item of interest. For example, in the find type-in for the list of all interfaces in your system, you may type “**Fingerprint**” to view the interface **Fingerprint**.
- instead of using CM3-IDE’s navigation via links, you may directly jump via a (full or partial) URL to any element. For example, while visiting package **MyPackage**, you may type “**/interface/IO**” in the find type-in to visit the interface **IO**. (See **CM3-IDE’s Web Namespace** on page 42 for more information on CM3-IDE URLs.)

2.2 CM3-IDE Start Screen

The CM3-IDE environment consists of many screens (called pages in web-speak.) The next two sections describe some of the more common CM3-IDE screens.

The *start screen* is the screen you see when you first start CM3-IDE. From any page in CM3-IDE, you can return to this screen by clicking on the CM3-IDE icon  located at the top-left corner of your page.

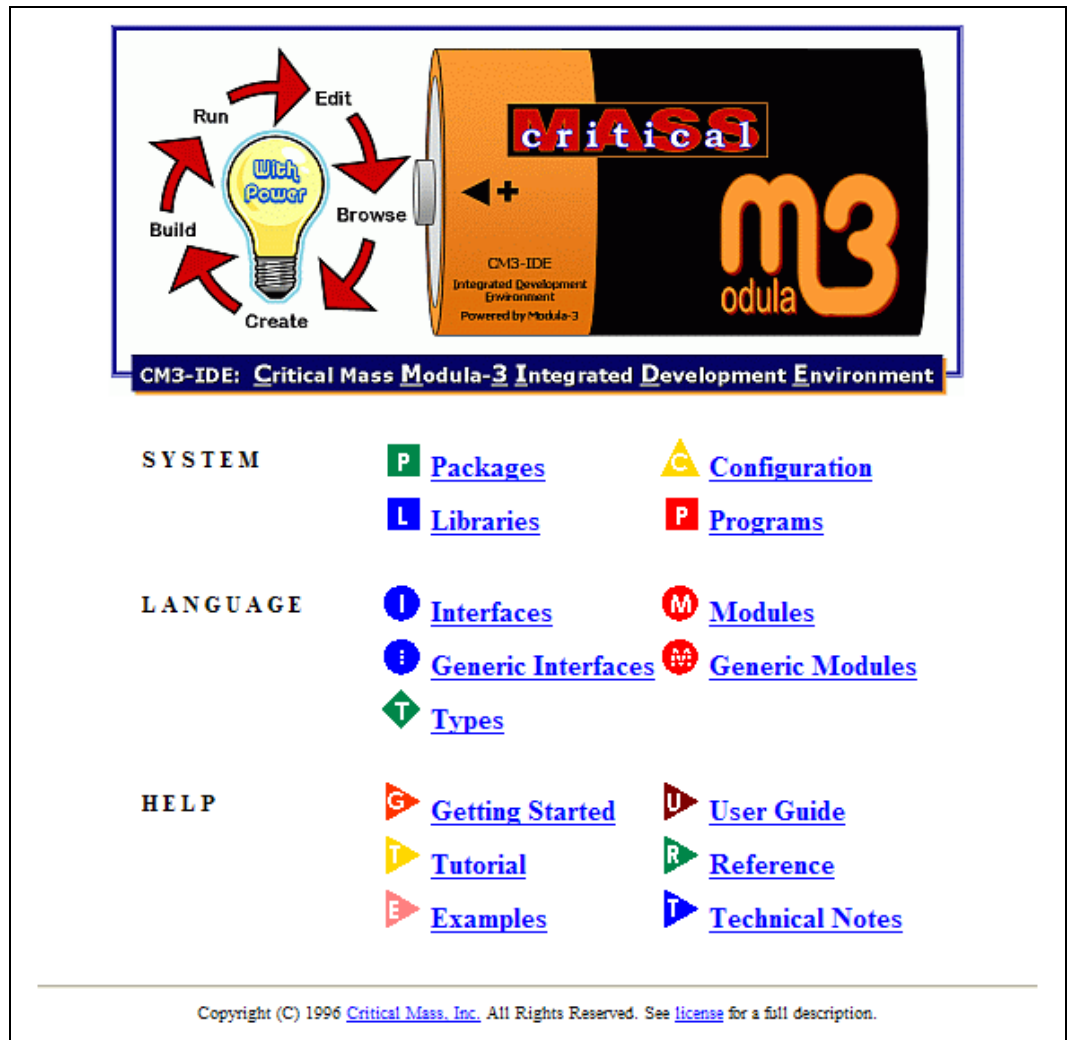






Figure 19. CM3-IDE's Start Screen

On the start screen you'll see three groups of icons. The groups are labeled System, Language, and Help. Each group includes a few links.


- System icons allow you to browse, modify, and customize the CM3-IDE system. You can create a package by following the  Packages link.
- Language icons provide access to program elements in your system. You can find out about all the available interfaces by following the  Interfaces link.
- Help icons refer you to on-line help and information sources. You can find many CM3-IDE examples by following the  Examples link.

Here we describe each of the icons on the start screen.


2.2.1 Start Screen: System

 **Packages.** Click on this icon to go to the Packages page, where you will see the list of packages in your system organized under package roots (such as public, the public package root, and proj, your private package root.) Click on a package name on the Packages page to visit a summary page for that package.


(For more information about packages, and their organization, see **Building and Sharing Packages** on page 47.)

 **Configuration.** Click here to go to CM3-IDE's Configuration page.

(For information about changing your configuration, see **Customizing CM3-IDE** on page 65.)



 **Libraries.** Click here to see a list of all libraries in your system. Clicking on the name within this list will take you to that library's summary.



(For more about libraries, see **Building and Sharing Packages** on page 47.)


 **Programs.** Click on the Programs to see a list of all programs in your system. Clicking on the name within this list will take you to that program's summary, where you can execute the program.

(For more information about programs, see **Learning the Basics** on page 5.)


2.2.2 Start Screen: Language

 **Interfaces,**  **Generic Interfaces.** Click on the Interfaces icon to view the list of all available interfaces where you may click on the name of any interface to view its summary page. Generics Interfaces works similarly.

 **Modules,**  **Generic Modules.** Click on the Modules icon to view the list of available modules where you may click on the name of any module to view its summary page. Generics Modules works similarly.

 **Types.** Click the Types icon to view the list of types available in all the programs on your system. Selecting a type from the list of all types causes CM3-IDE to display information about that type.

2.2.3 Start Screen: Help


 **Getting Started.** If you haven't read the *CM3-IDE User Guide*, click here for a quick starting point. The page includes an annotated start page, along with tips about many of the common CM3-IDE elements.


Novice users may use this page as their start page. See **Customizing CM3-IDE** on page 65 to learn how to change your start page.


Impatient users can follow this link to find basic information so that they can start using CM3-IDE immediately.

Advanced users may use its HTML source to learn more about customizing CM3-IDE. (See also **CM3-IDE's Web Namespace** on page 42.)

 **User Guide.** Click here to get to the on-line user guide. The *CM3-IDE User Guide* is available in PDF format. Your CM3-IDE distribution contains a copy of Adobe Acrobat PDF Reader for supported platforms.

 **Tutorial.** Following the tutorial link, you will see the list of available tutorials. Two tutorials are available currently: one on Modula-3 and another on Trestle, the portable user interface toolkit shipped with your CM3-IDE distribution.

 **Reference.** Click here to view an on-line, hypertext version of the Modula-3 language definition. This is the on-line version of the printed language definition you received as part of the CM3-IDE package.

 **Examples.** Click here to see a list of example packages for CM3-IDE. When you click on the name of any example package in the list, CM3-IDE creates a private package for you, allowing you to build, modify, and experiment with the example program without corrupting its sources.

You can use the example packages to study many of the programming concepts of CM3-IDE. Sources within examples contain on-line documentation that will guide you through learning various CM3-IDE concepts.

 **Technical Notes.** Click here to see a list of technical reports and information sources about the Modula-3 programming language. View them on-line or print them out as PostScript or Acrobat PDF documents.

2.2.4 Customizing the CM3-IDE Start Screen

You can change your CM3-IDE start screen at any time. Novices may like to use annotated start screens that include helpful comments; advanced users may want to tailor the start screen for more specific needs.

See **Customizing CM3-IDE** on page 65 for more information about changing your start screen.

2.3 Summary Screens

For each programming element in CM3-IDE, there is a corresponding summary screen. The rest of this section describes the common summary screens. Most of these screens are described as part of the hands-on examples in the previous section. Consider working through the examples in **Learning the Basics** on page 5 to learn more about CM3-IDE.

2.3.1 Package Summary

A package summary describes a package and its contents. From a package summary, you can access both sources and derived files in a package.

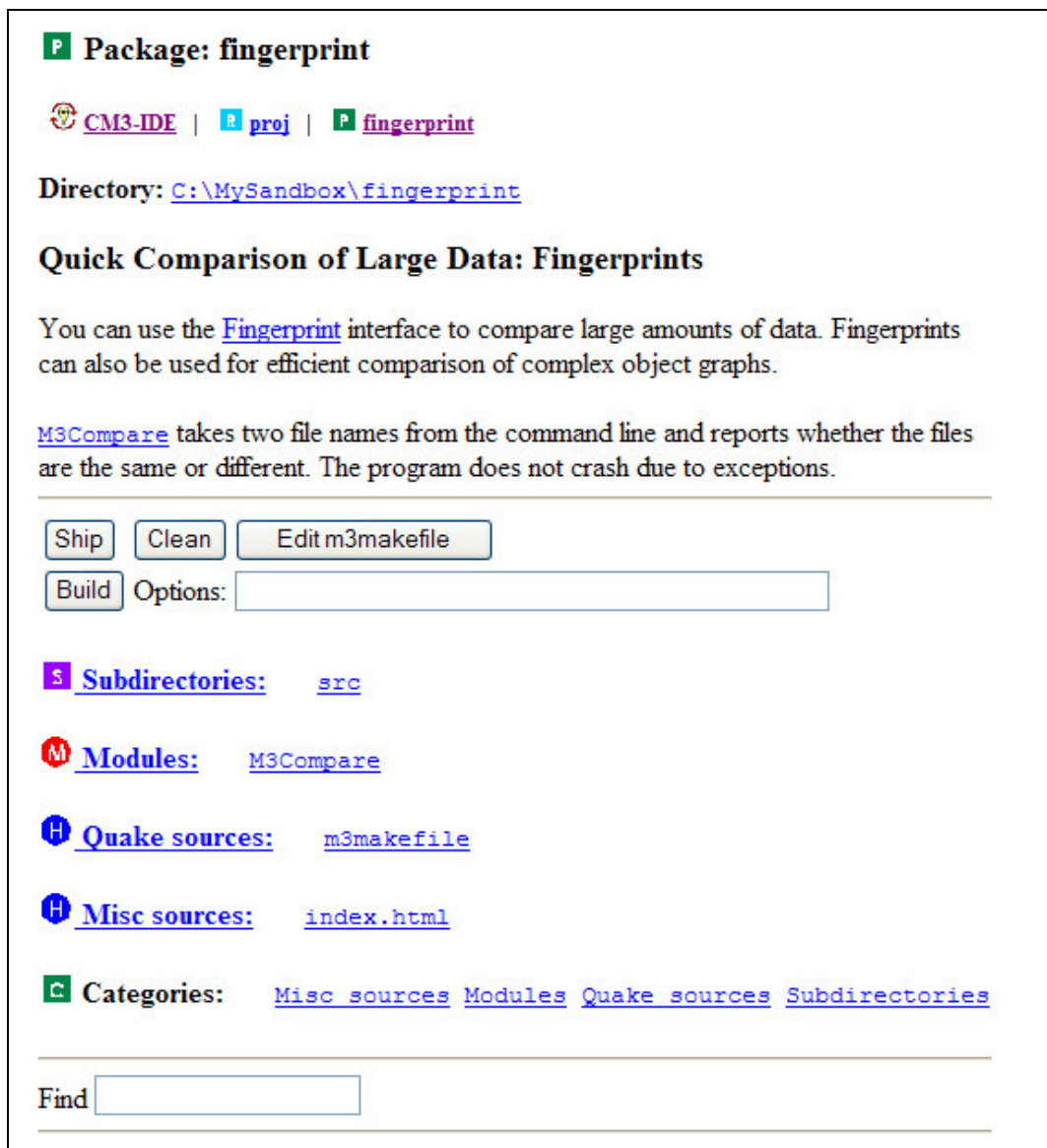


Figure 20. A Package Summary

Click on any package component (such as an interface or a module) to view it. When browsing package components, you can always return to this page by clicking on the package icon (marked by package-name) as part of the quick access icons.

From top to bottom, a package summary contains:

- quick access icons
- filesystem directory for the current package
- textual description of the current package. This information is extracted from an `index.html` or `README` source file in the current package. An `index.html` file for a package is interpreted within the web namespace for the package. Hence, it can include hypertext links to other package elements, or other names within CM3-IDE. See **CM3-IDE's Web Namespace** on page 42 for more information on CM3-IDE URLs.
- action buttons for this package, to build, ship, or clean the package
- a summary of the sources and derived files that comprise the current package. Clicking on the name of an element will point your browser to the files for that element.
- find type-in. Type in a regular expression to search within the names available in this package. For example, typing in “`m*`” will show you all the elements that start with “`m`”.

As is the case with any web page, the information available as part of a package summary may become stale. To bring a package summary up-to-date with its directory contents, reload the package summary page, or use the Rescan button if it is visible. (The Rescan button is only enabled if you have specifically configured CM3-IDE to display it. See **Customizing CM3-IDE** on page 65 for more information.)

2.3.2 Library Summary

A *library summary* describes a library—a group of sources and compiled files in a package that can be used in construction of other packages. The summary page also shows the sources and derived files that comprise the library, where the library resides, and when it was last modified.

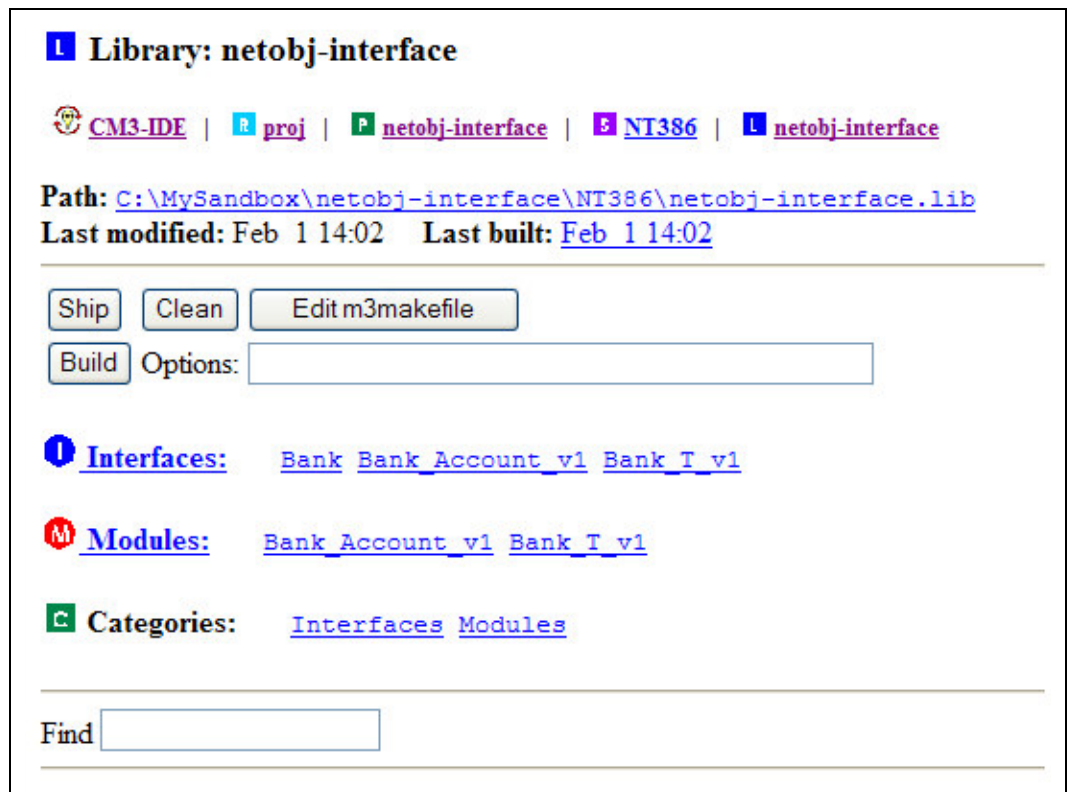


Figure 21. A Library Summary

Public libraries (filed under `/public` in CM3-IDE's namespace) may be imported by other packages in CM3-IDE. For more information about using and sharing libraries, see **Building and Sharing Packages** on page 47.

From top to bottom, a library summary consists of the following elements:

- quick access icons
- path to the library; last modified date; and last build date. If the last build was during the current session, the last built date is a link to the result of that build.
- action buttons for the library
- summary of the sources and derived files that comprise the current library. Clicking on the name of a library element listed here will point your browser to that element's summary page.
- the find type-in.

2.3.3 Program Summary

A *program summary* describes an executable program built using the CM3-IDE system. The summary page for a program shows the sources and derived files that comprise the program, where the program resides, and when it was last modified.

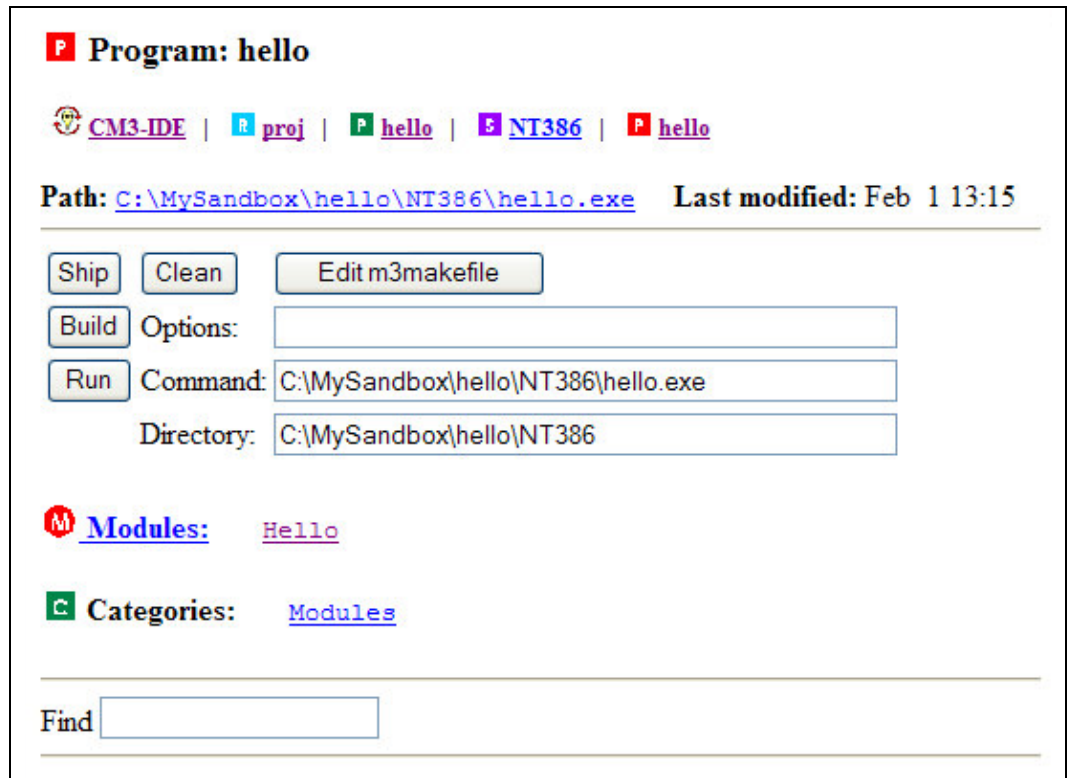


Figure 22. Top Portion of a Program Summary

Public programs (filed under `/public` in CM3-IDE's namespace) are visible to other developers in your group. For more information about using and sharing programs, see **Building and Sharing Packages** on page 47.

From top to bottom, a program summary consists of the following elements:

- quick access icons
- path to the program; last modified date; and last build date. If the last build was during the current session, the last built date is a link to the result of that build.
- action buttons for this page, including the Run button
- summary of the sources and derived files that comprise the current program. Clicking on the name of a program element listed here will point your browser to that element's summary page.

2.3.4 Interface Summary

An *interface summary* describes an interface. See **Learning the Basics** on page 5 if you would like to learn about interfaces.

 [CM3-IDE](#) |
  [public](#) |
  [libm3](#) |
  [src](#) |
  [fmtlex](#) |
  [Fmt](#)

Path: [c:\cm3\pkg\libm3\src\fmtlex\Fmt.i3](#) **Last modified:** Jan 24 14:44
Exported by: [Fmt.m3](#) **Imported by:** [357 units](#)

The `Fmt` interface provides procedures for formatting numbers and other data as text.
`\index{writing formatted data} \index{formatted data!writing}`

```

INTERFACE Fmt;

IMPORT Word, Real AS R, LongReal AS LR, Extended AS ER;

PROCEDURE Bool (b: BOOLEAN) : TEXT;

    Format b as {tt "TRUE"} or {tt "FALSE"}.

PROCEDURE Char (c: CHAR) : TEXT;

    Return a text containing the character c.

TYPE Base = [2..16];
  
```

Figure 23. An Interface Summary

From top to bottom, an interface summary consists of the following elements:

- quick access icons
- file information for this interface (physical path, last modified)
- export and import dependencies (if the interface is part of a compiled package). Click on “imported by” to see the sources that import this interface. Click on “exported by” to see the sources that export this interface.

- action buttons for this interface, such as Build, Ship, Clean, or Edit Source. These actions are only available when you are allowed to build the package that contains the current interface.
- the code for the interface.

You may click on any of the links within the body of an interface to reach the corresponding program element. For example, you click on a procedure name to view the procedure body within the corresponding module. Some cross-reference links may not be available until the enclosing package is built.

2.3.5 Module Summary

A *module summary* describes an module. See **Learning the Basics** on page 5 if you would like to learn about modules.

 **Module: NetObjServer**

 [CM3-IDE](#) |
  [proj](#) |
  [netobj-server](#) |
  [src](#) |
  [NetObjServer](#)

Path: [C:\MySandbox\netobj-server\src\NetObjServer.m3](#)
Last modified: Aug 14 18:53 **Last built:** [Feb 1 14:03](#)

Ship Clean Edit m3makefile Edit source
Build Options:

```

MODULE NetObjServer EXPORTS Main;
IMPORT Bank, NetObj, Thread;
IMPORT IO, Fmt;

Create an implementation object for the Bank.T network object.

TYPE
  BankImpl = Bank.T OBJECT
    accounts : ARRAY Bank.AcctNum OF Account;
  OVERRIDES
    findAccount := FindAccount;
  END;

```

Figure 24. Portions of a Module Summary

From top to bottom, a module summary consists of the following elements:

- quick access icons

- file information for this module (physical path, last modified)
- action buttons for this module, such as Build, Ship, Clean, or Edit Source. These actions are only available when you are allowed to build the package that contains the current interface.
- code for the module.

You may click on any of the links within the module body to reach the corresponding program element. For example, you click on a procedure name to view its declaration in the corresponding interface. Some cross-reference links may not be available until a package is built.

Following the links in the **MODULE** or **EXPORTS** clauses of the code, you will visit the interfaces exported by this module. Following the links in the **IMPORT** clause for a module will take you to the interfaces imported by this module.

Most modules include many links to other elements (inside or outside the module) referred to by the code in the module. CM3-IDE marks up your code dynamically to help you navigate your programs.

2.4 CM3-IDE's Web Namespace

This section describes CM3-IDE's web namespace. It is intended as a reference for advanced users.

The heart of CM3-IDE's user interface is its custom web server; CM3-IDE is driven by HTTP requests from your browser. To do this, CM3-IDE associates every one of its elements with a URL or a path.

To see the URL to the current CM3-IDE element, turn on the "Location" or the "URL" display on your browser. CM3-IDE URLs are not only useful for internal use by CM3-IDE itself, they can be used by you in the same way you use ordinary web URLs. You can send URLs referring to a CM3-IDE package, or interface, or even a procedure to a co-worker. Or you may save bookmarks to useful URLs within CM3-IDE. Moreover, you may refer to various elements within CM3-IDE's namespace from your own sources and documentation.

For example, you may include in the `index.html` file for a package, a link to a procedure in your package, with an implementation note for the procedure that points to a particular section of the language reference.

The possibilities for extending CM3-IDE based on this concept are endless. The rest of this section aims at explaining the basic syntax and semantics for CM3-IDE URLs.

2.4.1 CM3-IDE URLs

A CM3-IDE URL follows the format:

`http://host:port/path`

where *host* is the host name where you are running CM3-IDE (default: `localhost` which is the host running your browser), *port* is the port that the CM3-IDE server uses (default: `3800`), and *path* is the location of the element within the CM3-IDE namespace. A typical CM3-IDE path may be:

`http://localhost:3800/interface/IO`

By convention, we omit the protocol, host, and port information for a CM3-IDE URL, so the above example will be described as:

`/interface/IO`

If an expression matches multiple elements within CM3-IDE's namespace, CM3-IDE will display a list of matches and will allow you to choose one. If there is only one match for your query, CM3-IDE will automatically display its contents. (Note that even a fully-qualified name, such as `/module/Main`, may result in multiple matches.)

A URL also may contain a trailing action, specified via brackets “[” and “]”. For example, `/interface/IO/[edit]` opens your text editor with the file containing the `IO` interface, or `/proj/hello/[build]` builds the package `hello`.

2.4.2 Example CM3-IDE URLs

Here we briefly include some examples of URLs for CM3-IDE.

<code>/proj</code>	your private packages
<code>/public</code>	all public packages
<code>/interface/IO</code>	interface <code>IO</code>
<code>/module/M*</code>	all modules that start with “M”
<code>/intf/*/Get*</code>	all interface contents starting with “Get”
<code>/log</code>	the last 500 lines of the CM3-IDE log
<code>/help</code>	on-line help
<code>/reference</code>	language reference
<code>/</code>	the start page
<code>/proj/hello[build]</code>	build my private package <code>hello</code>
<code>/module/Hello.m3/[edit]</code>	edit <code>Hello.m3</code> module

2.4.3 Regular Expressions in CM3-IDE URLs

CM3-IDE's namespace lookup includes support for regular expressions. That is, in CM3-IDE, paths that express a regular expression (with the correct syntax) are valid URLs. This feature is useful in browsing the large amounts of information available in CM3-IDE.

In particular, you may use the regular expression language in the find type-in. For example, at any point of time, you can type in `"/interface/Rd*"` in a find type-in to search all interfaces whose name starts with `"Rd"`, or `"/interface/(Rd|Wr)/Get*"` to find all declarations in `Rd` and `Wr` which start with `Get`.

Regular Expressions Syntax. Here is a list of meta-characters that you may use in CM3-IDE URLs.

<code>*</code>	any string of zero or more characters
<code>@</code>	any single character
<code><expr1> <expr2></code>	strings matching <code><expr1></code> or <code><expr2></code>
<code><expr1>&<expr2></code>	strings matching <code><expr1></code> and <code><expr2></code>
<code>(expr)</code>	strings matching <code><expr></code>
	(and has higher precedence than or)
<code>*</code>	the single character <code>*</code> (asterisk)
<code>\@</code>	the single character <code>@</code> (at sign)
<code>\ </code>	the single character <code> </code> (vertical bar)
<code>\&</code>	the single character <code>&</code> (ampersand)
<code>\(</code>	the single character <code>(</code> (left parenthesis)
<code>\)</code>	the single character <code>)</code> (right parenthesis)
<code>\\</code>	the single character <code>\</code> (back slash)

To learn more about CM3-IDE URLs, examine the sources of fixed CM3-IDE screens, such as `/help/getting-started.html`, or `/example/*/src/index.html` or even the dynamically generated CM3-IDE pages.

2.5 Summary

This chapter describes the common user interface elements in CM3-IDE.

Quick Access Icons. To quickly navigate within the CM3-IDE screens, use the quick access icons. They are displayed on top of every screen. (See page 30.)

Action Buttons. Each CM3-IDE screen supports a number of actions, for example, a module may support an edit action, and a package may support a build action. Available actions for each CM3-IDE element are presented as action buttons in the summary page for that element. (See page 31.)

The Find Type-in. You may search for a particular element within the currently displayed list, or specify a path to a new place in the CM3-IDE namespace in the find type-in box. (See page 32.)

CM3-IDE Start Screen. The start screen is displayed when CM3-IDE starts up. It provides links to directories of elements in the CM3-IDE environment, such as interfaces, modules, packages, libraries, programs, and types. You may also customize CM3-IDE to use a different start screen. (See page 32.)

Summary Screens. Each programming element, such as a package or an interface, has a corresponding summary screen within CM3-IDE. Each summary displays a particular element, links to relevant information, and action buttons for the relevant actions. (See page 36.)

CM3-IDE's Web Namespace. The heart of CM3-IDE's user interface is a custom HTTP sever. CM3-IDE screens are normal web pages; you can send references to them in e-mail to your co-workers, or save a bookmark for them in your browser. CM3-IDE's URLs may contain regular expressions. For example `/interface/F*` displays a list of all interfaces that start with the letter "F". Actions for an element are enclosed in square brackets ("[" and "]"). You can invoke actions by using URLs also, for example, `/module/Hello[edit]` edits `Hello.m3` the file for the `Hello` module. (See page 42.)

This page left blank
intentionally.

Read this chapter if you know the basics of CM3-IDE and would like to learn how you can build packages and share them with others in your team.

3. Building and Sharing Packages

This chapter covers the basics of building and sharing packages in CM3-IDE. It also describes how CM3-IDE facilitates the building of large, multi-developer projects.



Each section of this chapter explores a particular aspect of building and sharing packages with CM3-IDE.

Building Packages on page 48 describes how to build a package by invoking CM3-IDE's builder.

Directory Structure of a Package on page 49 illustrates the directory structure of a basic CM3-IDE package.

CM3-IDE Makefiles on page 50 defines the basic syntax for CM3-IDE makefiles.

Managing Multiple Packages on page 53 shows how to divide your projects into multiple packages.

Shipping Packages on page 54 describes how to ship a package to make it available for importing.

Sharing Packages on page 57 explains how to share packages in a multi-developer team with CM3-IDE.



Builder Options on page 62 lists the command-line options available for CM3-IDE's builder, `cm3`.

3.1 Building Packages

Along with navigational links, many of CM3-IDE’s screens include associated actions. For example, a package summary page may allow a “build” action to bring the package up-to-date, or a module summary may allow an “edit” action to edit the source file for that module.

Most CM3-IDE pages include buttons for valid actions. For example, the  Build button denotes the “build” action on package pages.

Step

Starting from the CM3-IDE start screen, click on the  Packages icon. Following any of the links under  “proj” packages—your private directory—will lead you to a package summary page. If you have followed **Learning the Basics** on page 5 properly, you will see at least two links: **hello** and **MyPackage**.

Summary pages of your packages or their components always include a Build button.

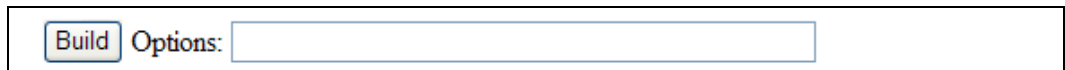


Figure 25. CM3-IDE’s Build Button

Clicking on the Build button will start CM3-IDE’s builder, and display the build results on the screen. If there are any errors, CM3-IDE displays hypertext links to errors in your source files.

CM3-IDE’s builder is called “**cm3**”, short for *Critical Mass Modula-3*. Indeed, **cm3** is a stand-alone compiler/builder for the Modula-3 language that is integrated within the CM3-IDE environment.

At the start of a build, **cm3** first looks for a makefile for the current package. (Makefiles in CM3-IDE are named “**m3makefile**”.) If it can’t find a makefile, it attempts to build a program from the files in your package directory. While you don’t always need to create a makefile, it is a good idea to create one for clarity.

CM3-IDE’s makefiles are discussed in more detail later in this chapter. You can find more information about customizing the behavior of the Build button in **Customizing CM3-IDE** on page 65, or in the **cm3** configuration file (**cm3.cfg**) in your CM3-IDE installation.

CM3-IDE's Builder

CM3-IDE's combined builder and compiler, **cm3**, has been designed specifically for the creation of robust and distributed programs.

When you click CM3-IDE's Build button, CM3-IDE invokes **cm3** to build your program. You may also invoke **cm3** from the command-line by issuing the command **cm3** from your command-line shell. See **Builder Options** on page 62 for more information on running **cm3** from the command-line.

3.2 Directory Structure of a Package

Each package in CM3-IDE resides in a directory, with sources in a source subdirectory, and generated files in a derived subdirectory.

Names of CM3-IDE's derived directories:

ALPHA_OSF
HPPA
IRIX5
IBMR2
LINUXELF
NT386
SOLsun
SOLgnu
SPARC

The source directory for a package is named “**src**”; its contents are the same on all platforms. In contrast, the name and contents of the derived directory for a package varies from one platform to another.

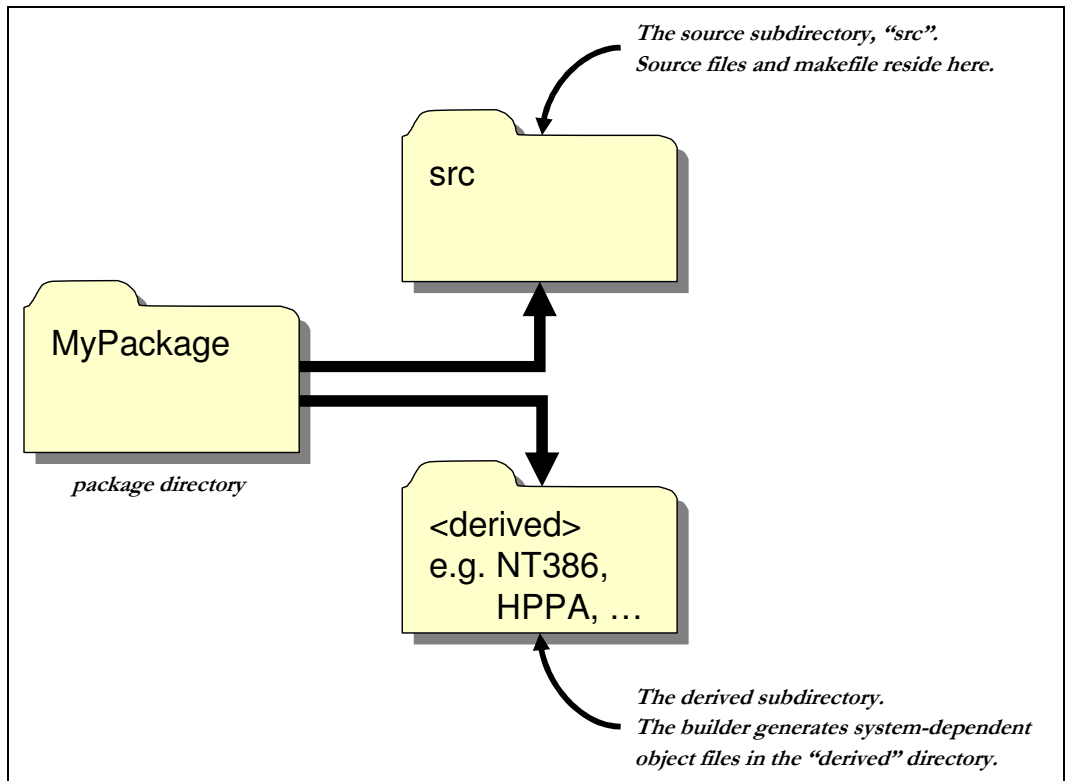


Figure 26. CM3-IDE Package Directory Structure

The name of the derived directory denotes the platform where CM3-IDE built the system, for example, **NT386** is the platform name for Win32 running on Intel x86 processors, and **HPPA** is the name for HP/UX running on HP Precision Architecture series.

The default names for derived directories are:

ALPHA_OSF	Digital Unix (OSF/1) on DEC Alpha
HPPA	HP/UX on HP Precision Architecture
IRIX5	SGI Irix on SGI/MIPS
IBMR2	AIX on IBM RS/6000
LINUXELF	Linux/ELF on Intel x86
NT386	Win32 (Win95 or NT) on Intel x86
SOLsun	Solaris 2 on SPARC (Sun C compiler)
SOLgnu	Solaris 2 on SPARC (GNU C compiler)
SPARC	SunOS 4 on SPARC

The separation of source and derived files is useful when building larger programs, because it:

- isolates source files for backup, revision control, and searching
- enables sharing the same source tree across operating systems and architectures, without confusing object files from different platforms.

This arrangement, combined with CM3-IDE's multi-platform libraries, simplifies the management of large, multi-platform programs.

3.3 CM3-IDE makefiles

If you have worked through the tutorials in **Learning the Basics** on page 5, you've already seen and used a basic makefile. This section describes makefiles in more detail.

A CM3-IDE makefile (named **m3makefile**) is a small script which tells how to build a package. Most instructions in a makefile are calls to pre-defined functions of CM3-IDE's builder, such as "**import**", or "**program**". Together with the builder, these predefined functions replace the need for a "**make**" utility. Function calls replace declarations in the makefile, and the builder takes care of the dependencies between modules.

Working with Makefiles

To view a makefile, navigate to the Package Summary page, and click the makefile name under “Quake sources”.

To edit a makefile from any package, click the button labeled “Edit m3makefile”. CM3-IDE will start your text editor and open the makefile for the current package.

Here is an example of a simple CM3-IDE makefile:

```
% m3makefile for simplePkg
import("libm3")
import("ui")
module("Editwindow")
implementation("Editor")
program("editor")
```

A CM3-IDE makefile is a script in a simple programming language called *Quake*, used by both CM3-IDE and **cm3**. You can find more information about Quake in the CM3-IDE on-line help under `/help/cm3/quake.html`. Nonetheless, coding simple makefiles will not require much knowledge beyond what is described here.

3.3.1 Basic Makefile Commands

The following are the commands used most often in CM3-IDE makefiles.

Most makefiles start with one or more import commands:

```
import( "package-name" )
```

The import command specifies a package to be imported in the build process. Any package that builds a library may be imported.

Most programs import the standard Modula-3 library, called **libm3**, via the command:

```
import( "libm3" )
```

Declaring Sources. The following commands declare the source files in your package that are to be included in the build:

```
interface( "X" )
```

Declares that the file **X.i3** contains an interface. All interface files that you want to include in your build must appear in interface commands. Don't forget to leave off the **".i3"** extensions.

```
implementation( "X" )
```

Declares that the file **X.m3** contains a module. All module files that you want to include in your build must appear in an implementation command.

```
module( "X" )
```

Declares **X.i3** and **X.m3** with one command. The **module** command is really a short-hand for doing both an **interface** and an **implementation**. This command is used most often, as many CM3-IDE modules consist of a single interface and implementation.

```
generic_interface( "X" )
generic_implementation( "X" )
generic_module( "X" )
```

Similar to their non-generic counterparts, the **generic_interface**, and **generic_implementation** commands declare the generic interface and implementation files to be included in the build. The command **generic_module** is a shorthand for calling **generic_interface** and **generic_implementation**. Generic interface and module files use the **".ig"** and **".mg"** extensions. See **Generics: Reusable Data Structures and Algorithms** on page 97 to learn more about generics.

Making sources visible to others. The above calls declare their arguments to be built, that is, but visible only within the current package. To declare an interface so that other team members can access it, you use the capitalized version of the same command, for example:

```
Interface("X")
```

and

```
Module ("X").
```

Programs and Libraries. To tell the builder to build an executable, include the **program** command at the end of your makefile:

```
program( "executable-name" )
```


A makefile may have only one **program** command. It specifies what to name the program executable. Use the capitalized version of this command, **Program**, to make the program available to other developers.

If the makefile is describing a collection of interfaces and modules that are designed to be a library to be used by other packages, use **library** instead of **program**:

```
library ( "library-name" )
```

3.3.2 Additional Makefile Commands

Many standard packages in your CM3-IDE distribution define new makefile commands. Importing these packages makes the new commands available. For example, the standard library **libm3** includes a makefile command **bundle** which will bundle a file in your source directory so that it's available at run-time.

For reference information about makefile commands and their syntax, see CM3-IDE's on-line help under `/help/cm3/cm3.help`.

3.4 Managing Multiple Packages

Building a large and complex project as one package is certainly possible but probably not wise. Even if you are programming on a project by yourself, you may want to divide your project into more manageable pieces.

Suppose you are building an editor library and a number of editor programs which, using the editor library, support editing of various file formats.

A natural division of your code would be to put the core editor functionality in one library package (called **libedit** in this example), and put each editor incarnation (called **html-editor** in this example) in its own program package. To create an editor program, you import the **libedit** library and add some additional formatting code and a main module. Building such a package results in an editor program.

The makefile for the **libedit** package would look like:

```
% makefile for edit library
Module("Edit")
...other makefile statements...
Interface("Format")
Library("libedit")
```

The commands **Module**, **Interface**, and **Library** are capitalized to denote that the corresponding interfaces, and the library should be made available to other packages.

The makefile for the `html-editor` package would look like:


```
% makefile for html-editor
import("libm3")
import("libedit")
module("HTMLFormat")
implementation("HTMLEditor")
program("html-edit")
```

How do you make the `libedit` library available to `html-editor`'s build? The simplest way to make the functionality of a package available to other packages is the `ship` command. To make the library package `libedit` available for reuse in the program package `html-editor`, you need to ship the `libedit` library, first.

After you've successfully built the `libedit` package, you may ship it by clicking on the  Ship button on its package summary page.

After shipping the `libedit` library, you can build the `html-editor` package, which depends on the `libedit` package.

3.5 Shipping Packages

Shipping a package makes the contents of the package available to other packages in your system. The  Ship button, located above the Build button on any package page ships the current package.

Shipping does not modify the private copy of your package. It simply copies the essential parts into a public version of the package.

You may continue working on your private copy, and ship another version at your convenience.

Once shipped, CM3-IDE will keep track of two copies of your package:

- a private copy, which you just built. You can continue to change and build this copy without affecting other packages. In the default settings, this copy would reside under `/proj` in CM3-IDE's namespace.
- a public copy of your package which is available to other packages. In the default settings, this copy would reside under `/public` in CM3-IDE's namespace.



Shipping Packages

Shipping a package makes its contents available to other packages in your system. Once shipped, a package can be imported into other packages, and it's listed with the public packages.

To ship a package from CM3-IDE, you can click the Ship button. CM3-IDE copies the contents of the package to the public package root. From the command-line shell, you use the command “**cm3 -ship**” to ship packages.

3.6 Package Roots

So far, we have discussed two kinds of packages:

- your private packages, listed under “**proj**” packages in the CM3-IDE  Packages page. The two you created in **Learning the Basics** on page 5, **hello** and **MyPackage**, are examples of private packages. Their respective URLs are `/proj/hello` and `/proj/MyPackage`.
- public packages, listed under “**public**” packages in the CM3-IDE  Packages page. The standard library package, **libm3** which you have imported into your own packages is an example of a public package. Its URL is `/public/libm3`.

In CM3-IDE, packages are kept in directories called *package roots*. The package roots **proj** and **public** are examples. CM3-IDE is pre-configured to use **proj** for your private packages, and **public** for the public packages. Indeed, you can create new package roots to organize your projects, or coordinate sharing with others. For example, you may use a package root **graphics** to contain all the graphics-related packages in your development group.

Within CM3-IDE, the name of a package root resides at the top level of CM3-IDE's namespace. For example, the package root **graphics** would map to `/graphics`, and the **html-editor** package contained in the **graphics** root would map to `/graphics/html-editor`.

On your filesystem, a package root is represented as a directory that contains zero or more packages. CM3-IDE supports building or browsing packages in multiple package roots. You can add new package roots by using CM3-IDE's configuration screen. (See **Customizing CM3-IDE** on page 65.)

3.6.1 Example: Creating a New Package Root


In this example, we create a new package root and configure CM3-IDE to add the new package root to its database.

Step

Create a directory for a package root. To create and configure CM3-IDE to use a package root, you must first create a directory for the package root. To do so, create a directory on your filesystem, such as:

```
D:\users\harry\graphics    (Win32)
/usr/harry/graphics        (Unix)
```

Step

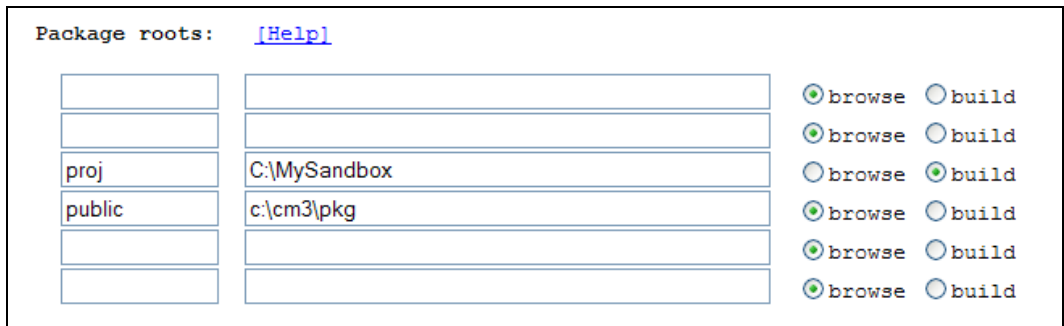
Navigate to the package roots settings. From the start page, click on the  Configuration icon to visit CM3-IDE's configuration screen. The second section of CM3-IDE Configuration is labeled "Package Roots." The Package Roots section of the configuration page is where you list the package roots in your system.

The package roots specified in this section are scanned periodically by CM3-IDE. Each root is either available for building or browsing. When you are sharing packages with others, you should use the "browse" option, so that you don't accidentally alter their code. You configure your own package roots to allow building.

By default, your CM3-IDE installation comes with two pre-defined roots:

- Your private packages, listed under "**proj**" packages in CM3-IDE, reside in the directory (**\$HOME/proj** on Unix, or **%HOME%\proj** on Win32).
- The public packages, listed under "**public**" packages in CM3-IDE, reside in the **pkg** subdirectory of your CM3-IDE installation.

When you ship a package, the contents of your package are copied to the public packages, making them available to other programmers in a controlled fashion.



Package roots: [Help]		
		<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build
proj	C:\MySandbox	<input type="radio"/> browse <input checked="" type="radio"/> build
public	c:\cm3\pkg	<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build

Figure 27. Package Roots Section of the Configuration Screen

Step

Back to the example. Next, choose one of the blank rows to specify your new package root. You need to specify three pieces of information about a root: its name in

CM3-IDE's namespace, its filesystem path, and whether it is to be used for building or browsing:

Step

- **Choose the name.**

Specify a short name for the new package root in the left-most field. You will use this short name to refer to this package root in CM3-IDE. Choose a short and descriptive name like “**graphics**” which will be known as **/graphics** within CM3-IDE. If the new root name collides with existing root names, CM3-IDE will substitute something less useful, like “**Root001**”.

Step

- **Specify the path to the package root.**

Specify the absolute path to the directory where this package root resides. CM3-IDE will periodically scan for packages from the path you specify. Examples are:

D:\USERS\HARRY\GRAPHICS	(Win32)
/usr/harry/graphics	(Unix)

Step

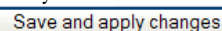
- **Enable either building or browsing for this package root.**

Check the option “build” to make packages in this root available for editing and building.

If you are configuring a new root to browse packages belonging to other developers, make sure to check the “browse” option so that you don't corrupt their packages inadvertently. (The public package root is an example of a browse-only package root, while your private package root, **proj**, allows building and editing of packages.)

Step

When you finish entering the information about the new root, click on

 Save and Apply Changes.

3.7 Sharing Packages

This section describes how to share a package with other developers and how to access its functionality.

Imagine that you are working on a large project as part of a team of programmers. You've been assigned to write a library that will be used by other programmers. To test and run their code, they need a stable copy of your library available to them at all times. What if you're still working on your code? How do you make sure they are using the most current version? How do you test and change your revised version of a library

while simultaneously allowing others to continue their work based on a stable release of your package?

The answer in CM3-IDE, as you might have guessed, is shipping. In CM3-IDE, you ship your packages to copy them to the public package root, where they are available to others in your team. Each CM3-IDE installation has one public package root. The code you ship becomes available for browsing and importing by others, but they may not edit or compile it.

In contrast, you, who shipped the package in the first place, are considered to be the owner of this package; you are responsible for its upkeep. The buildable sources for your packages remain in your private directory; they are in your complete control. In fact, shipping a package does not affect its contents in any way.

You may continue to work on your package after shipping it. Whenever you are comfortable with your changes, you may ship the package again, making the new changes available to others. CM3-IDE will overwrite the old shipped files in the public package root with the new ones.

3.7.1 Example: Adam's and Eve's Joint Project

In the following example, we examine a project that involves multiple programmers and multiple packages. We'll review some of the concepts discussed earlier in the context of a multi-developer project.

Imagine two developers Adam and Eve working on a shared project Garden. As smart developers, they have decided to use CM3-IDE for their Garden development. As organized developers, they first set up their environment to make sharing easy.

Creating directories for package roots. The first step is to create a directory that contains both of their package roots (presumably with group write permissions):

```
E:\GARDEN\      (Win32)
/proj/garden/    (Unix)
```

Next, they create package roots for themselves:

```
E:\GARDEN\EVE      (Win32)    /proj/garden/eve  (Unix)
E:\GARDEN\ADAM     (Win32)    /proj/garden/adam (Unix)
```

Packages in Eve's directory are her responsibility. Adam may be able to browse the code in Eve's packages, but he should not be able to modify the package contents. If Adam needs a change in one of Eve's packages, he should ask Eve to make the change.

Configuring package roots. Next, Adam and Eve use the configuration page of CM3-IDE to set up their package roots.

BUILDING AND SHARING PACKAGES

Eve types the package root name “eve”, with the path set to the full path to the directory she just created for her packages:

```
E:\GARDEN\EVE (win32)
/proj/garden/eve (Unix)
```

Eve checks the “build” option for her own package root, so that she can build new packages in her newly created package root.

Eve wants to be able to look at Adam’s packages but she wants to make sure she doesn’t accidentally do anything to them. So Eve includes Adam’s package root, but she is careful not to check the “build” option. The only other choice is the “browse” option.

Eve is ready to leave the Configuration page. The package root section of her configuration looks like:

Package roots: [Help]		
		<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build
proj	C:\MySandbox	<input type="radio"/> browse <input checked="" type="radio"/> build
public	c:\cm3\pkg	<input checked="" type="radio"/> browse <input type="radio"/> build
eve	C:\Garden\Eve	<input type="radio"/> browse <input checked="" type="radio"/> build
adam	C:\Garden\Adam	<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build
		<input checked="" type="radio"/> browse <input type="radio"/> build

Figure 28. The “Package Roots” section of Eve’s Configuration Page

Saving changes to the configuration. Before leaving the Configuration page, Eve clicks the Save and Apply button.

Adam sets up his package roots in a similar fashion, but in Adam’s configuration, Adam’s package root is available for building and Eve’s is only available for browsing.

Note that Adam and Eve could have chosen different names for their roots, but then communicating would have been harder.

Assigning project responsibilities. Now that they have organized their development environment, Adam and Eve meet to decide how to break up the work:

- Adam agrees to work on the end application, **pie**. The package **pie** will reside in Adam’s package root, and will be available in Adam’s and Eve’s CM3-IDE as **/adam/pie**.

- Eve agrees to work on the core library, named **apple**. The package **apple** will reside in Eve’s directory, and is available as `/eve/apple`.
- Adam’s package, **pie**, will need Eve’s package, **apple**.

Adam and Eve are the owners of their respective packages. No one else is allowed to modify sources in someone else’s package root. Working in a team however, each allows the other to browse the current state of their Garden-related packages. (CM3-IDE does not enforce this policy; you must use your filesystem to do that.)

Getting ready to code. Adam and Eve spend some time discussing the design of the application until they agree on an initial set of interfaces that Eve’s **apple** needs to support. The separation of interface from implementation in CM3-IDE is key to allowing Adam and Eve to work independently:

- Eve starts the first implementation of the **apple** interfaces.
- Adam starts the design and implementation of **pie**, assuming the agreed upon **apple** interfaces.

Shipping the first release. When Eve is satisfied with **apple**, she ships it. Now, there are two copies of the **apple** package, Eve’s private copy (`/eve/apple`) and the public copy (`/public/apple`). Once Eve ships her package:

- Adam can import **apple** in the makefile for **pie** and build it.
- Eve may continue working on **apple**, shipping it whenever she finds a proper checkpoint where the code is stable.

Testing before a release. If **apple** becomes too complex, Eve may need to “unit test” its functionality before shipping so that Adam’s **pie** is not affected by new bugs.

No problem: Eve can create her own package **juice** just for the purpose of testing the quality of **apple**. As a savvy developer, Eve also advertises **juice** as a sample program that uses **apple**. This is convenient for Adam since he can see the **juice** package as `/eve/juice` in his CM3-IDE’s namespace even if Eve doesn’t ship it. Moreover, because he configured **juice** for browsing only in his CM3-IDE configuration, Adam can’t corrupt **juice** by accident.

Eve is left with one problem: **juice** is supposed to test **apple** before **apple** is shipped, but since **juice** needs to import **apple**, **apple** needs to be shipped before **juice** can test it.

How does Eve overcome this problem? Simple: she needs to force **juice** to use her private version of **apple** (`/eve/apple`) instead of the publicly available version (`/public/apple`).

Overriding a build. When building **juice**, Eve will need to tell CM3-IDE not to look in the public package root for the shipped **apple**. Instead, she must specify where to get **apple**; this is called overriding.

To override a build, Eve must perform two steps:

First, she must create an overrides file (named “**m3overrides**”) in **juice**’s source directory that includes the names of the overridden packages and where to find them. In this example, Eve puts:

```
override("apple", "C:\\\\GARDEN\\EVE")      (Win32)
override("apple", "/proj/garden/eve")      (Unix)
```

in the **m3overrides** file.

Then, when building **juice**, Eve types “**-override**” as the build option to tell the builder that it should use the overrides file.

Finally, Eve compiles her **juice** package, tests **apple**, and when she is happy with the quality of **apple**, she ships it. Next time Adam builds **pie**, the builder will notice the fresh **apple** and will use it.

How Overrides Work. An overrides file contains a set of override commands to specify new paths for the builder to find packages. Each line of the override list contains the name of the package to replace and the path to the package root containing the new package, i.e.:

```
override(package, replacement-package-root)
```

where **package** and **replacement-package-root** are strings.

Note that you must escape the backslash character on Win32 by typing “\\” as your path delimiter.

When the **-override** option is specified, **cm3** looks for a file named **m3overrides** in the package’s source directory. If the file **m3overrides** exists, it is evaluated prior to evaluating **m3makefile**. (Both **m3overrides** and **m3makefile** are Quake scripts.)

CM3-IDE allows you to leave permanent overrides in your makefiles but it isn’t a good practice. By keeping all override calls in an **m3overrides** file and not in a makefile, you can readily switch between building packages based on private and public versions of imported packages without editing files.

The overrides in effect when a package was built are automatically carried forward into importers of the package, so there is no need to restate the complete set of overrides in

every package, only of those packages that are directly imported into the current package. CM3-IDE's builder will warn you if you overspecify your override options.

Shipping and the -override Option. Shipping a package that is built with overrides makes little sense, as it depends on packages that are not available in the public package root. CM3-IDE's builder will refuse to ship a package that was built using overrides. This safety check helps ensure that packages shipped to the public package root stay consistent.

3.8 Builder Options

CM3-IDE's builder options are listed here. You can find this information on-line by specifying “-help” as your build option, or typing “cm3 -help” at a shell command-line.

Modes.

-build	compile and link
-ship	install package
-clean	delete derived files
-find	locate source files
(default: -build)	

Compiler Options.

-g	produce symbol table information for the debugger
-O	optimize code
-A	disable code generation for assertions
-once	don't recompile to improve opaque object code
-w0..-w3	limit compiler warning messages
-Z	generate coverage analysis code
(default: -g -w1)	

Program and Library Options.

-C	compile only, produce no program or library
-a lib	build library lib
-o pgm	build program pgm
-skiplink	skip the final link step
(default: -o prog)	

Messages.

-silent	produce no diagnostic output
-why	explain why code is being recompiled
-commands	list system commands as they are performed
-verbose	list internal steps as they are performed
-debug	dump internal debugging information
(default: -why)	

Information and Help.

-help	print this help message
-?	print this help message
-version	print the version number header
-config	print the version number header

Miscellaneous.

-keep	preserve intermediate and temporary files
-times	produce a dump of elapsed times
-override	include the “.m3overrides” file
-x	include the “.m3overrides” file
-Dnm	define the quake variable nm with the value TRUE
-Dnm=val	define the quake variable nm with the value val
-console	produce a Windows CONSOLE subsystem program
-gui	produce a Windows GUI subsystem program
-windows	produce a Windows GUI subsystem program

CM3-IDE’s makefiles are discussed in more detail later in **CM3-IDE Makefiles** on page 50. For information about customizing the behavior of the Build button, see **Customizing CM3-IDE** on page 65, or review the configuration file **cm3.cfg** in your installation.

3.9 Summary

A *package* is the unit of building and shipping in CM3-IDE. Each package is represented by a directory on the filesystem. Packages typically have two subdirectories, a *source directory* (named “**src**”) and a *derived directory* whose name varies per platform. Each package may have a *makefile* in its source directory which is a set of instructions for compiling sources in the package.

To *build* a package, use the Build button on the summary page of the package.

To make a package available for importing by other packages, *ship* it, by clicking the Ship button on the summary page of the package.

CM3-IDE builder. The stand-alone program **cm3** is CM3-IDE’s builder. You may start the **cm3** builder by either clicking the Build button from any package page, or by invoking **cm3** from the command line. See **Builder Options** on page 62.

Package roots. You can organize packages for your various projects into package roots. A package root is a directory that can contain zero or more packages. See **Package Roots** on page 55.

The public package root is CM3-IDE’s shared repository for shipped packages. Packages in the public package root can be browsed and imported by anyone but they cannot be edited or compiled in-place.

Overriding. The **-override** option can be used to tell **cm3** to look for a file named **m3overrides** in the source directory and, if it exists, evaluate it immediately before evaluating **m3makefile**. See **How Overrides Work** on page 61.

4. Customizing CM3-IDE

Read this chapter to learn how to customize CM3-IDE.

This chapter contains information about CM3-IDE's configuration page.

This chapter describes how to customize CM3-IDE. The configuration screen is like a control panel for CM3-IDE—it controls many aspects of CM3-IDE's behavior. CM3-IDE's configuration settings are persistent; the configuration information is saved across sessions.

This chapter does not cover configuration of CM3-IDE's builder program, **cm3**. The builder can be invoked from outside CM3-IDE and its configuration settings may be modified independently of CM3-IDE's. See the **cm3.cfg** file in your CM3-IDE installation to configure the behavior of your CM3-IDE builder. To learn more details about configuring the builder, see **Building and Sharing Packages** on page 47.

Since changing your configuration settings has a permanent effect on your development environment, you should apply care when changing configuration parameters.

Navigating to the Configuration Page on page 66 points you to the CM3-IDE configuration page.

Saving Configuration Changes on page 66 shows you how to apply and save changes in your configuration.

Display Settings on page 66 controls the display aspects of CM3-IDE— start screen, and layout of dynamically configured pages.

Package Roots Settings on page 68 allows you to add, delete, or modify package roots.

Communication Settings on page 69 specifies the settings used by CM3-IDE to communicate with your web browser.

Miscellaneous Settings on page 69 allows you to change the number of threads for the CM3-IDE server and verbosity of messages.

Helper Procedures on page 70 tell CM3-IDE how to build, ship, or clean packages, run programs, or edit files.



4.1 CM3-IDE Configuration Screen

From time to time, you may need to change the configuration of your CM3-IDE development environment. For example, you may like to change the layout of lists in CM3-IDE to match your window settings, or you may want to share packages with a co-worker. For these kinds of tasks, you may alter CM3-IDE's behavior from the Configuration page.

4.1.1 Navigating to the Configuration Page

From the CM3-IDE start page, follow the  Configuration link in the System group to navigate to the configuration page.

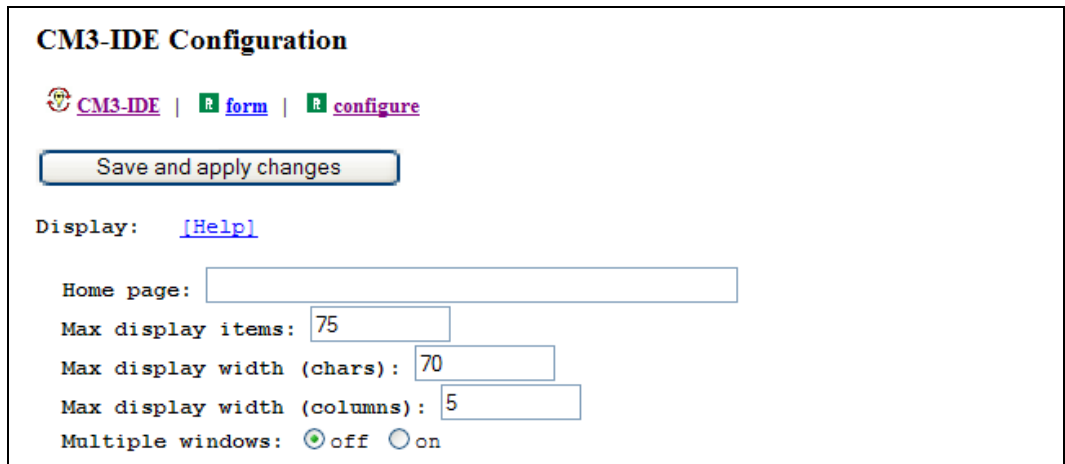
4.1.2 Saving Configuration Changes

At the top and bottom of this page are the Save and Apply Changes buttons (Figure 29). To save changes to your CM3-IDE settings, click on this button.


Important Note. Your personal configuration is kept in:

```
%HOME%\proj\CM3-IDE.cfg (Win32)
$HOME/proj/CM3-IDE.cfg (Unix)
```

If the HOME environment variable is not set when CM3-IDE was started, CM3-IDE won't be able to save your changes to the configuration. Otherwise, changes to your configuration will persist across sessions.



CM3-IDE Configuration

 [CM3-IDE](#) | [form](#) | [configure](#)

[Save and apply changes](#)

Display: [\[Help\]](#)

Home page:

Max display items:

Max display width (chars):

Max display width (columns):

Multiple windows: ☒ off ☐ on

Figure 29. Save Changes and Display Settings

4.1.3 Display Settings

You can control many aspects of the CM3-IDE screen layout (Figure 29). CM3-IDE's display settings, in conjunction with the display settings of your web browser will allow you to make your navigation in CM3-IDE more comfortable.

Start Page. The start page setting specifies the full path for a file containing the HTML that CM3-IDE displays initially. If a file is specified, CM3-IDE will use it as the start page; otherwise, CM3-IDE will use its own default start page.

When a start page is specified, its parent directory is available at the URL `/user` within your CM3-IDE's namespace, so you may refer to entries in `/user` in your CM3-IDE pages.

For examples of a start page, use your browser's "Save As" command (usually under the File menu) to save a copy of the default start page to your filesystem. You may then modify the file, and point CM3-IDE to the new version of the start page. Another good beginning for a start page is the  Getting Started page, located at the URL `/help/getting-started.html`.

See **CM3-IDE's Web Namespace** on page 42 for more information regarding the specification of URLs in your HTML files.

The rest of the display settings control how CM3-IDE displays lists of items. If you're using a small font or a large screen, you may want to adjust these values to better suit your preferences.

Max display items. Defines the maximum number of items for a list displayed on one page. For lists more than the maximum number of items, CM3-IDE will coalesce entries with common prefixes and suffixes, in an effort to fit the list to the screen size.

Max display width (chars). Defines the maximum number of characters that CM3-IDE will put into a single line of a dynamically generated list. CM3-IDE's lists are usually displayed in fixed-width fonts.

Max display width (columns). Defines the maximum number of columns that CM3-IDE will put into a single line of a dynamically generated list.

Multiple windows. Controls whether CM3-IDE will use non-standard window targeting to direct its output to multiple browser windows (*This is an experimental feature in release 4.1 of CM3-IDE.*)

CM3-IDE | [form](#) | [configure](#)

[Save and apply changes](#)

Display: [\[Help\]](#)

Home page:

Max display items:

Max display width (chars):

Max display width (columns):

Multiple windows: ☒ off ☐ on

Package roots: [\[Help\]](#)

<input type="text"/>	<input type="text"/>	<input checked="" type="radio"/> browse <input type="radio"/> build
<input type="text"/>	<input type="text"/>	<input checked="" type="radio"/> browse <input type="radio"/> build
proj	C:\MySandbox	<input type="radio"/> browse <input checked="" type="radio"/> build
public	c:\cm3\pkg	<input checked="" type="radio"/> browse <input type="radio"/> build
eve	C:\Garden\Eve	<input type="radio"/> browse <input checked="" type="radio"/> build
adam	C:\Garden\Adam	<input checked="" type="radio"/> browse <input type="radio"/> build
<input type="text"/>	<input type="text"/>	<input checked="" type="radio"/> browse <input type="radio"/> build
<input type="text"/>	<input type="text"/>	<input checked="" type="radio"/> browse <input type="radio"/> build

Communication: [\[Help\]](#)

Host name:

IP address:

Server port:

Figure 30. Package Root and Communication Configuration

4.1.4 Package Roots Settings

CM3-IDE supports browsing and building packages from multiple package roots. Here, you can specify the list of package roots used by CM3-IDE, and their characteristics (Figure 30). For setting each package root, you must specify the following:

- a short name that will be used by CM3-IDE to form URLs. The package roots map to the top of the URL hierarchy, i.e., the package root named **graphics** will map to **/graphics** within CM3-IDE.
- a full path in the file system where the package root resides
- a boolean specifying whether the user is allowed to build packages filed under this root

The default settings map **/proj** to your private packages and **/public** to the public packages in you system.

See **Building and Sharing Packages** on page 47 for more information about package roots.

4.1.5 Communication Settings

When started, CM3-IDE determines the local IP address and name of the machine it's running on, and uses this information for communication with your web browser. If networking is not installed properly, or you are accessing CM3-IDE remotely, you may need to override the default communication configuration (Figure 30).

Host name. Defines the name of the machine running CM3-IDE. The URLs generated by CM3-IDE will contain this name and the browsers attached to CM3-IDE will need the name. The default host name is **localhost**, referring to the host where the CM3-IDE is executing. This value should work on most platforms that support TCP/IP, even those that only have intermittent SLIP or PPP connections. If no value is specified for the host name, CM3-IDE attempts a reverse name server lookup using the host's IP address.

IP address. Defines the IP address of the machine running CM3-IDE. You should not need to explicitly define your IP address unless your networking installation is badly broken, or if you are trying to cross DNS naming boundaries. In case you need it, the IP address bound to **localhost** is usually **127.0.0.1**.

Server port. Defines the TCP port number that CM3-IDE will use. You may need to change CM3-IDE's default value if it conflicts with one of the TCP services already running on your machine. Usually, using port numbers below **1024** requires special privileges. Ports **80** and **8080** are often used by regular web servers.

4.1.6 Miscellaneous Settings

Verbose log. Determines how much information is generated in CM3-IDE's console log. It's best to leave the verbose log off, unless you are trying to track down a problem with CM3-IDE. CM3-IDE's logged messages display on your console window. The last 500 lines of the log are available at the URL **"/log"**.

Server threads. Determines the number of concurrent threads within CM3-IDE that can service HTTP requests. If CM3-IDE becomes sluggish because there are too many users contending for a limited number of server threads, it may help to increase the number of threads. However, it is more likely that CM3-IDE just needs to run on a machine with more memory.

Refresh interval. Determines the number of minutes for CM3-IDE to wait between full rescans of the package roots. Setting too small a value can overload the file system and degrade performance.

Misc: [\[Help\]](#)

Verbose log: ☒ off ☐ on

Automatic package scans: ☐ off ☒ on

Server threads:

Refresh interval (minutes):

CM3-IDE URL: <http://localhost:3800/>

System package root: c:\cm3\pkg

Build directory: NT386

Helper procedures: [\[Help\]](#)

Browser:

```
proc start_browser (initial_url) is
  cm3_exec ("start /wait C:\\progra~1\\intern~1\\iexplore.exe",
    initial_url)
  return TRUE &==> server terminates when browser terminates
end
```

Build:

```
proc build_package (pkg, options) is
  cm3_exec ("cd", pkg, "&& cm3", options)
end
```

Ship:

```
proc ship_package (pkg) is
  cm3_exec ("cd", pkg, "&& cm3 -ship")
end
```

Clean:

```
proc clean_package (pkg) is
  cm3_exec ("cd", pkg, "&& cm3 -clean")
end
```

Run:

```
proc run_program (dir, cmd) is
  cm3_exec ("cd", dir, "&& ", cmd)
end
```

Edit:

```
proc edit_file (file, line) is
  cm3_exec ("start C:\\Progra~1\\Window~1\\Accessories\\wordpad.exe",
    file)
end
```

Figure 31. Miscellaneous Settings and Helper Procedures

4.1.7 Helper Procedures

CM3-IDE uses a number of small helper procedures to interact with its external environment (Figure 31). These procedures are written in Quake, a small interpreted language. Modifying helper procedures should be easy and the default helper

procedures are quite short. CM3-IDE's use of these interpreted procedures maximizes your ability to configure and control CM3-IDE's behavior. For more information on Quake see the CM3-IDE on-line help under `/help/cm3/quake.html`.

Helper Procedure for Starting the Browser. When CM3-IDE starts, it calls the `start_browser` helper procedure, passing it the URL of the start screen. If `start_browser` returns FALSE, the server continues running; otherwise, the server terminates when the function returns. The default implementation starts your web browser automatically.

Helper Procedures for Build, Ship, Clean. The helper procedures `build_package`, `ship_package`, and `clean_package` are used to build, ship, and clean a package. They are called when the corresponding action buttons are pressed. The default versions of these procedures call the CM3-IDE builder (`cm3`) after changing the working directory to the package.

Helper Procedure for Run. CM3-IDE calls `run_program` to execute the program whose location is passed as a parameter. You may use this procedure to start debuggers or shell windows (via “`xterm -e`” command on Unix, or “`start`” command on Windows). The program summary page allows you specify the arguments passed to `run_program`.

Helper Procedure for Edit. CM3-IDE calls `edit_file(file,line)` to edit `file` initially positioned at `line`. The default procedure assumes that the editor supports a “`+line`” option common to most programmer's editors. If your editor doesn't support line placement, simply delete the portion of the default procedure that passes the `+line` option in the `exec` call.

4.2 Summary

CM3-IDE's Configuration. The CM3-IDE configuration screen acts as a central control panel for the CM3-IDE environment. You can change many of CM3-IDE's settings to alter its default behavior. To navigate to the configuration page, follow the Configuration link from the start page or go to `/form/configure` within CM3-IDE's namespace.

Save and Apply Changes. You must click on the “Save and Apply Changes” button before your changes to the configuration page can be applied. Once applied, changes persist across CM3-IDE sessions.

Display settings. Change CM3-IDE's start screen or alter the layout of dynamically generated pages by modifying the display settings.

Package Roots Settings. Add or delete package roots from your environment using this setting. See also **Building and Sharing Packages** on page 47.

Communication Settings. Different network environments require different communication settings. You may need to change these settings if you have specialized communication needs, for example, to run without networking or on a remote machine. CM3-IDE uses the host name “localhost” and the port number 3800 by default.

Miscellaneous Settings. Change settings for the number of threads used in CM3-IDE's web server, the verbosity of CM3-IDE's logged messages, and the refresh interval for background scanning.

Helper Procedures. Use the helper procedures to modify how CM3-IDE builds, ships, and cleans packages. You can also change how your editor is invoked, and how CM3-IDE runs programs.

Customizing CM3-IDE's Builder(cm3). To customize the behavior of CM3-IDE's builder (cm3), review and modify the `cm3.cfg` file in your installation of CM3-IDE. The file includes in-line comments regarding the significance of various settings, and should be easy to modify. The `cm3.cfg` configuration file usually resides in the same directory as the `cm3` executable in your file system. To find out for sure, invoke the builder from a shell command line as “`cm3 -config`”.


5. Beyond The Basics

Read this chapter to learn about advanced language concepts.

This chapter introduces more advanced language concepts as a starting point for your exploration of system and application programming with CM3-IDE.

This chapter treats the language concepts pedagogically; each concept is presented informally and is illustrated through a complete example program. The description of concepts and features in this chapter is incomplete; for complete and precise definitions of language features, see the *Language Reference*.



This chapter is divided into six parts. Each part describes a distinct feature of the language, and demonstrates the feature in a complete program. You can find the sources for the programs in this chapter in the  Examples section of your CM3-IDE environment.

Exceptions: Error Handling in CM3-IDE on page 74 illustrates the use of exceptions for building robust programs.

Object Types: Object Oriented Programming on page 81 showcases basic object-oriented programming in a simple program.

Threads: Managing Concurrent Activities on page 86 describes how to use threads to manage concurrent activities.

Opaque Types: Information Hiding and Encapsulation on page 88 demonstrates the use of opaque types to hide the implementation of a type from its clients. The section continues with partially opaque types, which can be used to reveal partial information about objects to select clients.

Generics: Reusable Data Structures and Algorithms on page 97 outlines the use of parametrized interfaces for creating polymorphic data structures and algorithms.

Unsafe Constructs: System Programming in CM3-IDE on page 102 introduces you to the unsafe subset of the language.

5.1 Exceptions: Error Handling in CM3-IDE

Error handling in CM3-IDE is done with language-level exceptions. Exceptions help separate error handling logic from the main code.

If you want your program to remain vigilant at all times for errors where exceptions are not available, you need to explicitly check for failure at every function call in your program. Due to the usual pressures of software development, returning or checking error codes for failure is seldom done thoroughly. Careful programmers who check for every error condition often design elaborate arrangements of if-then statements, or non-standard “**setjmp**” calls. Most of these methods make programming much more difficult for the careful programmer, hence encouraging sloppy treatment of errors.

Error-checking with exceptions enables the programmer to easily, and reliably handle all error conditions in their programs.

5.1.1 How Exceptions Work

In CM3-IDE, when a program encounters a situation deemed abnormal by the programmer, the runtime generates an exception, and begins to look for a handler for the exception to handle the abnormal condition.

If an exception is not handled in the current procedure, the calling procedure is searched. If no handler for that particular exception is found there, the runtime will continue following the chain of called procedures until it finds a procedure that handles that particular exception. If no handler is found, the computation terminates, for example, by entering the debugger.

Hence, it is possible to handle errors anywhere in a chain of procedure calls, without having to continually check error codes.

For a precise definition of the semantics of exceptions, see the *Language Reference*. Here we describe the syntax for raising and handling exceptions, and illustrate the use of exceptions in a simple program.

5.1.2 Declaring Exceptions

All exceptions must be declared at the top-level of an interface or a module. An exception may include a parameter.

```
EXCEPTION exception-name [ "(" exception-parameter ")" ] ;
```

In the following example, **DriveNotReady** is a parameter-less exception and **ReadError** is an exception that takes a **TEXT** parameter:

```

INTERFACE CDROM;

EXCEPTION DriveNotReady;
EXCEPTION ReadError(TEXT);

PROCEDURE Read(sector: INTEGER):TEXT
    RAISES {DriveNotReady, ReadError};
END CDROM.

```

5.1.3 Triggering Exceptions: RAISE Statement

You can trigger the handling of an exception through the use of the **RAISE** statement. The statement:

```
RAISE exception [ "(" exception-parameter ")" ];
```

raises an exception, passing control of the program to the innermost exception handler for that exception. Use the **RAISE** statement to raise an exception. If the exception is defined with a parameter, one must be supplied when the exception is raised.

5.1.4 Handling Exceptions: TRY-EXCEPT Statement

```

TRY
    guarded-statements
EXCEPT
    "|" exception-name { ",", exception-name ... } "=>" action-statement
    "|" exception-name "(" parameter-name ")" "=>" action-statement
[ ELSE statements ]
END

```

The **TRY-EXCEPT** statement guards statements between **TRY** and **EXCEPT** with the exception handlers between **EXCEPT** and **END**.

An exception raised by a **guarded-statement** is handled by the **action-statement** which has a matching handler for the exception, or by the **ELSE** clause, if present. If an exception is caught, execution continues with the statement following the **END**, otherwise the exception is passed on to the enclosing scope.

Example:

```

EXCEPTION Failure(Severity);
TYPE Severity = {Low, Medium, High};
...
TRY
    ...Code that may raise Failure, IO.Error, or Lex.Error ...
EXCEPT
    | IO.Error    => IO.Put("An I/O error occurred.")
    | Lex.Error   => IO.Put("Unable to convert datatype.")
    | Failure(x) => IF x = Severity.Low
                     THEN IO.Put("Not bad")
                     ELSE IO.Put("Bail out")
END
END;

```

Important Note. The language defines **RETURN** and **EXIT** in terms of exceptions, hence the **ELSE** clause of a **TRY-EXCEPT** statement may catch a **RETURN** or an **EXIT**. You should refrain from using the **ELSE** clause of a **TRY-EXCEPT** statement whenever possible.

5.1.5 Cleaning up: TRY-FINALLY Statement

The **TRY-FINALLY** statement is typically used for clean-up activities.

```

TRY
    guarded-statements
FINALLY
    cleanup-statements
END

```

TRY-FINALLY guarantees that **cleanup-statements** are called no matter what happens in the **guarded-statements**. If one of **guarded-statements** raises an exception, the same exception is re-raised by **TRY-FINALLY** after the **cleanup-statements** are executed.

TRY-FINALLY is useful for clean-up activities, like closing file handles, even when the I/O operations may fail:

```

rd := IO.OpenRead("myfile");
TRY
    WHILE NOT EOF(rd) DO
        IO.PutLine (IO.GetLine(rd));
    END;
FINALLY
    Rd.Close(rd);
END;

```

Important Note. As **RETURN** and **EXIT** semantics are defined in terms of exceptions, **TRY-FINALLY** will not only act upon an ordinary exception, but it also acts upon a **RETURN** or an **EXIT**. This is often handy for adding wrappers to a block of code to print debugging information no matter how the block of code behaves.

5.1.6 Trapping All Exits from a Block of Code

For example, starting with a procedure with multiple return paths:

```
PROCEDURE SomeProc(): BOOLEAN RAISES {Invalid} =
BEGIN
  FOR i := 1 TO 10 DO
    CASE option[i] OF
      | 'a'..'z' => RETURN TRUE;
      | '1'..'9' => EXIT;
      ELSE RAISE Invalid;
    END;
  RETURN FALSE;
END SomeProc;
```

Suppose you would like to print a message every time a procedure exits, no matter how it exits: It is easy to add a **TRY-FINALLY** statement to catch every exit from the procedure. Simply add a **TRY-FINALLY** around the procedure's body; the **FINALLY** clause will be called no matter how the program exits **SomeProc**'s scope.

5.1.7 An Example of Exception Handling

The next two programs illustrate how to use exceptions to make your programs more robust against failures. The first one is a simple file copy program that does not deal with exceptions. The second incarnation catches exceptions, and hence is more robust.

5.1.8 Programming without Exceptions

Here we review the implementation of the **Copy** program. This program may crash at run-time due to uncaught exceptions.

Indeed, if you compile the sources for this version of the copy program, you will notice a number of warnings regarding possible exception failures at run-time. It is easy to cause a run-time crash: simply attempt to copy a non-existent file. The compiler warnings notify you about possible run-time failures at *compile-time*. If you fix all the exception-related warnings in all the sources of a program, your program will never crash from an unhandled exception.

Let's start with the main module, named **Copy**. The **Copy** program is simple:

- Make sure that the user has specified arguments correctly.
- If parameters are wrong, return an error code and exit.
- Otherwise, pass them along to **FakeOS.Copy**.

BEYOND THE BASICS

```
MODULE Copy EXPORTS Main;
  IMPORT FakeOS, Params, Process, IO;
BEGIN
  IF Params.Count # 3 THEN
    IO.Put ("Syntax: copy <source> <destination>\n");
    Process.Exit (2);
  END;
  WITH source = Params.Get(1) DO
    WITH destination = Params.Get(2) DO
      FakeOS.Copy (source, destination);
    END;
  END;
END Copy.
```

As a user-defined interface, **FakeOS** provides access to the **FakeOS** module. **Copy** a file named **source** to a file named **destination**.

```
INTERFACE FakeOS;
  PROCEDURE Copy(source, destination: TEXT);
END FakeOS.
```

The **FakeOS** module supplies the body of the **Copy** procedure. The **Copy** procedure creates new reader and writer streams from the input and output files, reads the contents from the input, and writes it to the output.

The procedures **FileRd.Open** and **FileWr.Open** are used for reading and writing files, **Rd.GetText** and **Wr.PutText** for input and output, and **Wr.Close** and **Rd.Close** to close the I/O streams.

FakeOS.Copy does the following:

- Open a reader and a writer to the source and destination
- Read the contents of the reader
- Write the contents into the writer
- Flush and close the reader and the writer

```

MODULE FakeOS;
  IMPORT Rd, Wr, FileRd, FileWr;
  PROCEDURE Copy(src, dest: TEXT) =
    VAR rd: Rd.T; wr: Wr.T;
  BEGIN
    rd := FileRd.Open (src);
    wr := FileWr.Open (dest);
    WITH contents = Rd.GetText (rd, LAST(INTEGER)) DO
      Wr.PutText (wr, contents);
    END;
    Rd.Close (rd);
    Wr.Close(wr);
  END Copy;

BEGIN
END FakeOS.

```

Note that **FakeOS.Copy** does not handle any exceptions raised by **FileRd.Open**, **FileWr.Open**, **Rd.GetText** or **Wr.PutText**. You may review the definition of these procedures in the standard libraries to find out about the exceptions they may raise. You can easily create a situation where an uncaught exception, causes a run-time crash.

5.1.9 Making Programs Robust with Exceptions

The previous version of **FakeOS.Copy** has a serious problem: it crashes when any I/O exception (e.g., disk full, no permission to write) is raised. The new version of the **FakeOS** interface and implementation illustrates how to use exceptions to build robust programs. The main module **Copy** still calls the **FakeOS** interface. The required changes are:

1. Add an **Error** exception with a **TEXT** parameter to the **FakeOS** interface.
2. Change **FakeOS.Copy** to raise **Error** when there is a problem, and include a text string describing the problem.
3. Modify the **Copy** module to handle this exception by printing an error message for the user.

```

INTERFACE FakeOS;
EXCEPTION Error (TEXT);
PROCEDURE Copy(source, destination: TEXT) RAISES {Error};
END FakeOS.

```

Here is the implementation of the new **FakeOS**:

```
MODULE FakeOS;
IMPORT Rd, Wr, FileRd, FileWr;
IMPORT Thread, OSError;
```

FakeOS works similarly to the last version, but this time it catches exceptions using the **TRY-EXCEPT** clause. For each exception that is raised by the called procedures, we propagate an **Error** exception to the caller of **FakeOS.Copy**.

```
PROCEDURE Copy(src, dest: TEXT) RAISES {Error} =
VAR
  rd: Rd.T;
  wr: Wr.T;
  <* FATAL Thread.Alerted *>
BEGIN
  TRY
    rd := FileRd.Open (src);
    wr := FileWr.Open (dest);
    WITH contents = Rd.GetText (rd, LAST(INTEGER)) DO
      Wr.PutText (wr, contents);
    END;
    Rd.Close (rd);
    Wr.Close(wr);
  EXCEPT
    | Rd.Failure =>
      RAISE Error ("reading from " & src & " failed");
    | Wr.Failure =>
      RAISE Error ("writing to " & dest & " failed");
    | OSError.E =>
      RAISE Error ("a system problem occurred");
  END
END Copy;

BEGIN
END FakeOS.
```

Note that the exception **Thread.Alerted** is marked as fatal, because we didn't want to handle it. The **FATAL** pragma lets the programmer tell the compiler that a particular exception is intentionally not being handled. You should employ the **FATAL** pragma carefully; excessive use of the **FATAL** pragma results in crash-prone code. If **FakeOS.Copy** was to deal with multiple threads, it should deal with **Thread.Alerted** properly.

What happens to the **Close** statements if there is an exception raised while the files are open? Yes, that's a problem—if an exception occurs while copying, the files may be left open—and you can use a **TRY-FINALLY** statement to deal with it. Review the language reference or tutorial to learn about **TRY-FINALLY**.

While these kinds of issues are usually not important in short-lived programs, they are very important for long-lived, multi-threaded applications (for example, a network object server) where resource management is critical.

This program is now robust against various system exceptions raised by calls in **FakeOS** or **Copy** modules. If the program hadn't been handling a particular exception, you would have seen a warning at compile-time. This same program, without source modifications, will work without silent or unexpected errors due to system exceptions on all supported operating systems.

5.2 Object Types: Object-oriented Programming

This section assumes that you are familiar with the concepts of object-oriented programming.

An object associates some *state* with some *behavior*. A Modula-3 object is a record—its state—paired with a method suite—its behavior.

```

TYPE
  AnObjectType = [ parent-object-type ] OBJECT
    object-fields
    [ METHODS methods ]
    [ OVERRIDES overrides ]
  END

```

An object is a record paired with a *method suite*, a collection of procedures that operate on the object. The fields of an object are specified just like those of records. Methods look like fields that hold procedure values, with the following syntax:

```

methods = { method ";" ... }
method = identifier signature [ "!=" procedure-name ]

```

A *method signature* is similar to a procedure signature (See the section on procedure declarations in the *Language Reference*). If a method declaration includes "**!=**", the associated procedure must accept objects of this type as its first parameter; the rest of the parameters must match the signature. The first parameter is often referred to as the *self* parameter.

Overrides specify new implementations for methods declared by an ancestor object-type: (An ancestor is either the parent of this type, or its parent's parent, or ...)

```

overrides = { override ";" ... }
override = method-name "!=" procedure-name

```

Object types form a single-inheritance hierarchy. The type **ROOT** is the supertype of all object types. Objects are traced references by default, hence they are garbage-collected by default.

Example. Let's create an object type **Polygon** which contains an open array of coordinates. We also define an initialization method **init**, and a verification method **verify**, for all objects of this type. Subtypes of **Polygon** may override the **init** method, and must override the **verify** method.

BEYOND THE BASICS

```
TYPE
  Polygon = OBJECT
    coords: REF ARRAY OF Point.T;
  METHODS
    init( READONLY p: ARRAY OF Point.T): Polygon := Init;
    verify() := NIL; (* To be overridden by subclasses. *)
  END;

PROCEDURE Init (self: Polygon;
                READONLY p: ARRAY OF Point.T) =
BEGIN
  self.coords := NEW(NUMBER(p));
  self.coords^ := p;
  self.verify(); (* Verify that initialization was proper. *)
  RETURN self;
END;
```

The subtype **Drawable** adds the **draw** method and assigns the **Draw** procedure as the default implementation for the **draw** method.

```
TYPE
  Drawable = Polygon OBJECT METHODS
    draw() := Draw;
  END;

PROCEDURE Draw (self: Drawable) =
BEGIN
  WITH p = self.coords^ DO
    FOR i = FIRST(p) TO LAST(p) DO
      DrawLine(p[i], p[(i+1) MOD NUMBER (p)])
    END;
  END;
END Draw.
```

Type **Rectangle** is a concrete implementation of an object. It will override the **verify** method to make sure there are four sides to this polygon and that the sides have the right properties.

```
TYPE
  Rectangle = Drawable OBJECT METHODS
    OVERRIDES
      verify := Verify;
  END;

PROCEDURE Verify (self: Rectangle) =
BEGIN
  WITH p = self.coords^, dist = Point.DistSquare DO
    <* ASSERT NUMBER(p) = 4 *>
    <* ASSERT dist (p[0],p[2]) = dist (p[1],p[3]) *>
  END
END Verify;
```

Assuming `point` is a `ARRAY [1..4] OF Point.T`, to draw a new `Rectangle` object, you must do:

```
VAR
    rect: Rectangle;
BEGIN
    rect := NEW(Rectangle);
    rect.init(points);
    rect.draw();
END
```

Or the shorthand:

```
VAR
    rect := NEW(Rectangle).init(points);
BEGIN
    rect.draw();
END
```

5.2.1 Programming with Objects: A Complete Example

The complete program `Objects` is another example of the use of objects.

```
MODULE Objects EXPORTS Main;
IMPORT IO;
```

The type `Person` declares a new object type with fields `firstname`, `lastname`, and `gender`. `Person` also defines a method `fullname()` which is implemented by procedure `FullName`.

```
TYPE
    Person = OBJECT
        firstname, lastname: TEXT;
        gender: Gender;
    METHODS
        fullname(): TEXT := FullName;
    END;
    Gender = {Female, Male};

PROCEDURE FullName (self: Person): TEXT =
    CONST Title = ARRAY Gender OF TEXT {"Ms.", "Mr."};
BEGIN
    RETURN Title[self.gender] & " " & self.firstname &
        " " & self.lastname;
END FullName;
```

Any code that can see the declaration of the object type `Person` can create new instances of that type. So, anywhere in this module, you can create a new instance of the type `Person`. (You can use interfaces to control the visibility of object types. See **Opaque Types: Information Hiding and Encapsulation** on page 88 for more information.)

Here a new object is assigned to the variable **john**.

```
VAR
    john := NEW(Person,
                gender := Gender.Male,
                firstname := "John",
                lastname := "Smith");
```

Here is a procedure, **Describe**, which takes a **Person** object and a text description and prints a line to the standard output using the **fullname()** method.

```
PROCEDURE Describe (person: Person; description: TEXT) =
BEGIN
    IO.Put (person.fullname() & " is " & description & ".\n");
END Describe;
```

Describe calls the **fullname** method of its first parameter. Of course, different **Person** objects can have different implementations of the **fullname** method, so, ultimately you can pass different subtypes of **Person** into this procedure. Here we create one such subtype, named **Employee** which has some additional fields. Note that it shares the same implementation as **Person** for the **fullname()** method.

```
TYPE
    Employee = Person OBJECT
        company: TEXT;
    END;
```

Next, we create a new instance of this type, named **jane**. You can list the fields of an object in any order when you initialize it.

```
VAR
    jane := NEW(Employee,
                firstname := "Jane",
                lastname := "Doe",
                company := "ACME Ltd",
                gender := Gender.Female);
```

You can create new subtypes that override existing methods. In the next subtype, the **fullname()** method of **Doctor** object, implemented via **FullDoctorName**, skips the first name and uses a professional title for referring to a **Doctor** instance. Note that **PrintDoctorName**'s self argument is of type **Doctor**.

```
TYPE
    Doctor = Person OBJECT
        title: TEXT := NIL;
    OVERRIDES
        fullname := FullDoctorName;
    END;
```


BEYOND THE BASICS

```
PROCEDURE FullDoctorName(self: Doctor): TEXT =
  VAR
    result: TEXT := "Dr. " & self.lastname;
  BEGIN
    IF self.title # NIL THEN
      result := result & ", " & self.title & ", ";
    END;
    RETURN result;
  END PrintDoctorName;
```

Let's create a couple of instances of **Doctor**.

```
VAR
  dr_who := NEW(Doctor,
    lastname := "who",
    title := "Time Lord");

  dr_quinn := NEW(Doctor,
    lastname := "Quinn",
    title := "Medicine woman");
```

There is also a shorthand for creating one-of-a-kind objects types as part of a **NEW** call. That's how **joe** gets created.

```
VAR
  joe := NEW(Person, firstname := "Joe",
    lastname := "Schmo",
    fullname := AnAverage);

PROCEDURE AnAverage(self: Person): TEXT =
  BEGIN
    RETURN "An average " & self.firstname & " " &
      self.lastname;
  END AnAverage;
```

Finally, make a few calls to **Describe** just to show that it works.

```
BEGIN
  Describe (john, "a nice person");
  Describe (jane, "an employee of " & jane.company);
  Describe (dr_who, "a bit weird");
  Describe (dr_quinn, "not for real");
  Describe (joe, "probably good enough for working" &
    "on this project");
END objects.
```

Feel free to create your own subtypes of **Person**!

For more information on controlling visibility of object declarations, see **Opaque Types: Information Hiding and Encapsulation** on page 88.

5.3 Threads: Managing Concurrent Activities

A *Thread* is the fundamental concurrency abstraction in Modula-3.

Using threads, you can create concurrent activities within a single activation of a program.

Threads can be very useful for structuring real-world programs, because they often need to deal with multiple activities. For example, a program that needs to interact with the user and manage a long-running query to a database can use a thread for each task, so that one task can progress without waiting for the other tasks.

Threads enforce a separation of concerns of concurrent activities in your program, which helps you manage each task better.

Using the required **Thread** interface, you can create and manage threads in an operating-system-independent manner. The **Thread** interface provides operations for communication and synchronization between threads. Standard libraries distributed with your system are thread-friendly. Indeed, some libraries, such as the user interface toolkit **Trestle**, use threads in order to perform their background activities.

For a good introduction to threads, read Andrew Birrell's article, *Introduction to Programming with Threads*, included on-line as part of your CM3-IDE distribution. The sample program **ThreadExample** briefly outlines how you can program with multiple threads.

ThreadExample is a simple multi-threaded program:

```
MODULE ThreadExample EXPORTS Main;
IMPORT Thread, Fmt, IO;
```

Driven by commands from the standard input. For each user request, the program spawns a new thread which waits for a specified elapsed time.

ThreadClosure. The following fragment creates a new *closure* object, which embodies the state of a thread. In this case, the **Thread.Closure** subtype **TimeClosure** contains the state required for a timer thread: a length of time that a thread must pause.

By convention, thread closures override the **apply** method to designate the work to be done. **TimeClosure**'s implementation is in procedure **TimerApply**.

```
TYPE
  TimerClosure = Thread.Closure OBJECT
    time: LONGREAL;
  OVERRIDES
    apply := TimerApply;
  END;
```

`TimerApply` performs the work of the timer threads:

- print out a message that it has started
- wait for `time` seconds
- print out a message that it has finished.

The local variable `count` allows the user to match the start and finish messages.

```
PROCEDURE TimerApply (cl: TimerClosure): REFANY =
VAR
    count := Counter();
BEGIN
    Print("\nStarting timer " & Fmt.Int(count) &
        " for " & Fmt.LongReal (cl.time) & " seconds.");
    Thread.Pause (cl.time);
    Print ("\nFinished timer " & Fmt.Int(count) &
        " after " & Fmt.LongReal (cl.time) & " seconds.\n");
    RETURN NIL;
END TimerApply;
```

Thread Synchronization. The variable `timer_count` keeps track of the count for the threads created; `timer_count_mu`, a *mutex* (or a *lock*) protects the critical sections, where multiple threads may be contending for `timer-count`.

```
VAR
    timer_count: CARDINAL := 0;
    timer_count_mu := NEW(MUTEX);
```

`Counter` returns a new counter. `Counter`'s critical section (the place where multiple threads may be racing each other) is protected by a `LOCK` statement. Note that `mutex` is automatically unlocked upon exit from the scope of the `LOCK` statement, so `RETURN timer_count` effectively unlocks `timer_count` on its way out of the procedure.

```
PROCEDURE Counter(): CARDINAL =
BEGIN
    LOCK timer_count_mu DO
        INC (timer_count);
        RETURN timer_count;
    END;
END Counter;
```

Forking Threads. The main program waits for user input and forks threads for new timers when the user asks for one.

The main loop reads input from the user to determine how long the next forked thread should pause. A `closure` is created dynamically to be passed into `Thread.Fork`, which will fork a new thread and run `closure.apply()`.

```

VAR
  input: CARDINAL;
  closure: TimerClosure;
BEGIN
  LOOP
    IO.Put(Prompt);
    input := IO.GetInt();
    closure :=
      NEW(TimerClosure, time := FLOAT(input, LONGREAL));
    EVAL Thread.Fork (closure);
  END;
END ThreadExample.

```

There was no need to wait for the forked thread in this example. To wait for a forked thread to complete, you call **Thread.Join(th)** which returns the value returned by the thread's **apply** method.

```

th := Thread.Fork (cl);
... do other activities ...
result := Thread.Join (th)

```

Other calls in the **Thread** interface, such as **Signal**, **wait**, and **Broadcast** provide for more intricate synchronization patterns. For a thorough introduction, see Andrew Birrell's article, *Introduction to Programming with Threads*, available on-line in the Technical Notes section of your CM3-IDE distribution.

5.4 Opaque Types: Information Hiding And Encapsulation

Encapsulating implementation details is a key technique in managing the growth of large programs. Often, you may want to reveal references to data structures in a module, while hiding the structure and implementation of a datatype. Opaque types are a mechanism for enforcing such a separation. An *opaque type* is a name that denotes an unknown subtype of a known reference type. For example, all you know about an opaque subtype of **REFANY** is that it is a traced reference. (**REFANY** is the root of the traced heap.) The actual type denoted by an opaque type name is called its *concrete type*.

Different scopes can reveal different information about an opaque type. For example, what is known in one scope only to be a subtype of **REFANY** could be known in another scope to be a subtype of **ROOT**.

An opaque type declaration has the form:

```
TYPE T <: U
```

where **T** is an identifier and **U** an expression denoting a reference type. It introduces the name **T** as an opaque type and reveals that **U** is a supertype of **T**. The concrete type of **T** must be revealed elsewhere in the program.

5.4.1 Fully Opaque Types

The simplest way to hide information is to divide your program into two groups: portions of your code where everything about the structure of a type is revealed, and portions where nothing is revealed. This dichotomy is the essence of *fully opaque types*. A fully opaque type is a subtype of **REFANY**, or **ADDRESS**, corresponding to a traced or untraced reference to an unknown type.

By combining fully opaque types with interfaces, you can create abstract datatypes with full encapsulation: the interface pairs the name of the type, with a set of procedures that operate on that type.

In the next example, the **Person** interface exports an opaque type **Person.T**, and the operations **New** and **Describe**.

```
INTERFACE Person; IMPORT Wr;
```

Declare **Person.T** as an opaque type.

```
TYPE
  T <: REFANY;
```

The **Person** interface defines an opaque type **T** (often called an *abstract data type*) with two operations:

- **New** for creating new instances
- **Describe** for printing textual descriptions.

Since none of the clients can “see through” this opaque interface, it should describe precisely *what* the implementation does without revealing *how* the implementation works.

The statement **U <: V** means that **U** is a subtype of **V**. When you see such a declaration, you can assume that an instance of **U** supports at least as many operations as an instance of **V**. In this case, since **V** is **REFANY**, all you can assume about **Person.T** is that it is a traced reference. It’s traced, so you don’t have to worry about managing its memory. It’s a reference so you can store it, or compare it with another reference of the same type for equality.

```
Gender = {Female, Male};
```

Person.Gender is an enumeration type with elements **Female** and **Male**.

```
PROCEDURE New (firstname, lastname: TEXT;
               gender: Gender): T;
```

Create a new **Person.T** given a first name, a last name, and a gender.

```
PROCEDURE Describe(person: T; desc: TEXT; wr: Wr.T := NIL);
```

Write a textual description, **desc**, of a **Person.T** to the writer stream **wr**. If **wr** is not specified, write the description to the standard output.

```
END Person.
```

Note that nothing about the implementation of **Person.T** is visible to clients of **Person**. Indeed, the compiler will not know any more information while compiling clients of this interface; hence, if you change the implementation for this module, you don't have to recompile its clients.

```
MODULE Person;
IMPORT IO, Wr;
```

The **Person** module implements the opaque type **Person.T**. To do so, it reveals the representation of **Person.T** completely. Within the module, we can use the items declared in the interface without qualification. Hence, **T** in this module refers to **Person.T** and **Gender** refers to **Person.Gender**.

Since the only information specified in the **Person** interface is that **Person.T** is a reference—or more precisely, **Person.T** <: **REFANY**—the implementation can specify the full structure of the object. Indeed, the full revelation of **Person.T** is similar to an ordinary object type declaration.

```
REVEAL
  T = BRANDED OBJECT
    firstname, lastname: TEXT;
    gender: Gender;
  METHODS
    fullname(): TEXT := FullName;
  END;
```

The **BRANDED** keyword is required in all full revelations and ensures that instances of **Person.T** are distinct from all other types with the same structure. Essentially, the **BRANDED** keyword overrides Modula-3's structural equivalence for this type. See the language reference for more information about branding.

Next, we declare an array of title names. The outside world, of course, does not know about the existence of this array as it is not visible from the interface **Person**.

```
CONST
  Title = ARRAY Gender OF TEXT {"Ms.", "Mr."};
```

Procedure **FullName** is the implementation of method **fullname()** of **Person.T**. Since **FullName** is not exported by the **Person** interface it will not be visible to any outside modules.

```

PROCEDURE FullName (p: T): TEXT =
BEGIN
    RETURN Title[p.gender] & " " & p.firstname &
        " " & p.lastname;
END FullName;

```

The next procedure, **Describe**, is exported and hence visible to all clients of the **Person** interface. Since **Describe** is defined within this module, the representation of **p.fullname()** is visible within its body.

```

PROCEDURE Describe(p: T; desc: TEXT; wr: Wr.T := NIL) =
BEGIN
    IO.Put (p.fullname() & " is " & desc & ".\n", wr);
END Describe;

BEGIN
END Person.

```

5.4.2 Clients of an Opaque Type

OpaqueExample is a client of the **Person** interface:

```

MODULE OpaqueExample EXPORTS Main;
IMPORT Person;

```

There, we assign new **Person.T** instances to three variables **jane**, **june**, and **john** (using various combinations of positional and keyword parameter binding.)

```

VAR
    jane: Person.T :=
        Person.New(firstname := "June",
                    lastname := "Doe",
                    gender := Person.Gender.Female);
    june := Person.New("June", "Doe",
                      gender := Person.Gender.Female);
    john := Person.New("John", "Doe", Person.Gender.Male);

```

Next, we call **Person.Describe** a few times. Note that **jane.firstname**, **jane.lastname**, or even **june.fullname()** are not available in this module, even though they are available within the implementation of **Person**, since **Person.T** is an opaque type.

```

CONST
    address = "123 Main Street";
BEGIN
    Person.Describe (jane, "lives at " & address);
    Person.Describe (june, "lives at " & address);
    Person.Describe (john, "lives at " & address);

```

Indeed, if you were to invent your own **Person** type—even if its structure was the same as that of **Person.T**—the compiler would prevent you from passing the imposter into **Person.Describe**. The only way to get a new **Person.T** object is by calling **Person.New**.

END OpaqueExample.

By using opaque types, the **Person** interface achieves full encapsulation.

Sometimes full encapsulation is too strong. In the rest of this section, you learn how to encapsulate only parts of your objects.

5.4.3 Partially Opaque Types: Revealing Types in Moderation

Opaque types hide and reveal type information in an extreme manner. If the concrete implementation of an opaque type is revealed in your current scope, you know everything about the structure and the implementation of the type. If the concrete revelation is not available in your scope, then you may know nothing about the structure and implementation for the type.

In practice, you may need more control over the visibility of types in different parts of your programs. *Partial revelation* of opaque types enables fine-grained control over the visibility of fields and methods of your objects.

A natural extension of the opaque type concept, partially opaque types may be used to generalize the language-enforced visibility rules. Using partial revelation, you may define visibility rules that fit your particular application, instead of confining your object types to the hard-coded public, private, protected, and friend visibility rules common in other languages.

A new version of the **Person** interface illustrates partial revelation. A partially opaque type **Person.T** is defined, with two operations **init** and **fullname**.

```
INTERFACE Person;
```

Using the idiom **TYPE T <: Public; Public = OBJECT ... END**, the next fragment states that the **Person.T** supports at least operations **init** and **fullname**, without revealing the exact structure of **Person.T**, or revealing what other methods **Person.T** may support.

To declare **Person.T**, first, we declare a type **T** as a subtype of **Public**.

```
TYPE
  T <: Public;
```

Then, we define **Public**, the publicly available revelation of this **Person.T** to be an object type, with the methods **init** and **fullname**.

```
Public = OBJECT
  METHODS
    init(firstname, lastname: TEXT; gender: Gender): T;
    fullname(): TEXT;
END;
```


The name **Public** is used here by convention, not by a hard-coded rule. The method **init** initializes the object using **firstname**, **lastname**, and **gender** and returns the initialized object. The **init** method is used by convention to initialize values as they are.

The method **fullname** returns the full name of the **Person.T** object in question.

```
Gender = {Female, Male};
END Person.
```

In contrast with fully opaque definition of the person interface, there is no need to provide a **New** procedure in the interface. Clients of this interface can freely instantiate **Person.T** using the built-in **NEW** operation. Another difference is that clients of a partially-opaque type can invoke methods on it. In this case, the methods **init** and **fullname** are available to all clients of **Person**. After calling a built-in **NEW** operation, you can call **init** to initialize the newly instantiated object. The method **fullname** returns a text string containing the name of a person; hence, there is no need for the **Describe** procedure to exist inside this module.

Declaring a partially opaque type also allows clients of this interface to create new subtypes of **Person.T**. By calling **Person.T.init**, such subtypes can assure that the **Person.T** portion of the object is initialized properly.

```
MODULE Person;
```

The implementation of the **Person** interface implements the partially opaque type **Person.T**. To do so, it *reveals* the representation of **Person.T** fully.

The **Person** interface already defines the signatures for procedures **init** and **fullname**. It is the role of this module to implement these methods, and add the underlying implementation structure. The **BRANDED** keyword will ensure that instances of other types with identical structure cannot masquerade as **Person.T** objects.

```
REVEAL
  T = Public BRANDED OBJECT
    firstname, lastname: TEXT;
    gender: Gender;
  OVERRIDES
    init := Init;
    fullname := FullName;
  END;
```

Procedure **Init** initializes a **Person.T** object, and returns the **self** parameter just initialized.

```

PROCEDURE Init (self: T; firstname, lastname: TEXT;
                gender: Gender): T =
BEGIN
    self.firstname := firstname;
    self.lastname := lastname;
    self.gender := gender;
    RETURN self;
END Init;

```

Define the procedure **FullName**, the implementation of method **fullname()** of **Person.T**. Procedure **FullName** itself is not exported to the clients of **Person** interface, but the method **fullname()** of **Person.T** is visible to clients.

```

PROCEDURE FullName (p: T): TEXT =
BEGIN
    RETURN Title[p.gender] & " " & p.firstname & " " &
        p.lastname;
END FullName;

CONST
    Title = ARRAY Gender OF TEXT {"Ms.", "Mr."};
BEGIN
END Person.

```

5.4.4 Subtyping Partially Opaque Type

In the next module, **Employee.T** is defined as a subtype of a partially opaque type **Person.T**.

```

INTERFACE Employee;
IMPORT Person;

TYPE
    T <: Public;
    Public = Person.T OBJECT
    METHODS
        init(first, last: TEXT;
              gender: Person.Gender;
              company: TEXT): T;

    END;
END Employee.

```

The declaration of **Employee.T** is similar to that of **Person.T**. In this case, only a new method is declared. (As you will see, **Employee.T** also overrides the implementation of **Person.T.fullname()** method, however, **Employee's** clients need not know this. So, if the implementation of **fullname()** changes, **Employee's** clients don't need to be re-compiled.)

Employee.Public defines the publicly available definition of **Employee.T**, to be used in full revelation of **Employee.T** inside **Employee's** implementation.

BEYOND THE BASICS

```
MODULE Employee;
IMPORT Person;
IMPORT TextIntTbl, Fmt;

REVEAL
  T = Public BRANDED OBJECT
    company: TEXT;
    id: INTEGER;
  OVERRIDES
    init := Init;
    fullname := FullName;
  END;
```

The above statement gives the full implementation of **Employee.T**, along with its fields **company**, and **id**, and its method implementations. Note that **Employee.Public** is just a shorthand for **Person.T OBJECT METHODS init(...) END**; it is named **Public** only for convenience.

```
VAR
  employee_count := NEW(TextIntTbl.Default).init();
```

Init defines the implementation of the **Employee.T.init** method. It takes an extra **company** parameter.

```
PROCEDURE Init (self: T;
                first, last: TEXT;
                gender: Person.Gender;
                company: TEXT): T =
VAR
  emp_id := 0;
BEGIN
  EVAL Person.T.init(self, first, last, gender);
  self.company := company;
  EVAL employee_count.get(company, emp_id);
  INC(emp_id);
  self.id := emp_id;
  EVAL employee_count.put(company, emp_id);
  RETURN self;
END Init;
```

FullName defines the implementation of the **Employee.T.fullname** method. Note how it uses its supertype's **fullname** method by calling **Person.T.fullname(self, ...)**.

```
PROCEDURE FullName(self: T): TEXT =
BEGIN
  RETURN Person.T.fullname(self) & ", " &
    "employee #" & Fmt.Int(self.id) &
    " at " & self.company & ",";
END FullName;
```

```
BEGIN
END Employee.
```

5.4.5 Clients of a Partially Opaque Type

The main module, `PartiallyOpaque`, imports both `Person` and `Employee`.

```
MODULE PartiallyOpaque EXPORTS Main;
IMPORT Person, Employee;
IMPORT IO;
```

The next procedure, `Describe`, uses the `fullname` method of `Person.T` to print a textual description of a person. Note that `Person.T` does not reveal its internal structure, but it does reveal the `fullname` method, which is enough to allow the `Describe` procedure to be included this module instead of the `Person` module.

Instead of having to change `Person` every time a client requires a new `Describe` procedure, the new structure allows each client to implement its own `Describe` procedures without affecting `Person`.

```
PROCEDURE Describe(p: Person.T; desc: TEXT) =
BEGIN
  IO.Put (p.fullname() & " is " & desc & ".\n");
END Describe;
```

The next statements assign new `Person.T` instances to four variables `john`, `jane`, `june`, and `jack`. Note the use of the `v := NEW(T).init(...)` idiom in declaring, instantiating, and initializing these instances.

```
VAR
  john := NEW(Person.T).init("John", "Doe",
                             Person.Gender.Male);
  jane := NEW(Employee.T).init("Jane", "Doe",
                                Person.Gender.Female,
                                "ACME Ltd");
  june := NEW(Employee.T).init("June", "Doe",
                                Person.Gender.Female,
                                "Mass. State");
  jack := NEW(Employee.T).init("Jack", "Smith",
                                Person.Gender.Male,
                                "ACME Ltd");
```

Now, assign a new `Person.T` with a special `fullname` method to the variable `madonna`.

```
VAR
    madonna := NEW(Person.T, fullname := Madonna);

PROCEDURE Madonna(<*UNUSED*>self: Person.T): TEXT =
BEGIN
    RETURN "Madonna"
END Madonna;
```

The main body will make a few calls to `Describe`. Note the use of `john.fullname()` to call a method defined on a `Person.T`, and `Person.T.fullname(june)` to call a `Person.T` method on an `Employee.T`. Of course, the latter case is type-checked at compile-time.

```
CONST
    address = "123 Main Street";

BEGIN
    Describe (madonna, "a pop icon");
    Describe (john, "a resident at " & address);
    Describe (jane, "a resident at " & address);
    Describe (june, "a resident at " & address);
    Describe (jack, john.fullname() & "'s brother.");
    Describe (june, "called " &
        Person.T.fullname(june) & " outside work");
END PartiallyOpaque.
```

Partially opaque types are powerful structuring constructs for building large programs. For more information on partially opaque types, see the language specification, and **I/O Streams: Abstract Types, Real Programs** in *Systems Programming with Modula-3*.

5.5 Generics: Resuable Data Structures and Algorithms

A *generic* is a template for instantiating similar modules. Generics—called parameterized types, or templates in other languages—allow you to build generic data structure and algorithm code and readily use them in different contexts. For example, a generic hash table module could be instantiated to produce tables of integers, tables of text strings, or tables of a user-defined type. Different generic instances are compiled independently: the source program for the generic and its parameters is reused, but the compiled code for one instance has no relationship with other compiled instances.

To keep Modula-3 generics simple, they are confined to the module level: generic procedures and types do not exist in isolation, and generic parameters must be entire interfaces. In the same spirit of simplicity, there is no separate type checking associated

with generics. Implementations are expected to expand the generic and type-check the result.

Usually generic interfaces and modules contain code that operates independently of the type of data it operates on. The type that the generic will be operating upon is defined at compile-time.

5.5.1 Using Generics

In this example, we use the standard **List** generic interface to implement a set abstraction, and the standard **Table** generic interface to implement a mapping from names to action procedures.

Before exploring the program, let's review the **List**, **Atom**, and **Atom-List** interfaces. An abbreviated version of each interface is included here; see the on-line version of the interface for full comments.

5.5.2 A Generic Example: List

The generic interface **List** provides operations on linked lists of arbitrary element types.

```
GENERIC INTERFACE List(Elm);
```

Where **Elm.T** is not an open array type and the **Elm** interface contains:

```
CONST Brand = <text-constant>;
PROCEDURE Equal(k1, k2: Elm.T): BOOLEAN;
```

Brand must be a text constant. It will be used to construct a brand for any generic types instantiated with the **List** interface.

```
CONST Brand = "(List " & Elm.Brand & ")";
```

A **List.T** represents a linked list of items of type **Elm.T**.

```
TYPE T = OBJECT head: Elm.T; tail: T END;
PROCEDURE Cons(READONLY head: Elm.T; tail: T): T;
PROCEDURE List1(READONLY e1: Elm.T): T;
PROCEDURE List2(READONLY e1, e2: Elm.T): T;
PROCEDURE List3(READONLY e1, e2, e3: Elm.T): T;
PROCEDURE FromArray(READONLY e: ARRAY OF Elm.T): T;
PROCEDURE Length(l: T): CARDINAL;
PROCEDURE Nth(l: T; n: CARDINAL): Elm.T;
PROCEDURE Member(l: T; READONLY e: Elm.T): BOOLEAN;
PROCEDURE Append(l1: T; l2: T): T;
PROCEDURE AppendD(l1: T; l2: T): T;
PROCEDURE Reverse(l: T): T;
PROCEDURE ReverseD(l: T): T;
END List.
```

5.5.3 Parameter to a Generic: Atom

Interface **Atom** will be used as a parameter to the **List** generic interface, hence **Atom** must include a **Brand** constant and an **Equal** comparison procedure.

```
INTERFACE Atom;
```

An **Atom.T** is a unique representative for a set of equal texts (like a Lisp atomic symbol.)

```
    TYPE T <: REFANY;
    CONST Brand = "Atom-1.0";
    PROCEDURE FromText(t: TEXT): T;
    PROCEDURE ToText(a: T): TEXT;
    PROCEDURE Equal(a1, a2: T): BOOLEAN;
    PROCEDURE Hash(a: T): INTEGER;
    PROCEDURE Compare(a1, a2: T): [-1..1];
END Atom.
```

5.5.4 Instantiating a Generic: AtomList

Finally, we instantiate a **List** with an **Atom** parameter:

```
INTERFACE AtomList = List (Atom) END AtomList.
```

AtomList can be imported by other modules that use lists of atoms. The main program for this example imports **AtomList**.

Most generic interfaces have been pre-instantiated for common datatypes such as **Texts** and **Atoms**. Indeed, **AtomList** is one such interface. See the *Interface Index* for an overview of the available generic interfaces. The language tutorial and reference manual also describe the behavior of generics in more detail.

The rest of this section describes how you can use instantiated generics, and how you can instantiate generics with user-defined parameters.

5.5.5 Using Instances of Generics

The main module, **Generics**, uses an **AtomList** to keep track of names, as well as an instance of the **Table** interface that maps **Atoms** to **Actions**. The instantiation of the table with a user-defined type **Action** is described later. Import the **Action** interface, defined in this package, and the **AtomActionTbl**, a table mapping atoms to actions.

```
MODULE Generics EXPORTS Main;
IMPORT Atom, AtomList;
IMPORT Action, AtomActionTbl;
IMPORT Process, IO; <* FATAL IO.Error *>
```

Atom List operations. Insert an element into the list.

```
PROCEDURE Insert (VAR list: AtomList.T;
                  atom: Atom.T) =
BEGIN
  IF NOT AtomList.Member (list, atom) THEN
    list := AtomList.Cons (atom, list);
  END
END Insert;
```

Print all elements of the list by iterating over its members.

```
PROCEDURE Print(x: AtomList.T) =
BEGIN
  WHILE x # NIL DO
    IO.Put (Atom.ToText (x.head) & " ");
    x := x.tail;
  END;
END Print;
```

Command operations. Actions define initial values for the action table.

```
CONST
  Actions = ARRAY OF Action.T {
    Action.T { "show", Show},
    Action.T { "quit", Quit},
    Action.T { "reset", Reset},
    Action.T { "help", Help}};
```

Each procedure defines what each action should do. Note that the **proc** field of **Action.T** is defined to be a **PROCEDURE()**, so we can assign any of **Quit**, **Rest**, **Help**, or **Show** to fields of **Actions**.

```
PROCEDURE Quit() = BEGIN Process.Exit(0); END Quit;
PROCEDURE Reset() = BEGIN input_set := NIL; END Reset;

PROCEDURE Show() =
BEGIN
  Print(input_set); IO.Put ("\n");
END Show;

PROCEDURE Help() =
BEGIN
  IO.Put("Commands: show, reset, help, or quit.\n" &
    "Other items will be inserted into the list.\n");
END Help;
```


The variable `command_table` is an `atom`→`action` table; `input_set` is an `atom` list, containing all the elements that will be entered.

```

VAR
  command_table := NEW(AtomActionTbl.Default).init();
  input_set : AtomList.T := NIL;
BEGIN
  FOR x := FIRST(Actions) TO LAST(Actions) DO
    EVAL command_table.put(Atom.FromText (x.name), x);
  END;

  IO.Put ("welcome to the atomic database.\n");
  IO.Put ("Try any of commands: show quit reset help.\n");
  IO.Put ("Any other string will be entered into the" &
    "database.\n\n");

```

Loop, get the command line. If it's a command, do it. Otherwise insert the command line into the `input_set`. If `atom` is in the `command_table` then run the corresponding `Action.T`; otherwise, Insert the `atom` into the `input_set`.

```

LOOP
  IO.Put ("atom-db > ");
  IF IO.EOF () THEN EXIT END;
  VAR
    cmd := IO.GetLine();
    atom := Atom.FromText(cmd);
    action: Action.T;
  BEGIN
    IF command_table.get(atom, action)
    THEN action();
    ELSE Insert(input_set, atom);
    END;
  END;
END;
END Generics.

```

See the *Interface Index* for more information on various kinds of generics.

5.5.6 Instantiating Generics for User-Defined Types

Since `Action` is a user-defined type, there are no pre-instantiated interfaces available for it. CM3-IDE provides handy makefile procedures for instantiating various generics automatically.

The `Table` interface requires a *key* and a *value* parameter:

```

GENERIC INTERFACE Table (Key, Value) = ... END Table.

```

The **Key** in this case is `Atom`, the **Value** is an `Action`. Here, `Action` is a user-defined interface. `Action.T`, denoting actions for commands, is a procedure with no parameters and no results. `Action.Brand` is used by the table generic to create a composite brand for our table.

```

INTERFACE Action;
TYPE
  T = RECORD
    name : TEXT
    handler : PROCEDURE();
  END
CONST
  Brand = "Action";
END Action.

```

5.5.7 Instantiating Generics in a Makefile

Finally, the makefile for this package will instantiate an atom→action table with the name `AtomActionTbl`:

```

import ("libm3")
table("AtomAction", "Atom", "Action")
module("Action")
implementation ("Generics")
program ("atom-db")

```

(If you haven't built your package yet, you won't be able to see the contents of `AtomActionTbl` because it is generated as part of the build process.)

5.6 Unsafe Constructs: System Programming in CM3-IDE

This program illustrates the use of unsafe constructs, such as **LOOPHOLE**—an unsafe cast.

The default mode for programs in CM3-IDE is *safe*, i.e., the language and its runtime are responsible for checking run-time errors. For programming intricate systems, integrating legacy systems, or making programs more efficient, you may decide that you would like the freedom to perform tasks that circumvent language-enforced safety.

You have the freedom to perform unsafe operations in *unsafe* modules by using additional operations, such as **LOOPHOLE** (an unsafe cast to an arbitrary type) or **ADR** (address of a variable). These operations are restricted to unsafe modules because they violate invariants enforced and assumed by the language in the safe modules.

With the freedom in unsafe modules comes the responsibility for the programmer to check for run-time errors in place of the language runtime. *You* are now responsible for making sure that a **LOOPHOLE** is not causing run-time errors.

Separation of safe and unsafe codes is a key technique in writing portable programs that utilize unsafe or non-portable features of particular systems. Indeed it is common practice for systems programmers to divide their code into safe and unsafe portions,

even if the programs are written in C. This way, the bulk of porting to a new platform, lies in the unsafe portion. CM3-IDE extends this model by providing language support for separation of safe and unsafe modules. Both interfaces and modules can be marked as **UNSAFE**.

If you care about robustness of your code, you are best to code most (if not all) of your programs in the (default) safe mode, since it is much easier to understand and explain the behavior of safe programs, hence it is also easier to make them robust.

A safe module can only import safe interfaces, so in safe programming you can't mistakenly count on unsafe functionality in another unsafe module.

An unsafe module can make its functionality available to other safe modules by exporting a safe interface. This is how you can bridge the safety gap—otherwise if your program includes one unsafe module, then your whole program must be marked unsafe. When you export a safe interface from an unsafe module, you the programmer are guaranteeing the intrinsic safety of the calls in the safe interface.

One nice aspect of the inclusion of the unsafe features in the language is that you don't have to rely on calls to external, lower-level languages to make your programs more efficient. Indeed, the unsafe portions of your code will have as much control over the representation and layout of your data structures as you have when programming in an unsafe language like C. The support for unsafe modules has been used to implement operating systems, windowing systems, networking software, and the language runtime itself in Modula-3, a task that is not easily accomplished with other high-level languages.

5.6.1 Unsafe Coding Example

In this small example, an interface to the standard C library call **abs** is marked inside an unsafe interface **Clib**, which is imported by an unsafe main program, **UnsafeExample**. Note that **UnsafeExample** cannot be marked safe because it imports an unsafe interface.

```
UNSAFE INTERFACE Clib;
  <*EXTERNAL*> PROCEDURE abs(x: INTEGER): INTEGER;
END Clib.
```

The pragma **<*EXTERNAL*>** declares a procedure to be provided at link-time by external code. (In this case, by the C runtime.)

BEYOND THE BASICS

Following, the main module imports `Clib` and uses `Clib.abs`.

```
UNSAFE MODULE UnsafeExample EXPORTS Main;
FROM Clib IMPORT abs;
IMPORT IO, Fmt;

CONST
  an_integer = 10;
BEGIN

  IO.Put ("Absolute value of ");
  IO.PutInt (an_integer);
  IO.Put (" is ");
  IO.PutInt (abs(an_integer));
  IO.Put (".\n");

  IO.Put ("Absolute value of " & Fmt.Int(-an_integer) &
    " is " & Fmt.Int(abs(-an_integer)) & ".\n");
END UnsafeExample.
```

As an application programmer using CM3-IDE, you need not worry about unsafe modules unless you are writing code where efficiency is the first concern, or you are using non-portable features of an operating system or external component. The full details of the available unsafe features are described in the *Language Reference*.

5.7 Summary

Exceptions. A robust program must handle error conditions well. Exceptions are a convenient language construct for handling errors and abnormal conditions in a way that preserves your program structure.

Objects. A flexible structuring construct for building programs, objects in CM3-IDE are garbage-collected. They conform to a single-inheritance hierarchy.

Threads. Often you may need to manage multiple concurrent activities as part of your program. Threads provide the required features.

See Andrew Birrell's *Introduction to Programming with Threads*, in the Technical Notes section of your distribution, for a thorough introduction to multi-threaded programming.

Opaque types. For full encapsulation of the internal structure of types in one module from other modules, you can use opaque types. Opaque type visibilities are enforced by the language.

Partially Opaque Types. To allow multiple levels of visibility of objects in your programs, you can use partially opaque types. They are a generalization of hard-coded visibility rules such as public, private, protected, and friend modes in other languages.

Generics. Reuse is an important aspect of large program development. Known also as templates or parametrized types, generic interfaces and modules allow you to reuse data structures and algorithms. See **CM3-IDE Interface Index** on page 143 for a list of available generics.


Unsafe Programming. While safe programming is the default for CM3-IDE, at times you may not be able to avoid the need for unsafe code, for example, to write efficiency-critical portions of your program, or to interface with other languages. Unsafe modules allow you to perform tasks that are ordinarily disallowed by the language safety semantics. CM3-IDE provides mechanisms for managing unsafe portions of your code and for combining unsafe code with safe code.

This page left blank
intentionally.

Read this chapter if you know CM3-IDE well, and would like to use CM3-IDE for system development.

6. Development Recipes

Assuming that you are familiar with the CM3-IDE environment and the Modula-3 language, this chapter describes a number of recipes for building simple realistic systems applications.

A handful of complete programs illustrate the use of advanced facilities in CM3-IDE. You can find the sources for programs in this chapter in the  Examples section of your CM3-IDE environment.



Using a simple automated bank teller scenario, **Robust Distributed Applications: Network Objects** on page 108 illustrates how to build distributed applications with Network Objects.

Client/Server Computing: Safe TCP/IP Interfaces on page 115 describes the safe, multi-platform, and multi-threaded TCP/IP interfaces. A Finger client and a simple HTTP server are described.

Taking Persistent Snapshots of Objects: Pickles on page 120 demonstrates how to transcribe objects onto an I/O stream.

Quick Comparison of Large Data: Fingerprints on page 122 outlines how to take fingerprints of large data structures, and use the fingerprints to compare the data structures efficiently.

Portable Operating System Interfaces on page 124 illustrates the use of portable interfaces for operating system services, such as: file systems access, process management, thread creation, and environment variables. A complete and portable command-shell program is used as a demonstration.

Dynamic Web Applications: the Web Server Toolkit on page 135 outlines a simple contact database program based on the web toolkit.

Interacting with C Programs on page 137 shows how to have your code call C programs or be called by C programs. Examples illustrate the integration of C source code and libraries into CM3-IDE.

6.1 Robust Distributed Applications: Network Objects

Network Objects allows an object to be handed to another process in such a way that the process receiving the object can operate on it as if it were local. The holder of a remote object can freely invoke operations on that object just as if it had created that object locally.

Further, it can pass the object to other processes. Thus, the Network Objects system allows the development of not just simple client/server applications, but more general multi-tiered distributed applications.

When a program calls another through Network objects, we refer to the caller as *the client*, and the callee as *the server*. In the context of network objects, the names client and server signify roles in a particular interaction—a server may in fact be a client of another server.

The contract between the client and the server is defined by a *common interface*.

Here we describe a simple automated bank teller program as an example by outlining each component: the interface, the client and the server.

6.1.1 The Common Interface

The **Bank** interface defines the common contract between client and server in our example.

NetObj is the primary interface for building network object applications. **NetObj.Error** and **Thread.Alerted** may be raised by network object operations. A **Bank.T** is a network object which supports the operation **findAccount**, which returns a **Bank.Account** object. Type **Bank.Account** supports operations **deposit**, **withdraw** and **get_balance**.

Network object operations can raise user-defined exceptions such as **BadAmount**, and **InsufficientFunds**.


```

INTERFACE Bank;
IMPORT NetObj;
FROM NetObj IMPORT Error;
FROM Thread IMPORT Alerted;
TYPE
  T = NetObj.T OBJECT METHODS
    findAccount (acct: AcctNum): Account
      RAISES {Alerted, Error};
  END;
TYPE
  Account = NetObj.T OBJECT METHODS
    deposit (amount: REAL)
      RAISES {BadAmount, Alerted, Error};
    withdraw (amount: REAL) RAISES {BadAmount,
      InsufficientFunds, Alerted, Error};
    get_balance (): REAL RAISES {Alerted, Error};
  END;

TYPE
  AcctNum = [1..100];

EXCEPTION
  BadAmount;
  InsufficientFunds;
END Bank.

```

A simple makefile instructs CM3-IDE that **Bank.T** and **Bank.Account** are network objects. CM3-IDE will generate the required stubs automatically as part of this library, so a client or a server in this scenario may use **netobj-interface**.

Import netobj to bring in the network object libraries.

```
import("netobj")
```

For each network object type **I.T** you must call **netobj(I,T)**

```

interface("Bank")
netobj("Bank", "T")
netobj("Bank", "Account")
library("netobj-interface")

```

6.1.2 A Network Object Server

NetObjServer is a sample implementation of a network object server that exports an implementation of the **Bank** interface.

```

MODULE NetObjServer EXPORTS Main;
IMPORT Bank, NetObj, Thread;
IMPORT IO, Fmt;

```

BankImpl defines a full representation for the **Bank.T** network object.

```

TYPE
  BankImpl = Bank.T OBJECT
    accounts : ARRAY Bank.AcctNum OF Account;
  OVERRIDES
    findAccount := FindAccount;
  END;

```

Find an account in the table of accounts:

```

PROCEDURE FindAccount (self: BankImpl;
                       acct: Bank.AcctNum
                       ): Bank.Account =
BEGIN
  RETURN self.accounts[acct];
END FindAccount;

```

For **Bank.Account** network objects, **Bank.Account** uses a **MUTEX** to synchronize access to its balance. It also implements the operations **deposit**, **withdraw**, and **get_balance**.

```

TYPE
  Account = Bank.Account OBJECT
    lock : MUTEX;
    balance : REAL := 0.0;
  OVERRIDES
    deposit := Deposit;
    withdraw := Withdraw; (* not included *)
    get_balance := Balance; (* not included *)
  END;

```

Deposit the money, making sure to serialize access with others trying to operate on this account.

```

PROCEDURE Deposit (self: Account; amount: REAL)
  RAISES {Bank.BadAmount} =
BEGIN
  IF amount < 0.0
  THEN RAISE Bank.BadAmount;
  END;
  LOCK self.lock DO
    self.balance := self.balance + amount;
  END;
END Deposit;

```

DEVELOPMENT RECIPES

Withdraw the money, making sure to serialize access with others trying to operate on this account.

```
PROCEDURE withdraw (self: Account; amount: REAL)
  RAISES {Bank.BadAmount,
         Bank.InsufficientFunds} =
BEGIN
  IF amount < 0.0
  THEN RAISE Bank.BadAmount;
  END;
  LOCK self.lock DO
    IF self.balance < amount
    THEN RAISE Bank.InsufficientFunds
    END;
    self.balance := self.balance - amount;
  END;
END withdraw;
```

Get the balance, making sure to serialize access with others trying to operate on this account.

```
PROCEDURE Balance (self: Account): REAL =
BEGIN
  LOCK self.lock DO
    RETURN self.balance;
  END;
END Balance;
```

Create a new bank by instantiating all the account objects.

```
PROCEDURE NewBank () : BankImpl =
VAR b := NEW (BankImpl);
BEGIN
  FOR i := FIRST (b.accounts) TO LAST (b.accounts) DO
    b.accounts[i] :=
      NEW (Account, lock := NEW (MUTEX));
  END;
  RETURN b;
END NewBank;
```

DEVELOPMENT RECIPES

Print a summary of all the active accounts, i.e., ones that have a positive balance.

```
PROCEDURE PrintSummary() =
BEGIN
  IO.Put (BankName & ": active account information\n");
  FOR i := FIRST(bank.accounts) TO LAST(bank.accounts) DO
    IF bank.accounts[i].balance > 0.0 THEN
      IO.Put (Fmt.Int(i) & ".....$" &
        Fmt.Real(bank.accounts[i].balance) & "\n");
    END;
  END;
END PrintSummary;
```

Finally, the server's global variables and main body. The main body prints the summaries for accounts every 60 seconds. Since the network objects runtime forks and manages threads to handle incoming calls, the server can simply loop, printing its summary.

```
CONST
  BankName = "LastNationalBank";
VAR
  bank := NewBank();
BEGIN
  IO.Put ("Starting bank server.\n");
  TRY
    (* Export the bank object under "LastNationalBank". *)
    NetObj.Export (BankName, bank);
    IO.Put ("Bank server was exported as " &
      BankName & "\n");
    LOOP
      Thread.Pause (60.0D0);
      PrintSummary();
    END;
  EXCEPT (* If there is a problem, print an error and exit. *)
  | NetObj.Error =>
    IO.Put ("A network object failure occurred.\n");
  | Thread.Alerted => IO.Put ("Thread was alerted.\n");
  END;
END NetObjServer.
```

The makefile for the server is simple. Note that the server must import the library defining the common interface. In this case, it's called **netobj-interface**.

```
import("netobj")
import("netobj-interface") % the common interface
implementation("NetObjServer")
program("netobj-server")
```

6.1.3 A Network Object Client

`NetObjClient` is a sample implementation of a network object client.

```
MODULE NetObjClient EXPORTS Main;
IMPORT Bank, NetObj, Thread;
IMPORT IO, Fmt, Scan, Text, FloatMode, Lex;
VAR
    bank: Bank.T;
    acctnum: Bank.AcctNum;
    acct: Bank.Account := NIL;
    cmd: TEXT;
```

Print a prompt on the screen and asks for input from the user. If the current account is set, it will display the current account and the available balance.

```
PROCEDURE Prompt(txt: TEXT): TEXT RAISES {IO.Error} =
BEGIN
    TRY
        IF acct # NIL THEN
            IO.Put ("\n[acct:" & Fmt.Int(acctnum) &
                ", balance: $" &
                Fmt.Real (acct.get_balance()) & "]" );
        END;
    EXCEPT
        ELSE (* since it is only a prompt, we ignore all exceptions *)
    END;
    IO.Put (txt & " : ");
    RETURN IO.GetLine();
END Prompt;

EXCEPTION
    InvalidAccount;
    Quit;
```

This client will take input commands and make calls to network objects. As you can see, most of the work is in the reading of input from the user!

```
CONST
    BankName : TEXT = "LastNationalBank";
BEGIN
    IO.Put ("welcome to " & BankName & "\n");
    IO.Put ("Connecting to bank server...");
    TRY
        bank := NetObj.Import (BankName);
    EXCEPT
        IO.Put ("done.\n");
        IO.Put ("Bank Teller client started...\n");
        IO.Put ("valid commands are: \n" &
            "  account : set a current account for further " &
            "    transactions\n" &
            "  deposit : deposit into current account \n" &
            "  withdraw: withdraw from the current account\n" &
            "  balance : print balance for the current account\n" &
            "  quit    : quit bank teller client\n");
        IO.Put ("\n");
```

DEVELOPMENT RECIPES

```

LOOP
  TRY
    cmd := Prompt("Command: ");
    IF Text.Equal (cmd, "account")
    THEN (* new account *)
      WITH input = Scan.Int(Prompt("account number")) DO
        IF input < FIRST(Bank.AcctNum) OR
           input > LAST(Bank.AcctNum)
        THEN
          RAISE InvalidAccount;
        END;
        acct := bank.findAccount(input);
        acctnum := input;
      END;
    ELSIF Text.Equal(cmd, "deposit")
    THEN (* deposit *)
      IF acct = NIL THEN RAISE InvalidAccount END;
      WITH amount = Scan.Real(Prompt(" amount")) DO
        acct.deposit(amount);
      END;
    ELSIF Text.Equal(cmd, "withdraw")
    THEN (* withdraw *)
      IF acct = NIL THEN RAISE InvalidAccount END;
      WITH amount = Scan.Real(Prompt(" amount")) DO
        acct.withdraw(amount);
      END;
    ELSIF Text.Equal(cmd, "balance")
    THEN (* get balance *)
      IF acct = NIL THEN RAISE InvalidAccount END;
      IO.Put ("Balance is " &
              Fmt.Real(acct.get_balance()) & "\n");
    ELSIF Text.Equal(cmd, "quit")
    THEN (* quit by raising the "Quit" exception. *)
      RAISE Quit;
    ELSE (* invalid command *)
      IO.Put ("Valid commands are: account, " &
              "deposit, withdraw, balance, and quit.\n");
    END;
  EXCEPT
    | Bank.BadAmount => IO.Put("Can't withdraw or " &
                              "deposit negative amounts.\n");
    | InvalidAccount => IO.Put ("Select an account " &
                              "in the range [" &
                              Fmt.Int(FIRST(Bank.AcctNum)) & ".." &
                              Fmt.Int(LAST(Bank.AcctNum)) & "]" first.\n");
    | FloatMode.Trap, Lex.Error => IO.Put ("Cannot " &
                              "convert the number as specified.\n");
    | Bank.InsufficientFunds => IO.Put ("Insufficient " &
                              "funds available to perform this transaction\n");
  END;
END;
EXCEPT
  | NetObj.Error =>
    IO.Put ("A network object error occurred\n");
  | Thread.Alerted => IO.Put ("A thread was alerted\n");
  | IO.Error, Quit => IO.Put ("Goodbye.\n");
END;
END NetObjClient.

```

Finally, the makefile for a client:

```
import("netobj")
import("netobj-interface") % the common interface
implementation("NetObjClient")
program("netobj-client")
```

6.2 Client/Server Computing: Safe TCP/IP Interfaces

Using CM3-IDE's safe TCP/IP interfaces, it is easy to program multi-threaded TCP clients and servers. Two examples, a Finger client and a simple HTTP server illustrate the use of the TCP interfaces. These same programs will work with Unix sockets or the Windows Winsock libraries without requiring source changes.

6.2.1 A TCP/IP Client: Finger

Finger is a simple program which introduces TCP client services. It also shows you how to bind TCP/IP connections to input and output streams.

```
MODULE Finger EXPORTS Main;
IMPORT TCP, IP, ConnRW;
IMPORT IO, Params;
FROM Text IMPORT FindChar, Sub, Length;
IMPORT Thread; <* FATAL Thread.Alerted *>
```

Common Constants and Variables.

Port 79 is the internet standard for the finger socket port. Variables **user**, and **host** are used by code in this module.

```
CONST
    FingerPort = 79;
VAR
    user := "";
    host := "localhost";
    addr : IP.Address;
```

Command Line Parameters.

Exception **Problem** is used to flag problems with the parameters.

```
EXCEPTION
    Problem;
```

Parse the **user** and **host** from arguments. Raise **Problem** if they're bad.

DEVELOPMENT RECIPES

```
PROCEDURE GetUserHost() RAISES {Problem} =
BEGIN
  IF Params.Count # 2 THEN
    IO.Put ("Syntax: finger user@host\n");
    RAISE Problem;
  END
  IF Params.Count = 2 THEN
    user := Params.Get(1)
  END;
  WITH at = FindChar(user, '@') DO
    IF at = -1 THEN
      host := "localhost";
    ELSE
      host := Sub (user, at+1, LAST(INTEGER));
      user := Sub (user, 0, at);
    END;
  END;
END GetUserHost;
```

Main Implementation.

```
BEGIN
  TRY
    (* Get the values for user and host: *)
    GetUserHost();
    IO.Put ("(Checking for " & user &
      " finger information on host" & host & ")\n");
    (* Lookup host by name: *)
    IF NOT IP.GetHostByName (host, addr) THEN
      IO.Put ("Could not find hostname " &
        host & "\n");
      RAISE Problem;
    END;
    (* Connect to the endpoint at port 79 of host.
      Get a reader and a writer to that port. *)
    VAR
      endpoint := IP.Endpoint {addr, FingerPort};
      service := TCP.Connect(endpoint);
      rd := ConnRW.NewRd(service);
      wr := ConnRW.NewWr(service);
    BEGIN
      (* Send the user name to the writer; read the
        whole response until EOF from the reader *)
      IO.Put (user & "\n", wr);
      WHILE NOT IO.EOF (rd) DO
        IO.Put (IO.GetLine(rd) & "\n")
      END
    END
    (* Check for possible errors. *)
  EXCEPT
    | IO.Error, IP.Error =>
      IO.Put ("Problem communicating with " &
        host & "... \n");
    | Problem => (* Error has already been printed, do
      nothing. *)
  END
END Finger.
```


6.2.2 A TCP/IP Server: HTTPD

The program HTTPD implements a simple HTTP server by using the portable TCP/IP interfaces. The basic outline of the program is simple: After getting a connector, loop and do the following:

1. Use **TCP.Accept** to get a new service.
2. Get a reader and a writer to the service via the **ConnRW** interface.
3. Use **Lex.Match** to ensure that the requests start with a “GET”.
4. The rest of the input from the reader until the end of the line is the path requested by the web browser.
5. Given a path requested by a “GET” message, look in the current directory of your file system for the file in question. So, the URL
http://localhost:80/welcome.html
maps to the following HTTP request to the server running on port 80 of the machine “localhost”:
GET /welcome.html
which maps to the file **welcome.html** in your file system.
6. Open the file, and read its contents.
7. Write the contents to the writer that is hooked up to the network connection. Flush the writer upon completion.
8. Make sure to close the reader, the writer, and the server connection at the bottom of the loop.

Here is the implementation for HTTPD. Review the **TCP** and **IP** interfaces for more information regarding the TCP/IP calls.

```
MODULE HTTPD EXPORTS Main;
IMPORT TCP, IP, ConnRW;
IMPORT Rd, Wr, IO, Lex, FileRd, RdCopy;
IMPORT Thread, OSError, Text, Params, Process, Pathname;
```

Use **http://hostname:80/** to access this server:

```
CONST
  HTTP_Port = 80;
PROCEDURE Error (wr: Wr.T; msg: TEXT)
  RAISES {Thread.Alerted, wr.Failure} =
BEGIN
  wr.PutText (wr, "400 " & msg );
  wr.Flush (wr);
END Error;
```

DEVELOPMENT RECIPES

Create an endpoint on the **HTTP_Port**. Get a connector for the end point, and loop:

- Use **TCP.Accept** to wait for a new connection that can handle calls.
- Create a reader and a writer to the connection.
- Look for a **GET**, and then a path for the request. Parse pathname and print it. If there is a request for root, return a welcome string, otherwise find the file residing in a subdirectory.

Of course, catch all the possible exceptions.

DEVELOPMENT RECIPES

```
VAR
    endpoint := IP.Endpoint {IP.GetHostAddr(), HTTP_Port};
    connector: TCP.Connector;
    server: TCP.T;
    rd: Rd.T; wr: Wr.T;
    path: TEXT;
BEGIN
    TRY
        connector := TCP.NewConnector(endpoint);
        LOOP
            server := TCP.Accept(connector);
            rd := ConnRW.NewRd(server);
            wr := ConnRW.NewWr(server);
            TRY
                TRY
                    Lex.Match (rd, "GET ");
                    path := Lex.Scan (rd);
                    IO.Put ("path=" & path & "\n");
                    IO.Put (Rd.GetLine(rd) & "\n");
                    IF Text.Equal (path, "/") THEN
                        Wr.PutText (wr,
                            "<H1>welcome to our web server!" &
                            "</H1>Try <a href=welcome.html>" &
                            "this link" & "</a>.\n");
                    ELSE
                        WITH rd = FileRd.Open (Text.Sub (path,
                            1, Text.Length(path))) DO
                            TRY
                                RdCopy.ToWriter(rd,wr);
                            FINALLY
                                Rd.Close(rd);
                            END;
                        END
                    END;
                    Wr.Flush (wr); (* so the browser can see
                                   the results. *)
                EXCEPT
                    | Lex.Error => Error (wr,
                        "Only GET methods are supported\n");
                    | OSError.E => Error (wr,
                        "File not found or no permission.\n");
                    | Rd.EndOfFile => Error (wr,
                        "Request terminated prematurely.\n");
                END;
                FINALLY (* clean up on your way out. *)
                    Rd.Close (rd);
                    Wr.Close (wr);
                    TCP.Close (server);
                END;
            END
        EXCEPT
            | Thread.Alerted => IO.Put ("Thread was alerted\n");
            | IP.Error => IO.Put ("IP error\n");
            | Rd.Failure, Wr.Failure =>
                IO.Put ("Rd/Wr failure\n");
        END;
    END HTTPD.
```

6.3 Taking Persistent Snapshots of Objects: Pickles

Pickles can be used to load and save the state of objects via I/O streams bound to disk files, network connections, or in-memory data. To learn more about pickles, browse the **Pickle** interface.

This program uses pickles to snapshot a copy of its internal database to disk, and load it later. The internal database is kept as a list of *atoms*. An atom is a unique representation for a text string.

```
MODULE PickleExample EXPORTS Main;
IMPORT Pickle, wr, Filewr, Rd, FileRd;
IMPORT Atom, AtomList;
IMPORT Action, AtomActionTbl;
IMPORT Process, IO; <* FATAL IO.Error *>
```

Import the **Pickle** interface to take snapshots of objects and turn the snapshots back into live objects, **Wr** and **Filewr** to write snapshots to files, and **Rd**, and **FileRd** to read snapshots from files.

Import **Atom** and **AtomList** interfaces. An **Atom** is unique representation of a string, you can convert text to **Atom** and then compare it with other **Atoms** without using text operations.

Import the **Action** interface, defined in this package, and **AtomActionTbl**, a table mapping atoms to actions.

Atom List Operations. **Contains**, **Insert** and **Print** are utility functions which call **AtomList** operations.

Insert an element into the list.

```
PROCEDURE Insert (VAR list: AtomList.T; atom: Atom.T) =
BEGIN
  IF NOT AtomList.Member(list, atom) THEN
    list := AtomList.Cons (atom, list);
  END
END Insert;
```

Print out all elements of the list by iterating over its members.

```
PROCEDURE Print(x: AtomList.T) =
BEGIN
  WHILE x # NIL DO
    IO.Put (Atom.ToText (x.head) & " ");
    x := x.tail;
  END;
END Print;
```

Command Operations. Definition of what commands should do. **Actions** define initial values for the action table.

```

TYPE
  Commands = {Show, Quit, Reset, Help, Load, Save};

  Actions = ARRAY OF Action.T {
    Action.T { "show", Show},
    Action.T { "quit", Quit},
    Action.T { "reset", Reset},
    Action.T { "help", Help},
    Action.T { "load", Load},
    Action.T { "save", Save}};

```

Each procedure defines what each action should do. Note that **Actions** includes elements that happen to be procedures. Also that the **proc** field of **Action.T** is defined to be a **PROCEDURE()**, so we can assign any of **Quit**, **Reset**, **Help**, or **Show** to fields of **Actions**.

```

PROCEDURE Quit() = BEGIN Process.Exit(0); END Quit;
PROCEDURE Reset() = BEGIN input_set := NIL; END Reset;

PROCEDURE Show() =
BEGIN
  Print(input_set);
  IO.Put ("\n");
END Show;

PROCEDURE Help() =
BEGIN
  IO.Put("Commands: show, reset, " &
        "help, quit, load, save.\n" &
        "Otherwise: insert into the list.\n");
END Help;

```

Procedures **Save** and **Load** use pickles to save and load the database.

```

CONST DB = "db";

PROCEDURE Save() =
  VAR wr := IO.OpenWrite(DB);
BEGIN
  Pickle.Write(wr, input_set);
  wr.Close(wr);
END Save;

PROCEDURE Load() =
  VAR rd := IO.OpenRead(DB);
BEGIN
  input_set := Pickle.Read(rd);
  rd.Close(rd);
END Load;

```

Main Program. The principal data in this program: **command_table** is an atom→action table; **input_set** is an atom list, containing all the elements that will be entered.

```

VAR
  command_table := NEW(AtomActionTbl.Default).init();
  input_set : AtomList.T := NIL;
BEGIN
  (* Initialize Commands. *)
  FOR x := FIRST(Actions) TO LAST(Actions) DO
    EVAL command_table.put(Atom.FromText (x.name), x);
  END;

  IO.Put ("welcome to the atomic database.\n");
  IO.Put ("Try any of commands: show quit reset help.\n");
  IO.Put ("Any other string will be entered into " &
    "the database.\n\n");

```

Loop, get the user response from the command line. If it's a command, do it. Otherwise insert the command line into the **input_set**. If **atom** is in the **command_table** then run the corresponding **action**. Otherwise, **Insert** the **atom** into the **input_set**.

```

LOOP
  IO.Put ("persistent atom-db > ");
  IF IO.EOF () THEN EXIT END;
  VAR
    cmd := IO.GetLine();
    atom := Atom.FromText(cmd);
    action: Action.T;
  BEGIN
    IF command_table.get(atom, action)
    THEN action.proc();
    ELSE Insert(input_set, atom);
    END;
  END;
END;

END PickleExample.

```

6.4 Quick Comparison of Large Data: Fingerprints

You can use the **Fingerprint** interface to compare large amounts of data. Fingerprints can also be used for efficient comparison of complex object graphs.

The program **M3Compare** takes two file names from the command line and reports whether the files are the same or different. The program does not crash due to exceptions.

```

MODULE M3Compare EXPORTS Main;
IMPORT IO, Process, Fingerprint, Rd, Thread, Params;

```

Use `Fingerprint.FromText` to get a fingerprint of each file, then compare the finger prints.

```

PROCEDURE Compare (a, b: TEXT) =
  VAR aa, bb: TEXT;
BEGIN
  aa := Inhale (a);
  bb := Inhale (b);
  IF (aa = NIL) OR (bb = NIL) THEN
    (* already reported an error *)
  ELSIF Fingerprint.FromText(aa) =
    Fingerprint.FromText(bb)
  THEN
    IO.Put ("The files are the same.\n");
  ELSE
    IO.Put ("The files are different.\n");
  END;
END Compare;

```

Read a file and return its contents as text.

```

PROCEDURE Inhale (file: TEXT): TEXT =
  VAR rd: Rd.T; body: TEXT;
BEGIN
  rd := IO.OpenRead (file);
  IF (rd = NIL) THEN
    IO.Put ("\\" & file & "\" is not a file.\n");
    RETURN NIL;
  END;
  TRY
    body := Rd.GetText (rd, LAST (CARDINAL));
    Rd.Close (rd);
  EXCEPT Rd.Failure, Thread.Alerted =>
    IO.Put ("Unable to read\\" & file & "\".\n");
    RETURN NIL;
  END;
  RETURN body;
END Inhale;

BEGIN
  IF Params.Count # 3 THEN
    IO.Put ("syntax: m3compare <file1> <file2>\n");
    Process.Exit(2);
  END;
  Compare (Params.Get(1), Params.Get(2));
END M3Compare.

```

6.5 Portable Operating System Interfaces

Using the portable operating systems interfaces, you can write programs to get information about operating system facilities such as the files, directories, processes, paths, environment variables and command-line parameters. Following the interface specifications, you can write programs that do not depend on idiosyncrasies of different versions of Unix and Windows.

The program **M3Sh**, a simple command-line shell, operates like a normal DOS or Unix command shell, providing you with simple commands. Of course, **m3sh** does not depend on pre-processor macros.

```
MODULE M3sh EXPORTS Main;
```

M3sh is a simple shell utility which uses the safe, portable operating system interfaces. By using the portable interfaces, this program works on both Unix and Win32 platforms.

```
IMPORT Pathname, FS, IO, OSErrors;
IMPORT Stdio, RegularFile, Pipe;
IMPORT Process, Thread, Env, Params;
IMPORT FileWr, FileRd, Rd, Lex, Wr, Text, Atom, AtomList;
IMPORT TextRd, TextSeq;
```

Shell Commands. **Command** designates a **name** and an **action** procedure.

```
TYPE
  Command = RECORD
    name: TEXT;
    action: PROCEDURE (cmd: TEXT;
                       READONLY args: ARRAY OF TEXT
                       ): TEXT RAISES {OSErrors.E};
  END (* RECORD *);
```

Commands is an array of pre-defined **Command** designations for built-in shell commands. To allow aliasing of actions, multiple names may correspond to the same action.

```
CONST
  Commands = ARRAY OF Command {
    Command {"exit", exit},
    Command {"quit", exit},
    Command {"bye", exit},
    Command {"cd", chdir},
    Command {"chdir", chdir},
    Command {"dir", dir},
    Command {"ls", dir},
    Command {"pwd", pwd},
    Command {"directory", dir},
    Command {"type", type},
    Command {"cat", type},
    Command {"exec", exec},
```


DEVELOPMENT RECIPES

```
Command {"bg", background},  
Command {"help", help}};
```

Execute the shell command **cmd** with arguments **args**:

```
PROCEDURE ShellCommand(cmd: TEXT;  
                        READONLY args: ARRAY OF TEXT  
                        ): TEXT RAISES {OSerror.E} =
```

Check to see if **cmd** is a built-in. If **cmd** is not a built-in, then try to execute it.

```
BEGIN  
  FOR i := FIRST(Commands) TO LAST(Commands) DO  
    IF Text.Equal (cmd, Commands[i].name) THEN  
      RETURN Commands[i].action (cmd, args);  
    END;  
  END;  
  RETURN Execute (cmd, args);  
END ShellCommand;
```

DEVELOPMENT RECIPES

Run an external command, returning the result as a text string. See the **Process** interface for more information.

```
PROCEDURE Execute(cmd: TEXT;
  READONLY args: ARRAY OF TEXT
  ): TEXT RAISES {OSerror.E} =
  VAR hrChild, hwChild, hrSelf, hwSelf: Pipe.T;
  VAR result: TEXT := "";
BEGIN
  WITH full = FindExecutable(cmd) DO
    IF full # NIL THEN cmd := full; END;
  END;

  Pipe.Open(hr := hrChild, hw := hwSelf);
  Pipe.Open(hr := hrSelf, hw := hwChild);

  TRY
    WITH p = Process.Create (cmd, args, stdin := hrChild,
      stdout := hwChild, stderr := NIL) DO
      TRY
        TRY hrChild.close(); hwChild.close()
        EXCEPT OSerror.E => (* skip *)
        END;
        (* Here is the actual writing and reading,
           conveniently performed using I/O streams. *)
        WITH wr = NEW(FileWr.T).init(hwSelf),
            rd = NEW(FileRd.T).init(hrSelf) DO

          TRY wr.Close(wr)
          EXCEPT wr.Failure, Thread.Alerted => (*SKIP*)
          END;

          result := Rd.GetText(rd, LAST(INTEGER));

          TRY Rd.Close(rd)
          EXCEPT Rd.Failure, Thread.Alerted => (*SKIP*)
          END
        END;
      FINALLY EVAL Process.Wait(p);
      END
    END
  EXCEPT
    | Rd.Failure, Thread.Alerted => Error ("exec failed");
  END;
  RETURN result;
END Execute;
```

Check the number of arguments and raise **OSerror.E** if the wrong number of arguments are being passed.

```
PROCEDURE ArgCount(READONLY args: ARRAY OF TEXT;
  lo: CARDINAL;
  hi: CARDINAL := LAST(INTEGER)
  ) RAISES {OSerror.E} =
BEGIN
  IF NUMBER(args) < lo THEN Error ("Too few args");
  ELSIF NUMBER(args) > hi THEN Error ("Too many args");
  END;
END ArgCount;
```

Given a string, procedure **Error** raises **OSerror.E** with that string as a parameter.

```
PROCEDURE Error (name: TEXT) RAISES {OSerror.E} =
VAR
    err := AtomList.List2(Atom.FromText(name),
                          Atom.FromText("m3sh error"));
BEGIN
    RAISE OSerror.E(err);
END Error;
```

Built-in Commands. This section includes all the built-in shell commands, such as **dir** or **cd**.

```
PROCEDURE pwd(<*UNUSED*>cmd: TEXT;
              READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSerror.E} =
BEGIN
    ArgCount(args, 0, 0);
    RETURN Process.GetWorkingDirectory();
END pwd;

PROCEDURE dir(<*UNUSED*>cmd: TEXT;
              READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSerror.E} =
VAR
    dir: Pathname.T := ".";
    result: TEXT := "";
    name: TEXT;
    iter: FS.Iterator;
BEGIN
    ArgCount(args, lo := 0, hi := 1);
    IF NUMBER(args) > 0 THEN dir := args[0] END;
    IF NOT IsDirectory (dir) THEN
        Error (dir & " is not a directory");
    END;

    IO.Put ("Directory listing for " &
            FS.GetAbsolutePathname(dir) & "\n");
    iter := FS.Iterate (dir);
    WHILE iter.next (name) DO
        result := result & " " & name & "\n";
    END;
    iter.close();
    RETURN result;
END dir;

PROCEDURE chdir(<*UNUSED*>cmd: TEXT;
                READONLY args: ARRAY OF TEXT
                ): TEXT RAISES {OSerror.E} =
BEGIN
    ArgCount(args, 1, 1);
    IF NOT IsDirectory (args[0]) THEN
        Error (args[0] & " is not a directory\n");
    END;
    Process.SetWorkingDirectory(args[0]);
    RETURN NIL;
END chdir;
```

DEVELOPMENT RECIPES

Display the contents of a file or directory. If **arg[0]** is a file, return its contents. If **arg[0]** is a directory, prints its directory listing.

```
PROCEDURE type(cmd: TEXT;
               READONLY args: ARRAY OF TEXT
               ): TEXT RAISES {OSerror.E} =
VAR rd: Rd.T;
BEGIN
  ArgCount(args, 1, 1);
  IF IsDirectory (args[0]) THEN
    Error (args[0] & " is a directory\n");
  ELSE
    TRY
      rd := FileRd.Open(args[0]);
      TRY RETURN Rd.GetText(rd, LAST(INTEGER));
      FINALLY Rd.Close(rd);
    END;
    EXCEPT
      | Rd.Failure, Thread.Alerted =>
        Error ("type could not read a file\n");
    END;
  END;
  <* ASSERT FALSE *>
END type;

PROCEDURE exit(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
               ): TEXT RAISES {OSerror.E} =
BEGIN
  ArgCount(args, 0, 0);
  IO.Put ("Goodbye!\n");
  Process.Exit(0);
  <* ASSERT FALSE *>
END exit;

PROCEDURE exec(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
               ): TEXT RAISES {OSerror.E} =
BEGIN
  ArgCount(args, 1);
  IO.Put ("The command is " & args[0] & "\n");
  RETURN Execute (args[0],
                  SUBARRAY(args,1, NUMBER(args)-1));
END exec;

PROCEDURE help(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
               ): TEXT RAISES {OSerror.E} =
BEGIN
  ArgCount (args, 0, 0);
  RETURN HelpfulInfo();
END help;
```

Using Threads for the Background Command.

```

PROCEDURE background(<*UNUSED*>cmd: TEXT;
                    READONLY args: ARRAY OF TEXT
                    ): TEXT RAISES {OSError.E} =
VAR background_closure: BgClosure;
BEGIN
  ArgCount(args, 1);
  background_closure := NEW(BgClosure, cmd := args[0],
    args := NEW(REF ARRAY OF TEXT, NUMBER(args)-1));
  background_closure.args^ :=
    SUBARRAY(args, 1, NUMBER(args)-1);
  EVAL Thread.Fork (background_closure);
  RETURN NIL;
END background;

(* closure for the background thread. *)
TYPE
  BgClosure = Thread.Closure OBJECT
    cmd: TEXT;
    args: REF ARRAY OF TEXT;
  OVERRIDES
    apply := BackgroundApply;
  END;

(* work of the background thread. *)
PROCEDURE BackgroundApply (cl: BgClosure): REFANY =
BEGIN
  TRY
    RETURN Execute (cl.cmd, cl.args^);
  EXCEPT OSError.E => (* ignore background errors *)
  END;
  RETURN NIL;
END BackgroundApply;

```

PATH Navigation. Win32 and Unix use the **PATH** variable to define a list of directories to search for executables. Here we search for executables using the **PATH** as our guide.

Finds an executable program found by searching the directories contained in the **PATH** environment variable. **PATH** variable is looked up using the **Env** interface. To look up the separator for **PATH**, we need to find out what sort of system we are running. To do so, we check to see if **Pathname** uses / or \. (See also **SearchPath**.)

```

PROCEDURE FindExecutable (file: TEXT): TEXT =
VAR path := Env.Get ("PATH");
CONST UnixExts = ARRAY OF TEXT { NIL };
CONST winExts = ARRAY OF TEXT { NIL, "exe", "com", "cmd", "bat" };
VAR on_unix: BOOLEAN :=
  Text.Equal(Pathname.Join(""," ",NIL),"/");
BEGIN
  IF on_unix
  THEN RETURN SearchPath (file, path, ':', UnixExts);
  ELSE RETURN SearchPath (file, path, ';', winExts);
  END;
END FindExecutable;

```

DEVELOPMENT RECIPES

Return **TRUE** if the name corresponds to a file.

```
PROCEDURE IsFile (file: TEXT): BOOLEAN =
BEGIN
  TRY
    WITH stat = FS.Status (file) DO
      RETURN stat.type = RegularFile.FileType;
    END
  EXCEPT
    | OSError.E => RETURN FALSE;
  END
END IsFile;
```

Return **TRUE** if the name corresponds to a directory.

```
PROCEDURE IsDirectory (file: TEXT): BOOLEAN =
BEGIN
  TRY
    WITH stat = FS.Status (file) DO
      RETURN stat.type = FS.DirectoryFileType;
    END
  EXCEPT
    | OSError.E => RETURN FALSE;
  END
END IsDirectory;
```

DEVELOPMENT RECIPES

Search the items passed in as part of path for the file.

```
PROCEDURE SearchPath (file, path: TEXT;
                     sep: CHAR;
                     READONLY exts: ARRAY OF TEXT
                     ): TEXT =
VAR
  dir, fn: TEXT;
  s0, s1, len: INTEGER;
  no_ext: BOOLEAN;
BEGIN
  IF IsFile (file) THEN RETURN file; END;
  no_ext := Text.Equal (file, Pathname.Base (file));

  (* First try the file without looking at the path. *)
  IF no_ext THEN
    FOR i := FIRST (exts) TO LAST (exts) DO
      fn := Pathname.Join (NIL, file, exts[i]);
      IF IsFile (fn) THEN RETURN fn; END;
    END;
  END;

  IF path = NIL THEN RETURN NIL; END;
  IF Pathname.Absolute (file) THEN RETURN NIL; END;

  (* Try the search path *)
  len := Text.Length (path); s0 := 0;
  WHILE (s0 < len) DO
    s1 := Text.FindChar (path, sep, s0);
    IF (s1 < 0) THEN s1 := len; END;
    IF (s0 < s1) THEN
      dir := Text.Sub (path, s0, s1 - s0);
      IF no_ext THEN
        FOR i := FIRST (exts) TO LAST (exts) DO
          fn := Pathname.Join (dir, file, exts[i]);
          IF IsFile (fn) THEN RETURN fn; END;
        END;
      ELSE
        fn := Pathname.Join (dir, file, NIL);
        IF IsFile (fn) THEN RETURN fn; END;
      END;
    END;
    s0 := s1 + 1;
  END;

  (* SearchPath failed. *)
  RETURN NIL;
END SearchPath;
```

The main program and utility procedures.

```

PROCEDURE HelpfulInfo(): TEXT =
CONST
    Msg = "m3sh: a simple portable shell for POSIX and" &
          "Win32 written in Modula-3\n" &
          "syntax: m3sh [-prompt string | -help]\n" &
          "commands:";
VAR
    result := Msg;
BEGIN
    FOR i := FIRST(Commands) TO LAST(Commands) DO
        result := result & " " & Commands[i].name;
    END;
    RETURN result & "\n";
END HelpfulInfo;

VAR
    prompt: TEXT := "m3sh";

```

Echo the prompt. Get a command. If the command is not null, then execute it, and print its results on the screen.

```

PROCEDURE ProcessCommand()
    RAISES {OSerror.E, Rd.EndOfFile} =
VAR
    cmdname: TEXT; (* name of the command *)
    cmdargs: REF ARRAY OF TEXT; (* arguments of the command *)
    result: TEXT;
BEGIN
    IO.Put(prompt & "> ");
    GetCommand(cmdname, cmdargs);
    IF cmdname = NIL THEN RETURN END;
    result := ShellCommand (cmdname, cmdargs^);
    IF result # NIL THEN IO.Put (result & "\n") END;
END ProcessCommand;

```


DEVELOPMENT RECIPES

Read a command line; affect variables **name** and **args**. Set **name** and **args** to **NIL** if there is no input in this line. Raise **Rd.EndOfFile** if the end of file is reached.

```
PROCEDURE GetCommand (VAR name: TEXT;
                     VAR args: REF ARRAY OF TEXT
                     ) RAISES {Rd.EndOfFile} =
VAR
  cmd := NEW(TextSeq.T).init();
  rd: Rd.T;
BEGIN
  name := NIL; args := NIL;
  TRY
    (* Read a line and map it to the reader "rd". *)
    rd := TextRd.New(Rd.GetLine(Stdio.stdin));

    (* Tokenize the line into a sequence of strings. *)
    TRY WHILE NOT Rd.EOF(rd) DO
      Lex.Skip(rd);
      cmd.addhi(Lex.Scan(rd)); END;
    EXCEPT Rd.Failure => (* do nothing *)
    END;

    (* Turn the sequence into a (command, arguments) pair. *)
    IF cmd.size() = 0 THEN RETURN END;
    name := cmd.get(0);
    args := NEW(REF ARRAY OF TEXT, cmd.size()-1);
    FOR i := FIRST(args^)^ TO LAST(args^)^ DO
      args[i] := cmd.get(i+1);
    END;
  EXCEPT
    | Rd.Failure, Thread.Alerted =>
      IO.Put ("Problems in reading from input\n");
  END;
END GetCommand;
```

Print arguments to an **OSError.E**. Used by the main shell loop to print out errors.

```
PROCEDURE PrintError (al: AtomList.T) =
BEGIN
  WHILE al # NIL DO
    IO.Put (Atom.ToText(al.head) & ". ");
    al := al.tail;
  END;
  IO.Put ("\n");
END PrintError;
```

DEVELOPMENT RECIPES

Check the command-line parameters.

```
PROCEDURE ProcessParams() =
BEGIN
  CASE Params.Count OF
    | 1 => RETURN;
    | 2 => IF Text.Equal(Params.Get(1), "-help") THEN
        IO.Put (HelpfulInfo());
        RETURN
      END;
    | 3 => IF Text.Equal(Params.Get(1), "-prompt") THEN
        prompt := Params.Get(2);
        RETURN
      END;
  ELSE (* skip *)
  END;
  IO.Put ("Incorrect or bad number of parameters." &
    " Try -help to get more info.\n");
  Process.Exit(10);
END ProcessParams;
```

The main loop.

```
BEGIN
  ProcessParams();
  LOOP
    TRY
      ProcessCommand();
    EXCEPT
      | OSError.E (e) => PrintError(e);
      | Rd.EndOfFile => EXIT;
    END;
  END;
END M3sh.
```

Further Information. To learn more about operating system interfaces see **CM3-IDE Interface Index** on page 143, or interface definition for:

- **Process** interface for process management
- **Thread** interface for creating threads or “lightweight processes”
- **FS** interface for access to files and directories
- **Pathname** interface for manipulating pathnames in a portable fashion
- **Env** interface for environment variables
- **Params** interface for command-line parameters
- **OSError** interface for handling operating system errors

6.6 Dynamic Web Applications: the Web Server Toolkit

The web server toolkit defines a framework for building dynamic web servers. The program **WebContact** illustrates a simple web-based application of a dynamic contact database.

```
MODULE WebContact EXPORTS Main;
IMPORT HTTPApp, HTTPControl, HTTPControlValue, App;
IMPORT Text, TextTextTbl;
FROM IO IMPORT Put;
```

Create two fields for names and e-mail addresses. Each is displayed on an automatically generated form, and the call-back procedures are run when the form is submitted.

```
VAR
  name, email: TEXT := "";
```

Define a text control, **name_value**, and its **Get** and **Set**, and **Default** operations.

```
VAR
  name_value := NEW(HTTPControlValue.TextValue,
                    leader := "<strong>Name: </strong>",
                    id := "name", set := SetName,
                    get := GetName,
                    setDefault := Default);

PROCEDURE SetName(<*UNUSED*>self: HTTPControlValue.TextValue;
                 val: TEXT;
                 <*UNUSED*>log: App.Log) =
BEGIN
  name := val;
  IF NOT db.get(name, email) THEN
    email := "";
  END;
END SetName;

PROCEDURE GetName(<*UNUSED*>self: HTTPControlValue.TextValue
                 ): TEXT =
BEGIN
  RETURN name;
END GetName;
```

DEVELOPMENT RECIPES

Define another text control `email_value`, and its `Get`, `Set`, and `Default` operations.

```
VAR
    email_value := NEW(HTTPControlValue.TextValue,
                        leader := "<strong> Email:</strong>",
                        id := "email", set := SetEmail,
                        get := GetEmail,
                        setDefault := Default);

PROCEDURE GetEmail (self: HTTPControlValue.TextValue
): TEXT =
BEGIN
    RETURN email;
END GetEmail;

PROCEDURE SetEmail (self: HTTPControlValue.TextValue;
                    val: TEXT;
                    log : App.Log) =
BEGIN
    IF Text.Empty (val) THEN
        IF db.get(name, email) THEN
            email := "";
        END;
    ELSE
        EVAL db.put(name, val);
    END;
END SetEmail;

PROCEDURE Default (<*UNUSED*>x: HTTPControlValue.TextValue;
<*UNUSED*>log: App.Log) =
BEGIN
END Default;
```

The variable `root` defines the root of the HTTP server. `db` is a text-to-text table for mapping names to email addresses.

```
VAR
    root : HTTPControl.StaticForm := HTTPControl.RootForm();
    db := NEW(TextTextTbl.Default).init();
BEGIN
```

Initialize root default options.

```
    root.hasSubmitButton := TRUE;
    root.title := "Contact Database";
```

Add a title.

```
    root.addValue(NEW(HTTPControlValue.MessageValue).init(
        "\n" & "<H2>Contact Database</H2>"));
```

Add the two text fields.

```
root.addValue(name_value);
root.addValue(email_value);
```

Serve at port 80. If there is a problem, report it.

```
TRY
  HTTPApp.Serve(80);
EXCEPT
  App.Error => Put ("A problem occurred\n");
END;
END WebContact.
```

6.7 Interacting with C Programs

Most real programs need to interact with an existing body of code. Since CM3-IDE has provisions for describing unsafe operations, binding programs written using CM3-IDE with C is straightforward. In this section, we will describe two programs that call C code on Unix or Win32 platforms, and a Modula-3 program that is called from C.

6.7.1 Calling C: A Unix Example

In this example, we create an interface for accessing the `getcwd` function from Modula-3.

We then wrap a safe interface around the unsafe layer that calls C. This example only works on Unix, but a similar example can be written for Win32 as you will see later. Of course, if we were to only use Modula-3 facilities, the code could easily be ported.

The basic steps in writing this program are:

1. First, read the Unix man page on “`getcwd`” to get some information about its parameter, and what it does.
2. Interface `Ulib` contains the `<*EXTERNAL*>` Modula-3 signature for the “`getcwd`” function in our project. If we are to call this from client code, we’d have to make that unsafe, and have to deal with C data structures, which is probably not a good idea. So, in the next step, we build a safe wrapper around the C call.
3. Create an interface and an implementation “`Lib`”. This will be the Modula-3 wrapper for `Ulib`. The idea here is to create a function, `GetCWD()` which returns a `TEXT` containing your current working directory. `Lib.i3` should be pretty straightforward. All you do is declare the signature of the function.

4. **Lib.m3** is more subtle. What we need to do is allocate some space for the C buffer, and then pass it to “**getcwd**”, finally copy the contents of the **getcwd** buffer back into a **TEXT** and return it.

We can either use **Cstdlib.malloc** for allocating the right buffer, and **Cstdlib.free** to dispose it after copying the buffer into a **TEXT** via **M3toC.CopyStoT**.

5. Create a safe main module and call **Lib.GetCWD()** from it.

The good news is that most of the time, we can program in the safe mode, where the language takes care of things like garbage collection. This eliminates the need for separating safe and unsafe code.

Safe interface. Interface **Lib** provides a safe **GetCWD** interface. This means its implementation must have to deal with bridging from unsafe operations to safe operations.

```
INTERFACE Lib;
PROCEDURE GetCWD(): TEXT;
END Lib.
```

Unsafe implementation of a safe interface. The implementation of **Lib** interface includes the body of **GetCWD**, which calls **malloc** to allocate a string, sends it to **getcwd**, and converts the result to **TEXT** while handling memory management.

```
UNSAFE MODULE Lib;
IMPORT Ulib, M3toC;
FROM Ctypes IMPORT char_star;
FROM Cstdlib IMPORT malloc, free;

PROCEDURE GetCWD(): TEXT =
CONST size = 64;
VAR c_str := malloc (size);
BEGIN
  EVAL Ulib.getcwd(c_str,size);
  WITH result = M3toC.CopyStoT(c_str) DO
    free(c_str);
  RETURN result;
  END;
END GetCWD;

BEGIN
END Lib.
```

Unsafe interface to Unix libraries. Interface **Ulib** defines an external function `getcwd`.

```
INTERFACE Ulib;
FROM Ctypes IMPORT char_star, int;

<*EXTERNAL*>
PROCEDURE getcwd(result: char_star;
                 size: int
                 ): char_star;

END Ulib.
```

The main module. The main body of the code is simple, because **Lib** takes care of bridging the safety gap.

```
MODULE CallingC EXPORTS Main;
IMPORT IO, Lib;
BEGIN
  IO.Put (Lib.GetCWD() & "\n");
END CallingC.
```

Makefile. The makefile is quite ordinary.

```
import("libm3")
interface("Ulib")
module("Lib")
implementation ("CallingC")
program ("m3pwd")
```

6.7.2 Calling C: A Win32 Example

In this example, we create an interface for accessing the **MessageBox** function from the Win32 API. To do so, we import the interface **WinUser** which defines the signature of the **MessageBoxA** call. We then call **WinUser.MessageBox** from the main module, OK.

This example only works on Win32, since **MessageBox** is a Win32 call. Since this call is not available on other platforms your program is not portable. Of course, if you were to only use the portable interfaces available in Modula-3, you would not have any portability problems.

WinUser defines basic Win32 API user-level calls. **M3toC** defines mappings from Modula-3 to C strings.

OK.m3.

```

UNSAFE MODULE OK EXPORTS Main;
IMPORT WinUser, M3toC;
IMPORT Params;
VAR
    message: TEXT := "";
BEGIN
    FOR i := 1 TO Params.Count-1 DO
        message := message & Params.Get(i) & " ";
    END;
    EVAL WinUser.MessageBox(NIL,
                            M3toC.TtoS(message),
                            M3toC.TtoS("A CM3-IDE Example"),
                            0);
END OK.

```

Here is the portion of the **WinUser** interface where **MessageBox** is defined:

```

INTERFACE WinUser;
...
PROCEDURE MessageBoxA (hwnd: HWND;
                       lpText : LPCSTR;
                       lpCaption: LPCSTR;
                       uType : UINT
                       ): int;

CONST MessageBox = MessageBoxA;
...
END WinUser.

```

6.7.3 Calling Modula-3 from C

This example demonstrates how to call Modula-3 procedures from C. The C procedure in the example takes a single parameter which itself is a parameter-less procedure that returns an integer. Have the C function call the passed the procedure, add one to the result and return the new value. The makefile will assume that the C code is in a file named **Cstuff.c**.

```

INTERFACE Cstuff;

TYPE
    IntProc = PROCEDURE (): INTEGER;

<*EXTERNAL*>
PROCEDURE add_one (p: IntProc): INTEGER;
(* Returns "1 + p()". *)

<*EXTERNAL*>
PROCEDURE add_one_again (): INTEGER;
(* Returns "1 + m3_proc()". *)

<*EXTERNAL*>
VAR m3_proc: IntProc;

END Cstuff.

```


DEVELOPMENT RECIPES

```
MODULE CcallsM3 EXPORTS Main;
IMPORT IO, Cstuff;
VAR
    x: INTEGER := 33;
    i: INTEGER;

PROCEDURE Foo (): INTEGER =
BEGIN
    INC (x);
    RETURN x;
END Foo;

BEGIN
    IO.Put ("calling add_one.\n");
    i := Cstuff.add_one (Foo);
    IO.Put ("add_one () => ");
    IO.PutInt (i);
    IO.Put ("\n");
    IO.Put ("calling add_one_again.\n");
    Cstuff.m3_proc := Foo;
    i := Cstuff.add_one_again ();
    IO.Put ("add_one_again () => ");
    IO.PutInt (i);
    IO.Put ("\n");
END CcallsM3.
```

Cstuff.c calls (and is called by) the Modula-3 code.


```
#include <stdio.h>
typedef int (*PROC)();
int add_one (p)
    PROC p;
{
    int i;
    printf ("in add_one, p = 0x%x\n", p);
    i = p ();
    printf (" p() => %d\n", i);
    return i+1;
}

PROC m3_proc;
int add_one_again ()
{
    int i;
    printf ("in add_one_again, m3_proc = 0x%x\n", m3_proc);
    i = m3_proc ();
    printf (" m3_proc () => %d\n", i);
    return i+1;
}
```

Makefile. The call **c_source** compiles a C program with your C compiler. See the Operations Guide for **cm3** at </help/cm3/cm3.html> or the **cm3.cfg** file in your installation for more information.

```
import ("libm3")
c_source ("Cstuff")
interface ("Cstuff")
implementation ("CcallsM3")
program ("ccallsm3")
```

6.8 Summary

This chapter described a handful of complete programs to illustrate the use of advanced programming facilities in CM3-IDE. You can find the sources for the programs in this chapter in the  Examples section of your CM3-IDE environment.

Robust Distributed Applications. Network Objects can be used to build robust distributed applications. See the **NetObj** interface for more information (See page 108).

Client/Server Computing. Using the safe TCP/IP interfaces, you can build multi-threaded client/server applications that use the socket interfaces. TCP/IP interfaces abstract away the differences between Unix sockets and Winsock implementations. See the **TCP**, **IP**, and **ConnRW** interfaces for more information (See page 115).

Data Manipulation. Using the **Pickle** interface, you can take snapshot of complex object graphs, and later load them into memory. The **Fingerprint** interface allows you to compare large data structures efficiently. The **IO**, **Rd**, **Wr**, **Lex**, **Scan**, and **Fmt** interfaces help you read and write from I/O streams (See page 120).

Portable Operating System Interfaces. CM3-IDE provides interfaces for accessing operating system services in a platform-neutral manner. See the **Process** interface for managing processes, **File**, **FS**, **FileWr**, **FileRd** for filesystem access, **Thread** for creating new threads and concurrency control, **Params** for command-line parameters, **Env** for environment variables, and **Time** for the system clock. Using these interfaces, you can write portable programs that access various operating system facilities (See page 124).

Dynamic Web Applications. You can build dynamic web applications using the web toolkit. Read the **HTTPApp** interface as a start (See page 135).

Accessing Legacy C code. Binding unsafe portions of your program to C code is straightforward, but tedious. To aid portability and robustness of your application, you should avoid using legacy C code as much as possible (See page 137).

7. CM3-IDE Interface Index

Read this chapter to learn about the most-frequently-used interfaces in CM3-IDE.



CM3-IDE includes many interfaces. Finding the right one for a particular task is not always easy. This index provides an overview of some of the most frequently used interfaces available in CM3-IDE.

Data types, Data Structures, and Algorithms on page 144 outlines:

- Basic Data Types
- Collections, Lists, Tables, Sets
- Linked Lists, Sorted Linked Lists
- Property lists
- Sequences
- Priority queues
- Sets
- Tables, Sorted tables
- Sorting Lists, Tables, and Arrays.

Standard Libraries on page 148 describes:

- Math, Geometry, Statistics, Random numbers
- Floating point
- Environment, Command line parameters
- I/O streams, Reading and Writing, Files
- Formatting, I/O Conversion
- Threads.

Systems Development on page 150 outlines:

- Distributed and Client/Server Development
- Databases and Persistence
- Operating System, Files, Processes, Time
- Interoperability with C
- Low-level Run-time Interfaces.

Miscellaneous on page 153 describes:

- Main interface, Weak References, Performance Tuning, Configuration.

7.1 Data Types, Data Structures, and Algorithms

7.1.1 Basic Data Types

The following table maps basic data types to the corresponding interfaces:

Data Type	Interface
INTEGER	Integer, Int32
BOOLEAN	Boolean
CHAR	Char
REFANY	Refany
REAL	RealType, Real
LONGREAL	LongrealType, LongReal
EXTENDED	Extended
TEXT	Text

The on-line CM3-IDE interface index includes hypertext references to CM3-IDE interfaces. You can find it at: </help/interfaces.html>

The list of all available interfaces in your CM3-IDE distribution is available at: </interface>

To locate a particular interface within CM3-IDE, find </interface/name> for example: </interface/Text>

Interfaces for built-in types are used mainly as arguments for instantiating generic collections.

Text Strings. Strings are represented as values of type **TEXT**. Text strings are immutable; they are automatically garbage collected.

The **TEXT** type is used extensively throughout Modula-3 libraries. The **Text** interface defines the basic operations on this type. Operations to convert between other encodings of text strings are available in the **TextConv** interface. The internal representation of text strings is exposed by **TextF**. You should avoid using the **TextF** interface whenever possible.

ASCII Characters. ASCII includes constant definitions for the character codes of non-printable characters, such as **ASCII.NL** for new-line. It classifies characters into groups, like digits or punctuation; each group is represented as a set of characters. Finally, it provides mapping tables that translate lower-case letters into upper-case and vice versa.

Machine Words and Bit Manipulation. **Word** allows bit manipulation on machine words; **Swap** is useful for writing code that must deal explicitly with byte ordering and/or word length issues.

Atoms. While not built-in types, atoms are handy for efficient comparison of text strings. The **Atom** interface describes the set of operations available for atoms.

Symbolic Expressions. The **SX** interface provides symbolic expressions represented as a recursive linked list structure, similar to Lisp systems. **SX** includes routines for reading and printing symbolic expressions, as well as some convenience procedures for manipulating them.

See also **Formatting, I/O Conversion** on page 149 for interfaces that produce or parse text representations of the built-in types.

7.1.2 Collections, Lists, Tables, Sets

CM3-IDE libraries contain interfaces supporting the following data structures: linked lists, sorted linked lists, Lisp-like property lists, tables, sorted tables, sequences, priority queues, and sets.

Many of these data structures are available as generic interfaces. They can be instantiated with whatever types you need. For most of the generic data structures, the libraries already include instances for text strings, integers, atoms, and arbitrary references. Many of the generics are packaged with makefile commands that make it simply a matter of writing a line in your makefile to instantiate them.

7.1.3 Linked Lists

A generic implementation of singly-linked lists is available in the **List** interface, and implementation. There are predefined instances for atoms, integers, references, and text strings, as well as makefile commands to create custom lists.

7.1.4 Sorted Linked Lists

Lists may be sorted by using the generic **ListSort** interface and implementation. Like the **List** interfaces, there are predefined instances for atoms, integers, references, and text strings.

7.1.5 Property lists

Property lists, simple linked lists of (name, value) pairs are available from the **Property** interface. The related interfaces, **PropertyV**, **MProperty**, **PropertyF**, and **MPropertyF** provide more features.

Tables

A flexible and highly-reusable generic **Table** interface provides efficient mappings from values of one type to values of another. Like the list interfaces, the table interfaces are provided with predefined instances for the full cross product of the four basic types:

From \ To	atoms	integers	references	texts
atoms	AtomAtomTbl	AtomIntTbl	AtomRefTbl	AtomTextTbl
integers	IntAtomTbl	IntIntTbl	IntRefTbl	IntTextTbl
references	RefAtomTbl	RefIntTbl	RefRefTbl	RefTextTbl
texts	TextAtomTbl	TextIntTbl	TextRefTbl	TextTextTbl

Predefined makefile commands are available to create custom tables.

7.1.6 Sorted tables

Sorted tables are like tables with the addition of operations to iterate through the elements of the table in a sorted order. The generic **SortedTable** interface and implementation are available. Like the other table interfaces, the sorted table interfaces are provided with predefined instances for the full cross product of the four basic types:

From \ To	atoms	integers	references	texts
atoms	Sorted-AtomAtomTbl	Sorted-AtomIntTbl	Sorted-AtomRefTbl	Sorted-AtomTextTbl
integers	Sorted-IntAtomTbl	Sorted-IntIntTbl	Sorted-IntRefTbl	Sorted-IntTextTbl
references	Sorted-RefAtomTbl	Sorted-RefIntTbl	Sorted-RefRefTbl	Sorted-RefTextTbl
texts	Sorted-TextAtomTbl	Sorted-TextIntTbl	Sorted-TextRefTbl	Sorted-TextTextTbl

Predefined makefile commands are available to create custom sorted tables.

7.1.7 Sequences

The **Sequence** interface and implementation provide generic extensible arrays. Elements can be added or removed from either end or directly indexed.

The **SequenceRep** interface exposes the full details of the underlying representation of sequences. For maximum portability and implementation independence, programs should avoid using **SequenceRep**.

Predefined instances for atoms, integers, references, and text strings are available, so are the makefile commands to create custom sequences.

7.1.8 Priority queues

Priority queues, or sequences that keep their elements sorted, are available in the generic **PQueue** interface and implementation. The standard predefined instances for atoms, integers, references, and text strings are available.

When it is necessary to access the underlying implementation, the **PQueueRep** interface defines the full details. For maximum portability and implementation independence, programs should avoid using this interface. Predefined instances for atoms, integers, references, and text strings are available, as well as makefile commands to create custom priority queues.

7.1.9 Sets

Sets are collections of values without duplicates. A generic **Set** interface and implementation are available. Sets are implemented using two implementation strategies: **SetDef** uses a hash-table, and **SetList** uses a list representation for the set.

generic	integer	text	reference	atom
Set	IntSet	TextSet	RefSet	AtomSet
SetDef	IntSetDef	TextSetDef		AtomSetDef
SetList	IntSetList	TextSetList	RefSetList	AtomSetList

Predefined makefile commands are available to create custom sets.

See also **Atoms** on page 144 for an overview of the **Atom** interface, which provides a unique value for all equal text strings.

7.1.10 Sorting Lists, Tables, and Arrays

Sorted generic lists, tables, and arrays allow iteration in a sorted order.

SortedLists. The generic interface **ListSort**, implemented by generic module **ListSort** extends the **List** interface. As usual, there are instantiations **AtomListSort**, **IntListSort**, **RefListSort**, and **TextListSort**.

SortedTables. The interface **SortedTable** allows you to iterate through tables in a sorted order.

SortedArrays. `ArraySort` works similarly, but for arrays; it is instantiated as `IntArraySort` and `TextArraySort`.

7.2 Standard Libraries

7.2.1 Math, Geometry, Statistics, Random numbers

Modula-3 provides a rich set of interfaces for mathematical and statistical programming. The **Math** interface provides access to the C math libraries. Many geometric abstractions are also available: **Axis**, **Interval**, **Point**, **Rect**, **Transform**, **Path**, **Region**, **PolyRegion**, **Trapezoid**.

The generic interface **Sqrt** defines a square root operation, instantiated as **RealSqrt** and **LongSqrt** for **REAL** and **LONGREAL**s. The interface **Stat** defines a set of tools for collecting elementary statistics of a sequence of real quantities. The interface **Random** and **RandomPerm** provide random permutations of numbers. **RandomReal** includes machine specific algorithms for generating random floating-point values.

7.2.2 Floating point

Real, **LongReal**, and **Extended** are interfaces corresponding to the built-in floating-point types; their representations are in **RealRep** and **LongRealRep**.

The interface **FloatMode** allows you to test the behavior of rounding and of numerical exceptions. On some platforms it also allows you to change the behavior, on a per-thread basis.

The interface **FloatExtras**, **RealFloatExtras**, and **LongFloatExtras** contain miscellaneous functions useful for floating point arithmetic. The generic interface **Float** and its instantiations **RealFloat**, **LongFloat**, and **ExtendedFloat** provide interfaces to floating-point arithmetic.

IEEESpecial defines variables for the IEEE floating-point values **-infinity**, **+infinity**, and **NaN** (not a number) for each of the three floating-point types.

7.2.3 Environment, Command line parameters

The **Env** and **Params** interfaces provide access to the environment variables and command-line parameters given to a process when it is started.

See also the **PROCESS** interface in **Processes, Pipes, O/S Errors** on page 152.

7.2.4 I/O streams, Reading and Writing, Files

I/O Streams allow you to read and write to disk, network, another thread, another process, etc.

Basic Input and Output. **IO** interface is a simple high-level I/O interface. **Stdio** declares the standard input, output, and error streams.

Input and Output Streams. **Rd** is the interface for input streams, known as readers. **RdClass** allows you to create new kinds of readers. **UnsafeRd** is an internal interface, providing non-serialized access to readers.

Wr is the safe interface to output streams, or writers. **WrClass** can be used to implement new streams. **UnsafeWr** allows unserialized access to a writer.

File Streams. **FileRd** and **FileWr** read from and write to files.

Text Streams. **TextRd** and **TextWr** read from and write to **TEXT** strings. They are designed for applying string procedures to streams or stream operations on strings.

Empty Streams. **NullRd** and **NullWr** represent empty streams.

Message Streams. **MsgRd** and **MsgWr** present message stream abstractions. A message is a sequence of bytes terminated by an end of message marker.

Stream and File Utilities. **TempFiles** creates temporary files which get deleted automatically upon termination of the process. **RdCopy** copies from readers to writers efficiently. **AutoFlushWr** flushes the output in the background. **RdUtils** adds a few utilities for manipulating readers.

7.2.5 Formatting, I/O Conversion

Most formatting interfaces in CM3-IDE work with strings, readers, and writers. **Fmt** formats basic data types to strings. **Scan** converts strings into basic data types. **FmtTime** returns a string denoting the current date and time. **FmtBuf** is similar to **Fmt** interface, with the exception that it uses character buffers instead of **TEXT** strings. **FmtBufF** exposes its representation.

Lex provides lexical operations for reading tokens and basic datatypes, and matching or skipping blanks from a reader. **Formatter** performs pretty-printing, the printing of structured objects with appropriate line breaks and indentation.

Convert converts binary and ASCII representation of basic values. **CConvert** provides lower-level access to the conversion functions in C.

7.2.6 Threads

CM3-IDE provides language-level support for multi-threaded applications. CM3-IDE's runtime and standard libraries on all platforms are multi-threaded. The **Thread** interface describes the portable interface for creating new threads (also called light-weight processes.) Interfaces **Scheduler**, **ThreadF**, and **ThreadContext** provide access to the internal representation of threads and some control over the thread scheduler.

7.3 Systems Development

7.3.1 Distributed and Client/Server Development

Network Objects. CM3-IDE's Network Objects system allows an object to be handed to another process in such way that the process receiving the object can operate on it as if it were local. The holder of a remote object can freely invoke operations on that object just as if it had created that object locally. Further, it can pass the object to other processes.

NetObj is the basic interface for defining network objects. A few makefile commands help you integrate network objects within your programs. The **NetObjNotifier** interface notifies a server if its clients die. **StubLib** contains procedures to be used by stub code for invoking remote object methods and servicing remote invocations.

The current implementation of Network Objects is built on top of TCP.

TCPNetObj implements network objects on top of TCP/IP. Network Objects are designed to make adaptation to specialized network protocols easy.

Network Streams. Network Streams provide a set of high-level abstractions for sending and receiving messages across the network. **ConnRW** creates reader and writer streams from a connection; **ConnMsgRW** creates message streams from a connection.

TCP / IP Socket Interfaces. Using the TCP/IP interfaces, you can write safe, multi-threaded clients and servers for client/server computing. The same programs work whether you use Unix sockets or Windows winsock. The interface **IP** defines the addresses used for communicating with the internet protocol family. **TCP** provides bidirectional byte streams between two programs, implemented using internet protocols. **TCPSpecial** is a utility interface.

7.3.2 Databases and Persistence

Databases, Persistent Storage. CM3-IDE includes a number of facilities for saving data in persistent forms: Relational Databases, Pickles, Simple Snapshot Persistence, Stable Objects, and Bundles.

Relational Database Interface. The **DB** interface provides serialized access to relational databases. **DB** allows multiple connections within one application and each may be used concurrently by multiple threads. An implementation based on Microsoft's ODBC is available for both Windows and Unix; a sample implementation for Postgres'95 on Unix is also included. You can modify the backend of the interface to suit any relational database.

Pickles: ObjectTranscription (or "Serialization"). The **Pickle** interface provides operations for reading and writing arbitrary values as streams of bytes. Writing a value as a pickle and then reading it back produces a value equivalent to the original value. In

other words, pickles preserve value, shape and sharing. You can write pickles for values that have cyclic references (such as doubly-linked lists), or that are arbitrary graph structures.

Two implementations of the **Pickle** interface are available. **Pickle2** is an implementation geared toward heterogeneous platforms; it works across platforms, reconciling machine word encoding, little-endian, big-endian, size differences. **Pickle** is a more efficient implementation for homogeneous setups. Pickles are used by Network Objects for transferring objects across processes.

SmallDB: Simple Snapshot Persistence. **SmallDB** stores objects in a file in a recoverable fashion. If a crash occurs while the objects are being written to disk, their state can be restored from the latest consistent snapshot the next time they are used. For binary snapshots, use the combination of **SmallDB** and **Pickle**.

Stable Objects. Stable Objects extends the lightweight object storage provided by **Pickles** and **SmallDB** to allow for recoverable storage of objects through logging and check-pointing. Updates to objects are logged to stable storage automatically. When the state of an object is restored from disk, the restoration process checks to see if a crash occurred before the entire state of the object was written to disk. If so, the state of the object is recovered from the log of modifications to the object.

The generic interface **Stable** defines a subtype of a given object type that is just like that object type, but stable. Makefile operations are provided to create stable versions arbitrary object types. **LogManager** manages readers and writers for the log and checkpoint files used by stable objects. **StableRep** defines the representation of stable objects. **Log** provides debugging operations for the log. **StableLog** contains procedures for reading and writing logs for stable objects.

Logs are written on readers and writers. **StableError** defines the various error scenarios and corresponding exceptions.

Bundles. Bundles package up arbitrary files at compile-time so that their contents can be retrieved by a program at run-time without accessing the file system. The interface **Bundle** allows runtime access to the stored data.

You can bundle files with your program by using operations defined in the bundle makefile templates. **BundleRep** exposes the representation of bundles.

7.3.3 Operating System, Files, Processes, Time

CM3-IDE provides a set of high-level, portable interfaces to the underlying OS facilities such as files, processes, directories, terminals, and keyboards. The interfaces to these operating system functions are identical whether you are running on Windows or Unix.

See also **Microsoft Windows** on page 152 and **Unix** on page 153 for lower-level, non-portable interfaces to operating system services.

File-system Interfaces. **File** defines a source and/or sink of bytes. File handles provide an operating-system independent way to perform raw I/O. For buffered I/O, use the **FileRd** and **FileWr** interfaces instead.

Pathname defines procedures for manipulating pathnames in a portable fashion. The **FS** interface provides access to persistent storage (files) and naming (directories).

RegularFile defines regular file handles which provide access to persistent extensible sequences of bytes—usually disks. A **Terminal** handle is a file handle that provides access to a duplex communication channel usually connected to a user terminal.

Processes, Pipes, O/S Errors. The **PROCESS** interface manages operating-system processes (e.g., creating processes, awaiting their exit). A **Pipe** is a file handle that can be used to communicate between a parent and a child process or two sibling processes.

OSError defines an exception raised by a number of operating system interfaces.

Time, Date, Ticks. **Time** defines a moment in time, reckoned as a number of seconds since some epoch or starting point. **Date** defines a moment in time, expressed according to the standard (Gregorian) calendar, as observed in some time zone. **Tick** defines a value of a clock with sub-second resolution, typically one sixtieth of a second or smaller.

Timestamps, Capabilities, Fingerprints. Timestamps provided by the **TimeStamp** interface are totally ordered in a relation that approximates the real time when the value was generated. If two timestamps are generated in the same process then the ordering of the timestamps is consistent with the order that **TimeStamp.New** was called. The interface **Capability** defines unique global identifiers that are extremely difficult for an adversary to guess. The **Fingerprint** interface allows efficient comparison of large strings, and more general data structures such as graphs. **MachineID** returns a unique number for the machine running the Modula-3 program.

Platform-Specific Interfaces. Not all programs are portable. CM3-IDE allows access to lower-level interfaces available from the host operating system. Most of these interfaces are unsafe.

Microsoft Windows. The Windows distribution of CM3-IDE includes interfaces for accessing many of the calls from the Win32 API: **winBaseTypes**, **winDef**, **winError**, **winNT**, **winBase**, **winCon**, **winGDI**, **winNetwk**, **winReg**, **winUser**, **winVer**, **NB30**, **CDErr**, **CommDlg**, **winSock**.

Intermediate interfaces provide access to middle layers of Modula-3 libraries on Win32 are also available: `Filewin32`, `Timewin32`, `OSwin32`, `TCPwin32`, `OSerrorwin32`.

Unix. The Unix distribution of CM3-IDE includes interfaces for accessing many of the calls from common Unix APIs: `Udir`, `Uipc`, `Uprocess`, `Usignal`, `Uugid`, `Uerror`, `Uman`, `Upwd`, `Usocket`, `Uuio`, `Uexec`, `Umsg`, `Uresource`, `Ustat`, `Uutmp`, `Ugrp`, `Unetdb`, `Usem`, `Utime`, `Uin`, `Unix`, `Ushm`, `Utypes`.

Intermediate interfaces provide access to middle layers of Modula-3 libraries on Unix are also available: `FilePosix`, `TimePosix`, `OSPosix`, `TCPPosix`, `OSerrorPosix`.

7.3.4 Interoperability with C

Several standard C libraries are available from CM3-IDE. `Cerrno`, `Cstddef`, `Cstdlib`, `Ctypes`, `Cstdarg`, `Csetjmp`, `Cstdio`, and `Cstring` are Modula-3 interfaces for C standard libraries. `M3toC` converts between C strings and Modula-3 TEXT types.

7.3.5 Low-level Run-time Interfaces

Several interfaces provide low-level access to the run-time.

Allocator	<code>RTAllocator</code> , <code>RTAllocStats</code>
Heap Management	<code>RTHeap</code> , <code>RTHeapDep</code> , <code>RTHeapInfo</code> , <code>RTHeapMap</code> , <code>RTHeapRep</code> , <code>RTHeapDebug</code> , <code>RTHeapStats</code> , <code>RTHeapEvent</code>
Garbage Collector	<code>RTCollector</code> , <code>RTCollectorSRC</code>
Type Management	<code>RTTipe</code> , <code>RTType</code> , <code>RTMapOp</code> , <code>RTTypeFP</code> , <code>RTTypeMap</code> , <code>RTUtils</code> , <code>RTTypeSRC</code>
Code and Execution	<code>RTLinker</code> , <code>RTProcedureSRC</code> , <code>RTModule</code> , <code>RTProcedure</code> , <code>RTException</code>
System Interface	<code>RTParams</code> , <code>RTArgs</code> , <code>RTProcess</code> , <code>RTStack</code> , <code>RTMachine</code>
Low-level	<code>RTHooks</code> , <code>RT0</code> , <code>RTSignal</code>
Miscellaneous	<code>RTIO</code> , <code>RTPacking</code> , <code>RTMisc</code>

7.4 Miscellaneous

Main interface. The `Main` interface is the entry point for executable programs. All programs must include a module that exports this interface.

Weak References. Using the `WeakRef` interface, you can register cleanup procedures to be run when the garbage collector is about to collect an object.

Performance Tuning. `ETimer` keeps track of elapsed time. It can be used for performance measurements. `PerfTool` and `LowPerfTool` control access to performance monitoring tools.

Configuration. The `M3Config` interface exports the configuration constants defined when the system was built.

Where to Go Next?

There are many more Modula-3 interfaces than described in this index. You may continue learning about Modula-3 interfaces by browsing the list of interfaces available in your CM3-IDE system.

Read this chapter to learn where to find further information about modula-3.

This chapter contains list of pointers to other information available about CM3-IDE and Modula-3. Much of the material cited in this chapter is included in your CM3-IDE distribution, whether in print, or on-line.

For continuously up-to-date information, see:

- Critical Mass Modula-3 Home Page, <http://modula3.elegosoft.com/cm3/>
- Modula-3 Resource Page, <http://modula3.org/>
- Modula-3 Internet Newsgroup: **comp.lang.modula3**
- Modula-3 Home Page, <http://www.cs.arizona.edu/~collberg/Research/Modula-3/modula-3/html/home.html>
- HP Labs (formerly Digital Equipment Corporation Systems Research Center [DEC SRC]), archive of Technical Reports, http://www.hpl.hp.com/techreports/Compaq-DEC/?jumpid=reg_R1002_USEN#src



Books on page 156 lists some of the introductory and advanced books on Modula-3.

Technical Documentation on page 157 includes references to a number of technical information sources. Much technical documentation is available as part of your CM3-IDE distribution.

If you are new to Modula-3, you may consider reading articles referenced in **Introductory Programming Articles** on page 158.

Systems Built Using Modula-3 on page 160 includes references to some of the systems written in Modula-3.

Parallel Programming on page 161 cites references to articles written about parallel programming in Modula-3.

Garbage Collection on page 161 describes the local and distributed memory management algorithms used in CM3-IDE.

Comparisons to Other Languages on page 162 may be useful if you would like to find out about similarities and differences between Modula-3 and other languages.

8.1 Books

8.1.1 System Programming with Modula-3

Greg Nelson (editor), Prentice Hall Series in Innovative Technology
ISBN 0-13-590464-1, L.C. QA76.66.S87, 1991.

This book is the definitive language reference. It includes the language reference manual and papers on the I/O library, threads, and the Trestle window system. On the newsgroups and in informal discussion it is often referred to as “SPwM3”.

Here is the table of contents:

- Introduction
- Language Definition
- Standard Interfaces
- An Introduction to Programming with Threads
- Thread Synchronization: A Formal Specification
- I/O Streams: Abstract Types, Real Programs
- Trestle Window System Tutorial
- How the Language Got its Spots

8.1.2 Modula-3

Samuel P. Harbison
ISBN 0-13-596396-6, Prentice Hall, 1992.

A complete Modula-3 textbook covering the full language, with examples and exercises. Includes a style manual and a user’s guide for SRC Modula-3. The first edition of the book contains many typos. A list of errata is available on-line for anonymous FTP (in TeX, compressed PostScript, or DVI format) from [gatekeeper.dec.com](http://gatekeeper.dec.com/pub/DEC/Modula-3/errata/) in the directory `pub/DEC/Modula-3/errata/`.

8.1.3 Algorithms in Modula-3

Robert Sedgewick
Addison-Wesley
ISBN 0-201-53351-0, L.C. QA76.73.M63S43, 1993.

Sedgewick’s classic text on algorithms, with examples in Modula-3.

8.1.4 Programming with Modula-3:

An Introduction to Programming with Style

Laszlo Boeszoermenyi and Carsten Weich

577 pages

Springer-Verlag

ISBN 3-540-57912-5 (English version)

ISBN 3-540-57911-7 (German version), 1995.

This book is an introductory programming text that uses Modula-3 for its examples. To quote the authors, “The main concern of the book is to give a clean and comprehensive introduction to programming for beginners of a computer science study. We start with more traditional programming concepts and move toward advanced topics such as object-oriented programming, parallel & concurrent programming, exception handling, and persistent data techniques. The book also presents a large number of complete examples written in Modula-3.”

8.2 Technical Documentation

Several technical reports describe various aspects of CM3-IDE. Most of these reports are available on-line in the Technical Notes section of your CM3-IDE environment.

8.2.1 Reactor White Paper

Critical Mass, Inc. August 15, 1996.

<http://www.cmass.com/reactor>

Reactor combines an innovative application development system with a rich and robust distributed infrastructure. This report outlines the features and benefits of Reactor’s high-productivity, distributed application development system.

8.2.2 Some Useful Modula-3 Interfaces

Jim Horning, Bill Kalsow, Paul McJones, Greg Nelson

Systems Research Center, Digital Equipment Corporation

Report #133, December 1993, 103 pages.

This manual describes a collection of interfaces defining abstractions that Modula-3 programmers have found useful over a number of years of experience with Modula-3 and its precursors. We hope the interfaces will be useful as a “starter kit” of abstractions, and as a model for designing and specifying abstractions in Modula-3.

8.2.3 Network Objects

Andrew Birrell, Greg Nelson, Susan Owicki, Edward Wobber
Systems Research Center, Digital Equipment Corporation
Report #115, February 1994.

This report describes the design and implementation of a Modula-3 network objects system, which allows you to write programs that communicate over a network, while hiding the messy details of network programming. Network objects provide functionality similar to remote procedure call (RPC), but they are more general and easier to use. The system is implemented in Modula-3.

8.2.4 Trestle Reference Manual

Mark S. Manasse and Greg Nelson
Systems Research Center, Digital Equipment Corporation
Report #69, December, 1991.

This report is the working definition of the Trestle toolkit for doing graphics in Modula-3.

8.2.5 VBToolkit Reference Manual: A toolkit for Trestle

Marc H. Brown and James R. Meehan (editors)
Systems Research Center, Digital Equipment Corporation

This report is the working definition of the VBToolkit toolkit. VBToolkit is a collection of widgets for building graphical user interfaces in Modula-3. See the FormsVBT Reference Manual below, which describes a system for easily composing these widgets.

This document is available on-line as part of the CM3-IDE distribution.

8.2.6 Obliq-3D Tutorial and Reference Manual

Marc Najork
Systems Research Center, Digital Equipment Corporation
Report #129, December 1994, 110 pages.

This report describes Obliq-3D, an interpreted language based on the Anim3D library for building 3D animations quickly and easily.

8.3 Introductory Programming Articles

8.3.1 Modula-3 Reference and Tutorial

Stephen Schaub

An on-line reference and tutorial, available in Tutorials section of your CM3-IDE Environment.

This tutorial is available on-line from the Modula-3 Home Page.

8.3.2 Net Balance: A Network Objects Example

Farshad Nayeri, March 1996

The sources of a simple client and server that use network objects.

A gzipped tar archive of the sources and web page is also available.

This example program is available on-line from the Modula-3 Home Page.

8.3.3 Building Distributed OO Applications: Modula-3 Objects at Work

Michel R. Dagenais

Draft, March 1995.

A draft of a book describing the latest object-oriented techniques for developing large interactive distributed applications. The focus is on the Modula-3 libraries and Network Objects, but the first two chapters give an introduction to object-oriented programming in general, and the object methodologies in particular.

8.3.4 Partial Revelation and Modula-3

Steve Freeman

Dr. Dobbs's Journal, 20(10):36-42, October 1995.

This article describes how Modula-3's partial revelations promote encapsulation and code reuse. The article is one of five on "object-oriented programming" contained in the same issue (the other four languages are C++, Ada 95, S, and Cobol'97).

8.3.5 Initialization of Object Types

Greg Nelson

Threads: A Modula-3 Newsletter, Issue 1, Fall 1995.

This article describes the rationale for the way object types are initialized in Modula-3. Modula-3 doesn't have type constructors, but you can specify default values for object fields in the type definition. The article also describes the init method convention.

8.3.6 Trestle Tutorial

Mark S. Manasse and Greg Nelson

Systems Research Center, Digital Equipment Corporation

Report #69, May 1992, 70 pages.

This report is a tutorial introduction to programming with Trestle, a Modula-3 window system toolkit implemented over the X Window System and Microsoft Windows. It assumes that you have some experience as a user of window systems, but no previous experience programming with X or Win32.

This article is available on-line as part of the CM3-IDE package.

8.3.7 Trestle by Example

Ryan Stansifer, October 1994.

An on-line introduction to the Trestle window system.

This tutorial is available in CM3-IDE environment at </tutorial/ui/tutorial.html>.

8.4 Systems Built Using Modula-3

8.4.1 The Juno-2 Constraint-Based Drawing Editor

Allan Heydon and Greg Nelson

Systems Research Center, Digital Equipment Corporation

Report #131a, December 1994.

This report describes Juno-2, a constraint-based drawing editor implemented in Modula-3. For more information, see the Juno-2 Home Page at <http://www.research.digital.com/SRC/juno-2/>.

8.4.2 Zeus: A System for Algorithm Animation and Multi-View Editing

Marc H. Brown

Systems Research Center, Digital Equipment Corporation

Report #129, February 1992.

8.4.3 Writing an Operating System with Modula-3

Emin Gün Sirer, Stefan Savage, Przemyslaw Pardyak, Greg P. DeFouw, and Brian Bershad

November 1995.

<http://www.cs.washington.edu:80/research/projects/spin/www/papers/WCS/m3os.ps>

Describes the experiences of the SPIN group at the University of Washington using Modula-3 to build a high-performance extensible operating system. Debunks some of the myths surrounding Modula-3 by arguing that the SRC reference implementation introduces some inefficiencies that are not imposed by the Modula-3 language itself.

8.4.4 The Whole Program Optimizer

Amer Diwan

Threads: A Modula-3 Newsletter, Issue 1, Fall 1995.

This article motivates and describes an optimizer for Modula-3 programs based on whole-program analysis. On benchmark programs, up to 50% of method invocations can be converted to direct calls.

8.5 Parallel Programming

8.5.1 An Introduction to Programming with Threads

Andrew D. Birrell

Systems Research Center, Digital Equipment Corporation

Report #35, January 1989.

This paper provides an introduction to writing concurrent programs with threads. A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs.

A must-read for anyone programming a concurrent system, this paper is included in the Technical Notes section of the CM3-IDE distribution.

8.5.2 Synchronization Primitives for a Multiprocessor: A Formal Specification

A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin

Systems Research Center, Digital Equipment Corporation

Report #20, August, 1987, 21 pages.

Formal specifications of operating system interfaces can be a useful part of their documentation. This document illustrates this by documenting the thread synchronization primitives available in Modula-3.

8.6 Garbage Collection

8.6.1 Compacting Garbage Collection with Ambiguous Roots

Joel F. Bartlett

Western Research Laboratory, Digital Equipment Corporation

Report #88/2, February 1988.

This report describes one of the algorithms used as the basis for CM3-IDE's garbage collector.

8.6.2 Distributed Garbage Collection for Network Objects

Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, Edward Wobber
Systems Research Center, Digital Equipment Corporation
Report #116, December 1993.

This report describes a fault-tolerant and efficient garbage collection algorithm for distributed systems. It is the algorithm used to garbage collect Network Objects.

8.6.3 Portable, Mostly-Concurrent, Mostly-Copying Garbage Collection for Multi-Processors

Antony L Hosking
Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
hosking@cs.purdue.edu

<add description here>

8.7 Comparisons to Other Languages

8.7.1 A Comparison of Modula-3 and Oberon-2

Laszlo Boeszoermenyi
Compares Modula-3 and Oberon-2, two successors to Modula-2.
Structured Programming, Springer-Verlag, 1993. Pgs 15-22.

8.7.2 A Comparison of Object-Oriented Programming in Four Modern Languages

Robert Henderson and Benjamin Zorn
Technical Report CU-CS-641-93
Department of Computer Science, University of Colorado, 1993

The paper evaluates Oberon, Modula-3, Sather, and Self in the context of object-oriented programming. While each of these programming languages provide support for classes with inheritance, dynamic dispatch, code reuse, and information hiding, they do so in very different ways and with varying levels of efficiency and simplicity. A single application was coded in each language and the experience gained forms the foundation on which the subjective critique is based. By comparing the actual run-times of the various implementations it is also possible to present an objective analysis of the efficiency of the languages. Furthermore, by coding the application using both explicit dynamic dispatch and static method binding, it is possible to evaluate the cost of dynamic dispatch in each language. The application was also coded in C++, thereby providing a well-known baseline against which the execution times can be compared.

8.8 Summary

Modula-3, the core of CM3-IDE, has been in extensive use for over a decade, and many of these experiences have been recorded in various articles, books, technical documents and network postings.

Many of these documents are distributed in Technical Notes or Tutorial sections of your CM3-IDE distribution.

Systems Programming with Modula-3. The definitive book on Modula-3 is Systems Programming with Modula-3, often abbreviated as SPwM3. The language reference section from this book is included as part of CM3-IDE.

Some Useful Interfaces. Also included in your CM3-IDE distribution, describes many of the standard Modula-3 interfaces.

Network Objects. To find out about the operation of Modula-3 Network Object system, see the Network Objects manual.

Language Reference. Included as part of your CM3-IDE distribution, the *Language Reference* provides precise information about the semantics of Modula-3 programs.

Other Information Sources. There are many other sources of information on the Internet:

- Critical Mass Modula-3 Home Page, <http://modula3.elegosoft.com/cm3/>
- Modula-3 Resource Page, <http://modula3.org/>
- Modula-3 page on Wikipedia, <http://en.wikipedia.org/wiki/Modula-3>
- Modula-3 Internet Newsgroup: post to **comp.lang.modula3**.

If you have found a useful publication or information source which is not listed in this chapter, please inform us at m3-support@elego.de.

FURTHER INFORMATION

This page left blank
intentionally.

