

S C R I P T I N G M A N U A L



ETHERNET
M-MODULE
CARRIER/
LXI BRIDGE

MODEL
EM405-8

COPYRIGHT

C&H Technologies, Inc. (C&H) provides this manual "as is" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. C&H may make improvements and/or changes in the product(s) and/or program(s) described in this manual at any time and without notice.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Copyright © 2009 by C&H Technologies, Inc.

The information and/or drawings set forth in this document and all rights in and to inventions disclosed herein which might be granted thereon disclosing or employing the materials, methods, techniques, or apparatus described herein, are the exclusive property of C&H Technologies, Inc.

A Reader's Comment Form is provided at the back of this publication. If this form has been removed address comments to:

C&H Technologies, Inc.
Technical Publications
445 Round Rock West Drive
Round Rock, Texas 78681-5012

C&H may use or distribute any of the information you supply in any way that it believes appropriate without incurring any obligations whatever.

DOCUMENT REVISION NOTICE

C&H Technologies, Inc. makes every attempt to provide up-to-date manuals with the associated equipment. Occasionally, throughout the life of an instrument, changes are deemed necessary to equipment related documentation. The latest revision of our documentation is available for download from our web site at <http://www.chtech.com>.

NOTE

The contents of any amendment may affect operation, maintenance, or calibration of the equipment.

INTRODUCTION

This manual describes the operation and use of the scripting features of the EM405-8 Ethernet M-Module Carrier / LXI Bridge (Part Number 11029380). The scripting features described throughout this manual are available on all EM405-8's containing firmware version 3.0 and later. Contained within this manual are the functional details and the programmatic guidelines to adequately develop and deploy script-based applications for the EM405-8.

The EM405-8 is one of many test, data acquisition and control modules provided by C&H Technology and is one of a number of M-Module carriers in C&H Technologies' product line. All manufacturing options of the EM405-8 include the scripting features; however, only options that include an internal mass storage device provide non-volatile storage of scripts and autonomous operation using the startup script feature.

The part numbers of the EM405-8's that are covered by this manual are:

<u>Part Number</u>	<u>Description</u>
11029380-0001	EM405-8 with external triggers
11029380-0002	EM405-8 without external triggers
11029380-0003	EM405-8 with external triggers and 16GB mass storage

EM405-8's that contain a firmware revision prior to 3.0 can be updated in the field to the most recent firmware revision containing the scripting support. Please contact C&H Technologies Technical Support for update files and detailed instructions.

TABLE OF CONTENTS

1.0	GENERAL DESCRIPTION	1
1.1	PURPOSE OF EQUIPMENT	2
1.2	THE LUA SCRIPTING LANGUAGE.....	2
1.3	KEY FEATURES OF EM405-8 SCRIPTING	2
1.4	APPLICABLE REFERENCES	3
2.0	USING AND MANAGING SCRIPTS	4
2.1	COMMANDS	4
2.1.1	? Help	4
2.1.2	Data.....	5
2.1.3	Halt.....	5
2.1.4	List	5
2.1.5	Read	6
2.1.6	Remove	6
2.1.7	Retrieve.....	6
2.1.8	Run.....	7
2.1.9	Socket?.....	8
2.1.10	Upload.....	8
2.1.11	Ver.....	9
2.2	STORING SCRIPTS AND SUPPORTING FILES.....	9
2.3	SCRIPTING SOCKET INTERFACE	10
2.3.1	Configuring the Scripting Socket Interface	10
2.3.2	Connecting to the Scripting Socket Interface	11
2.3.3	Interactive Mode	11
3.0	DEVELOPING SCRIPTS	13
3.1	DEVELOPMENT SOFTWARE – EM405-8 IVI DRIVER SOFT FRONT PANEL	13
3.2	EM405-8 EXTENSIONS LIBRARY	13
3.3	PASSING DATA.....	19
3.4	CALLING M-MODULE DRIVERS	20
3.4.1	Using Alien to Call M-Module Drivers	21
3.4.2	Dealing with data	23
3.4.2.1	Buffers.....	24
3.4.2.2	Arrays.....	25
3.4.2.3	Pointer Unpacking.....	26
3.4.2.4	Callback Functions.....	26
3.4.3	Building M-Module Drivers into a Linux Shared Library.....	27
3.5	SHARING DATA BETWEEN SCRIPTS	27
3.6	STARTUP SCRIPT	27
3.7	USING THE MASS STORAGE DEVICE.....	28
4.0	USING LUA TO WRITE CUSTOM WEB PAGES	29
4.1	URL OF SCRIPT BASED PAGES	29
4.2	ADDING A LINK TO THE C&H NAVIGATION MENU.....	30
4.3	URL OF PICTURES AND OTHER SUPPORTING FILES	31
4.4	PASSING ARGUMENTS TO THE SCRIPT	31
4.5	DEVELOPING THE PAGE CONTENT	32

LIST OF FIGURES

Figure 1 EM405-8 Scripting Utilities Architecture Diagram	1
Figure 2 Retrieve Script Download Format.....	7
Figure 3 Script Upload Format	9
Figure 4 Network Configuration Webpage	10
Figure 5 EM405-8 IVI Driver Scripting Panel Screenshot	13
Figure 6 Data Passing Example.....	20
Figure 7 M-Module Driver Example Code	23
Figure 8 M-Module Driver Example Using Buffers	25
Figure 9 M-Module Driver Example Using Arrays	26
Figure 10 Simple Lua Based Web Page	29
Figure 11 Example Adding Link to the Navigation Menu	30
Figure 12 Example Linking to a User Image.....	31
Figure 13 Example Retrieving Arguments from the URL	32
Figure 14 Example Using “l_em405web.lua” Library	34

LIST OF TABLES

Table I Use and Management Commands.....	4
Table II. Functions in the EM405-8 Extensions Library	14
Table III. Alien Data Types	22
Table IV l_em405web.lua Library Functions.....	32

1.0 GENERAL DESCRIPTION

The scripting utilities of the EM405-8 LXI M-Module Bridge provide enhanced programming capabilities allowing the user to easily embed software on the bridge to improve performance and further integrate a set of M-modules. The scripting utilities are based off of the Lua programming language and include extensions and customizations designed specifically for the EM405-8. An architecture diagram of the scripting utilities is shown in Figure 1. Since Lua is an interpretive language, EM405-8 users can easily develop scripts without the need for an expensive development environment.

By embedding a script or a set of scripts on the EM405-8, the burden for monitoring, configuring and controlling the M-modules is taken off the network and placed onto the EM405-8's embedded processor. This provides significant performance improvements and allows the EM405-8 and associated M-modules to be customized and configured. It can also act independently for a long period of time without the need for a network client to monitor the device(s).

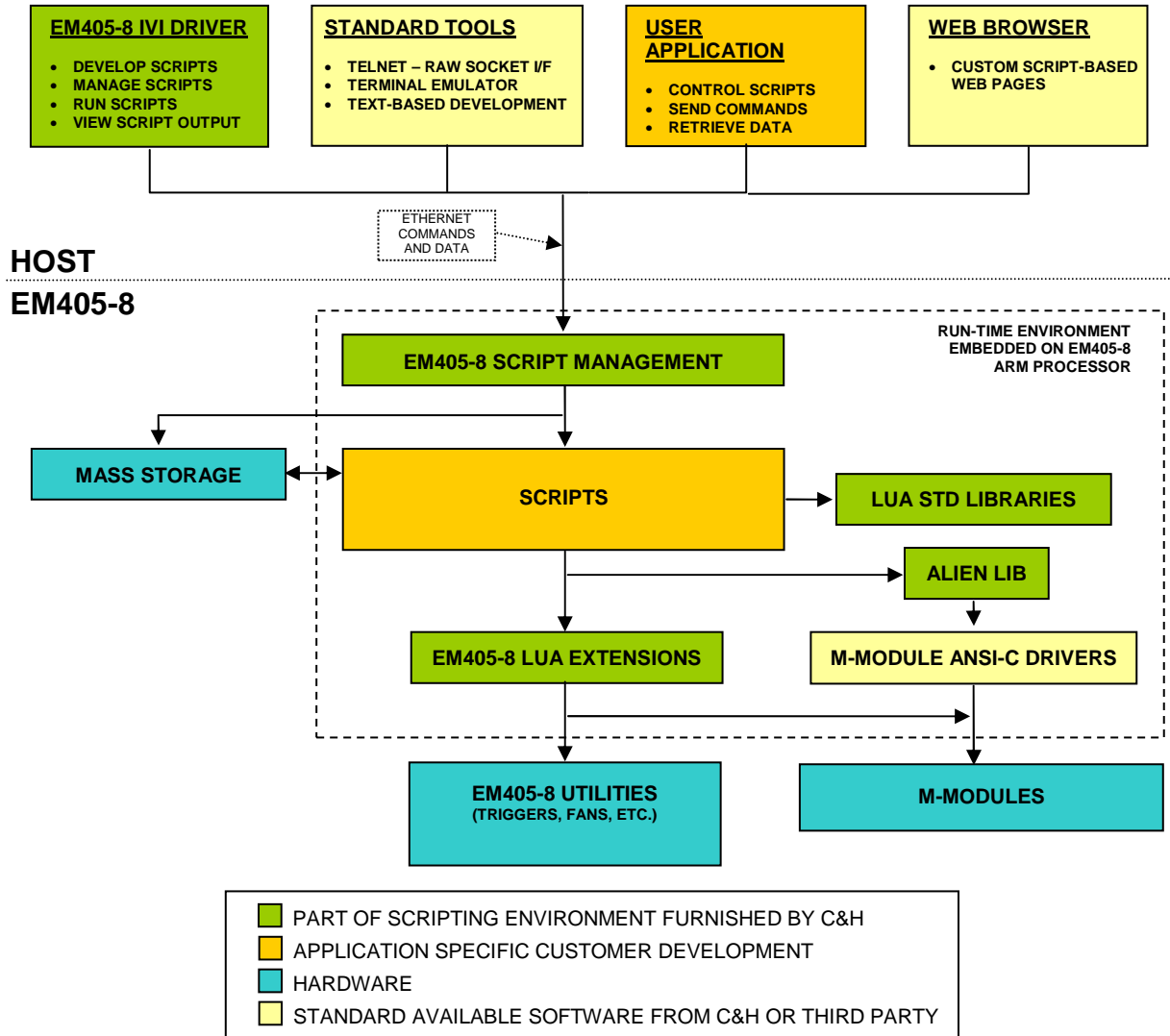


Figure 1 EM405-8 Scripting Utilities Architecture Diagram

1.1 PURPOSE OF EQUIPMENT

The EM405-8 was designed for measurement, automation and control applications. The EM405 easily interfaces a VITA 12-1996 standard M-Module to a typical Ethernet network. The carrier allows the numerous functions available in the M-Module mezzanine format to be remotely located near the unit-under-test, easing many system integration issues. Over 100 M-Modules are available from numerous manufacturers.

The scripting utilities were added to the EM405-8 to allow users to fully utilize the embedded processor of the EM405-8 and to allow for easy customization of the integrated unit.

1.2 THE LUA SCRIPTING LANGUAGE

Lua (pronounced LOO-ah) is a powerful, fast, lightweight embeddable scripting language. It is important to note that the name Lua is not an acronym. Lua was written in ANSI-C and is designed to easily integrate with other software written in C and other conventional programming languages. Thus, one of Lua's primary strengths is its extensibility. Lua's extensibility is two-fold. Lua can be used to extend an application by embedding Lua scripting into the application. Also, Lua itself is extendible allowing Lua scripts to utilize libraries and components written in other programming languages.

Lua is an interpreted language in that Lua programs are stored as text that is interpreted at runtime. In reality, Lua code is compiled at runtime into an intermediate form; however, the fact that compilation is performed at runtime allows scripts to be run directly from text and allows Lua to retain the designation of an interpreted language.

Lua is free software distributed under the terms of the MIT license. The MIT license is short and very liberal, allowing the user unrestricted use the software.

Further details, including documentation and references can be found at www.lua.org.

1.3 KEY FEATURES OF EM405-8 SCRIPTING

The EM405-8 implementation of scripting includes the basics of Lua including all standard libraries, as well as EM405-8 specific extensions and EM405-8 specific script development tools. Key features include:

- Lua 5.1 programming language interpreter
- Lua 5.1 standard libraries (math, table, string, i/o, operating system, etc.)
- EM405-8 extensions to control M-modules and EM405-8 utilities
- Call high-level M-module drivers and other libraries
- TCP/IP based interactive interpreter for development
- Develop scripts using any text editor. No compilation required
- Store and retrieve data to/from mass storage device (mass storage is an EM405-8 option)
- Receive commands and pass data over the Ethernet interface
- Network based commands to use and manage scripts (store, retrieve, run, halt, etc.)
- Create custom web pages to control the instrument

1.4 APPLICABLE REFERENCES

In addition to this reference manual there are several sources of further information on both the EM405-8 and the Lua programming language. These documents should be referenced for details of topics not covered or not elaborated on by this manual. The most applicable references are:

- **EM405-8 User's Manual**, C&H Technologies, Inc., C&H part No: 11029384.
- **Programming in Lua – 2nd Edition**, Ierusalimschy, Roberto, Rio de Janeiro, 2006
- **Lua Reference Manual**, Ierusalimschy, Roberto, De Figueiredo Luiz Henrique, Celes Waldemar, Lua.org, 2006
- **Lua.org**, <http://www.lua.org>
- **Alien**, <http://alien.luaforge.net/>, <http://luaforge.net/projects/alien/>

2.0 USING AND MANAGING SCRIPTS

The scripting utilities are accessible via the standard EM405-8 command interfaces (VXI-11 and raw socket). In addition, a special *scripting socket interface* is available to help develop and debug scripts. A set of use and management commands are defined and any of these commands can be sent via any of the available interfaces with a few exceptions.

2.1 COMMANDS

The scripting commands allow the user to develop, use and manage scripts. These commands are the primary interface to the scripting utilities and are used during both development and deployment of scripts. They are available, with a few exceptions, via any of the EM405-8's command interfaces including the special scripting socket interface. A complete list of use and management commands is shown in Table I. Details of each command can be found in the subsequent sections.

Table I Use and Management Commands

? help		list command help
data	<data>	send up to 16Kbytes to a script (<i>VXI-11 only</i>)
halt	<script>	halt the script of the specified name
	-l <script>	halt last script started of specified name
	-nx <script>	halt script number x of specified name
	-a [script]	halt all running scripts
list		list all scripts
	-l [script]	list details (script name optional)
	-r [script]	list if running (script name optional)
read	<script>	read the specified script
remove	<script>	remove the script
retrieve	<script><port>	retrieve the script using the specified port
	-d	delete the file upon retrieval
run	<script>	run specified script
	-e <command>	execute command
	-i	enter interactive mode (<i>scripting socket only</i>)
socket?		returns 1 if scripting socket running, 0 if not
	-p	returns the port of the scripting socket
upload	<script><port>	upload the script using the specified port
	-x	execute script after upload
	-o	overwrite script if file exists
ver		returns versions of Lua and EM405-8 extensions

Note: An asterisk () is required before the command if sent via the VXI-11 or raw socket interface. The EM405-8 contains a binary command protocol. An asterisk (*) character denotes that the remaining command consists of ASCII characters. If the first byte is anything other than an asterisk (*) the EM405-8 perceives it as a binary command. The binary commands are not available via the scripting socket therefore the asterisk (*) is not required nor recognized.*

2.1.1 ?| Help

The help command or simply '?' will return a list of commands along with argument options and descriptions similar to Table I. The result will be output via the same interface from which the command was received.

help or ?

2.1.2 Data

The `data` command allows the user to transfer up to 16 kilobytes of data to a running script. This command is only available over the VXI-11 or raw socket interface. It has no effect if sent over the scripting socket interface.

```
data <up to 16K of data>
```

The command requires a running script that is expecting or polling for data. If such a script is not running then this command will result in an I/O Error. For details on transferring data to/from a running script refer to section 3.3.

2.1.3 Halt

The `halt` command is used to stop running scripts. Scripts always run in their own thread allowing multiple scripts to be run at the same time. The `halt` command has the following syntax:

```
halt <options>
```

```
options:
```

```
<script>      halt the first script started of the specified name
-l <script>    halt the last script started of the specified name
-nX <script>   halt script number x of specified name
-a [script]    halt all scripts or all scripts of the specified name
```

In addition to running more than one script at a time, the same script can be concurrently run multiple times, each in its own thread. The system maintains a sequence list indicating the order in which each instance of a running script was started. Using the `halt` command, the user can specify the specific instance to halt, based on this start sequence. The system does not maintain any further details other than the start order. It is up to the user to understand the details of a specific instance, such as the responsibilities of that instance, and the order in which that instance was started.

With the `-a` option the user can halt all running scripts of a specified name, by passing the script name argument or all running scripts within the system by not specifying a script name.

2.1.4 List

The `list` command will return a list of all scripts stored on the system. The result will be output via the same interface from which the command was sent.

```
list [options]
```

```
options:
```

```
-l          show details of each file
-r          show only scripts that are currently running
[script]    if the optional [script] is specified only that script
            will be shown
```

Arguments of the `list` command are optional. If no arguments are provided, all scripts will be listed without details. Use the `-l` option to show the details of the scripts. Details include the file size, date, the system/user designation and a run/idle status indication. The `-r` option shows only those scripts that are currently running. Both options may be included in a single `list` command. If the optional `[script]` argument is passed only scripts of the specified name will be listed.

In the output of the `list` command, each script listing is terminated with a “new line” character (`\n` or `0xA`). The entire list is terminated with a carriage return (`\r` or `0xD`).

2.1.5 Read

The `read` command will return the source code of the specified script via the interface from which the command was sent. For example, if in the scripting socket interface you type `read myscript.lua` the source code of `myscript.lua` will appear in the terminal window. If you send the same command via the VXI-11 interface, the next VXI-11 read will return the source code of the script. Note that the VXI-11 interface has a 16 kilobyte limit therefore any script greater than 16 kilobytes will not be returned properly. An alternative to the `read` command is the `retrieve` command detailed in section 2.1.7.

```
read <script>
```

2.1.6 Remove

The `remove` command will delete the specified script from both volatile and non-volatile memory. A removed script is not recoverable therefore care must be taken to ensure a script is saved on a host computer before sending the `remove` command.

```
remove <script>
```

2.1.7 Retrieve

The `retrieve` command is used to download a script from the EM405-8 to the host computer. A download port must be specified.

```
retrieve [options] <script> <port>
```

```
options:
```

```
-d          delete the file upon retrieval
```

Upon receiving the `retrieve` command, the EM405-8 will search for the specified script and return a 4-byte acknowledge (“`ack\n`” if found or “`nck\n`” if not found) via the same interface from which the command was received. If the script was found, the EM405-8 will launch a small TCP/IP server at the specified port. The host application must then connect to this server to download the script. The TCP/IP server will automatically exit once the script has been downloaded. Therefore, a new `retrieve` command must be used for each script to be downloaded.

The script is sent over the TCP/IP server by the EM405-8 in a raw format preceded by 4 bytes specifying the size of the file as shown in Figure 2. Figure 2 also shows the simple procedure to download the file.

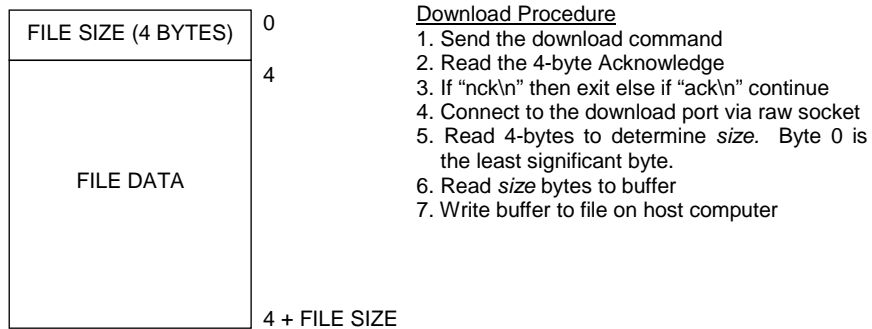


Figure 2 Retrieve Script Download Format

If the `-d` option is found, the script will be deleted at the completion of the download. A deleted script is not recoverable therefore care must be taken before sending the `-d` option

2.1.8 Run

The `run` command starts the specified script, launches interactive mode or executes the specified Lua command. One of the three arguments is required for the `run` command. Only one argument can be passed in a single run command.

```
run <argument>
arguments:
  <script>          run the specified script
  -i                enter interactive mode (scripting socket only)
  -e <command>    executes the specified command
```

Running a specific script will launch the script in its own process so that it may run independently and so that the user may launch more than one script. In reality, ***the run command is optional***. Scripts can be run by simply sending the name of the script to the EM405-8 as a command itself. Further, if the file extension of the script is `.lua` than the extension may be omitted. For example, if the script file is `myscript.lua` then the following three commands perform the same action of launching the script

```
run myscript.lua
myscript.lua
myscript
```

Interactive mode is a special mode of the scripting socket in which the user can type Lua source code directly to the interpreter to be run immediately. Details on interactive mode can be found in section 2.3.3. The `-i` option is only valid from the scripting socket interface and does not return until the socket is closed.

The `-e` argument allows the user to input Lua source to be run immediately. It is different than interactive mode in that it does not give the user a different command prompt in the scripting socket and more importantly once the command(s) is complete the Lua state is lost. In other words, variables, loaded libraries and other constructs that are normally valid from one command to the next are only available during that one command.

2.1.9 Socket?

The `socket?` command will return an ASCII '1' if the scripting socket interface is available and an ASCII '0' if not. Further passing the `-p` argument will return the TCP/IP port of the scripting socket interface. Note that the `-p` argument will return a port number regardless of whether or not the scripting socket is available.

```
socket? [options]
```

```
options:
```

```
-p          returns the port number of the raw socket interface
```

It is useful to send the `socket?` command via the VXI-11 or raw socket interface to determine if the scripting socket interface is available and to determine at which TCP/IP port to connect to it.

2.1.10 Upload

The `upload` command is used to upload a script from a host computer to the EM405-8. An upload port must be specified.

```
upload [options] <script> <port>
```

```
options:
```

```
-x          execute the script after upload  
-o          overwrite the script if the file exists
```

Upon receiving the `upload` command, the EM405-8 will first determine if a script of the same name already exists. If the EM405-8 finds a script of the same name and the `-o` is not provided a 4-byte negative acknowledge (“nck\n”) is returned. If it does not find a script of the same name or if the `-o` option is provided, a 4-byte positive acknowledge (“ack\n”) is returned. The acknowledge, whether it is positive or negative, is sent via the same interface from which the command was received.

If a positive acknowledge is returned the, EM405-8 will launch a small TCP/IP server at the specified port. The host application must then connect to this server to upload the script. The TCP/IP server will automatically exit once the script has been uploaded. Therefore, a new `upload` command must be used for each script to be uploaded.

The host computer must send the script in a raw format preceded with 4 bytes specifying the size of the file as shown in Figure 3. Figure 3 also shows the simple procedure to upload the file.

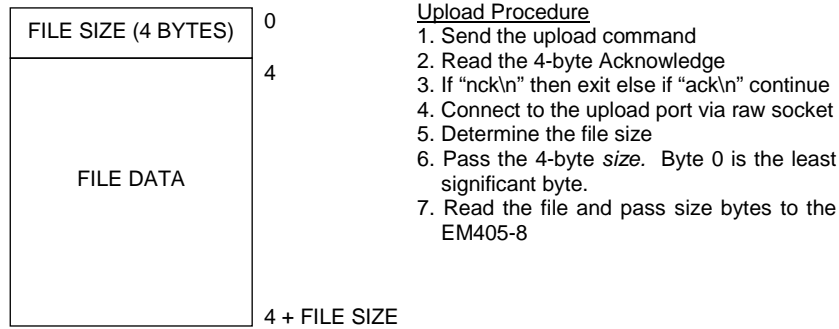


Figure 3 Script Upload Format

If the `-x` option is found, the script will be automatically executed at the completion of the download. A minor limitation of the `-x` option is that it does not allow you to pass arguments to the script upon running it. If this functionality is needed, simply use the `upload` command without the `-x` option followed by the `run` command.

2.1.11 Ver

The `ver` command will return version information of both the Lua interpreter and the EM405-8 extensions. The result will be output via the same interface from which the command was received. There are no arguments to the `ver` command.

```
ver
```

2.2 STORING SCRIPTS AND SUPPORTING FILES

All scripts and supporting files are stored in a single script pool. Since there are no subdirectories in the script pool each file must have a unique name. Using the `upload` command detailed in section 2.1.10, the user can upload and store any file into the script pool. All file related commands such as `list`, `read`, `remove`, `retrieve` and `run` operate on the script pool.

Files in the script pool are labeled either `sys` or `user`. The system files are scripts that are furnished by C&H and are stored on the EM405-8 as part of the EM405-8 firmware. These files include EM405-8 utility scripts. User files are those uploaded by the user. Using the `-l` argument of the `list` command, the `sys` versus `user` type of each file can be determined.

If the EM405-8 being used includes the internal mass storage option, the script pool is automatically stored on the mass storage device in a non-volatile fashion. If the EM405-8 does not contain the internal mass storage option, the script pool, with the exception of the system scripts, is stored in volatile memory and must be uploaded after every power-up. In both cases, the absolute path of the user script pool is: `/usr/scripts/user`.

There are several symbolic links in the EM405-8 web server host directory and link to the script pool. These symbolic links allow scripts to be used to create custom web pages and allow these custom webpage to link to images and other files uploaded by the user. Details on the symbolic links and the use of scripts to develop custom web pages can be found in section 4.0.

2.3 SCRIPTING SOCKET INTERFACE

The EM405-8 supplies a TCP/IP socket interface that allows the user to interact with the scripting utilities of the carrier. This is the primary interface used for developing scripts. This interface accepts commands that allow the user to run scripts and view the script output. If a script is started from the scripting socket interface, the scripting socket interface acts as the stdout of the script. The commands accepted over the scripting socket interface are detailed in section 2.1. In addition, the interface implements an interactive mode that allows the user to input script commands line by line.

2.3.1 Configuring the Scripting Socket Interface

The scripting socket interface provided by the EM405-8 is disabled by default. It can be enabled via the webpage of the EM405-8 under “Network Configuration.” Figure 4 shows the Network Configuration webpage.



Home	EM405-8 LAN Configuration																						
LAN Configuration																							
Status/Control																							
	<table border="1"><tr><td>Hostname</td><td>192.168.1.4</td></tr><tr><td>Description</td><td>C&H EM405-8 - SN1011</td></tr><tr><td>TCP/IP Mode</td><td><input checked="" type="checkbox"/> DHCP <input checked="" type="checkbox"/> Auto IP <input type="checkbox"/> Static IP</td></tr><tr><td>IP Address</td><td>192.168.1.4</td></tr><tr><td>Subnet Mask</td><td>255.255.255.0</td></tr><tr><td>Default Gateway</td><td>192.168.1.1</td></tr><tr><td>DNS Server(s)</td><td>0.0.0.0 0.0.0.0</td></tr><tr><td>Raw Socket</td><td><input type="radio"/> Enable <input checked="" type="radio"/> Disable</td></tr><tr><td>Raw Socket Port</td><td>10001</td></tr><tr><td>Scripting Socket</td><td><input checked="" type="radio"/> Enable <input type="radio"/> Disable</td></tr><tr><td>Scripting Socket Port</td><td>10011</td></tr></table>	Hostname	192.168.1.4	Description	C&H EM405-8 - SN1011	TCP/IP Mode	<input checked="" type="checkbox"/> DHCP <input checked="" type="checkbox"/> Auto IP <input type="checkbox"/> Static IP	IP Address	192.168.1.4	Subnet Mask	255.255.255.0	Default Gateway	192.168.1.1	DNS Server(s)	0.0.0.0 0.0.0.0	Raw Socket	<input type="radio"/> Enable <input checked="" type="radio"/> Disable	Raw Socket Port	10001	Scripting Socket	<input checked="" type="radio"/> Enable <input type="radio"/> Disable	Scripting Socket Port	10011
Hostname	192.168.1.4																						
Description	C&H EM405-8 - SN1011																						
TCP/IP Mode	<input checked="" type="checkbox"/> DHCP <input checked="" type="checkbox"/> Auto IP <input type="checkbox"/> Static IP																						
IP Address	192.168.1.4																						
Subnet Mask	255.255.255.0																						
Default Gateway	192.168.1.1																						
DNS Server(s)	0.0.0.0 0.0.0.0																						
Raw Socket	<input type="radio"/> Enable <input checked="" type="radio"/> Disable																						
Raw Socket Port	10001																						
Scripting Socket	<input checked="" type="radio"/> Enable <input type="radio"/> Disable																						
Scripting Socket Port	10011																						
	<input type="button" value="Submit Changes"/> <input type="button" value="Restore Defaults"/> <input type="button" value="Change Password"/>																						

Figure 4 Network Configuration Webpage

The “Scripting Socket” enable/disable selection determines whether the socket interface is available and the “Scripting Socket Port” specifies the TCP/IP port number at which the user will connect to the interface. The default port number is **10011**. If this port is used by a different application in the system then this setting can be modified.

Port numbers are assigned by the Internet Assigned Numbers Authority (IANA) and many are reserved for specific functions such as HTTP, SMTP, or telnet. Care must be taken not to choose a port number that is reserved or that will be commonly used on the network.

2.3.2 Connecting to the Scripting Socket Interface

Application software can utilize the scripting socket interface using a standard TCP/IP raw socket connection to the EM405-8's IP address and the scripting socket port number. The application can then transmit ASCII commands as detailed in section 2.1 and receive standard output directly from the scripts.

Alternatively and more commonly, the user will utilize the scripting socket interface using a telnet type connection in which the user will be able to manually type in commands and view the script output. For example, using the Windows command prompt or a Linux terminal, the user can connect to the scripting socket interface by typing the following:

```
$ telnet 192.168.1.236 10011
```

In this example *192.168.1.236* is the IP address of the EM405-8 and *10011* is the configured port number of the scripting socket interface. ***For proper operation and display, the telnet interface must be configured for local echo and for CR+LF termination of commands.***

The user can directly input commands as detailed in section 2.1 at the command prompt. In addition, the scripting socket interface forwards the standard output console (`stdout` and `stderr`) to the socket allowing all data destined for the `stdout` and `stderr` to be displayed on the scripting socket interface. The data includes `stdout` data (`print()` function) from the scripts themselves and warnings, errors and messages from the Lua interpreter.

In contrast, the standard EM405-8 VXI-11 and raw socket interfaces accept the same commands that are available via the scripting socket interface; however these interfaces do not allow the user to interactively control the scripting utilities with a telnet type connection nor do they allow the user to receive console messages and errors.

2.3.3 Interactive Mode

From the scripting socket interface, the user can enter interactive mode with the command

```
run -i.
```

Once in interactive mode, the user can input Lua statements directly at the command prompt. Interactive mode reads lines that the user inputs and executes them as they are read like the following example (**bold** indicates the output):

```
>print(5+10)  
15
```

Interactive mode recognizes incomplete Lua statements and in-turn prompts the user for more data until the statement is complete or a syntax error is found. This allows the user to input multi-line statements such as for loops and if-else conditional statements. For example an if-then

statement will look like the following (**bold** indicates the output):

```
>a=10
>if a == 5 then
>> print("pass")
>>else
>>print("fail")
>>end
fail
```

Interactive mode also supports a special prefix “=” that will cause the interpreter to print the value of the expression to the right of the “=” sign. For example (**bold** indicates the output):

```
>a=10+5
>=a
15
```

There are a few limitations to interactive mode. Notably, each statement is executed as its own Lua script; therefore local variables are not carried over to the next statement. The following example illustrates this (**bold** indicates the output):

```
>local b=10
>print(b)
nil

>c=10
>print(c)
10
```

Another notable limitation is the inability to print tables.

There is no command to exit interactive mode. The user may exit interactive mode by simply disconnecting from the scripting raw socket. When the user re-connects to the scripting raw socket he/she will return to normal mode. Interactive mode is a great tool to learn and experiment with the Lua language; however most script development will be done by writing scripts into files that are uploaded and run.

3.0 DEVELOPING SCRIPTS

3.1 DEVELOPMENT SOFTWARE – EM405-8 IVI DRIVER SOFT FRONT PANEL

The EM405-8 IVI Driver Soft Front Panel contains a scripting panel that allows the user to interface with the scripting utilities and provides tools to ease script development. A screenshot of this scripting panel is shown in Figure 5. With this panel the user can interactively use and manage the scripts. The panel provides the capability to, among others, upload, retrieve, run, halt, and remove scripts. The panel also contains debug utilities that allow the user to run a script in the scripting socket interface and view the script output in a standard command prompt. Further, the panel allows the user to open a command prompt to the scripting socket interface where he/she can directly input use and management commands.

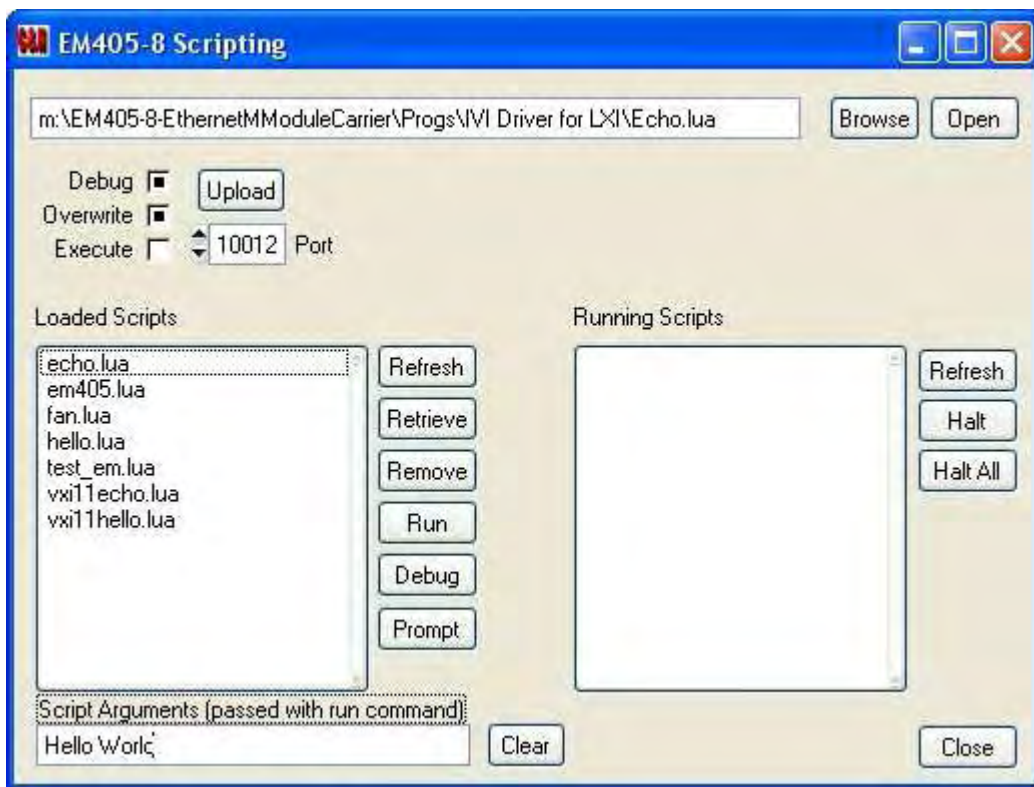


Figure 5 EM405-8 IVI Driver Scripting Panel Screenshot

3.2 EM405-8 EXTENSIONS LIBRARY

The EM405-8 Extension Library is a set of functions in the form of a Lua library that can be included and called from a Lua script like any of the standard Lua libraries. The functions in the EM405-8 Extension Library allow a Lua script to utilize and interact with the EM405-8 utilities and M-module hardware. Functions include the ability to read and write M-module registers, configure the EM405-8 triggers, send and receive data via the VXI-11 interface, control the EM405-8 fans, and read the EM405-8 temperature sensors.

To use the Lua libraries simply `require` the library in the script as follows:

```
require "lua_em405"
```

Once the library is loaded any of the library functions are available for use in the script with the following syntax:

```
lua_em405.function()
```

To further enhance readability, the library can be assigned to a local or global variable and the library functions can be called with this variable as the prefix as in the following example. This method will be used throughout this manual.

```
em405 = require "lua_em405"
em405.function()
```

Table II summarizes the functions that are in the EM405-8 Extensions Library. Details of each function including a function description, valid parameter ranges and return types follow the table.

Numerical constants in the Lua language are written much like they are in ANSI-C. Examples of numerical constants are:

```
7      7.893      756.87e-6      0.784e1      0x76
```

The detailed descriptions below follow the same conventions.

Table II. Functions in the EM405-8 Extensions Library

<u>Function</u>	<u>Description</u>
emclose	Free resources allocated during library load
emfans	Control the speed of the fans
emread	Read an EM405-8 carrier control register
emtemp	Read an EM405-8 on-board temperature sensor
emwrite	Write an EM405-8 carrier control register
input	Input data to the script from the VXI-11 interface
mread	Read an M-Module register
mreadblock	Read a block of M-Module registers
mreadfifo	Repeatedly read an M-Module register
mreadid	Read the ID PROM of an M-Module
mwrite	Write an M-Module register
mwriteblock	Write a block of M-Module registers
mwritefifo	Repeatedly write an M-Module register
output	Output data to VXI-11 interface
version	Get the version of the EM405-8 Extension Library

emclose ()	
Description:	This function must be called before exiting any script that loads the EM405-8 Extensions. It closes and frees resources that were allocated during library load.
Parameters:	None
Return:	None

emfans (full_on)	
Description:	Controls the speed of the EM405-8 fans. The fans can be full-on or variable
Parameters:	full_on Integer with 1 = fan set to full-on and 0 = fan set to variable
Return:	None

status, value = emread (offset)	
Description:	Reads an EM405-8 control register. These registers are shared between all M-Modules and control the carrier utilities such as triggers and fans. Refer to the EM405-8 User's Manual for register details.
Parameters:	offset Integer specifying the register offset to read
Return:	status integer indicating status of the read (0 = success) value the integer value read from the register

temp = emtemp (sensor)	
Description:	Read one of three temperature sensor and return the result in degrees C
Parameters:	sensor Integer indicating the temperature sensor to read. 0 = Fan Area, 1 = Logic Area, 2 = M-Module Area
Return:	temp Real value of temperature in degrees C

status = emwrite (offset, value)	
Description:	Writes an EM405-8 control register. These registers are shared between all M-Modules and control the carrier utilities such as triggers and fans. Refer to the EM405-8 User's Manual for register details.
Parameters:	offset Integer specifying the register offset to read value Integer value to write to the register
Return:	status Integer indicating status of the read (0 = success)

num_input, buffer = input (length)

Description: Inputs *length* bytes from the VXI-11 interface and returns the data in *buffer*. The data may be ASCII or binary. Refer to section 3.3 for details passing data via the VXI-11 interface.

Parameters: length Integer number of bytes from the buffer to be input

Return: num_input Integer number of bytes successfully input; 0 = no data is available.
buffer Buffer of data that is input from the VXI-11 interface

Note: Only one script at a time should attempt to input data from the VXI11 interface

status, value = mread (module, width, offset)

Description: Reads an M-Module register from a specified M-Module position.

Parameters: module Integer indicating the M-Module position to read. (0 – 7)
width Integer specifying the access width in bytes
2 = 16-bit
4 = reserved for future use
offset Integer specifying the register offset to read. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.

Return: status Integer status of the M-Module read (0 = success)
value Integer value read from the M-Module register

status, buffer = mreadblock (module, width, offset, length)

Description: Read a block of M-Module registers from a specified M-Module position. The function will read *length* words incrementing the address between each read

Parameters: module Integer indicating the M-Module position to read. (0 – 7)
width Integer specifying the access width in bytes. This value also determines the increment of the address for each read in the block.
2 = 16-bit
4 = reserved for future use
offset Integer specifying the register offset to read. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.
length Integer indicating the number of words of size *width* to read.

Return: status Integer status of the block read (0=Success)
buffer Buffer of data from the block read

status, buffer = mreadfifo (module, width, offset, length)		
Description:	Read a single M-Module register repeatedly from a specified M-Module position. The function will read <i>length</i> words from the specified offset without incrementing the address between each read	
Parameters:	module	Integer indicating the M-Module position to read. (0 – 7)
	width	Integer specifying the access width in bytes. 2 = 16-bit 4 = reserved for future use
	offset	Integer specifying the register offset to read. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.
	length	Integer indicating the number of words of size <i>width</i> to read.
Return:	status	Integer status of the FIFO read (0=Success)
	buffer	Buffer of data from the FIFO read

status, value = mreadid (module, word)		
Description:	Reads one 16-bit word of the specified M-Module's ID PROM	
Parameters:	module	Integer indicating the M-Module position to read. (0 – 7)
	word	Integer specifying the 16-bit word of the ID PROM to read (0 for first 16-bit word, 1 for second 16-bit word and so forth)
Return:	status	Integer status of the ID read (0 = success)
	value	Integer value read from the M-Module ID PROM

status = mwrite (module, width, offset, value)		
Description:	Writes an M-Module register to a specified M-Module position.	
Parameters:	module	Integer indicating the M-Module position to write. (0 – 7)
	width	Integer specifying the access width in bytes 2 = 16-bit 4 = reserved for future use
	offset	Integer specifying the register offset to write. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.
	value	Integer value to write to the M-Module register
Return:	status	Integer status of the M-Module read (0 = success)

status = mwriteblock (module, width, offset, length, buffer)

Description: Write a block of M-Module registers to a specified M-Module position. The function will write *length* words incrementing the address between each write

Parameters:

module	Integer indicating the M-Module position to write. (0 – 7)
width	Integer specifying the access width in bytes. This value also determines the increment of the address for each read in the block. 2 = 16-bit 4 = reserved for future use
offset	Integer specifying the register offset to write. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.
length	Integer indicating the number of words of size <i>width</i> to write.
buffer	Buffer of data for the block write

Return: status Integer status of the block read (0=Success)

status = mwritefifo (module, width, offset, length, buffer)

Description: Write a single M-Module registers repeatedly to a specified M-Module position. The function will write *length* words from the specified offset without incrementing the address between each write

Parameters:

module	Integer indicating the M-Module position to write. (0 – 7)
width	Integer specifying the access width in bytes. 2 = 16-bit 4 = reserved for future use
offset	Integer specifying the register offset to write. This value must be bounded by the width parameter. For example, if width = 2 then this value must be a multiple of 2 (0, 2, 4...). If width = 4 then this value must be a multiple of 4 (0, 4, 8...). Valid values are 0 - 0xFF.
length	Integer indicating the number of words of size <i>width</i> to write.
buffer	Buffer of data for the FIFO write

Return: status Integer status of the FIFO read (0=Success)

num_output = output (buffer , length)

Description: Outputs *length* bytes to the VXI-11 interface from the data in *buffer*. The data may be ASCII or binary. Refer to section 3.3 for details passing data via the VXI-11 interface.

Parameters:

length	Integer number of bytes from the buffer to be output
buffer	Buffer of data that is output to the VXI-11 interface

Return: num_output Integer number of bytes successfully output

version = version ()		
Description:	Get the version information of the EM405-8 Extension Library (this library).	
Parameters:	None	
Return:	version	String describing the version of the library

3.3 PASSING DATA

Data, including commands, can be passed to a script when a script is launched by simply placing the data in the script arguments. But, what if we want to pass data to a script at some indeterminate time after launch and what if the script wants to return data? The EM405-8 Extensions Library, discussed in section 3.1, provides both input and output utilities that allow for data passing between the script and the host computer using the VXI-11 interface. For the purpose of this discussion, the direction of data is in reference to the script. In other words, input refers to passing data from host computer to the script and output refers to passing data from the script to the host computer.

The EM405-8 extensions implement two FIFO's for data passing, one for each direction. To place data in the input FIFO, the host computer must send the data preceded by the `*data` command via the VXI-11 interface. The `*data` command is discussed in section 2.1.2. For example, to pass the string "Hello World" to a script, the host computer can perform a VISA call as follows:

```
printf(string, "*data Hello World");
viWrite(vi, strlen(string), string);
```

The data can be ASCII like the above example or binary like the following example that sends binary 1, 2, 3, and 4.

```
printf(string, "*data');
string[5] = 1;
string[6] = 2;
string[7] = 3;
string[8] = 4;
viWrite(vi, 9, string);
```

Note: If a script is not running to accept that data, the VISA function `viWrite` will return an I/O error.

To retrieve data from the input FIFO, the script must call the EM405-8 Extensions Library function `input`. The EM405-8 Extensions Library functions are detailed in section 3.1. The `input` function takes a single parameter specifying the number of bytes to input. The `input` function is non-blocking meaning that if the specified number of bytes is not available in the FIFO, the function will return anyway. In addition, being non-blocking also means that the script must poll for input data either on a regular basis or when data is expected. The return value of `input` specifies the number of bytes retrieved from the FIFO. There is only one input FIFO, therefore only one script should input data at any given time. If more than one script calls `input` at the same time, the behavior is indeterminate.

To output data from a script to the host computer, the script can use the EM405-8 Extensions Library function `output` to place data into the output FIFO. The host computer can in-turn use the VISA function `viRead` to retrieve the data from the output FIFO. Unlike `input`, more than one script can output data to the VXI-11 interface; however, there is no built-in indication as to which script a piece of data is from nor is there a defined delimiter to separate data from one script with data from another script. All data is dumped into a single buffer to be read from the host. It is guaranteed however, assuming no errors, that all data written within a single call to `em405.output` will be contiguous in the buffer. With that said, there is nothing preventing the scripts themselves from adding an ID tag and a delimiter to the data itself. In this case, it will be up to the host application to parse the data. Also, in contrast to `input`, there is always something (i.e. the VXI-11 firmware) waiting to retrieve data from the output FIFO, therefore the user may call `output` at any time without error.

To illustrate data passing we have created an example script that simply retrieves data from the VXI-11 interface using the `input` function then echo's it back to the VXI-11 interface using the `output` function. The Lua source code is found in Figure 6. The host side simply uses `viWrite` to send the data and `viRead` to retrieve the echo.

```
em405 = require "lua_em405"
io.write("Echo Example\n")
io.write("VXI-11 data will be echoed via VXI-11\n")
io.flush()

while 1 do
    num_read, buffer = em405.input(256)
    if num_read > 0 then
        em405.output(buffer, num_read);
    end
end
```

Figure 6 Data Passing Example

3.4 CALLING M-MODULE DRIVERS

The scripting utilities include a Lua library known as *Alien* that allows a Lua script to call dynamic libraries (.so, dll, etc.). Using *Alien*, we can make calls to any of C&H Technologies' standard M-Module drivers. *Alien* uses the Foreign Function Interface (FFI) library to dynamically call shared library functions. *Alien* was written by a third party and like Lua, is open source software.

The alternative to using *Alien* is to write a “wrapper” around the driver that makes the library a native Lua component. Tools do exist that can automatically generate a wrapper however they still require knowledge of Lua C components and the tools necessary to cross compile the source code for ARM based Linux. By using *Alien*, we can utilize M-module drivers as-is without the need of building more software.

The downside to using Alien is that ANSI-C code is inherently not “safe.” In other words, the code can perform actions that result in undefined behaviors, such as dereferencing an invalid address. In a C program or library this will result in a crash of the application. In contrast, Lua is a safe language where no matter the code, the behavior is defined. Alien is written to be as safe as possible; however it is still easy to crash Lua from within a library, thus, to avoid frustration, care must be taken when calling drivers using Alien.

Crashes due to a misbehaving C library typically only occur during development. Once a program is debugged, it is rare that a crash will occur. Recovering from a crash typically involves rebooting or power-cycling the EM405-8. A system reboot can be performed remotely via the EM405-8 webpage under the Status/Control link.

3.4.1 Using Alien to Call M-Module Drivers

The usage of Alien to call M-Module drivers can be divided into three basic steps:

1. Load the driver library
2. Define function prototypes for the functions to be called
3. Perform function calls.

To load a driver library from within a script, we must first load the Alien Lua module using the `require` function like we would any other Lua library. In addition to the alien library, we may also sometimes load a library called “alien.struct.” This library can be used to convert Lua values to and from C structures. It is a good idea, in most cases, to `require` alien.struct by default. Once Alien is loaded, we can use the `alien.load` function to load the M-Module driver. The Lua source code below shows how to perform these steps. Note that there are multiple ways to perform the same tasks in Lua source code. For clarity, we have chosen to present the most straight forward way.

```
require "alien"  
require "alien.struct"  
libm228 = alien.load("/usr/scripts/user/libm228.so")
```

Note that in the above example code, the path `/usr/scripts/user/` is the absolute path of the script pool where `libm228.so` will be uploaded. The absolute path must be specified for Alien to find the shared library.

To use a function in the driver you first must tell Alien the function prototype of driver. This is performed with the call `lib.func:types(return_type, arg_types...)`. For each driver function that is to be called. The types are specified as a string and can be any of the types specified in Table III. Most correspond directly to C data types.

Table III. Alien Data Types

Alien Data Type	C Data Type
“void”	void
“int”	int
“double”	double
“char”	char
“string”	const char*
“pointer”	void*
“ref int”	int <i>passed by reference</i>
“ref double”	double <i>passed by reference</i>
“ref char”	char <i>passed by reference</i>
“callback”	<i>generic function pointer</i>
“short”	short
“byte”	signed char
“long”	long
“float”	float

The following example code illustrates how to define the function prototype of the M228 driver initialize function whose C prototype is:

```
int m228_Init(int path, int id_query, int reset, M228_HANDLE *handle);
```

Note that in the below example, we create a variable of the same name as the C function that is equal to *lib.func*. This is not required but makes for more readable code as from that point on, the function can be called with the function name.

```
m228_Init = libm228.m228_Init  
m228_Init:types("int", "int", "int", "ref int")
```

Function prototypes only need to be defined once and can be defined anywhere in the script as long as it is before the function is called. With that said, it makes for more readable code if all prototypes are defined in one section near the top of the code.

Once the prototype is defined the function can be called as follows:

```
path = 0  
id_query = 1  
reset = 1  
error, handle = m228_Init(path, id_query, reset, 0)
```

You may find it strange that there are two variables specified on the left side of the equal sign. The first variable is the return value of the function. The second and subsequent variables are the return values of the parameters that are passed by reference. Alien supports by-reference types by allocating space for the argument on the Lua stack then initializing this stack space to the Lua number (typecasting where needed) specified in the function call (0 in the example above). When the C library function returns, Alien takes the value and returns it (converting where appropriate) after the functions normal return value.

To put this all together, the example source code in Figure 7 uses the M228 M-module driver to read 20 locations of the M-modules ID PROM. This example uses three driver functions:

m228_Init, m228_GetIdentWord, and m228_Close. Note that these examples use the m228_GetIdentWord function not the EM405-8 Extensions Library function mreadid. This same example can be performed using just the EM405-8 Extensions Library. The example prints a line to the standard output for each ID location indicating the ID address and the value read. Note that the functions print and string.format are part of the Lua standard libraries.

```

require "alien"
require "alien.struct"

local libm228 = alien.load("/usr/scripts/user/libm228.so")

-----
-- Function Prototypes
-----

local m228_Init = libm228.m228_Init
m228_Init:types("int", "int", "int", "int", "ref int")

local m228_GetIdentWord = m228_lib.m228_GetIdentWord
m228_GetIdentWord:types("int", "int", "int", "ref int")

local m228_Close = m228_lib.m228_Close
m228_Close:types("int", "int")

-----
-- Start of Script --
-----

id_query = 1
reset = 1
module = 1

result, handle = m228_Init(module,id_query,reset, 0)

for offset=0, 19, 1
do
    result, value = m228_GetIdentWord(handle, offset, 0)
    print(string.format('0x%x:  0x%x', offset, value))
end

m228_Close(handle)
print("Done!")

```

Figure 7 M-Module Driver Example Code

3.4.2 Dealing with data

The data types shown in Table III are straightforward and cover most function calls to M-Module drivers. But there are M-Module driver function calls that, for example, require special data types (C structures) or require the calling software to allocate memory that the driver then modifies. In both cases, there are no data types in Table III to handle this type of data.

The Alien library provides a host of functions and utilities that can be used to deal with data types not covered by the standard types shown in Table III. The following sections discuss the functions and utilities that are most pertinent to M-Module Drivers. These sections are not meant to replace Alien documentation which may include further details and a more complete list of utilities.

3.4.2.1 Buffers

Alien buffers allow the script to allocate a block of memory that can then be passed to a function in place of any string or pointer type. A buffer is allocated with `alien.buffer()`. If nothing is passed to `alien.buffer()` a buffer of default length is allocated. If a number is passed a buffer of that length is allocated. If a string or userdata is passed to `alien.buffer()`, a buffer will be allocated and initialized with a copy of the string or data. Once the driver function is called, the function `buf:tostring()` can be called to retrieve the data from the buffer. Individual bytes of the buffer can be accessed using `buf[i]` where `i` is 1-based not 0-based. Further you can get or set non-byte values using the `buf:get(offset, type)` and `buf:set(offset, type)`. Note that the `buf:get()` and `buf:set()` the offset is 1-based and is in bytes not elements so if the buffer has integers, their offsets are 1, 5, 9, etc. An example of using buffers in a M-Module driver call is shown in Figure 8.

```

require "alien"
local libm228 = alien.load("/usr/scripts/user/libm228.so")

-----
-- Function Prototypes
-----
local m228_Init = libm228.m228_Init
m228_Init:types("int", "int", "int", "int", "ref int")

local m228_ReadFifoBuffer = m228_lib.m228_ReadFifoBuffer
m228_ReadFifoBuffer:types("int", "int", "int", "int", "pointer",
                          "pointer", "pointer", "ref int")

local m228_Close = m228_lib.m228_Close
m228_Close:types("int", "int")

-----
-- Start of Script --
-----

result, handle = m228_Init(1,1,1, 0)

time_buff = alien.buffer(256 * alien.sizeof("int"))
real_buff = alien.buffer(256 * alien.sizeof("double"))
raw_buff = alien.buffer(256 * alien.sizeof("short"))

result, num_read = m228_ReadFifoBuffer(handle, 0, 256, time_buff,
real_buff, raw_buff, 0)

for offset=0, 256, 1
do
    print(string.format('0x%x', raw_buff:get((i *
        alien.sizeof("short")) + 1, "short")))
end

m228_Close(handle)
print("Done!")

```

Figure 8 M-Module Driver Example Using Buffers

3.4.2.2 Arrays

Alien arrays build upon buffers by adding an extra layer of safety and elements that describe the array. The function `alien.array(type, length)` allocates an array where `type` is the Alien type of the elements and `length` is the number of elements in the array. Once allocated, the user may use `arr.type` to get the type of the elements, `arr.length` to get the number of elements in the array, `arr.size` to get the size in bytes of each element and `arr.buffer` to get the underlying buffer. The user can access the elements of an array directly with `arr[i]`. The example in Figure 9, perform the same exact function as the example in Figure 8 but with arrays instead of buffers.

```

require "alien"

local libm228 = alien.load("/usr/scripts/user/libm228.so")

-----
-- Function Prototypes
-----
local m228_Init = libm228.m228_Init
m228_Init:types("int", "int", "int", "int", "ref int")

local m228_ReadFifoBuffer = m228_lib.m228_ReadFifoBuffer
m228_ReadFifoBuffer:types("int", "int", "int", "int", "pointer",
                          "pointer", "pointer", "ref int")

local m228_Close = m228_lib.m228_Close
m228_Close:types("int", "int")

-----
-- Start of Script --
-----
result, handle = m228_Init(1,1,1, 0)

time_array = alien.array("int", 256)
real_array = alien.array("double", 256)
raw_array = alien.array("short", 256)

result, num_read = m228_ReadFifoBuffer(handle, 0,
time_array.length, time_array.buffer, real_array.buffer,
raw_array.buffer, 0)

for offset=0, 256, 1
do
    print(string.format('0x%x', raw_array[i]))
end

m228_Close(handle)
print("Done!")

```

Figure 9 M-Module Driver Example Using Arrays

3.4.2.3 Pointer Unpacking

Pointer can be de-referenced and converted to a Lua type using one of the functions `alien.tostring`, `alien.toint`, `alien.toshort`, `alien.tolong`, `alien.tofloat`, and `alien.todouble`. These functions should be used with care as they do not check whether the dereference or the typecast is safe. More robust unpacking can be performed using the `alien.struct.unpack` function.

3.4.2.4 Callback Functions

In some rare cases an M-Module driver will take a function pointer as a parameter and call that function to perform a certain task. This is known as a callback function. Alien provides the

function `alien.callback()` that lets the user create a callback by passing it the function and prototype of the callback function. The function `alien.callback()` will return an object that can be passed to any argument of callback type.

3.4.3 Building M-Module Drivers into a Linux Shared Library

In order for Alien to call a function within a M-Module driver, the driver must be built into a Linux Shared Library (.so) file for the XScale ARM processor on the EM405-8 and the file must be hosted on the EM405-8. C&H Technologies, writes M-Module drivers in an architecture that makes it easy to port the driver to various operating systems and processors (See the application note titled M-Module Instrument Driver Architecture for details). Thus, it is easy to build one of C&H's M-Module drivers into a Linux shared library for the XScale ARM processor. Instructions to do so are beyond the scope of this document. If an M-module's driver is not available as a Linux shared library then please contact C&H Technologies for assistance.

The Alien library is configured to look for dynamic libraries in the same location that scripts are stored. Therefore hosting an M-module driver on the EM405-8 is as simple as uploading the driver like any script using the `upload` command as detailed in section 2.1.10.

3.5 SHARING DATA BETWEEN SCRIPTS

Lua scripts have access to the complete set of Lua standard libraries and the EM405-8's Linux kernel upon which the Lua interpreter is running contains all the common utilities such as a file system, messaging utilities including pipes and FIFO's, and sockets. These utilities can be very useful to pass data between independent scripts; however, the details of using the standard utilities and recommendations as to which method is best to use are beyond the scope of this document. The intent of this paragraph is simply to highlight that these utilities are available and may be very useful to the system developer.

3.6 STARTUP SCRIPT

Upon bootup or system reset, the EM405-8 will search for a startup script named *startup.lua* in the pool of user stored scripts. If *startup.lua* is found, the EM405-8 will automatically run this script. This utility allows the system to run autonomously without the need for a host computer's intervention. In reality, this utility will allow the system to run without the network cable plugged in for a long period of time.

The startup script can perform any task a normal script can perform, including, configuring M-Modules, sending and receiving VXI-11 data, acquiring data, storing data to the mass storage device, starting other scripts and so forth. In addition, the startup script is listed with the `list -r` command and can be halted like any other script.

To utilize the startup script functionality, simply upload a script with the name *startup.lua*. Since the script must be present at bootup, the startup script function requires the mass storage option of the EM405-8 which provides non-volatile storage of scripts.

3.7 USING THE MASS STORAGE DEVICE

In addition to providing non-volatile storage of scripts, the mass storage device (ordering option -0003 only) provides the user with a large amount of non-volatile memory for storage of user data. The EM405-8 system only uses the mass storage device for the script pool. The remainder of the device is available for user data.

Scripts can store and retrieve data to/from the mass storage device using the standard file I/O functions found in the Lua I/O Library. The mass storage device is mapped to the directory `/media`. A single subdirectory `/media/scripts` is created by the system for the script pool. Care should be taken not to overwrite or erase the script pool. If desired, the user can create subdirectories from the Interactive Mode of the scripting socket or from within a script using the `execute` function in the Lua Operating System Library as show in the following example:

```
os.execute("mkdir /media/userdir")
```

4.0 USING LUA TO WRITE CUSTOM WEB PAGES

Using Lua, the user may create custom web pages to be served by the EM405-8's embedded web server. In addition to creating the pages, themselves, the user can add links to the standard navigation menu that is found on every one of the default web pages of the EM405-8. The scripts that create the custom web pages have full access to the EM405-8 scripting utilities like any standard script. Scripts can, among other things, configure hardware, retrieve status, write/read files, and send/receive data.

Lua based web pages are called from a CGI executable that maps the standard console output (stdout) to the requesting web browser. In other words, all standard output from the standard I/O library functions such as `print()` and `io.write()` go directly to the web browser. Therefore, a Lua script can create a webpage by simply outputting the HTML like the example in Figure 10.

```
print([[
  <html>
  <head>
    <title>An HTML Page</title>
  </head>
  <body>
    <a href=http://www.chtech.com">C&H Technologies, Inc.</a>
  </body>
</html>
]])
```

Figure 10 Simple Lua Based Web Page

Note that in the example in Figure 10, double square brackets ([[&]]) start and stop the print statement instead of the standard double quotes (“”). In Lua, double square brackets denote a literal string where there are no special characters or escape sequences. This is especially useful when using Lua to create a webpage as it allows the developer to input the HTML exactly as it would be in a standard HTML file.

Unlike standard scripts, web-based scripts do not run in their own thread; therefore they must not loop forever, nor can they be queried using `list -r` or halted. In fact, for most, if not all, browsers, a webpage will not be displayed until the script is complete. If a script needs to be run in a loop, the developer should use a Meta Refresh Tab to instruct the web browser to automatically refresh the page (i.e. run the script) after a certain period of time.

4.1 URL OF SCRIPT BASED PAGES

As mentioned in section 4.0, Lua web page scripts are called from a CGI executable. This executable is named *script.cgi* and takes the script filename as a parameter. There are actually two locations of *script.cgi* one for public pages at `/cgi-bin/script.cgi` and one for private password-protected pages at `/cgi-bin/private/script.cgi`. The script filename is passed to *script.cgi* within the URL using the standard name/value pair parameter passing.

For example, if your script is named *myscript.lua* then the URL would be:

```
/cgi-bin/script.cgi?script=myscript.lua  
or  
/cgi-bin/private/script.cgi?script=myscript.lua
```

Of course, typing these URL's directly into a browser would require the above to be preceded with the IP address; however, these are the absolute paths from within the device itself and therefore the paths that should be used for all links within the HTML.

The CGI executable *script.cgi* will search the standard pool of Lua scripts. Therefore, scripts that create web pages will be intermixed with non-webpage scripts and the user may use the standard upload command to store the script on the device.

Arguments to be passed to the script are also passed within the URL by adding a name/value pair within the URL. Details on passing arguments to the script can be found in section 4.4.

4.2 ADDING A LINK TO THE C&H NAVIGATION MENU

The user can add links to the main navigation menu displayed on all C&H standard web pages by simply creating a file named "web_navbar.lua" and adding links to this file. This file should be uploaded to the standard script pool like any other script.

The navigation menu bar is an HTML table with each link in a separate row. To add a link, "web_navbar.lua" must include the HTML to create a new row. The example in Figure 11 illustrates how to add two links to the navigation menu.

```
print([[  
  <tr><td align="left" bgcolor="EFEFEF">  
    <a href="/cgi-bin/script.cgi?script=myscript.lua">  
      My Page  
    </a>  
  </td></tr>  
  
  <tr><td align="left" bgcolor="EFEFEF">  
    <a href="/cgi-bin/script.cgi?script=myscript_2.lua">  
      My Page #2  
    </a>  
  </td></tr>  
  
]])
```

Figure 11 Example Adding Link to the Navigation Menu

4.3 URL OF PICTURES AND OTHER SUPPORTING FILES

Sometimes a script based webpage will need to link to other supporting files such as graphics and images. A directory name *scripts* on the root level of the URL is provided to give the user a location for supporting files.

In reality, */scripts* is simply a symbolic link to the standard pool of scripts in which all scripts are stored. Therefore, the user may simply store images or other supporting files like he/she would scripts and link to them with the URL */scripts/user/file*.

The example in Figure 12, shows the script for a webpage that links to an image names *logo.jpg*. The user must upload both this script and the image *logo.jpg* for the webpage created by this script to display properly.

```
print([[
  <html>
  <head>
  <title>An HTML Page</title>
  </head>
  <body>
    
  </body>
  </html>
]])
```

Figure 12 Example Linking to a User Image

4.4 PASSING ARGUMENTS TO THE SCRIPT

Arguments are passed to the script via URL by simply adding name/value pairs to the URL. For example:

```
/cgi-bin/script.cgi?script=myscript.lua&arg1=val1&arg2=val2
```

In the above example, the script receives the arguments *val1* and *val2*. The labels (*arg1* and *arg2*) are discarded before running the script therefore they may be any identifier. Also, because the labels are discarded, the order of the arguments is important as the script has no way to identify an argument other than the order in which it was received.

Arguments are received by the script in the same manner is if the script was called from the command line. When a script is called, a Lua table named *arg* is defined and populated with the arguments. Table item *arg[0]* contains the name of the script. The arguments are placed in *arg[1]*, *arg[2]* and so forth in the order in which they appear in the URL.

Figure 13 illustrates the Lua source code required to retrieve the arguments. In this example, the table *arg* is printed to the webpage as the index followed by the argument.

```

print([[
  <html>
  <head>
  <title>An HTML Page</title>
  </head>
  <body>
]])

for i=0, table.getn(arg) do
  print(i,arg[i])
  print("<br>")
end

print([[
  </body>
  </html>
]])

-----
URL:
  /cgi-bin/script.cgi?script=mymyscript.lua&arg1=val1&arg2=val2

Output:
  0 myscript.lua
  1 val1
  2 val2

```

Figure 13 Example Retrieving Arguments from the URL

4.5 DEVELOPING THE PAGE CONTENT

The user may customize the webpage to look however he/she likes by creating the entire HTML content. Alternatively, the user can maintain the EM405-8 default look and feel of the webpage by using the `l_em405web.lua` Lua library. This library contains a set of functions that can be used to generate the main HTML structure that is used in the EM405-8’s default pages. By using this library, the developer needs only to concentrate on the added content of the custom page.

The library `l_em405web.lua` includes four functions as summarized in Table IV. Details of each function follow the table.

Table IV l_em405web.lua Library Functions

Function	Description
Top	Create the HTML header info and display the top banner
Bottom	Display the bottom banner & close the HTML
Contents_Pre	Setup a nested table including the navigation menu
Contents_Post	Close the nested table

Top(refresh, refresh_time, refresh_url)

Description: This function creates the HTML header information and displays the top banner including the C&H logo and banner graphic. The parameter allow the user to control the Refresh Meta Tag.

Parameters:

refresh	1 or 0 indicating whether or not to include the Refresh Meta tag in the HTML header
refresh_time	The refresh time placed in the CONTENT section of the Refresh Meta Tag. This is the number of seconds to wait before the browser should refresh the page.
refresh_url	The URL that the page load when the refresh time is complete. This allows the page to redirect to a different URL. Pass an empty string to this parameter to refresh to the same URL.

Bottom()

Description: This function display the bottom banner including the LXI logo and closes the page with the </body> and </html> tags.

Parameters: None

Contents_Pre()

Description: This function creates a nested table that includes the navigation menu and the page content. This function leaves the table open so that the user may insert his/her content. The last html tag output by this function is a <td>. The user can follow this function with his/her content then close th <td> tag the table by calling the function Contents_Post().

Note: The user may call Top() and Bottom() without calling Contents_Pre() Contents_Post() since Top() does not leave any HTML tags open.

Parameters: None

Contents_Post()

Description: This function closes the <td> tag left open by Contents_Pre() and then subsequently closes the nested table. The user should always call Contents_Post() if Contents_Pre() was used before the page content. The first HTML tag output by this function is </td>.

Note: The user may call Top() and Bottom() without calling Contents_Pre() Contents_Post() since Top() does not leave any HTML tags open.

Parameters: None

As discussed in the above descriptions, the l_em405web.lua functions create a nested table with the user content being one table cell. The best way to learn the effects of using the l_em405web.lua library is to add them to your script, load the page in a web browser then view the source using the browsers utility to view source.

Figure 14 shows an example web page using the l_em405web.lua library. This example performs the same function as the example in Figure 13; however it uses the l_em405web.lua library instead of explicitly outputting the full HTML.

```
em405web = require "l_em405web"

em405web.Top(0,0," ")

em405web.Contents_Pre()

-- Contents Begin Here --
for i=0, table.getn(arg) do
    print(i,arg[i])
    print("<br>")
end

-- Contents End Here --

em405web.Contents_Post()

em405web.Bottom()
```

Figure 14 Example Using “l_em405web.lua” Library

It is highly recommended that you use the W3C Markup Validation Service to validate you HTML. This is a free utility available at: <http://validator.w3.org/>. By validating your HTML, you are ensuring that it will be properly displayed on any W3C compliant web browser.

NOTES:

READER'S COMMENT FORM

Your comments assist us in improving the usefulness of C&H's publications; they are an important part of the inputs used for revision.

C&H Technologies, Inc. may use and distribute any of the information that you supply in any way that it believes to be appropriate without incurring any obligation whatsoever. You may, of course, continue to use the information, which you supply.

Please refrain from using this form for technical questions or for requests for additional publications; this will only delay the response. Instead, please direct your technical questions to your authorized C&H representative.

COMMENTS:

Thank you for helping C&H to deliver the best possible product. Your support is appreciated.

Sincerely,

F. R. Harrison
President and CEO

INSTRUCTIONS

In its continuing effort to improve documentation, C&H Technologies, Inc. provides this form for use in submitting any comments or suggestions that the user may have. This form may be detached, folded along the lines indicated, taped along the loose edge (DO NOT STAPLE), and mailed. Please try to be as specific as possible and reference applicable sections of the manual or drawings if appropriate. Also, indicate if you would like an acknowledgment mailed to you stating whether or not your comments were being incorporated.

NOTE: This form may not be used to request copies of documents or to request waivers, deviations, or clarification of specification requirements on current contracts. Comments submitted on this form do not constitute or imply authorization to waive any portion of the referenced document(s) or to amend contractual requirements.

————— (Fold along this line) —————

————— (Fold along this line) —————

Place Stamp Here

C&H Technologies, Inc.
Technical Publications
445 Round Rock West Drive
Round Rock, Texas 78681-5012