Atlas Reference Guide

For Videogame Script Insertion

8/10/2010

By Klarth (Steve Monaco)

Table of Contents

Background and Purpose

The history of Atlas began during a conversation between Gideon Zhi and I back in 2002 about the need for a better script inserter. We came up with some ideas together and I started the implementation soon after. The first success of Atlas arrived shortly after in the form of AGTP's Shin Megami Tensei translation for the SFC. Atlas remained private for longer than year and several more AGTP releases until I motivated myself to push it out to the public. In doing so, the entire code base of Atlas was reprogrammed, additional feature sets added, and tweaking of the old features as well. The first known game inserted by the public Atlas version was the FF6 retranslation by ChrisRPG and Sky Render, about two weeks after Atlas's initial release. Thus began a new era of Romhacking where many more games became possible to translate without custom utilities.

Atlas is a single purpose tool: Insert uncompressed scripts back into videogames with the ability to modify all pointer changes. A bonus is that once pointer types are setup, it can do all pointer writes automatically for the simplest scripts. For the complex scripts, it will require some manual processing but allows you to revise, insert, and test without reprocessing the whole script.

Command Line Usage

```
Atlas [switches] TargetFile ScriptFile.txt

Valid switches
      -d filename – Sets debugging output to filename.  If “stdout” is used, then
the output is displayed to the console window.

Example:
      Atlas –d debug.log ff1.nes ff1script.txt
```

Version History

V1.12 – 4/25/2013

- Added INSERTBINARY command: inserts a file at the current script position
- Added FILL command: fills remainder of text block with a specified byte
- JMP: Now you can name text blocks to make Atlas output clearer
- Static linking with VC++ runtimes
- Underscore is now allowed in variable names
- Crash fix: Scripts that contained over 1024 characters on a line
- Crash fix: Enabling autowrite before a table was loaded
- Hang fix: Unix line breaks in table files now work
- Table parser largely rewritten
- Returns nonzero error codes for makefile/batch error detection
- If target or pointer file does not exist, Atlas creates a new file
- Extension support will be removed after this version

V1.11 – 8/10/2010

- UTF-8 support
- Added ($XX) style hex codes to natively support WindHex dumps
- Debug logs now contain script text
- Skips linked, dakuten, and handakuten table entries instead of failing

V1.1 – 6/2/2010

- Documentation completely rewritten
- Added ENDIANSWAP to support both endian types of pointers
- Added FIXEDLEN for fixed length string support
- Added STRINGALIGN to align strings to byte boundaries
- Added SETPTRFILE and SETTARGETFILE for multifile support
- Added EMBPTRTABLE/WRITE to make embedded pointer tables cleaner
- Added a new WRITE for pointer tables for out of order tables
- Added overloads for WXX commands to support custom pointers to aid with inserting scripts that use hardcoded pointers
- Added WHW to write the high word of pointers
- Added detection of initialized but unwritten embedded pointers
- Fixed WXX single byte writing commands
- Fixed autowrite with Pascal strings
- Fixed line detection for missing table entries in script
- Fixed statistics relating to embedded pointers
- Fixed bug in the table library that required a newline at the table's end
- Optimized insertion speed for games using lots (10k+) of embedded pointers

Building Sample Files

Exercise 1 – Basic Script

The purpose of this sample will be to familiarize you with the steps of setting up a basic Atlas script. We will create a pointer type and use Atlas commands to update pointers for our script.

Game Specs

Platform: Super Nintendo
Header Size: $200
Dialogue Location: $40200
Pointer Table: $50200
Pointer Type: 24bit LOROM

Script File

```
// script.txt - Script for our game
#VAR(dialogue, TABLE)
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)

#VAR(Ptr, CUSTOMPOINTER)
#CREATEPTR(Ptr, "LOROM00", $0, 24)

#HDR($200)
#JMP($40200)

#WRITE(Ptr, $50200)
Now that our script is all setup…<LINE>
We can begin inserting the script!<END>

#WRITE(Ptr, $50203)
And just as nice, Atlas is<LINE>
updating the pointers too!<END>

#WRITE(Ptr, $50206)
And here is the end of the sample.<END>
```

```
Comment sample
Sets up a TABLE variable
Adds a table file
Makes the table active

24bit LOROM pointer without
offsetting

Sets a $200 byte header
Sets insertion offset

Writes the current insertion
offset to $50200 using Ptr.
HDR is used only internally
inside of pointer writing
commands to adjust for the
dialogue's position.
```

Exercise 2 – Pointer Tables

This sample will demonstrate Atlas's ability to create pointer tables. This eliminates the need for managing memory addresses within the main body of the script. We will also cover a more advanced pointer type than previously.

Game Specs

Platform: Nintendo
Header Size: $10
Dialogue Location: File - $2010, NES Memory - $C000
Pointer Table: File - $2810
Pointer Type: 16 bit – NES does not have an Atlas pointer type so we will use LINEAR and offset it by adding $A000 to get the correct address. You will likely need to create a new pointer type with different offsetting for every text bank in a game like this.

Script File

```
#VAR(dialogue, TABLE)                          Setup the table
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)

#VAR(Ptr2000, CUSTOMPOINTER)                   Creates a LINEAR (no mapping)
#CREATEPTR(Ptr2000, "LINEAR", $-A000, 16)      16bit pointer, adding $A000

#VAR(PtrTbl2810, POINTERTABLE)                 Creates a pointer table with 2
#PTRTBL(PtrTbl2810, $2810, 2, Ptr2000)         byte increment at $2810

#HDR($10)
#JMP($2010)

#WRITE(PtrTbl2810)
It's much easier to write to an Atlas<LINE>
Pointer table!<END>

#WRITE(PtrTbl2810)
Just make sure all of the strings are<LINE>
in the same order as the pointers.<END>

#WRITE(PtrTbl2810)
Next lesson we'll learn how to do this
without an Atlas command every string!<END>
```

Exercise 3 – Pointer Autowrite

This sample will build upon Exercise 2 and demonstrate usage of Atlas autowrite. This feature detects end strings and automatically writes a pointer to an associated pointer table or list. If you have a simple script, this is definitely the way to go to cut down on editing. One thing to note is that you must change your table to accommodate this. If your end string value is FF, then place /FF=<END> in your table.

Game Specs

See Exercise 2

Script File

```
#VAR(dialogue, TABLE)
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)

#VAR(Ptr2000, CUSTOMPOINTER)
#CREATEPTR(Ptr2000, "LINEAR", $-A000, 16)

#VAR(PtrTbl2810, POINTERTABLE)
#PTRTBL(PtrTbl2810, $2812, 2, Ptr2000)

#AUTOWRITE(PtrTbl2810, "<END>")

#HDR($10)
#JMP($2010)
// Automatically written pointer
It's easy as pie to use autowrite.<END>
// Automatically written pointer
It'll save you lots of commands if<LINE>
your script is simple.<END>
// Automatically written pointer
You can even disable autowrite...<END>
#DISABLE(PtrTbl2810, "<END>")

#WRITE(PtrTbl2810)
And continue writing to the pointer<LINE>
table manually.<END>
```

Create the table

Create the pointer

Create the pointer table

Initiate autowrite for the specific end string tag.

You can also manually write to a pointer table while autowrite is active.

Exercise 4 – Embedded Pointers

This sample will introduce embedded pointers. You use embedded pointers when pointers are embedded within the script, especially conditionals that have different strings. Because in this case the pointer's location will vary on each edit, you cannot define the pointer normally. This is also necessary when pointer banks are between script banks. The sample below does not demonstrate that you can indeed use EMBWRITE before EMBSET.

Game Specs

Platform: Genesis, ROM not interleaved
Header Size: No Header
Dialogue Location: Start - $80000, End - $81FFF
Pointer Type: 24bit LINEAR, Big Endian

Script File

```
// Pointers to individual strings not shown
#VAR(dialogue, TABLE)
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)
#ENDIANSWAP("TRUE")
#EMBTYPE("LINEAR", 24, $0)

#JMP($80000, $81FFF)

Let's start embedded pointers!<LINE>
<option><yesno>
#EMBSET(0)
#EMBSET(1)
#EMBWRITE(0)
That's the spirit!  You'll learn them<LINE>
in no time!<END>
#EMBWRITE(1)
Not ready yet, young grasshopper?<END>

<if-print><hasitem><atlas>
#EMBSET(2)
#EMBSET(3)
#EMBWRITE(2)
Now you're cooking with gas!<END>
#EMBWRITE(3)
No Atlas?  Get out of here!<END>
```

```
Setup the table


Swap endian for pointers
Set embedded pointer type to
linear, 24 bit, 0 offsetting
Jump into text block.  Atlas
limits text insertion to $81FFF

Yes/No control code containing
two pointers afterwards.  First
prints the Yes option, second
the No option.

Conditional text with two
pointers.  First points to text
displayed if you have the item
and second if you don't have it.
```

Exercise 5 – Embedded Pointer Tables

Let's build upon embedded pointers and discover how to use embedded pointer tables. This method allows you to simplify the code in situations where you must use a lot of embedded pointers. Generally this will be a pointer table situated directly between banks of text.

Game Specs

Platform: SNES
Header Size: No Header
Dialogue Location: Start - $21000, End - $23FFF
Pointer Type: 16bit HIROM, subtract $1000

Script File

```
// Skip over table code
#VAR(Ptr, CUSTOMPOINTER)
#CREATEPTR(Ptr, "HIROM", $1000, 16)
#VAR(PtrTbl, EMBPOINTERTABLE)

#JMP($21000, $23FFF)                    Embedded pointer table with 16
#EMBPTRTBL(PtrTbl, 16, Ptr)             pointers

#WRITE(PtrTbl, 0)                       Zero indexed, so this is the first
Now Atlas is setup.<END>                pointer

#WRITE(PtrTbl, 1)
You can manually specify the
pointer.<END>

#WRITE(PtrTbl)                          This will write the third pointer.
Or if the pointers are in order,<LINE>
you can do away with that part!<END>

#VAR(PtrTbl2, EMBPOINTERTABLE)          Setup new embedded pointer table
#EMBPTRTBL(PtrTbl2, 16, Ptr)            between the blocks of text

#WRITE(PtrTbl2)                         Writes the first pointer for block 2
And now we're on the second block!<END>
```

Exercise 6 – Multifile

This sample explains how to use Atlas's multifile features. By default, the game file specified via command line will be the file where script insertion and pointer writing takes place. You can also use pointer tables, lists, and autowrite with multifile support. The sample uses manual writes for brevity.

Game Specs

Platform: PSX
Header Size: No Header
Dialogue Location: Start - $100, End - $FFF, in both script1.bin and script2.bin
Pointer Type: 32 bit LINEAR, in event.bin. No offsetting
Pointer Location: $1000 for script1.bin, $2000 for script2.bin
Command Line: Atlas script1.bin script-e.txt

Script File

```
#VAR(dialogue, TABLE)                          Setup the table
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)
#VAR(Ptr, CUSTOMPOINTER)                       Pointers are simply linear
#CREATEPTR(Ptr, "LINEAR", 0, 32)

#SETPTRFILE("event.bin")                       Write pointers to event.bin!
#JMP($100, $1000)

#WRITE(Ptr, $1000)                             This pointer goes to event.bin
Here's some sample text from                   The text goes to script1.bin
script1.bin<END>

#WRITE(Ptr, $1004)
The pointers go to event.bin!<END>

#SETTARGETFILE("script2.bin")                  Insert script into script2.bin
#JMP($100, $1000)                             Must use a JMP after a target
                                               file change!
#WRITE(Ptr, $2000)
Now we're in script2.bin.<END>

#WRITE(Ptr, $2004)
Multifile is pretty easy!<END>
```

Exercise 7 – Fixed Length Strings

Here we cover Atlas's support of fixed length strings. In the event that a string is longer than the set limit, Atlas automatically truncates the string. If the string is shorter than the required size, Atlas will pad out the remaining space using the value from FIXEDLENGTH's second parameter. If you wish to turn off fixed length strings, use FIXEDLENGTH with string length 0. Most importantly, you must have an end token defined in your table file for string detection. You define an end token like so: /00=<END> if it has a hex value, or /<END> if there is no hex value.

Game Specs

Platform: NES
Header Size: $10
Text Location: Start - $2010, max string length 10 bytes
Pointer Type: None
Pointer Location: N/A
End String Value: None, so use /<END> in your table

Script File

```
#VAR(dialogue, TABLE)
#ADDTBL("game.tbl", dialogue)
#ACTIVETBL(dialogue)

#FIXEDLENGTH(10, 0)
#JMP($2010)

Short Sword<END>
Long Sword<END>
Falchion<END>
Claymore<END>
Scimitar<END>
```

```
Fixed length strings with a
length of 10, padded with 00s

Truncated to "Short Swor"
```

Table and Script Reference

Encoding

Atlas supports both ASCII and UTF-8 scripts and tables. Other encodings might work as long as all Atlas commands, script hex values, and comments (the // part) are ASCII-compatible. This is because the right hand side of table entries (ex: the 'a' in 10=a) and text portions of Atlas scripts are treated as a neutral encoding.

Variable names

The first character of a variable name must be A-Z, a-z, or an underscore. Afterwards, characters may be A-Z, a-z, underscore, and 0-9.

Hex Input

Both <$XX>/<$xx> and ($XX)/($xx) codes are supported for hex input.

Supported Table Features

The hex and text portions of a table entry are practically unlimited in length. The hex portion must contain an even count of numbers. Table entries such as "012345678901=This is a long, long string" are fully supported.

Supported table codes are as followed:
New line values: *FE or *FE=<LINE>
End String Values (necessary for autowrite): /FF=<END>
End String Values without text insertion: /<END>

Entry types skipped by Atlas:
Script bookmarks: (8000h)Text1
Script dump bookmarks: [8000h-8450h]Block 1
Script insertion bookmark: {8000h-TextDump.txt}Block 1
Dakuten code: !XX
Handakuten code: @XX
Linked values (used to dump control codes): $XX=x.

Pointer Reference

Pointer Lists

Pointer lists are a method you can use when the pointers in your script are out of order or scattered throughout. Basically, you create a text file (with ASCII encoding) listing the pointer locations one by one on separate lines. This is easier done by a program that can automate this for you. Below is a sample of how the text file should appear:

$10000
$10404
$23306
$18302
…And so on.

Low Level Pointers

Low level pointers (ie, the Wxx commands) are a shorthand way of writing pointers than custom pointers. They do have support for changing addressing types, but no support for offsetting. To use them, use SMA first and then the Wxx command of choice later.

Function Reference

General Commands

JMP(number Address)
JMP(number Address, string Name)
JMP(number Address, number MaxAddress)
JMP(number Address, number MaxAddress, string Name)
Changes the current position for text insertion.

Address – New file position
MaxAddress – Uppermost address for text insertion
Name – Can be used to identify text blocks in output statistics

---

HDR(number HeaderSize)
Modifies the current header size. This value is only used to effect pointer value calculations.

HeaderSize – New size for the header

ADDTBL(string TblFileName, table TableId)
Opens, parses, and loads a table file into Atlas. Use ACTIVETBL to activate.

TblFileName – The filename for the table file
TableId – table variable

ACTIVETBL(table TableId)
Activates a table for text encoding

TableId – table variable initialized by ADDTBL

VAR(variable VarName, variabletype Type)
Creates a new variable

VarName – The variable name to initialize
Type – Type of variable to create
Values – CUSTOMPOINTER, POINTERTABLE, POINTERLIST, EMBPOINTERTABLE, TABLE, EXTENSION

String Commands

FIXEDLENGTH(number Length, number FillCharacter)

Sets a fixed length for each string's insertion.

Length – Max constant length of the string. Use 0 to disable fixed length strings.
FillCharacter – Value (0-255) to fill the padding space with.

STRINGALIGN(number AlignValue)

Aligns the start of each string to a multiple of AlignValue

AlignValue – Byte boundary to align the string address to

STRTYPE(string StringType)

Sets the string type used during text insertion. Default is ENDTERM.

StringType – "ENDTERM" or "PASCAL"

PASCALLEN(number NewLength)

Sets the length used in Pascal strings

NewLength – Values: 1, 2, 3, or 4

High Level Pointer Commands

ENDIANSWAP(string DoSwap)

Performs endian swapping of all pointer commands. Default on Windows is little endian.
Values: “TRUE”, “FALSE”

CREATEPTR(custompointer PtrName, string AddressType, number Offsetting, number PtrSize)

Creates a custom pointer

PtrName – Name of the custompointer to create
AddressType – Address type of the pointer, see SMA for reference
Offsetting – Offsets the pointer by this value. Positive values subtract, negatives add.
PtrSize – Size of the pointer in bits. Valid values: 8, 16, 24, 32

WRITE(custompointer Ptr, number Address)

Writes a pointer to the address specified

Ptr – Name of the custompointer
Address – Physical location to write the pointer

PTRTBL(pointertable PtrTbl, number Start, number Offsetting, custompointer Ptr)

Creates a pointer table

PtrTbl – Name of the pointertable variable
Start – Beginning address of the pointer table
Offsetting – Number of bytes to advance in the table after each write
Ptr – Name of the custompointer used for pointer calculation

WRITE(pointertable PtrTbl)

Writes the next pointer to PtrTbl

PtrTbl – Name of the pointertable variable

WRITE(pointertable PtrTbl, number PtrNum)

Writes the next pointer to PtrTbl. PtrNum is used to specify a particular pointer in the table. 0 is the first pointer, 1 is the second, and so on.

PtrTbl – Name of the pointertable variable
PtrNum – Index into the pointertable

PTRLIST(pointerlist List, string Filename, custompointer Ptr)
Creates a pointerlist from Filename using Ptr to calculate pointers

List – Name of the pointerlist
Filename – Name of the file containing offsets. See Pointer Reference for details
Ptr – Name of the custompointer to calculate pointers

WRITE(pointerlist)
Writes the next pointer to the next address in the pointer list.

List – Name of the pointerlist

AUTOWRITE(pointertable PtrTbl, string EndTag)
Sets up autowrite for PtrTbl. Writes a pointer everytime EndTag is encountered. Must setup EndTag as an end token in your table file. See Table Reference for details.

PtrTbl – Name of a previously created pointertable
EndTag – Text string representing your end tag, ie “<END>”

AUTOWRITE(pointerlist PtrList, string EndTag)
Sets up autowrite for PtrList. Writes a pointer each time EndTag is encountered. Must setup EndTag as an end token in your table file. See Table Reference for details.

PtrTbl – Name of a previously created pointertable
EndTag – Text string representing your end tag, ie “<END>”

DISABLE(pointertable, PtrTbl, string EndTag)
Disables autowrite for specified PtrTbl and EndTag

DISABLE(pointerlist PtrList, string EndTag)
Disables autowrite for specified PtrList and EndTag

Low Level Pointer Commands

SMA(string AddressType)
Changes the pointer machine addressing type for low level pointers.
Values: “LINEAR”, “LOROM00”, “LOROM80”, “HIROM”, “GB”

W32(optional custompointer Ptr, number Address)
Writes a 32bit pointer to Address. If a custompointer is provided, it will be used for calculation.

W24(optional custompointer Ptr, number Address)
Writes a 24bit pointer to Address. If a custompointer is provided, it will be used for calculation.

W16(optional custompointer Ptr, number Address)
Writes a 16bit pointer to Address. If a custompointer is provided, it will be used for calculation.

WLB(optional custompointer Ptr, number Address)
Writes the lowest pointer byte to Address. If a custompointer is provided, it will be used for calculation. (ie, the 33 in value 00112233)

WHB(optional custompointer Ptr, number Address)
Writes the high pointer byte to Address. If a custompointer is provided, it will be used for calculation. (ie, the 22 in value 00112233)

WBB(optional custompointer Ptr, number Address)
Writes the bank pointer byte to Address. If a custompointer is provided, it will be used for calculation. (ie, the 11 in value 00112233)

WUB(optional custompointer Ptr, number Address)
Writes the upper pointer byte to Address. If a custompointer is provided, it will be used for calculation. (ie, the 00 in value 00112233)

WHW(optional custompointer Ptr, number Address)
Writes the high pointer word to Address. If a custompointer is provided, it will be used for calculation. (ie, the 0011 in value 00112233)

Embedded Pointer Commands

EMBTYPE(string AddressType, number Size, number Offsetting)
Sets addressing type, pointer size, and offsetting for all future embedded pointers.

AddressType – “LINEAR”, “LOROM00”, “LOROM80”, “HIROM”, “GB”
Size – Pointer size, in bits. Values: 8, 16, 24, or 32
Offsetting – Offsets the pointer by this value. Positive values subtract, negatives add.

EMBSET(number PointerNum)
Allocates the current text position as an embedded pointer as PointerNum.

PointerNum – Number representing the embedded pointer.

EMBWRITE(number PointerNum)
Writes the current pointer to the embedded position. If not allocated already via EMBSET, it will write it upon using EMBSET for the same PointerNum.

PointerNum – Number representing the embedded pointer.

EMBPTRTBL(embpointertable PtrTbl, number PtrCount, custompointer Ptr)
Creates an embedded pointer table at the current text position. The size allocated is PtrCount pointers using the size in bits from Ptr.

PtrTbl – Name for the embedded pointer table to be created
PtrCount – Number of pointers to allocate in the table.
Ptr – Custom pointer to calculate pointers

WRITE(embpointertable PtrTbl, optional PtrNum)
Writes the current pointer to PtrTbl. If PtrNum is specified, it will write the pointer to that particular pointer in the table (the first pointer is 0, second is 1, and so on). If PtrNum is not specified, it will write the pointer to the next pointer address (starting with number 0).

PtrTbl – Name of the embedded pointer table
PtrNum – 0-based pointer index

File Commands

SETTARGETFILE(string Filename)

Sets the text insertion file to Filename. By default, this is the file loaded via command line.

SETPTRFILE(string Filename)

Sets the pointer insertion file to Filename. All pointers except embedded pointers will be written to this file. By default, this is the file loaded via command line.

INSERTBINARY(string Filename, number StartOffset, number BytesToCopy)

Opens the file and directly inserts data into the current script output location. StartOffset is a 0-based file offset. If BytesToCopy is 0, the command inserts the entire file starting at StartOffset. Use StartOffset = 0, BytesToCopy = 0 to insert an entire binary file.

FILL(number FileByte)

Once enabled, this command fills the remainder of every text block using FillByte. FillByte must be between 0 and 255. The current text block is automatically filled after a new JMP or the end of file.

Extension Commands

EXTLOAD(extension Ext, string Filename)
Loads an extension by filename. Only DLLs are supported.

EXTEXEC(extension Ext, string Function)
Executes a DLL function from Ext

Ext – The loaded extension
Function – The DLL exported name of the function to execute

AUTOEXEC(extension Ext, string Function, string EndTag)
Executes Function for every time EndTag is encountered. EndTag must be defined in the table as a string end tag for this to work. Only one function per EndTag supported.

Ext – The loaded extension
Function – The DLL exported name of the function to execute
EndTag – Name of the end token in the table

DISABLE(string Function, string EndTag)
Disables AUTOEXEC for Function and EndTag.

Extension Reference

Caution

Extensions in their current form are quite primitive. I provide the documentation here for the curious, but I recommend not using them. It would be easier to download the free version of MSVC++, make the code changes to the source, and recompile.

What Are Extensions Good For?

The best advantage offered by Extensions is to provide a way to execute code on end tags. Possible uses include writing unsupported pointers, table switching inside the middle of text, and access to the internal script buffer. I highly recommend not attempting an Extension for compression (especially block-type) due to how Atlas flushes script to file before every Atlas command. You would need to reprogram the Atlas execution engine to make compression work smoothly: keep a text buffer in memory until JMP is reached (or flushed manually) and keep track internally of where the text position would be if it were written to file to keep Atlas pointer commands from breaking.

One extra use I have for extensions is for scripts with fixed-sized blocks that you can resize if need be. Make an extension and run it before the script is inserted to resize a segment to allow for more script space. This could be useful for games that use file systems like PSX.

How to Create an Extension?

First, create a blank DLL project with your compiler. Then #include “Atlas.h” and use the function declaration below:

__declspec(dllexport) unsigned int MyFunction(AtlasContext** Context)

Return codes:
NO_ACTION: Does nothing as you’d expect.
REPLACE_TEXT: Replaces Atlas’s internal script buffer with .StringTable
WRITEPOINTER: Writes a pointer of .PointerSize bytes at location .PointerPosition with .PointerValue

How to Use Your Extension?

Load the extension dll using EXTLOAD from an Atlas script. Then use EXTEXEC with the exported DLL function name to execute. You are currently unable to pass arguments to your extension. If you wish for automatic triggering of your extension via tags, then setup AUTOEXEC. Put your string value into the table as an end token like the following: “/<extexecute>”. This will setup an end token without inserting anything into the script.

Modification Reference

Environment

Atlas was created in MSVC++ 2010 and includes the project file. The core program has remained the same since the initial release of VS.net, so it should compile under earlier and future versions with only minor modifications. Support is not offered to compile under other compilers.

Creation of New Atlas Commands

Addition of a new Atlas command involves the following steps: Adding the command and type information to the parser, adding the command to the execution core, and implementing the function's code.

AtlasTypes.h contains the information for the parser. Add a new define for the CMD, increase CommandCount, add the string (what Atlas recognizes the command by) to CommandStrings, add the type info for each parameter to Types, add the number of parameters for the command to ParamCount. If your function takes no arguments, set ParamCount to 1. You'll be passed a P_VOID parameter. Check #BREAK() for an example of this. That's it for the parser.

AtlasCore.cpp contains the execution engine in the form of AtlasCore::ExecuteCommand. You'll receive each parameter parsed (no whitespace) as a std::string. Add a case statement for the CMD you defined in AtlasTypes.h.

Implementation of the function is up to you! Good luck!