NetStreams™ DigiLinX™

Writing *StreamNet*™ Device Drivers

Copyright

Title: Writing *StreamNet* Device Drivers
Document Number: 020014A
Original Publication Date: August 14, 2006
Revision Date: February 27, 2007

3600 W. Parmer Lane, Suite 100

Austin, TX 78727

USA

Phone: +1 512.977.9393

Fax: +1 512.977.9398

Toll Free Technical Support 1-866-353-3496

Contents

Chapter

1

Introduction

NetStreams® ships drivers for a variety of third party systems with the *DigiLin*X™ system. These drivers include Lutron® for lighting, Aprilaire® for HVAC, and GE Concord® for security. Custom installers can use these included drivers with a *ControLin*X™ to enable *DigiLin*X control of external systems. Sometimes, the specific needs of a job require *DigiLin*X to control different third party systems. The *NetStreams StreamNet*™ device driver Application Programming Interface (API) is designed to enable programmers to interface RS-232 and network devices with the *NetStreams DigiLin*X system. This gives custom installers the flexibility they need for their jobs.

The *DigiLin*X system is a combination of hardware, software, and firmware. To understand how to interface third party systems with *DigiLin*X, programmers must understand the architecture of the system (see Figure 1-1).

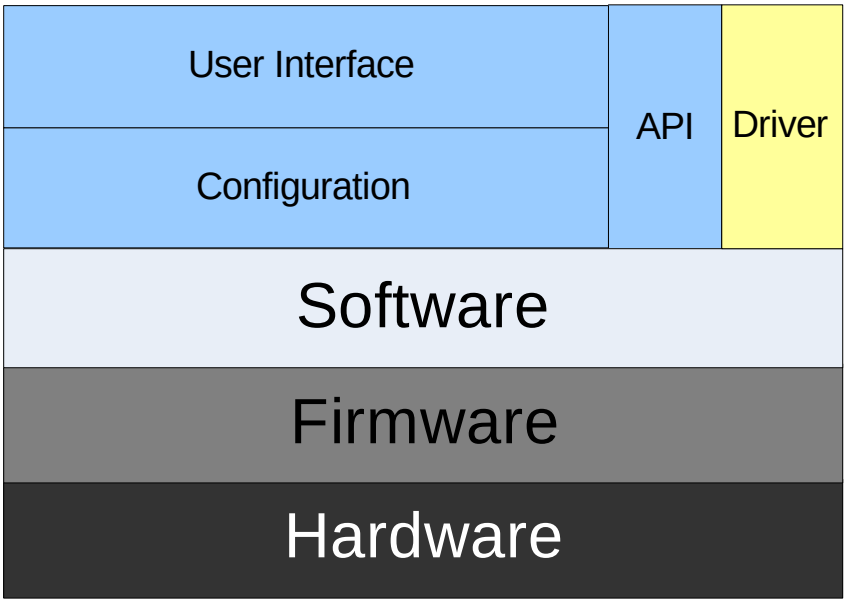

Figure 1-1 *DigiLin*X Hardware/Software model

In the *DigiLin*X system, the hardware layer is controlled by firmware. Firmware is controlled and updated by software, which in turn is controlled by configuration files, and drives the User Interface. *NetStreams*' engineering team has created an API that

enables *DigiLinX* to control and be controlled by third party systems using a driver (seeFigure 1-2).

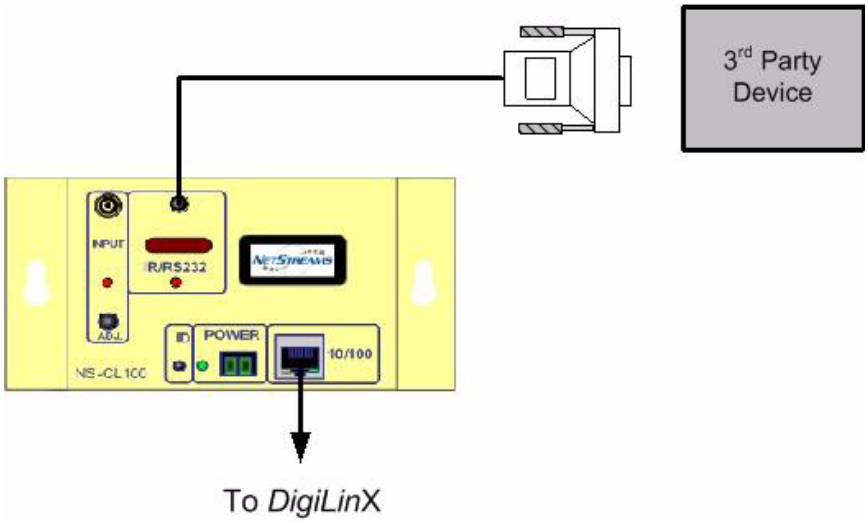

Figure 1-2 Connecting 3rd Party Systems to *DigiLinX*

Device drivers allow two-way interaction with third party hardware devices over a communications subsystem, such as RS-232. The *ControLinX* device includes the hardware needed to control external systems.

The *DigiLinX* API is written in a programming language called Lua. Lua is a free, lightweight, procedural language that is designed to be fast and easy to learn for most programmers. Third party drivers must be written in Lua to work with the *DigiLinX* API.

Third party drivers are composed of objects called Controls. These Controls are represented by all of the functions and variables in the driver's code. Drivers may also contain code to control subcomponents, called subNodes. *DigiLinX* treats subNodes as logical entities for both accepting commands and producing status messages. For example, in a lighting system, a subNode might represent a specific lighting load. Commands addressed to this subNode would then only affect that lighting load and all status messages related to the load would be addressed from the subNode.

Introduction to Lua

Lua is a byte-code interpreted language, similar to Java. It provides a small scripting and user-customizable interface to a subset of an application written in C or C++.

Where Java is primarily focused on providing all of the tools necessary to write a complete application, Lua is focused on providing minimal system interfaces and a tightly coupled interface to an underlying application.

The benefits of Lua include:

- Extension Language – Lua is designed from the ground up to be an extension language. This is a language solely used to extend the functionality of a larger application. This means there is a two-way interface that easily allows Lua code to invoke native subroutines and vice-versa. The vast majority of the functional

portion of the control system can be written in C/C++ while a minimum of device specific code is written in interpreted Lua.

- Minimal System Interface and Dependence – The Lua virtual machine and its associated parser/compiler are written in ANSI-standard C with a minimum system interface. There is no operating system (OS) interface defined at the script level.
- Native String Manipulation – The bulk of the necessary string operations (concatenation, searching, formatting, regular expressions, etc.) are implemented in ANSI C, rather than being implemented in the script language. This provides for substantially improved performance given that the majority of control applications are string parsing and generation.
- Virtual Machine (VM) implementation – The Lua scripting language is compiled (either at runtime or in advance) into a VM-interpreted language that is subsequently run on a virtual machine designed with the language constraints in mind. This substantially improves performance over similar languages that are fully interpreted.
- Object-oriented – While Lua is not an object-oriented language, there are sufficient language features that it can be treated as such.

The Lua script engine and its associated tools are licensed under a MIT-style license, which states that it is copyrighted material but free license is granted for commercial and non-commercial usage.

For the full Lua license see Appendix A, *Lua 5.0 License* on page A-1.

Further information on Lua is available either from the web site http://www.lua.org or *Programming in Lua* by Roberto Lerusalimschy.

Chapter

2

Activating Drivers within *DigiLinX* Dealer Setup

Starting with version 1.70, *DigiLinX* Dealer Setup uses an intelligent lookup to determine if there are custom written drivers present in the system.

In order to load custom written drivers through *DigiLinX* Dealer Setup, you must first create a new folder called “drivers” under your *DigiLinX* Dealer Setup install directory and copy the .lua driver file to this directory. In most cases, the directory will be:

C:\Program Files\DigiLinX Dealer Setup\drivers\yourdriverfile.lua

After you have created this folder and copied your driver file into the folder, build your project normally in *DigiLinX* Dealer Setup. Add a *ControLinX* or *MediaLinX* to your project, selecting the GUI type from the dropdown menu (for example, Tuner, Lutron, etc.). When you enter the details for your *ControLinX* or *MediaLinX*, Dealer Setup will detect the custom driver in the drivers folder and present the option to select a new .lua driver file.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / *www.netstreams.com.*

Chapter

3

Handling Commands

The generation and processing of ASCII-formatted commands is central to the interaction between various *StreamNet* devices and hence, the heart of a device driver is how it receives and process ASCII commands. Figure 3-1 shows how the *StreamNet* commands flow.

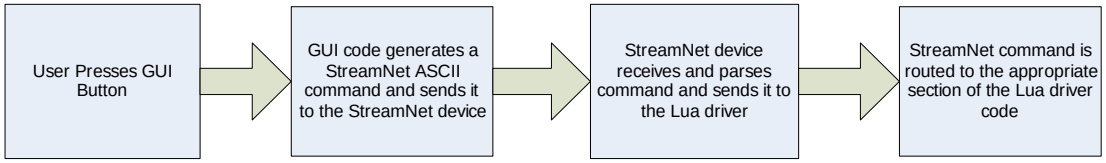

Figure 3-1 *StreamNet command flow*

As commands are received by the host *StreamNet* device, they are parsed to separate out the various addressing fields, ASCII command, and command parameters. Once the command is completely parsed, it is passed to the device driver for handling by the appropriate Lua code. Refer to Figure 3-2 to see how commands are routed within the driver.

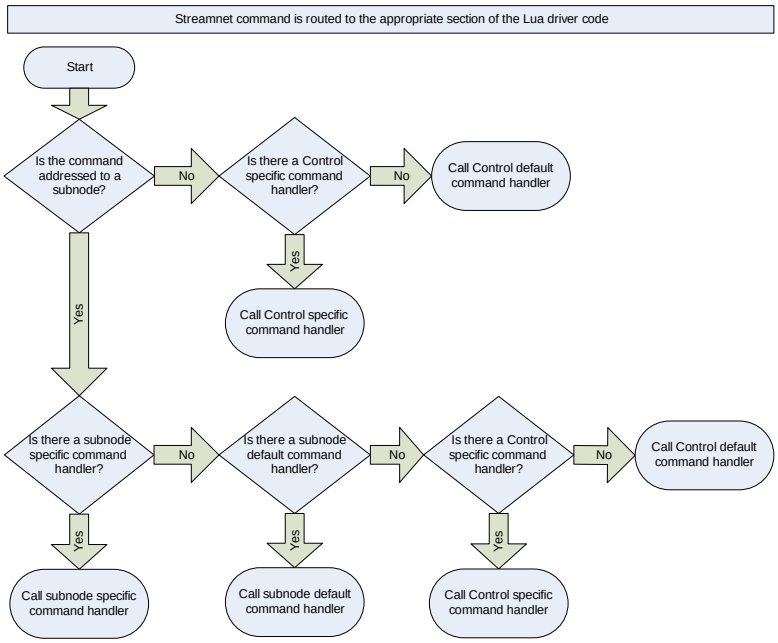

Figure 3-2 *Command routing within the driver*

Handling Commands

NOTE

It is always possible to define an appropriate command-specific handler, so that the default_command_handler is not needed. Programming best-practices state that a fallback default_command_handler should be defined even if it is not intended for use.

For a device driver to receive and handle commands, it is necessary to define Lua functions with predefined names based on the command to be handled. The routine that needs to be defined is “handle_<command>” (where <command> is replaced with the lower cased version of the ASCII command). For example, if a driver needs to handle #SET commands, it needs to define a function handle_set within the driver. As a fall-back mechanism, if no command-specific function is defined, the function “default_command_handler” will be invoked if defined.

If the command is generic to the entire device, the necessary function should be defined in the outermost control scope of the Lua file, and will be invoked with only the command as a parameter.

Example: Handle_Set

```
function handle_set(cmd)
if(cmd.params[1] == “heat”) then
      … handle the #set heat command
end
end
function default_command_handler(cmd)
```

```
if(cmd.command == “debug”) then
      … handle the #debug command
end
end
```

If the command is specific to a particular subNode of the driver, it should be defined as an element in the subNode. It will be passed the subNode itself as its only argument.

```
subNode.handle_set = function(self, cmd)
if(cmd.params[1] == “heat”) then
      … handle the #set heat… command
end
end
subNode.default_command_handler = function(self,
cmd)
if(cmd.command == “debug”) then
      … handle the #debug command
end
end
```

NOTE

In the previous example, Lua provides a convenient short-cut notation with identical functionality:

```
function subNode:handle_set(cmd)
if(cmd.params[1] == “heat”) then
      …
end
end
```

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / *www.netstreams.com.*

Chapter

4

Handling Communications

Most device drivers will need some mechanism for talking to a controlled device. In the *StreamNet* system, that capability is provided by a generalized I/O Stream mechanism.

A stream can be created using a string-based configuration string, the precise format of which depends on the details of the connection. Once the stream is created, the details of the underlying transport mechanism (such as RS-232, TCP/IP, or some other mechanism) is identical for all streams. This allows the same driver and code to be used to control a device regardless of the physical connection type.

A stream is created with the createStream function:

```
stream = createStream("comm://
0;baud=9600;parity=n");
```

Once created, the stream can be read from and written to using the read and write functions:

```
input = stream:read()
stream:write(input)
```

NOTE

Use the ":" syntax to pass the stream as an argument to the read and write functions.

Many devices have asynchronous outputs, i.e., outputs that are not in direct response to some command being sent down. The preferred mechanism of dealing with such responses is to define and enable an asynchronous response handler:

```
function input(stream, line)
… handle async response in "line"
end

stream:startAsyncInput(input, {endString = "\n"})
```

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Chapter

5

Configuration

As for any other *StreamNet* service, a *StreamNet* service with a Lua driver is configured by including an appropriate service tag within the config_current.xml file. In the case of standard *StreamNet* device types, the clause will be automatically generated, but in the case of custom device types, the clause may need to be manually generated.

```
<service serviceNumber=”2” serviceName=”AprilAire”
     serviceType=”gpio” enabled=”1”>
<control controlType=”SCRIPT”>
     <SCRIPT file=”AprilAire.lua”>
          <SCRIPT_DATA>
-- lines here are passed to the lua interpreter
before
-- the driver file is loaded
config = {};
config.port = “comm://0;baud=9600”;
          </SCRIPT_DATA>
     </SCRIPT>
</control>
</service>
```

The SCRIPT_DATA block defines a section of Lua that will be passed to the interpreter before the driver is loaded, allowing for installation specific parameters, such as the port settings above, to be passed to the driver.

NOTE: The serviceType in this example is currently required to be “gpio,” and the serviceNumber is a device specific constant (always 2 in the case of a CL100 or CL100A)

Chapter

6

Installing the Driver

After writing a driver, you must load it for use with the *DigiLinX* network. This chapter provides step by step procedures for installing and running an example driver for a Panamax® power conditioner. If you have any questions concerning *DigiLinX* Dealer Setup, refer to the *DigiLinX* Dealer Setup Manual located on the Dealer Documents page of the *NetStreams* website.

WARNING! You can only load drivers with *DigiLinX* Dealer Setup versions 1.5 or higher.

1. Add a *ControLinX* configured for General Purpose Driver, Serial (see Figure 6-1).

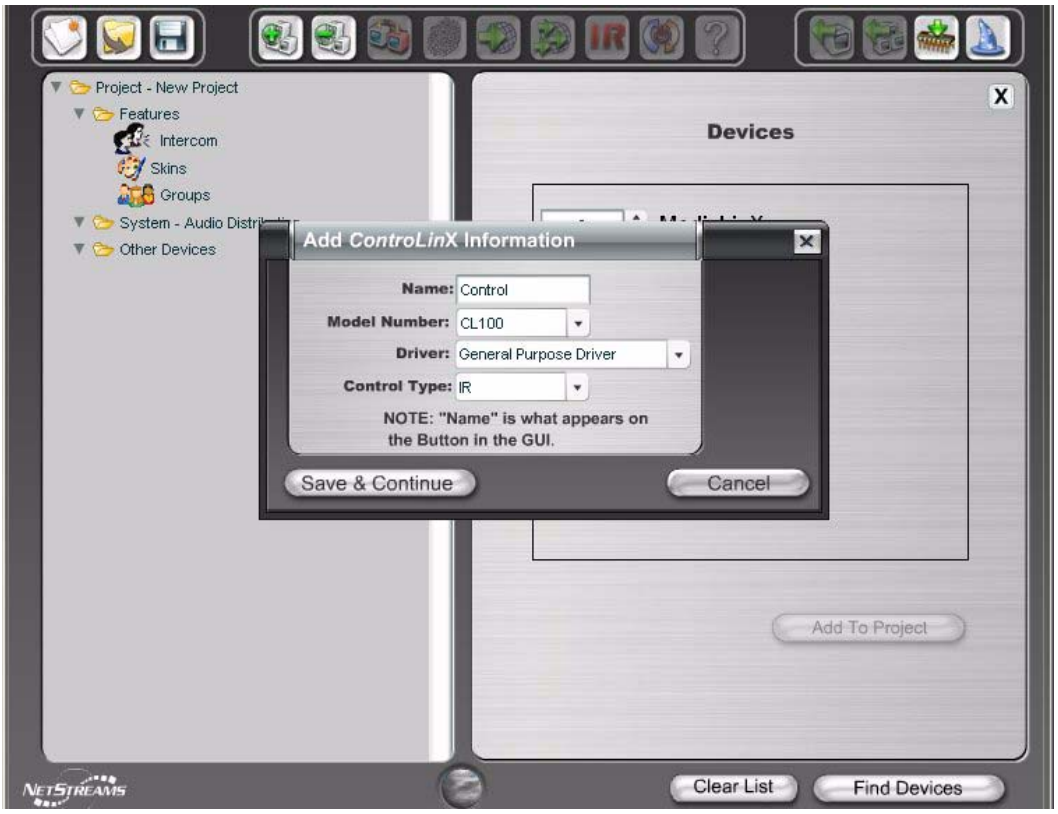

Figure 6-1 Add a *ControLinX*

2. Using Windows Explorer, copy the driver you have written to C:\Program Files*DigiLinX* Dealer Setup\upgrades\<latest folder date>\drivers.

NOTE: In this case you are copying to \07_17_2006\drivers.

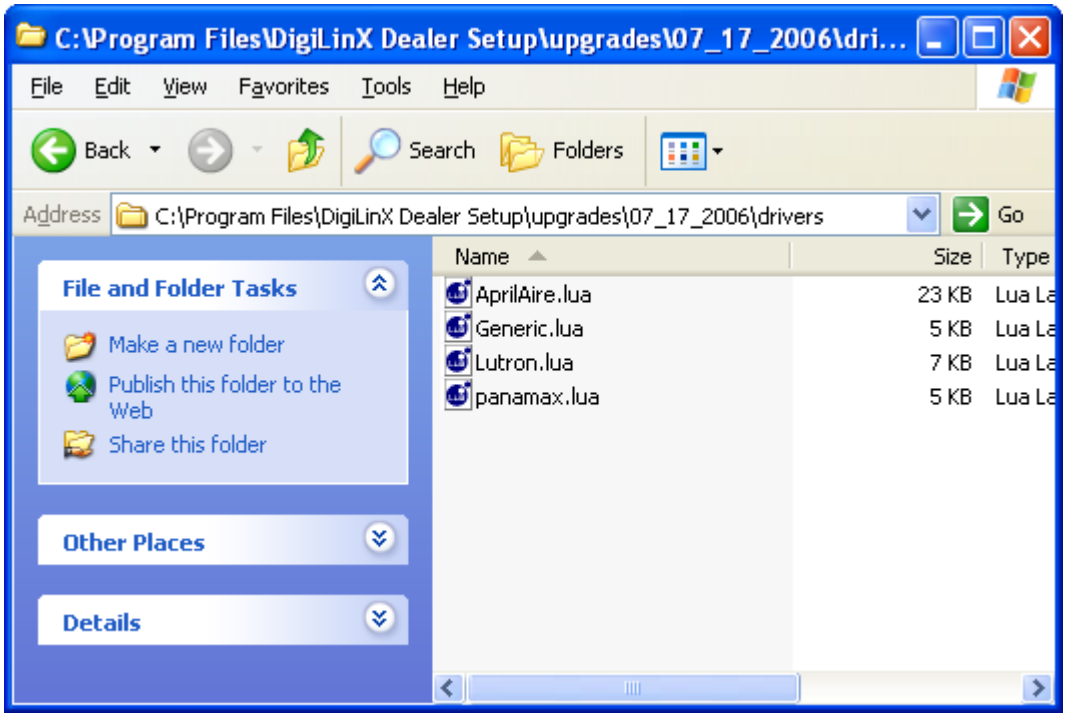

Figure 6-2 Copying the driver to the hard drive

3. From *DigiLinX* Dealer Setup, highlight the *ControLinX* and select the IR/RS-232 tab.
4. Enter the filename of your driver file in the driver file text box of your *ControLinX* configuration:

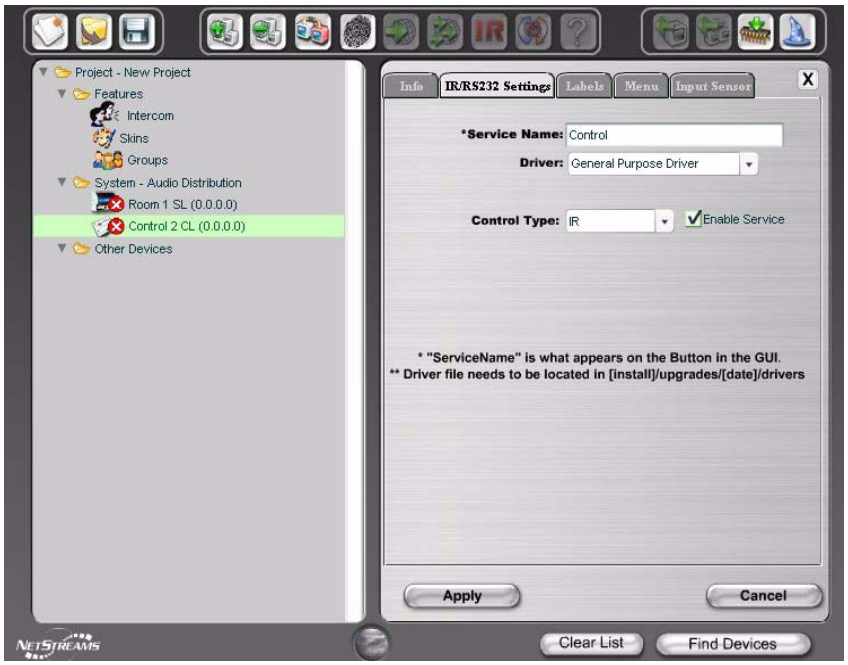

Figure 6-3 Entering the driver file name

5. Select **Labels** and enter button presets.

NOTE: This procedure creates the individual buttons that will control this device. The ID field corresponds to the driver subNode, while the Button Label is the first label that appears on the GUI buttons.

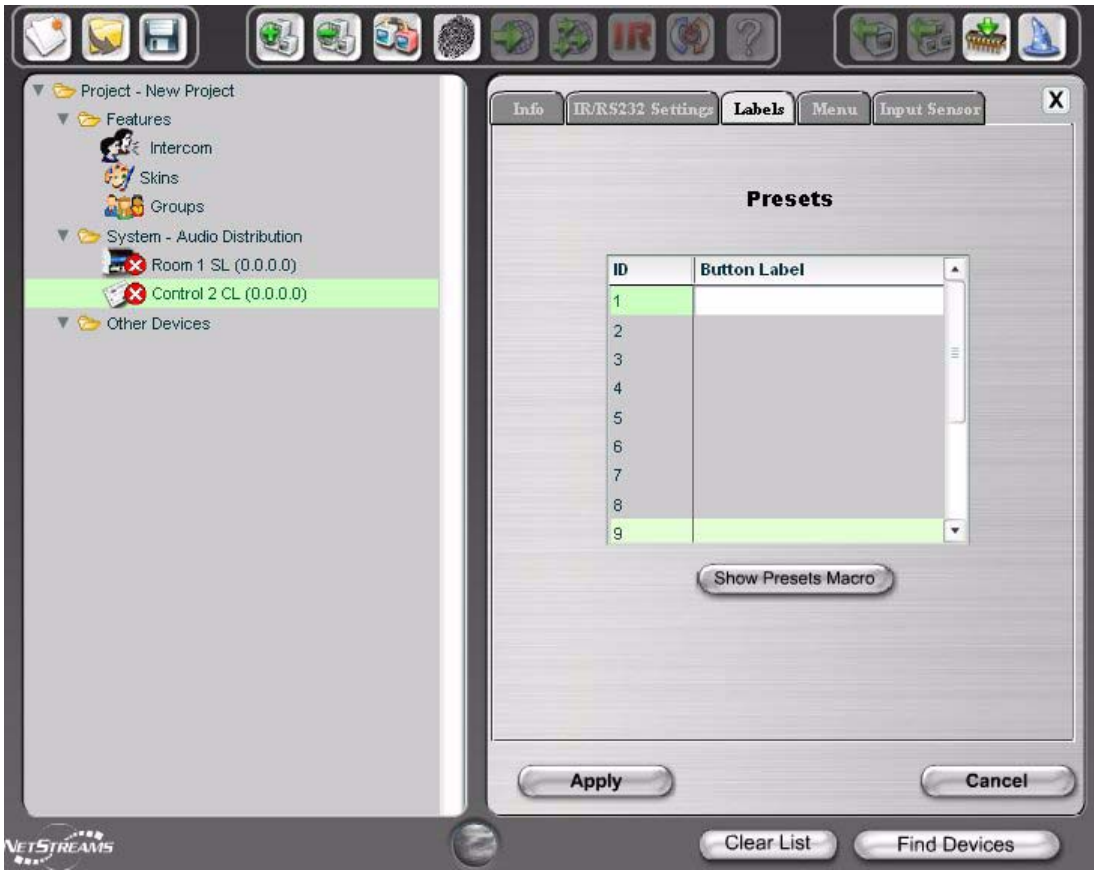

Figure 6-4 Entering Button Presets

6. Select the Menu tab and select **Enable a Menu for this Room**.

NOTE: Ensure that rooms requiring control of this device have this menu enabled.

Figure 6-5 Enable menu for rooms

7. Select the  button to send the configuration to all devices (see Figure 6-6).

NOTE: Wait for the *ControLinX* to restart.

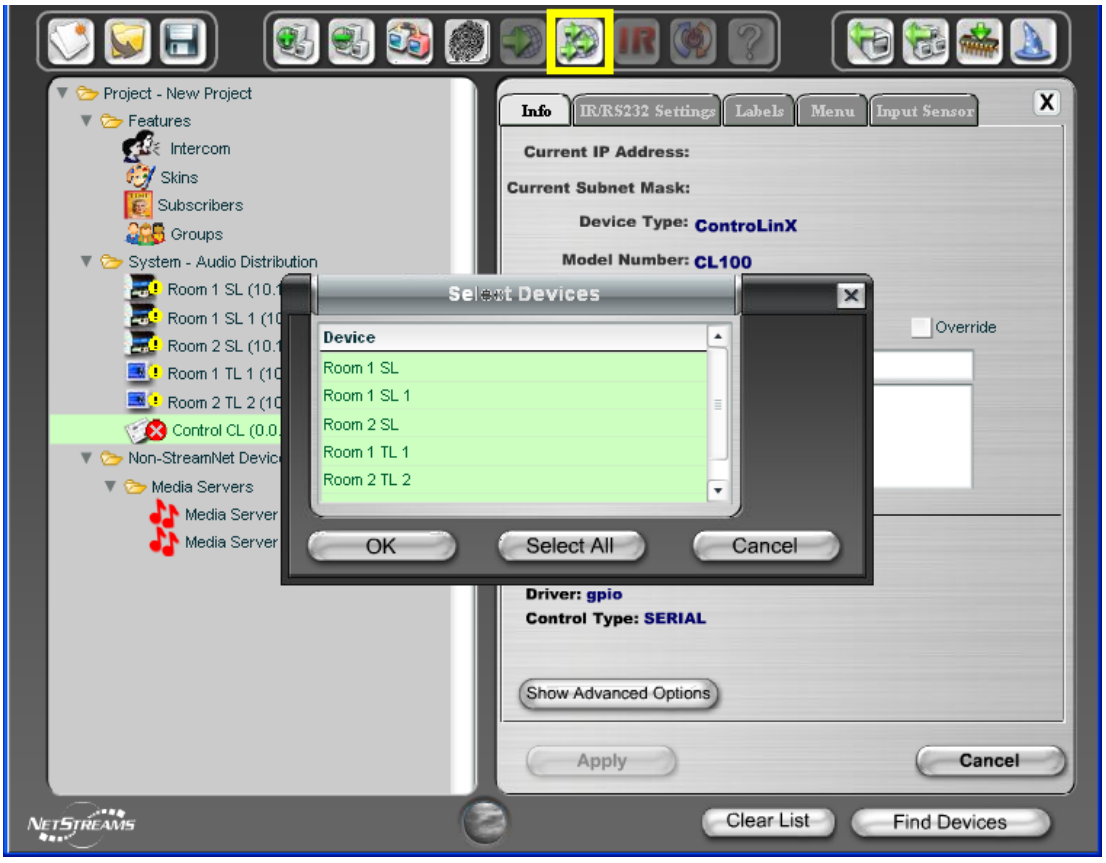

Figure 6-6 Sending configuration to all devices

The Control icon (see Figure 6-7) and buttons (see Figure 6-7) display.

Figure 6-7 Control icon

Figure 6-8 Buttons

Useful Development Tips

- Always compile your script (using your editor's compile option, or luac.exe) before transferring the driver to the *ControLin*X. While this only checks syntax errors, it will save time when they appear.
- Use the categorized debug logging to filter your messages, and turning on only the level you need.
- If you have communication problems, even if they are only in one direction, verify that you are using the right COM port parameters, and that you don't need a null modem adapter.
- You can add the attribute `autoStart` to the `SCRIPT` block in config_current.xml and set it equal to 0 to keep the script from loading automatically.
- You can send the ASCII commands `#script start` and `#script stop` to the service for your driver to start and stop the service. This is useful for starting the script with `autoStart=0`. You can also stop your script, copy a new version, and then start it again to run the new version without restarting the device. Something to keep in mind when using this method is that global variables are kept between stop and start. Although this can be useful (your current debug filter settings are kept for example) you need to take care when performing operations that modify the global state.

Chapter

7

Generating Status Reports

In almost all cases, the generation of the status reports used by the User Interface is automated. A status report takes the form of a #report ASCII command:

```
#report {{<report type="state" field=value…}}
```

To update the generated status report is to update the responsible status object with the appropriate field/value pairs. The status report is then generated and transmitted.

As examples, if the base device needs to generate status reports with a time stamp field, include the code:

```
setStatus("timeStamp", os.uptime())
```

Care is automatically taken to insure that excessive status reports are not generated, rather they are specifically only generated as the status itself is modified.

As another example, if a subNode needs to report a status containing a temperature field, include the code:

```
zone:setStatus("temperature",
zone.currentTemperature)
```

NOTE: When a subNode is modifying its status variables it should use the ":" notation to insure that the zone is passed to the setStatus routine.

Chapter

8

Diagnostic Messages

Diagnostic messages can be output to the debug viewer using the `debug` function. The `debug` function operates similarly to `print` when only one parameter is passed. When two or more parameters are passed, the first argument is converted to a string and use to a filter the display of that message, based on the current filter settings. Filter settings can be changes using the `setDebug` function or the `#debug` ASCII command. Filtering for all categories defaults to off.

Chapter

9

Reference

This section defines the *StreamNet* extensions to the standard Lua function calls defined in the Lua manual, which can be found at http://www.lua.org/manual/5.1/.

Syntax

In the descriptions below, the following notations are used:

- aType = indicates that the function returns an item of the indicate type, or that a variable is of the indicated type
- aType: indicates the indicated function should be invoked or defined with the ":" operator as the "self" argument of type "aType" must always be given.
- aType. indicates the standard lua table lookup rules apply

Control

For purposes of simplicity, all the basic control functions are directly defined in the outer scope of a Lua driver. In most cases, the only required entries here are the handle_command entries necessary to handle the commands important to the driver itself.

default_handle_command (command)

default_handle_command is the last effort command handler and is used only if there is no handle_*command* function defined to specifically handle a given command.

Strictly speaking, this function need never be defined or used since any command will first be handled by the type specific handler.

NOTE

It is always possible to define an appropriate command-specific handler, so that the default_command_handler is not needed. Programming best-practices state that a fallback default_command_handler should be defined even if it is not intended for use.

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1. The driver can define a handle_command method to handle the incoming ASCII command “#command …”. The driver can define a handle_command method to handle the incoming ASCII command “#command …”command method to handle the incoming ASCII command “#command …”

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1.

For more information on command handling see Chapter 2, *Handling Commands* on page 3-1.

handle_command(command)

The driver can define a handle_*command* method to handle the incoming ASCII command “*#command …*”

The single argument is an AsciiCommand table as documented in The driver can define a handle_command method to handle the incoming ASCII command “#command …”. The driver can define a handle_command method to handle the incoming ASCII command “#command …”_command method to handle the incoming ASCII command “#command …”

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1.

aNode = getSubNode(strNodeName)

Returns the node associated with strNodeName by fetching subNodes[strNodeName].

Device drivers may replace this routine with one of their own if they wish to create nodes on the fly as they are addressed by commands. Such an implementation might look like:

```
function getSubNode(strNodeName)
if(subNodes[strNodeName] == nil) then
      node = createSubNode(strNodeName)
      … do some more initialization here
else
      node = subNodes[strNodeName]
end
return node
end
```

setStatus(strField, value)

This extension updates the value of a particular field in the associated status object. This is shorthand for status.setField(field, value). A status field can be deleted by setting it's value to nil.

aString = getStatus(strField, value)

Returns the value for the field specified, or nil if the field has not yet been defined.

SubNode

aSubNode = createSubNode(strName)

createSubNode is invoked to create a new named subNode. The newly created subNode is automatically registered with the control by placing it in the subNodes table using strName as an index. The newly created subNode will have an empty status object associated with it and only minimal commands will be handled.

aTable = subNodes

subNodes is a table containing each of the subNodes created with createSubNode, indexed by subNode name.

aString = aSubNode.name

Holds the name of the subNode as set in createSubNode. Once created the name of the subNode should not be altered.

aStatus = aSubNode.status

This extension holds the status object for the subNode. Under normal circumstances the setStatus and getStatus methods should be used in lieu of directly accessing the status object.

aSubNode:setStatus(strField, value)

Updates or creates a status report field with the indicated value. If the value is actually being changed this will trigger the automatic generation of a status report when the execution of the current command, timer, or async input function is completed. A field that is no longer appropriate can be deleted by calling aSubNode:setStatus(strField, nil)

aString = aSubNode:getStatus(strField, value)

Fetch the most recently assigned value of a status report field. If the report field is currently undefined, nil will be returned.

aSubNode:default_handle_command(command)

default_handle_command is the last effort command handler and is used only if there is no handle_*command* function defined to specifically handle a given command. Strictly speaking, this function need never be defined or used since any command will first be handled by the type specific handler.

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1. The driver can define a handle_command method to handle the incoming ASCII command “#command …”command method to handle the incoming ASCII command “#command …”

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1.

aSubNode:handle_command(command)

The driver can define a handle_*command* method to handle the incoming ASCII command “*#command …*”

The single argument is an AsciiCommand table as documented in Chapter 2, *Handling Commands* on page 3-1.

AsciiCommand

An AsciiCommand object contains the information and parameters contained in an incoming ASCII command.

aString = command.command

The lower-cased string version of the command itself. For example, in the case of “#SET FAN,OFF” command.command would contain “set”.

aTable = command.params

A 1-based vector of the parameters specified to the command; e.g., in the case of “#SET FAN,OFF”, command.params would contain:

```
command.params = {
      [1] = “FAN”,
      [2] = “OFF”
}
```

Individual parameters can be referenced as command.params[n] where n is between 1 and the number of command specified.

aString = command.to

The entire to address as specified in the original command. Contains nil if no address was specified.

aString = command.toNode

The node portion only of the address specified in the original command. If the to address given was “@Aprilaire~zone1”, then toNode would contain “Aprilaire”. Contains nil if no address was specified.

aString = command.toSubNode

The subNode portion of the address specified in the original command. If the to address given was “@Aprilaire~zone1”, then toNode would contain “zone1”. Contains nil if no address was specified or if no subNode was specified in the original address.

aString = command.from

The entire from address as specified in the original command. Contains nil if no address was specified.

aString = command.fromNode

The node portion only of the from address specified in the original command. If the from address given was “:Aprilaire~zone1”, then fromNode would contain “Aprilaire”. Contains nil if no address was specified.

aString = command.fromSubNode

The subNode portion of the from address specified in the original command. If the from address given was “:Aprilaire~zone1”, then fromNode would contain “zone1”. Contains nil if no address was specified or if no subNode was specified in the original address.

Stream

Creating a stream allows a device driver to communicate with an external device by RS-232 or TCP/IP. More connection types may be defined in the future. Once the stream has been created, data can be read from or written to the stream with no regard

for the underlying transport mechanism. Facilities are also provided for handling of asynchronous streams with defined message boundaries with no need on the drivers' part for explicitly polling the stream.

aStream = createStream(strPort)

Creates a new stream from the string specification in strPort. The form of strPort is:

```
{protocol}://{address}[;{options}…]
```

Where protocol and address are taken from the table below:

Protocol	Address	Comments
comm	0 .. # of supported device ports	Opens an RS-232 serial connection
socket	{host}:{port}	Opens a TCP/IP connection to the indicated host and port.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

In the case of the socket protocol, no options are currently supported.

In the case of the comm protocol, the following options may be specified:

Key	Value	Default
baud	300,1200,2400,9600,19200	9600
parity	odd,even,none	None
bits	8	8
stop	1	1

aString = stream:read(max)

Reads data from the stream and returns it in the form of a string. If max is specified, no more than max characters will be returned. This call will never block, but rather will return whatever data is immediately available up to the specified maximum.

NOTE

Mixing synchronous input via read and asynchronous input via startAsyncInput will yield inconsistent and variable results. If mixing is required, async input should always be stopped via stopAsyncInput before read is used.

stream:write(aString)

Writes data from a string to the stream.

anInteger = stream:available()

Returns the number of bytes of data immediately available on the port.

stream:startAsyncInput(aFunction, options)

NOTE

Mixing synchronous input via read and asynchronous input via startAsyncInput will yield inconsistent and variable results. If mixing is required, async input should always be stopped via stopAsyncInput before read is used.

Enables asynchronous input on the associated stream. Has completed "messages" are received aFunction will be invoked with the stream and message as parameters. The options parameter contains an optional table that can be used to define a "message":

Key	Default	Notes
endString	Nil	If non-nil defines a string which will be used to recognize the end of a message. Common values might be "\r", "\n", or "\r\n" Completed messages will contain all the data between endString terminators and the terminator itself.
timeout	0	If > 0 specifies the maximum time to wait for any input. Every "timeout" ms any available data will be passed to the input function.
readIdle	0	If > 0 specifies the maximum time to wait while no data is being received. If the readIdle timeout elapses with no new data being received any available data will be passed to the input function.
maxRead	1024	The maximum # of characters to process before invoking the input function.
trailing	0	If > 0 specifies that additional data should be returned after a message has be recognized based on an endString. So specifying an endString of "\n" and a trailing of 2 would return the all text up to and including the newline and the following two characters.

The options parameter is in the form of a table, such as:

```
Stream:startAsyncInput(onInput, {
                                    endString = "\r",
                                    trailing = 2
                                })
```

NOTE

Asynchronous input is not truly asynchronous as the input function will never interrupt execution of an asynchronous stream handler, command handler, or timer handler.

stream:stopAsyncInput()

Stops asynchronous input previously started with startAsyncInput.

Status

aStatus = createStatus()

Creates and initializes a new status object.

Under normal circumstances this function is unneeded by the developer since the status objects for the control and subNodes are automatically created.

aStatus = status

Contains the status object for the control.

status:setField(field, value)

Updates a field in the status report and triggers status report generation “soon” Assigning a field the value of nil will remove it from the report.

aString = aStatus:getField(field)

Returns the value of a field in the status report.

Timer

Timers are used to schedule a function to be called at some predetermined time in the future.

NOTE

Due to the scheduling inconsistencies in a multi-tasking environment, the only guarantee that can be made about when the timer will be executed is that it will be executed no sooner than the time specified in the createTimer call. No real guarantees can be placed on the maximum time that will elapse before the timer is executed.

NOTE

Timers are not truly asynchronous events as they will never interrupt execution of an asynchronous stream handler, command handler, or another timer handler.

aTimer = createTimer(nMS, aFunction)

Creates a timer object and schedules it for execution in nMS milliseconds. When the timer executes it will invoke aFunction with the timer itself as it's sole parameter. If the timer function returns a non-zero value, the return value will be used to reschedule the timer at another future time.

anInteger = aTimer.duration

The duration last specified for the timer, either in createTimer, queue, or as a return value from timer execution.

aTimer:cancel()

Cancels the timer and prevents any future execution until queue is called.

aTimer:queue(nMS)

Queues the timer object to be executed in nMS milliseconds subject to the vagaries of operating system scheduling. If the timer is already queued for execution, the timeout will be rescheduled.

Debug

debug(category, …)

NOTE

The category is coerced to a lower case string before any processing is done. The special category “all” controls all categories. The default filtering for all messages is off.

Displays in the Debug Viewer the remaining (…) parameters if debug filtering is on for `category`. If there is only one parameter, it is always displayed in the debug viewer. Parameters are displayed concatenated with a tab character.

setDebug(category, <”on”|”off”|”toggle”>)

Enables, disables or toggles display of messages for the given `category`. If the given category is “all” then the operation will apply to all messages. The equivalent ASCII command is:

```
#@Service#debug category <on|off|toggle>
```

Appendix

A

Lua 5.0 License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Appendix

B

Editing Environments

There are a number of editors with varying degrees of support for editing Lua. Some include:

Name	URL	Notes
Gnu emacs	http://www.gnu.org/software/emacs/emacs.html http://luaforge.net/projects/lua-mode/	For UNIX
Lua Edit	http://www.luaedit.net/	Designed for Lua
UltraEdit	http://www.ultraedit.com/ http://lua-users.org/files/wiki_insecure/editors/UltraEdit_wordfile_Lua5.txt	Very powerful and commercial developer's editor.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Appendix

C

Control Driver Example

This example shows how to control a Panamax MAX 5410 Pro AC power source via its RS-232 interface. The driver will create six control subNodes which will represent six buttons, each having a distinct function. These buttons can then be displayed on the *DigiLin*X GUI. The subNodes will represent the following functions:

subNode Function Description

1	Power On	Initiates power-on sequence
2	Power Off	Initiates power-off sequence
3	Switched	Cycles power to the switched outlets
4	High Current	Cycles power to the high current outlets
5	All Off	Turns off all outlets
6	Cycle	Initiates a power-cycle sequence

The example uses a user created function, createButton, to encapsulate the repetitive tasks for creating each of the button subNodes. Each subNode will have four variables, Name, the name of the subNode used for addressing, xmit, the RS-232 string to transmit when the button is pressed, Label, the text that will appear on the GUI button face, and indicatorState, which will determine whether or not the indicator LED on the GUI button will be on or off. Each subNode will also contain one function, handle_button, which is called whenever the corresponding GUI button is pressed.

Because of the way Lua parses the source files, the functions that the example uses must be defined before they are actually used, so that is why the functions will appear first.

The example also uses two timer driven functions, pollResponse and queryStatus. The first function, pollResponse, reads data from the serial port every second looking for messages from the device. Depending on what the response is, the driver may turn the indicatorState variable on or off (1 or 0 respectively). The second function queryStatus will send a query to the device every 10 seconds to ensure that the driver stays in sync with the device. These functions are created by calling the createTimer function.

```
--
    -- The service itself is set up with a subNode for
       each of the switchable
    -- power supplies (switchable outlets and high power
       outlets) and a subNode
    -- for all power.  Each zone responds to a #BUTTON
       PRESS by sending out
    -- the requested command and then issuing an
       ?OUTLETSTAT to find out which
    -- outlets are on.

    -- To allow the GUI to display an appropriate label
       on the button, each
    -- subNode also reports status in the common form:
    -- <report type="indicatorState" label="<label
       text>" indicatorState="<0/1>" />
    --


    --


    -- We have to create our functions first then we can
       use them below

    -- function createButton( Name, Label, xmitString )
    --
    -- create a button node
    --

    function createButton( Name, Label, xmitString )

    -- create a subNode for each button
    local node = createSubNode(Name)


    --
    -- the serial to xmit string that dealer setup writes
       for us
    --   has escape sequences so that special characters
       like newlines
    --   and nulls can be embedded in the string to
       send.  Each escape
```

```
-- consists of %XX where XX is two hexadecimal
   characters.  We
-- have to replace each escape with it's
   corresponding single
-- character
--
if(xmitString) then
      node.xmit = decodeString(xmitString)
end

-- set the newly created button up to handle the
   #BUTTON command
function node:handle_button(cmd)
      --
      -- we just handle #BUTTON PRESS by sending our
   string
      --
      if(cmd.params[1]:upper() == "PRESS" and
   self.xmit ~= nil) then

            g_serial:write(self.xmit)

            -- Only change to the button set up
   routine is to call
            --    queryStatus after sending the text
   to find out what
            --    state the panamax is currently in
            pollResponse()

      end
end

-- set the label field in the status report
node:setStatus("label", Label)

-- assume an initial "off" state
node:setStatus("indicatorState", "0")
end

-- function pollResponse()
-- Function that checks to see if there is anything
   on the COM port
--
function pollResponse()
-- read and parse the response
```

```
local response = g_serial:read()

if(response ~= nil) then
      if(response:find("MAINACON")) then

   getSubNode("1"):setStatus("indicatorState", "1")

   getSubNode("2"):setStatus("indicatorState", "0")

      elseif(response:find("MAINACOFF")) then

   getSubNode("1"):setStatus("indicatorState", "0")

   getSubNode("2"):setStatus("indicatorState", "1")

      end

      if(response:find("SWITCHEDON")) then

   getSubNode("3"):setStatus("indicatorState", "1")

      elseif(response:find("SWITCHEDOFF")) then

   getSubNode("3"):setStatus("indicatorState", "0")

      end

      if(response:find("HICURRENTON")) then

   getSubNode("4"):setStatus("indicatorState", "1")

      elseif(response:find("HICURRENTOFF")) then

   getSubNode("4"):setStatus("indicatorState", "0")

      end
end

-- check for responses again in a second
```

```
return 200
end

-- function queryStatus()
--
-- query the Panamax to see if anything has changed
--

function queryStatus()
   -- request the status so we know the initial state
      of everything
g_serial:write("?OUTLETSTAT\r");

-- try again in a second if this is from a timer
return 10000
end

-- Now run the main code

-- Check to see if the COM port has been overridden
   by Dealer Setup

if( config == nil ) then
config = {}
end

-- If the COM port doesn't exist, setup the default

if( config.port == nil ) then
config.port = "comm://0;baud=9600;parity=none"
end


--
-- open the serial port and ready it for writing
--
g_serial = createStream(config.port)
if(g_serial == nil) then
print("Unable to open stream \""..config.port.."\"")
return
end
-- Create buttons

createButton("1", "Power On","POWERON\r")
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

```
createButton("2", "Power Off","POWEROFF\r")
createButton("3", "Switched","CYCLESW\r")
createButton("4", "High Current","CYCLEHC\r")
createButton("5", "All Off","ALLOFF\r")
createButton("6", "Cycle","CYCLE\r")

-- create a timer to call queryStatus every 10
   seconds

createTimer(10000, queryStatus)

-- every one second call pollResponse to check for
   asynchronous responses
-- (stuff it sends without being asked)

createTimer(1000, pollResponse)
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Appendix

D

Audio Driver Example

The following example audio and control drivers in commented code-form are available as further resources.

All drivers can be found in your *DigiLinX* Dealer Setup folder under the ./upgrades/[MM-DD-YYYY]/drivers/ folder. Open the .lua file with your preferred text editor to see the complete commented code created by *NetStreams* engineers.

Parasound zTuner

```
    -- zTuner.lua
    -- driver for Parasound Ztuner V1 and V2
    -- file version 1.0.0
    -- There are many places where we check to see if we are a V1 or V2 tuner since their protocols
    -- are different.  The V2 protocol is more straight forward than the V1 as will be evident in
    -- how much extra work is required to setup the outgoing tune commands and with the extra work
    -- required to know what piece of information we are currently expecting back from the tuner
    -- (this is the FREQUENCY_FIRST, LAST, BAND, etc stuff)
    --
    -- DigiLinX command handler:
    --
    -- function handle_band(command)
    -- function handle_clear(command)
    -- function handle_key(command)
    -- function handle_next(command)
    -- function handle_preset(command)
    -- function handle_prev(command)
    -- function handle_scan(command)
    -- function handle_seek(command)
    -- function handle_tune(command)
    --
    -- Other functions:
    --
    -- function CheckPower()
    --  Send command to check power state of tuner
    -- function OnAsyncInput(stream, message)
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

```
    --  Get and Process Input from Serial Port
    -- function PerformReset()
    --  Perform a soft reset of the zTuner when we seem to have lost
communications
    -- function QueryStatus()
    --  Periodically query for the frequency of the tuner. This is more
important with V1 tuners that have no unsolicited messages
    -- function RefreshFrequency()
    --  Send command to get the frequency from tuner
    -- function RequestV2Status()
    --  Send command to refresh the V2 tuner's status info
    -- function SendPower(bOn)
    --  Send command to power the tuner on (or off)
    -- function WriteMessage(message)
    --  Decode and send commands to the tuner



    -- uncomment the next line for any hope of debugging startup issues
    -- setDebug("all", "on")
    setDebug("error", "on")

    debug("Driver loading")

    -- setup our constants
    IDLE = 0
    FREQUENCY_FIRST = 1
    FREQUENCY_LAST = 2
    BAND = 3
    POWER_STATE = 4

    FM = 0
    AM = 1
    BANDNAME = {
    [FM] = "FM ",
    [AM] = "AM "
           }

    -- setup our module variables
    serialPort = nil
    bIsV2 = false;
    szFreq = "---.-"
    nResponseTimeout = 0
    nExpectedRX = IDLE
    nRefreshPowerTimeout = 5
    nRefreshFreqTimeout = 0
    nTimeouts = 0
    nBand = FM
    szInput = ""
    szPwrOn = "0"

    -- We have to create our functions first then we can use them below
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

```
-- function OnAsyncInput(stream, message)
-- Diego Alfarache
-- 09-20-2006
-- Get Input from Serial Port

function OnAsyncInput(stream, message)
debug("verbose", message)
local start = 0

if( message == nil ) then
    return
end

if( message:find("[*]")) then

    bIsV2 = true

    local Power = message:match("PW(.)")
    local Band = message:match("BD(.)")
    local Frequency = message:match("FR(......)")
    local Preset = message:match("PR(..)")

    if( Power ) then
            debug("verbose", "Power = "..Power)
            if( Power == "0" ) then
                    -- power is off
                    bPwrOn = "0"
                    SendPower( true )
            else
                    bPwrOn = "1"
            end
    end

    if( Band ) then
            debug("verbose", "Band = "..Band)
            if( Band == "1" ) then
                    nBand = FM
            elseif( Band == "2" ) then
                    nBand = AM
            end
    end

    if( Frequency ) then
            -- skip past any leading zeros
            local start = Frequency:find("[123456789]")
            Frequency = Frequency:sub(start,-1)

            if( nBand == AM ) then

                    szFreq = "AM "..Frequency
```

```
                elseif( nBand == FM ) then

                        szFreq = "FM "..Frequency
                else

                        szFreq = Frequency
                end

                setCaption(szFreq)
                debug("verbose", "Frequency is "..szFreq)

        end

    else

        -- This is all V1 Stuff

        if( nExpectedRX == FREQUENCY_FIRST ) then

                if( message:len() >= 2 ) then

                        message = message:match("%d+")

                        local nMessage = tonumber(message)

                        -- if the first part of the frequency is less than 20
then we assume that the band is AM
                        if( nMessage < 20 ) then

                                nBand = AM
                                szFreq = "AM "
                        else

                                nBand = FM
                                szFreq = "FM "
                        end

                        szFreq = szFreq..message

                end

                nExpectedRX = FREQUENCY_LAST

        elseif( nExpectedRX == FREQUENCY_LAST ) then

                if( message:len() > 2 ) then

                        message = message:match("%d%d")

                        if( nBand == AM ) then
```

```
                        szFreq = szFreq..message
                    else

                        szFreq = szFreq.."."..message
                    end

                    setCaption(szFreq)
                end

                nExpectedRX = IDLE
                nResponseTimeout = 0
                nTimeouts = 0

            elseif( nExpectedRX == POWER_STATE ) then

                if( message:find("[0]") ) then

                    SendPower( true )
                end

                nExpectedRX = IDLE
                nResponseTimeout = 0
                nTimeouts = 0
            else

                debug("verbose", "Unexpected Response "..message)
            end
        end

        UpdateSongReport()
        end

        --  function UpdateSongReport()
        --  Cristian Prundeanu
        --  10/23/2006
        --
        function UpdateSongReport()
        local songReport = songReportInstance()

        local strCaption = BANDNAME[nBand] or ""
        songReport:setField("band", strCaption)
        strCaption = strCaption..szFreq

        -- currently, the only used field in the zTuner GUI is "caption"
        songReport:setField("caption", strCaption)

        songReport:setField("frequency", szFreq)

        songReport:setField("pwrOn", szPwrOn)
        end
```

```
-- function WriteMessage(message)
-- Diego Alfarache
-- 09-20-2006
-- Decode string and write to the stream (serial port)

function WriteMessage(message)
debug("verbose", "Sending "..message)
-- call decodeString() to convert %XX to actual ASCII values
xmitMessage = decodeString(message)

serialPort:write(xmitMessage)

end


-- function handle_tune(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #TUNE commands

function handle_tune(command)

if( command == nil ) then
    return
end


local AsciiMessage = ""

local dir = command.params[1]:upper()

if( dir == "DN" ) then

    if( bIsV2 == true ) then
            AsciiMessage = "W 1 6 4%0D"
    else
            AsciiMessage = "W 1 7 4%0D"
    end

elseif( dir == "UP" ) then

    if( bIsV2 == true ) then
            AsciiMessage = "W 1 6 3%0D"
    else
            AsciiMessage = "W 1 7 3%0D"
    end

    else
        -- look to see if the frequency has a dot in it.  If it does, assume it
is FM
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

```
        local dot = command.params[1]:find("[.]")

        if( bIsV2 == true ) then
                AsciiMessage = "W 1 7 "
        else
                AsciiMessage = "W 1 6 "
        end

        if( dot ~= nil) then
                -- FM --
                AsciiMessage = AsciiMessage..command.params[1]:sub(1,dot-1)
                if( bIsV2 ) then
                        AsciiMessage = AsciiMessage.."."
                else
                        AsciiMessage = AsciiMessage.." "
                end

                AsciiMessage = AsciiMessage..command.params[1]:sub(dot+1,-1)

                -- Check to see if we need a trailing zero
                if( command.params[1]:len() < dot + 2 ) then
                        AsciiMessage = AsciiMessage.."0"
                end

                AsciiMessage = AsciiMessage.."%0D"
        else
                -- AM --
                if( bIsV2 ) then
                        AsciiMessage = AsciiMessage..command.params[1]
                else
                        if( command.params[1]:len() < 4 ) then

                                AsciiMessage                                    =
AsciiMessage..command.params[1]:sub( 1, 1 )..." "..command.params[1]:sub( 2, 4 )
                        else

                                AsciiMessage                                    =
AsciiMessage..command.params[1]:sub( 1, 2 )..." "..command.params[1]:sub( 3, 5 )
                        end
                end

                AsciiMessage = AsciiMessage.."%0D"

        end

    end

    WriteMessage(AsciiMessage)

    -- if we are a V1 we need to update the frequency
    if( false == bIsV2 ) then
```

```
    nRefreshFreqTimeout = 0
    RefreshFrequency()
end
end

-- function handle_seek(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #SEEK commands

function handle_seek(command)

if( command == nil ) then
    return
end

local AsciiMessage = ""
local dir = command.params[1]:upper()

if( dir == "DN" ) then

    if( bIsV2 == true ) then
            AsciiMessage = "W 1 6 2%0D"
    else
            AsciiMessage = "W 1 7 2%0D"
    end

elseif( dir == "UP" ) then

    if( bIsV2 == true ) then
            AsciiMessage = "W 1 6 1%0D"
    else
            AsciiMessage = "W 1 7 1%0D"
    end

end

WriteMessage(AsciiMessage)
-- if we are a V1 we need to update the frequency
if( false == bIsV2 ) then

    nRefreshFreqTimeout = 0
    RefreshFrequency()
end
end

-- function handle_scan(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #SCAN commands
```

```
    -- since the Parasound Ztuner doesn't have a scan feature we will just call
our seek function

    function handle_scan(command)
    handle_seek(command)
    end

    -- function handle_preset(command)
    -- Diego Alfarache
    -- 09-20-2006
    -- Receive and process #PRESET commands

    function handle_preset(command)

    if( command == nil ) then
        return
    end

    local AsciiMessage = ""
    local dir = command.params[1]:upper()

    if( dir == "DN" ) then

        AsciiMessage = "W 1 3 4%0D"

    elseif( dir == "UP" ) then

        AsciiMessage = "W 1 3 3%0D"

    end

    WriteMessage(AsciiMessage)
    -- if we are a V1 we need to update the frequency
    if( false == bIsV2 ) then

        nRefreshFreqTimeout = 0
        RefreshFrequency()
    end
    end

    -- function handle_next(command)
    -- Diego Alfarache
    -- 09-20-2006
    -- Receive and process #NEXT commands

    function handle_next(command)

    if( command == nil ) then
        return
    end
```

```
local AsciiMessage = "W 1 3 3%0D"

WriteMessage(AsciiMessage)
-- if we are a V1 we need to update the frequency
if( false == bIsV2 ) then

    nRefreshFreqTimeout = 0
    RefreshFrequency()
end
end

-- function handle_prev(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #PREV commands

function handle_prev(command)

if( command == nil ) then
    return
end

local AsciiMessage = "W 1 3 4%0D"

WriteMessage(AsciiMessage)
-- if we are a V1 we need to update the frequency
if( false == bIsV2 ) then

    nRefreshFreqTimeout = 0
    RefreshFrequency()
end
end

-- function handle_band(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #BAND commands

function handle_band(command)

if( command == nil ) then
    return
end

local AsciiMessage = ""

if( command.params[1] == "a" or command.params[1] == "A" ) then

    AsciiMessage = "W 1 8 2%0D";

elseif( command.params[1] == "f" or command.params[1] == "F" ) then
```

```
    AsciiMessage = "W 1 8 1%0D";

else

    AsciiMessage = "W 1 8 10%0D";

end

WriteMessage(AsciiMessage)
-- if we are a V1 we need to update the frequency
if( false == bIsV2 ) then

    nRefreshFreqTimeout = 0
    RefreshFrequency()
end
end

-- function handle_key(command)
-- Diego Alfarache
-- 09-20-2006
-- Receive and process #KEY commands

function handle_key(command)

if( command == nil or command.params[1] == nil) then
    return
end

local keyvalue = command.params[1]:upper();

if( szInput:len() > 6 ) then

    szInput = szInput:sub(2,-1)

end

szInput = szInput..keyvalue

end

-- function handle_key(command)
-- Diego Alfarache
-- 09-22-2006
-- Receive and process #CLEAR commands

function handle_clear( command )

szInput = ""

end
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

```
    -- function handle_enter(command)
    -- Diego Alfarache
    -- 09-20-2006
    -- Receive and process #ENTER commands

    function handle_enter(command)

    if( command == nil ) then
        return
    end

    debug("verbose", "szInput is "..szInput)

    local AsciiMessage = ""

    if( bIsV2 ) then

        AsciiMessage = "W 1 7 "
    else

        AsciiMessage = "W 1 6 "
    end

    if( nBand == AM ) then

        if( bIsV2 ) then

                AsciiMessage = AsciiMessage..szInput
        else
                if( szInput:len() < 4 ) then

                        AsciiMessage  =  AsciiMessage..szInput:sub(  1,  1  ).."
"..szInput:sub( 2, 3 )

                else

                        AsciiMessage  =  AsciiMessage..szInput:sub(  1,  2  ).."
"..szInput:sub( 3, 4 )
                end
        end
    else
        if( szInput:sub(1,1) == "1" ) then

                if( bIsV2 ) then

                        AsciiMessage   =   AsciiMessage..szInput:sub(   1,   3
).."."..szInput:sub( 4, 6 )
                else
                        AsciiMessage  =  AsciiMessage..szInput:sub(  1,  3  ).."
"..szInput:sub( 4, 6 )
                end
```

```
                -- make sure we have enough trailing zeros
                if( AsciiMessage:len() < 12 ) then
                        AsciiMessage = AsciiMessage.."0"
                end

        else
                if( bIsV2 ) then

                        AsciiMessage    =    AsciiMessage..szInput:sub(    1,    2
)..".."..szInput:sub( 3, 5 )
                else
                        AsciiMessage  =  AsciiMessage..szInput:sub(  1,  2  )..."
"..szInput:sub( 3, 5 )
                end

                -- make sure we have enough trailing zeros
                if( AsciiMessage:len() < 11 ) then
                        AsciiMessage = AsciiMessage.."0"
                end

        end
    end

    -- clear out szInput
    szInput = ""

    AsciiMessage = AsciiMessage.."%0D"

    WriteMessage(AsciiMessage)
    -- if we are a V1 we need to update the frequency
    if( false == bIsV2 ) then

        nRefreshFreqTimeout = 0
        RefreshFrequency()
    end

    end

    -- function handle_menu_list(command)
    -- Cristian Prundeanu
    -- 10-23-2006
    -- handler for #MENU_LIST indexStart, indexEnd, path, searchParams
    -- path format: "PRESET"|"MEDIA"|"MEDIA2" [ ">" <submenu1> [ ">" <submenu2>
[...] ] ]; ex: "MEDIA>ALLDISCS>Disc_10>Title_5>Chapter_2"

    function handle_menu_list(command)
    debug("verbose", "handle_menu_list: command=",command)
    debug("verbose", '#MENU_LIST '..
          (command.params[1] and ('"'..command.params[1]..'"') or 'nil')..', '..
          (command.params[2] and ('"'..command.params[2]..'"') or 'nil')..', '..
          (command.params[3] and ('"'..command.params[3]..'"') or 'nil')..', '..
```

```
            (command.params[4] and ('"'..command.params[4]..'"') or 'nil')
          )
    if (#command.params > 4) then
        debug("warning",    "Extended    #MENU_LIST    params    received:    ",
command.params)
    end

    local nStart = tonumber(command.params[1])
    local nEnd = tonumber(command.params[2])

    if (command.vPath[1]:upper() == "PRESETS") then
        -- default preset handling is sufficient
        defaultHandleMenuListPresets(command, nStart, nEnd)
    elseif (command.vPath[1]:upper() == "MEDIA" or
                command.vPath[1]:upper() == "MEDIA2") then
        debug("error", "Media menus not supported in zTuner")
        command:sendFinalMenuResp()
    else
        debug("error", "Unknown root node in #MENU_LIST: ",command.params[3] )
    end
    end

    -- function handle_menu_sel(command)
    -- Cristian Prundeanu
    -- 10-24-2006
    -- handler for #MENU_SEL path
    -- path format: see handle_menu_list
    handle_menu_sel = function(command)
    debug("verbose", '#MENU_SEL '..
          (command.params[1] and ('"'..command.params[1]..'"') or 'nil')
         )
    if (#command.params > 1) then
        debug("warning", "Extended #MENU_SEL params received: ", command.params)
    end

    if(command.vPath[1]:upper() == "PRESETS") then
        -- handle "#MENU_SEL {{presets>...}}" commands, where the
        -- presets are stored in the table passed by the dealer setup
        command:handlePresetMenuSel(command.vPath[2])
    elseif (command.vPath[1]:upper() == "MEDIA" or
                command.vPath[1]:upper() == "MEDIA2") then
        debug("error", "Media menus not supported in zTuner")
    end
    end

    -- function handle_menu_set(command)
    -- Cristian Prundeanu
    -- 10-24-2006
    -- handler for #MENU_SET index
    handle_menu_set = function(command)
    debug("verbose", '#MENU_SET '..
```

```
            (command.params[1] and ('"'..command.params[1]..'"') or 'nil')
          )
    if (#command.params > 1) then
        debug("warning", "Extended #MENU_SET params received: ", command.params)
    end
    local strDisplay = (BANDNAME[nBand] or "")..szFreq
    command:handlePresetMenuSet(strDisplay, "#TUNE "..szFreq)

    -- permanently save presets
    presetSaveOverrides()
    end

    -- function handle_user(command)
    -- Cristian Prundeanu
    -- 10-18-2006
    -- handler for #USER functioncall [functioncall [...]] - Lua-execute one or
more functioncalls and print each result

    setDebug("USER", "on")
    function handle_user(command)
    local f
    for i = 1, #command.params do
        local strLine = string.format('debug("USER", "statement %d result: ",
%s) ', i, command.params[i])
        f = loadstring(strLine)

        if (not f) then
                f = loadstring(command.params[i])--  this  enables  execution  of
assignments and declarations
        else
                command.params[i] = strLine
        end
        if (not f) then
                debug("USER",    "error:    could    not    load    string:    ",
command.params[i])
        end
    end
    -- loading all statements as one chunk enables local variable/scope usage
    f     =     loadstring(string.format(string.rep('%s     ',#command.params),
unpack(command.params)))
    if (f) then
        f()
    end
    debug("USER", "execution finished")
    end

    --  function default_command_handler(command)
    --  Cristian Prundeanu
    --  10/23/2006
    --
    function default_command_handler(command)
    debug("error", "Unhandled command received: ", command)
```

```
end

--  function CheckPower()
--  Diego Alfarache
--  09/21/2006
--
function CheckPower()

-- Ignore commands if we are waiting on a response
if(true == bIsV2 or nResponseTimeout ~= 0) then

    return
end

WriteMessage("R 1 1%0D")

nResponseTimeout = 5
nExpectedRX = POWER_STATE
nRefreshPowerTimeout = 5
end

--  function PerformReset()
--  Diego Alfarache
--  09/21/2006
--
--  Note: This uses an undocumented command sequence to request a
--        hard reset of the zTuner.
function PerformReset()


WriteMessage("W 1 20 1%0D")

nTimeouts = 0
nResponseTimeout = 0
nRefreshPowerTimeout = 10
nRefreshFreqTimeout = 12

end

--  function CheckPower()
--  Diego Alfarache
--  09/21/2006
--
function RequestV2Status()

WriteMessage("R 1 13%0D")

nRefreshFreqTimeout = 5
end
```

```
--  function RefreshFrequency()
--  Diego Alfarache
--  09/21/2006
--
function RefreshFrequency()

-- Ignore commands if we are waiting on a response
if(nResponseTimeout ~= 0) then

    return
end

if( false == bIsV2 ) then
    WriteMessage("R 1 6%0D")

    nExpectedRX = FREQUENCY_FIRST;
    nResponseTimeout = 5
    nRefreshFreqTimeout = 5
else

    RequestV2Status()
end

end


--  function SendPower(bOn)
--  Diego Alfarache
--  09/21/2006
--
function SendPower( bOn )


debug("warning", "Setting Power state to %d", bOn)

local szPower = ""
local nPower = 0;

if( true == bOn ) then
    nPower = 1
end

szPower = string.format("W 1 1 %d%%0D", nPower)

WriteMessage(szPower)
nRefreshPowerTimeout = 5;
nRefreshFreqTimeout = 6;

end
```

```
-- function QueryStatus()
-- Diego Alfarache
-- 09-21-2006
-- Do a periodic check for status
function QueryStatus()
-- request the status so we know the initial state of everything
if( nResponseTimeout ~= 0 ) then

    nResponseTimeout = nResponseTimeout - 1

    if(nResponseTimeout <= 0) then

            nTimeouts = nTimeouts + 1

            -- errorLog(&SystemLog, "CControlZTuner: Timeout waiting ZTuner
response, %d, %d", m_nExpectedRX, m_nTimeouts);
            nRefreshPowerTimeout = 0
            if(nTimeouts >= 10) then

                    PerformReset()

            else

                    CheckPower()
                    RequestV2Status()
            end

    end

else

    if(nRefreshPowerTimeout > 0) then

            nRefreshPowerTimeout = nRefreshPowerTimeout - 1
    end

    if(nRefreshFreqTimeout > 0) then

            nRefreshFreqTimeout = nRefreshFreqTimeout - 1
    end

    if(nRefreshFreqTimeout <= 0) then

            RefreshFrequency()
    elseif(nRefreshPowerTimeout <= 0) then

            CheckPower()
    end
end
-- try again in 5 seconds if this is from a timer
return 5000
```

```
end


-- Now we can run our main code


-- Check to see if the COM port has been overridden by Dealer Setup
if( config == nil ) then
config = {}
end

-- If the COM port doesn't exist, setup the default
if( config.port == nil ) then
config.port = "comm://0;baud=9600;parity=none"
end


-- open the serial port and ready it for writing
serialPort = createStream(config.port)
if(serialPort == nil) then
debug("error", "Unable to open stream \""..config.port.."\"")
return
end

-- setup the stream to call OnAsyncInput whenever a carrige return is
detected
serialPort.startAsyncInput(serialPort, OnAsyncInput, {endString = "\r"})

-- load saved presets and add them to the ones defined in SCRIPT_DATA
presetLoadOverrides()

-- create our timer to call QueryStatus every 5 seconds
createTimer(5000, QueryStatus)
```


Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.

Main +1 512.977-9393 / fax +1 512.977.9398 / Toll Free Technical Support +1 866-353-3496
3600 W. Parmer Lane, Suite 100; Austin, TX 78727 / www.netstreams.com.