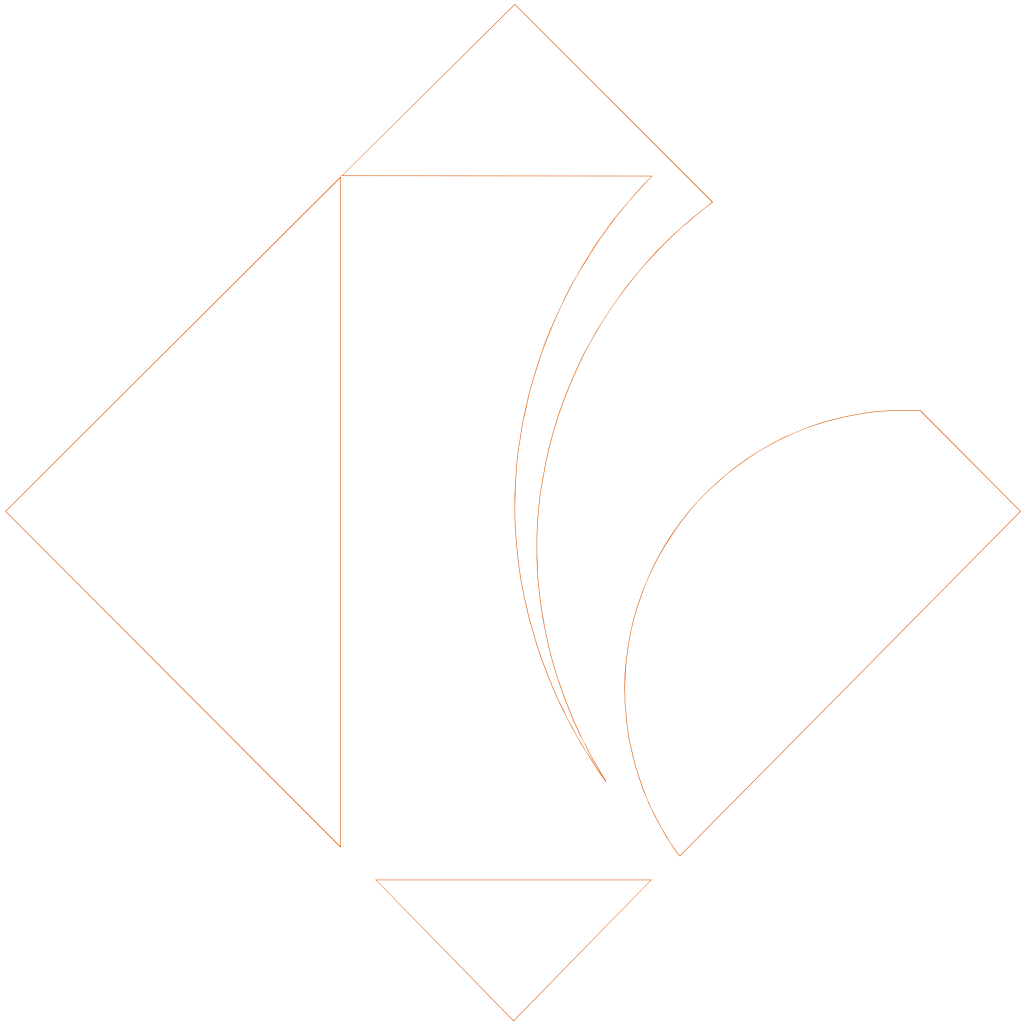


1. Table of Contents

2. M4223

2.1. M4223 Features

2.1.1. Ethernet port

Allows for remote connections to the module.

2.1.2. USB port

Allows for compatible USB devices to be attached to the module.

Only USB storage devices with an NTFS file system are compatible.

2.1.3. Embedded Linux

The module runs on embedded Linux operating system which provides a familiar interface for users. This system does **not** include a local compiler.

2.1.4. Web interface

The module includes a web interface that allows for firmware to be easily upgraded. This is covered in detail in 2.7 Web Interface.

2.1.5. Programmable indicators with Lua

The M4223 comes with Lua 5.1.5, a powerful lightweight scripting language, and supporting libraries that simplify the process of writing scripts to control the R400 and interface with the operator.

This feature vastly increases the capabilities of the R400 and allows it to be customised extensively to perform specific tasks.

This is covered in deal in 6. Rinstrum Lua Libraries.

2.1.6. Online and offline capabilities

The M4223 is designed for networking capability, but can also be used offline for purposes where user control is necessary, but it is not feasible to connect to a network.

2.1.7. Lua multiplexer

The M4223 uses a LUA multiplexer to allow for multiple connections (via a user application or View400) to a single R400 device, giving users the ability to set up multiple connections to local LUA scripts as well as remote control applications.

2.2. Logical Architecture

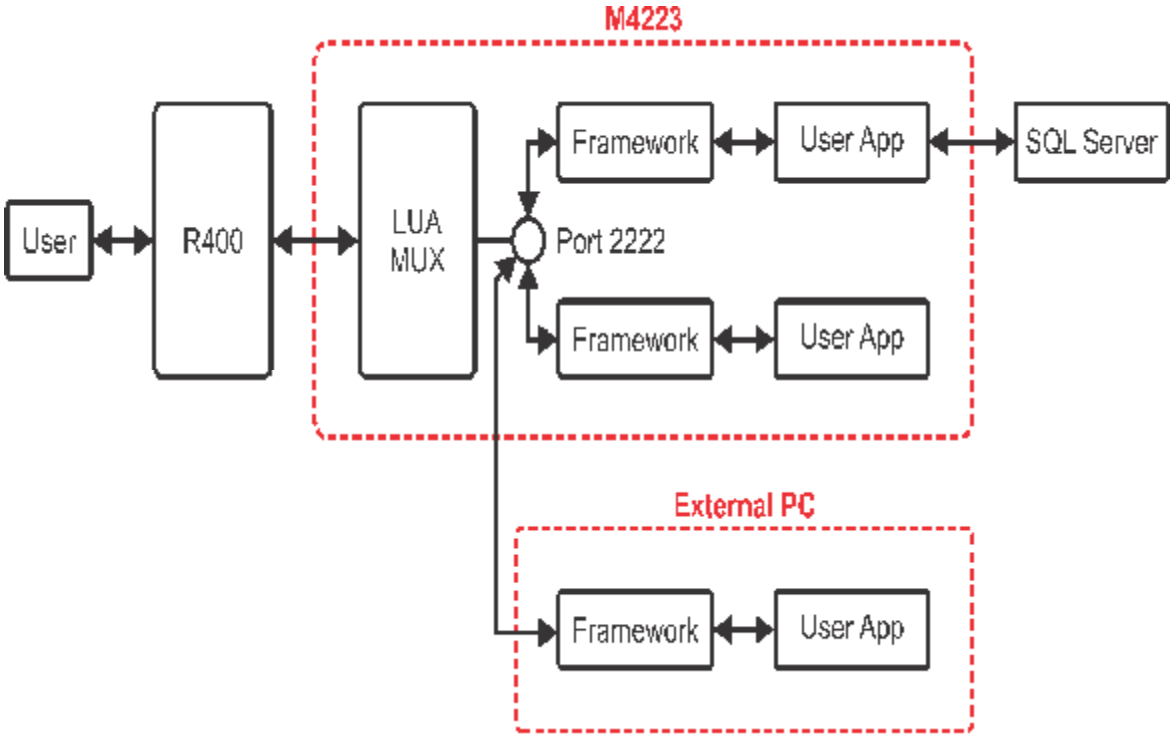


Figure 1: Logical Architecture with Framework (rinLib and rinSystem)

2.3. Physical Architecture

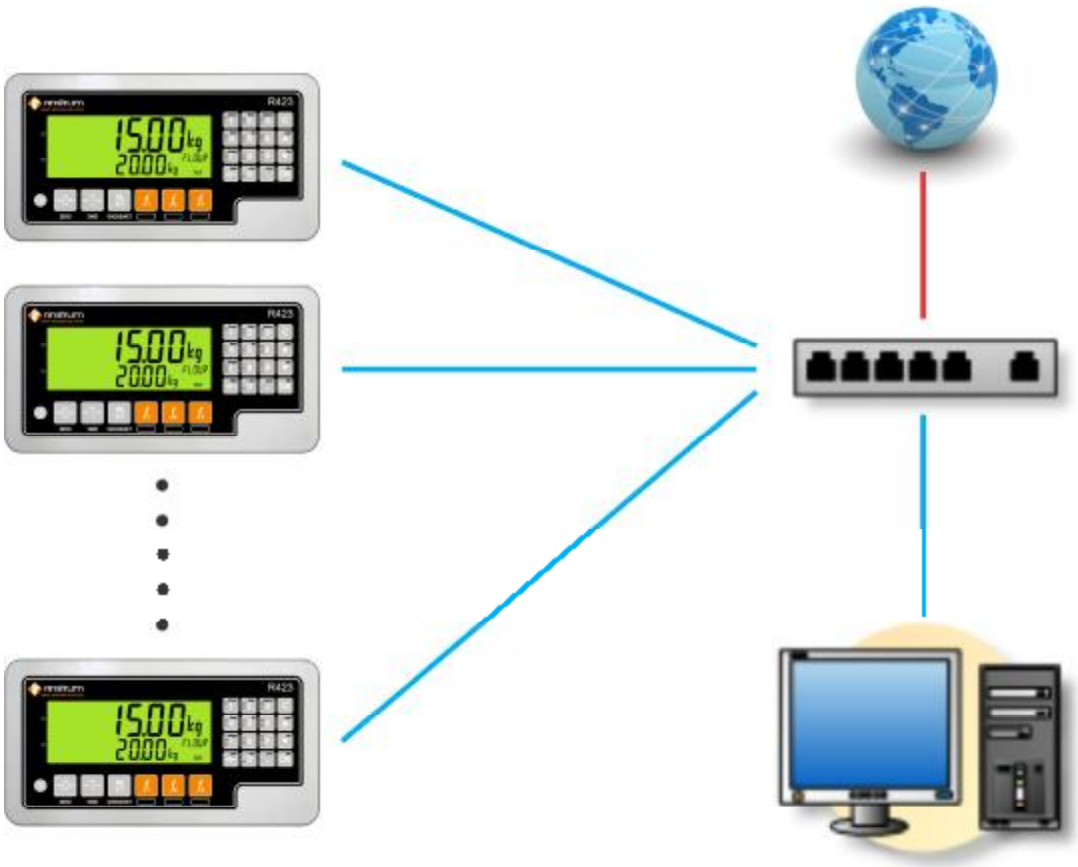


Figure 2: Physical Architecture

2.4. Connecting the M4223

1. Plug the M4223 into the back R400 and tighten the screws to secure the module.
2. Plug an Ethernet cable into the M4223.
3. Restart the R400

2.5. Verify the Connection

1. Bring up the modules menu by holding the 0 key
2. Use the arrow keys to navigate until TYPE displays M4223
3. Press the +/- key until STATUS is shown (should be OK)

   If the STATUS displayed is ETH.ERR this indicates the M4223 is not talking to the R400 properly. Check that the M4223 is correctly plugged into the back of the device, turn the device off, wait 10 seconds, and then turn it back on.

4. Press +/- once more so the IP is displayed (referred to as <IP> from here)

   If the IP does not change from 0.0.0.0 within at least a minute after start-up, this indicates the module has not got an IP address. This may be because the Ethernet cable is not plugged in properly, or the network is not configured properly.

2.6. Remote Interface

2.6.1. Logging In to the Remote Interface

1. Open a connection to the module
   a. Windows
      i. Download and open PuTTY
      ii. Select 'Telnet'
      iii. Enter <IP>, leave port as 23
      iv. Press 'Open'
   b. Linux
      i. Open a terminal
      ii. Type:

      telnet <IP>
2. Enter the username and password
   a. Default username/password: root/root

2.6.2. Mounting a USB

This allows for mounting an **NTFS formatted** USB storage disk.

From a remote interface type:

```
blkid
```

This will list the connected drives in the format of:

```
<partition>: UUID="<ID>" TYPE="<FILETYPE>"
```

One entry should have TYPE="ntfs" (and will typically be at `/dev/sda1`).

To mount the drive to the filesystem:

```
mkdir /mnt/usb
mount -t ntfs-3g <partition> /mnt/usb
```

2.7. Web Interface

2.7.1. Web Interface Features

- Display syslog

  This displays the kernel and application messages for user debugging.
- Change web interface password
- Reboot device
- List installed packages

  Lists the firmware packages that have been installed on the device, and allows users to remove them.
- Install new packages

  Allows users to install new firmware provided by Rinstrum

2.7.2. Logging in to the Web Interface

- Get the IP of the M4223 using 2.6.1 Logging In to the Remote Interface.
- Type this into a web browser
- A prompt should appear asking for a username and password
  - The default is admin/password

2.7.3. Upgrading Firmware

- Press 'Installed Packages'
- Check if the firmware you are trying to install already exists
  - If the firmware you are trying to install is already there, uninstall it
- Press 'Firmware Upload'
- Press the 'Choose File' button and navigate to the firmware you wish to install (should be an .rpk file)
- Press the 'Upload' button

3. Lua

3.1. Features

Lua is designed to be a fast, lightweight scripting language that is powerful enough to be used for complex projects but simple and flexible enough for new users to quickly overcome the learning curve and start writing effective scripts.

As such, the language features a minimum number of built-in libraries but has large support for user-written libraries.

Further reading: http://www.lua.org/about.html

3.2. Introduction to Lua

This is only intended to be a brief overview to Lua, and showcase the basic functionality. For more in-depth guides there are tutorials available online at http://lua-users.org/wiki/TutorialDirectory, and a reference manual is available online at http://www.lua.org/pil/contents.html.

Introductory Example

```
-- Variable scope
globalVar = "Hello "                    -- Global variable
local temp = "World"                    -- Local variable

-- Data types
varNum = 123                            -- Number
varString = "456"                       -- String
varBoolean = true                       -- Boolean

-- Printing
print(varNum, varString, varBoolean)    -- 123  456 true

-- String handling
newString = globalVar .. temp           -- Concatenate to "Hello World"
newString = varNum .. varString         -- Concatenate to "123456"

-- Simple if/else statement
if (varNum > 5) then
    print("Greater than 5")
else
    print("Less than or equal to 5")
end

-- While loop
local i = 0
while i < 5 do                          -- Print 0 to 4 on new lines
    print(i)
    i = i + 1
end

-- For loop
for i = 0,10,2 do                       -- Start at 0, end at 10, increase i by 2
    print(i)                            -- on each interation
end

-- Function that will return double the number
function multi(x)
    local y = 2*x                       -- y is local to this function
    return y                            -- and cannot be accessed outside of it
end
```

3.3. Function Arguments and Returns

Lua has simple ways of handling overflow of para

Function Arguments and Returns Example

```
function sum1(a, b, c)
    return a+b+c
end

print(sum1(1,2,3))      -- 6

--print(sum1(1,2))      -- Lua fills unused parameters with nils
                        -- This will error, as 1+2+nil does not add

print(sum1(1,2,3,4))    -- 6 (The extra argument is discarded)

-- The function has be improved by using default argument
-- This works by using short circuit evaluation of the 'or' operator
function sum2(a, b, c)
    a = a or 0          -- if a is non-nil, 'or 0' will not evaluate
    b = b or 0          -- if b is nil, the 'or 0' will evaluate and give b = 0
    c = c or 0

    return a+b+c
end

print(sum2(1,2,3))      -- 6

print(sum2(1,2))        -- 3

--print(sum2('a', 2))   -- This will error, as 'a' cannot be added

-- The function can be made robust by checking values given to it are numbers
function sum3(a, b, c)
    a = a or 0          -- if a is non-nil, 'or 0' will not evaluate
    b = b or 0          -- if b is nil, the 'or 0' will evaluate and give b = 0
    c = c or 0

    if (type(a) ~= 'number' or
            type(b) ~= 'number' or
            type(c) ~= 'number') then
        return nil, "non-numeric argument"
    end

    return a+b+c
end

print(sum3(1,2,3))      -- 6

print(sum3(1,2))        -- 3

print(sum3('a', 2))     -- This will print nil and an error message
                        -- but will not crash lua.

-- To read values out of the functions, variables can be comma separated
-- This can be used to see if the function has returned an error
val, err = sum3(1, 2)
print(val, err)         -- 3, nil

val, err = sum3('a', 2)
print(val, err)         -- nil, "non-numeric argument"
```

3.4. Standard Libraries

Lua comes with a minimum number of standard libraries included.

These include the fairly standard core, math, string, OS, and IO libraries, and the more advanced table, and coroutines libraries.

3.5. Advanced Concepts

3.5.1. Tables

A table in Lua is an associative array that maps a key to a value.

Tables are very simple to create, and can easily be used to store and retrieve data.

Table Example

```
t = {}                          -- Initialise the table
t["foo"] = 3                    -- Set the value of "foo" to 3
print("foo: " .. t["foo"])      -- foo: 3

t.foo = 5                       -- This is equivalent to t["foo"] = 5
print("foo: " .. t["foo"])      -- foo: 5
print("foo: " .. t.foo)         -- This is the same as the previous line

--t.5 = 1                      -- This line is not allowed, and will error
t["5"] = 1                      -- This works though
print("foo: " .. t["5"])

table.insert(t, 7)              -- Insert a value by explicitly using table

for key,value in pairs(t) do    -- This will print the values in no particular
    print(key,value)            -- order. Unordered storage is a property of
end                             -- storing data as a table.

t.foo = nil                     -- This will remove "foo" from the table

t.innerTab = {}                 -- Tables can also contain other tables, which
                                -- can be accessed and traversed as above.
t.innerTab["foo"] = "abc"
t.innerTab.bar = "def"

-- A typical use might be to setup a config data table
local config = {
         var1 = 5,                      -- global settings
         var2 = 'Test',
         general = { name = 'Fred'},    -- [general] group settings
         comms = {baud = '9600',
                  bits = 8,
                  parity = 'N',
                  stop = 1},            -- [comms] group settings
         batching = {target = 1000,
                     freefall = 10}     -- [batching] group settings
         }
```

Tables as Arrays

```
t = {"a", "b", "c", "d", "e"}    -- Initialise the array with 5 elements
                                 -- This is equivalent to:
                                 -- t = {}
                                 -- t[1] = "a"
                                 -- t[2] = "b"
                                 -- etc.

print (#table)                   -- Length of the table (5)

t[1] = nil                       -- Remove element "a"
t[2] = nil                       -- Remove element "b"

print(#table)                    -- 'Length' of the table (0). Isolated elements
                                 -- are not counted.

table.insert(t, "f")             -- Add a new element to the end of array (6, f)

for key,value in pairs(t) do     -- Print the array, not necessarily in order
    print(key,value)
end

t[1] = "a"
t[2] = "b"

for key,value in pairs(t) do     -- Print the array, will be in order as keys are
    print(key,value)             -- consecutive
end
```

3.5.2. Modules

Tables are also the basis for modules in Lua, and are used to return a collection of module variables and functions.

samplemodule.lua

```
local _M = {}

_M.moduleVar = 5

function _M.double(num)
    return 2*num
end

return _M
```

This module can then be required by other Lua scripts, and the module variables can be read and modified.

Calling Sample Module

```
local sample = require "samplemodule"

print(sample.double(5))          -- 10
print(sample.moduleVar)          -- 5
```

3.5.3. Coroutines

Coroutines in Lua allow for cooperative multi-threading. This is different to pre-emptive multithreading traditionally used in computing as it requires the thread that currently has control to yield rather than have control taken from it. This approach is faster and requires much less overhead than using multithreading.

For most applications coroutines are not necessary, but they can be extremely useful for tasks that require processing a large amount of data while concurrently handling events.

A typical use for coroutines is IO dispatching, where each connection has its own coroutine and a dispatcher resumes the coroutines when data comes in on the connection.

4. Lua API

4.1. Introduction

Comprehensive details of how to use the Lua API are contained in programmers documentation automatically generated from structured comments in the libraries themselves using a utility available onboard the M4223 called ldoc.

All functions in the API and are covered by the GNU GPL (http://www.gnu.org/licenses/gpl.html).

The LUA API libraries are structured in layers and designed so that most applications can be coded using the high level functions. These high level functions are explored in this chapter with details of the lower layers explored in subsequent chapters.

4.2. myApp

myApp is an application template that contains all the boilerplate configuration setup for the most common types of applications.

myApp uses the rinAPP framework.

To start a new project, copy myApp.lua into your project directory, rename to your project name and add in the details of your application.

4.3. rinApp

rinApp creates all the application framework. It loads in the lower level libraries required to implement timers and communications sockets so that you do not need to do that explicitly in your application.

rinApp.addK400()

addK400 is called to establish the connection to the R400 instrument. When called addK400 loads in all and configures all the libraries needed to control that instrument.

If the connection is to a remote instrument then specify the IP address of that instrument. Otherwise the default operation of the function is to establish connection with the local host instrument using a local linux socket.

rinApp.running is a global flag maintained by rinApp which is **true** while the application is running and is set **false** in the event that the application has been instructed to exit.

rinApp.cleanup()

A function that should be called at the end of your application that releases the R400 instrument from the control of your application, frees up the communications sockets and generally tidies up for a clean exit.

Debug:

rinApp loads and configures a copy of the rinDebug library with the debug level set by a command line parameter passed into the application.

Eg lua myApp.lua info

runs the myApp script with debug configured to show INFO messages.

Terminal Commands:

rinApp establishes a dedicated posix connection for the application that allows for interaction with the running application using the ssh/telnet terminal. To use this type in the commands and press enter directly from the terminal as follows:

exit sets the running flag false

DEBUG, INFO, WARN, ERROR, FATAL set the debug level to determine what types of messages are logged.

4.3.1. Streaming

Streams allow for the contents of up to 5 registers to be transferred to the LUA engine in the one transaction. The stream can be configured to update at 1Hz, 3Hz and 10Hz, or on change.

`addStream()`

Add a register to the stream set and setup a callback function to process the data. The callback function can be configured to be called whenever data is received or only when the received data is different from previous update.

`removeStream()`

Remove a register from the stream set.

`setStreamFreq()`

Call to set the frequency of the stream update. By default the frequency is set to update on change.

4.3.2. Status Change Events

By default, rinApp adds the instrument status register to the stream list and so configures the K400 library to monitor changes in instrument status.

The following functions allow you to modify which status bits are monitored and register callback functions to respond to status changes.

setStatusCallback()

Register a function to be called on the change of a particular status bit. Callback function gets given the status bit and the current state.

setRTCStatus()

By default changes in the real time clock are not monitored in the status bits. setRTCStatus allows you to enable or disable change in RTC monitoring. When enabled the status stream will contain a bit every time the instrument Real Time clock updates each second.

setRDGStatus()

Use this routine to configure a status bit that changes every time a set number of weight readings have been made.

setIOStatus()

Use this routine to configure a status bit that alerts the application if any of a defined subset of the 32 IO points has changed.

4.3.3. Keyboard Events

By default, rinApp adds the instrument keyboard register to the stream list and configures the instrument to send all keyboard events to the LUA engine before processing. This allows the LUA application to define the behaviour of any key in the instrument. Keys events that are not processed in the LUA application are sent back to the instrument to invoke the default actions.

The following functions enable your application to respond directly to operator key presses:

Callback functions can be linked to a single key or to groups of keys (eg all function keys or all number keys).

Keyboard event handling is a special implementation of stream handling, and allows for key presses to be streamed up to the M4223 so that custom actions can be made via callbacks using `setKeyCallback`. If there is no callback associated with a key, or the callback does not return true, then the key is passed back to the R400 to be handled normally.

As well as being supporting callbacks associated with a single key, the DWI library supports associating callbacks with key groups using `setKeyGroupCallback`. The fixed key groups (in order of priority), are `cursor`, `numpad`, `keypad`, `functions`, `primary`, and `all`. If callbacks for multiple, overlapping key groups are set, the callbacks will be called in order of priority until one of them returns true.

There are three types of key events: `short`, `long`, and `up`. A normal key press results in short and up key events while `long` and `up` events are triggered when the key is held down for 2 seconds or more.

4.3.4. User Dialogue

The following library services are provided for regular user interface tasks. These are modal processes focused on the user that do not return to the main application until the user responds but keep all the non-user background activities running.

GetKey() Waits for a key from a particular key group to be pressed.

edit() Prompt user to enter data of a particular type and press OK

delay() Delay for a specified number for millisecs but keep background activities running while you wait.

askOK() Prompt user to press OK or CANCEL

selectOption Prompt user to select from a list of options

4.3.5. Setpoint Support

The R400 supports up to 32 I/O control points that can be configured as outputs. It is possible to directly control individual outputs from within your LUA application. Alternatively there are functions to setup the realtime setpoint functions built into the instrument firmware.

Direct Control

enableOutput(), releaseOutput() Set or release a particular IO for direct LUA control

turnOn(), turnoff() Turn on or off a particular IO point that has been configured for LUA control by enableOutput()

turnOnTimed() As with turnOn() but takes a parameter to determine how long the output is to remain on before turning off.

RealTime Control

setNumSetp() Set the number of realtime setpoints

setpName() set the name of the setpoint

setpIO() set the physical IO point controlled by the setpoint

setpType(), setpSource(), setpLogic(), setpAlarm(), setpHys(), setpTarget() set the setp control parameters

For a complete description of the functionality of the built in setpoint features refer to the Reference Manual for the particular R420 firmware.

4.3.6. Analogue I/O Control

The M4401 provides analogue output either 4-20mA or 0..10 V.
It is possible to control the analogue output values directly from LUA as follows:

setAnalogSrc() Set this to COMMS to enable local LUA control
setAnalogType() Voltage or Current
setAnalogClip() controls whether output is clipped to nominal limits or allowed to exceed these.
setAnalogVal() 0.0 .. 1.0 corresponds to analogue output range
setAnalogPC() 0 .. 100%
setAnalogVolt() 0 .. 10.0V
setAnalogCur() 4.0 .. 20.0 mA

4.3.7. Serial Ports

The R420 supports up to 2 serial ports each with a bidirectional and a transmit only port. These are designated as 1A,1B, 2A, 2B with 'A' ports being bidirectional.

printCustomTransmit Instruct R420 to expand the token string supplied and transmit out the designated serial port. See the R420 Reference Manual for a full list of print tokens.

reqCustomTransmit Instruct R420 to expand the token string supplied and return.

In addition it is possible to configure the R420 to buffer incoming serial traffic. A status bit is available in the system status register to indicate that serial data is available. Read the associated buffer register to collect the serial data.

Write to the serial buffer register to send serial data out the R420 ports.

It is also possible to use the USB port to manage USB serial ports directly from Lua.

4.3.8. LCD Control

To control the LCD display directly the display mode needs to be set to TOP.

The instrument LCD is divided into 4 areas that will display whatever text is written to them:

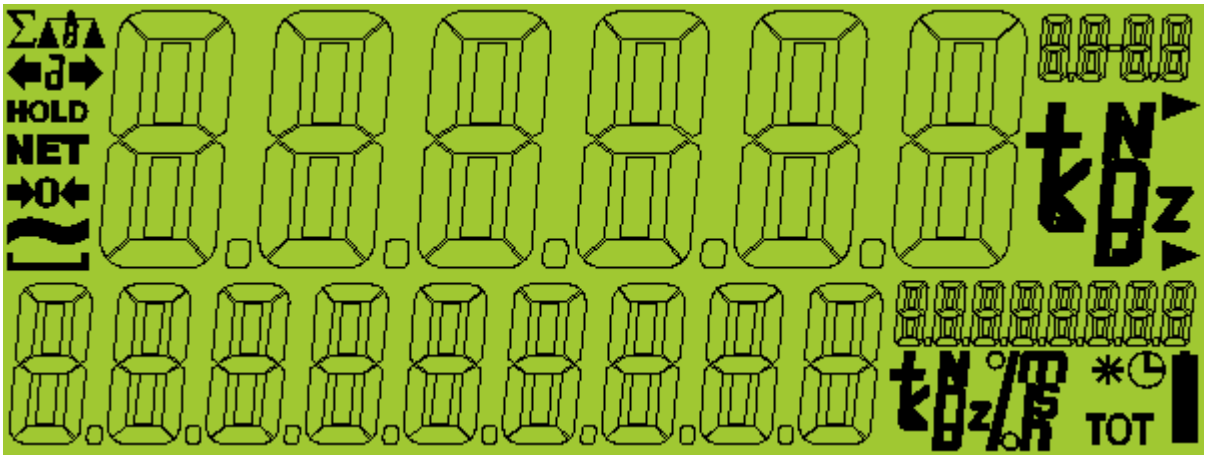
`writeTopLeft(),` **`writeTopRight(),`**

`writeBotLeft()` **`writeBotRight()`**

In addition to the main data areas there are also two separate units groups and two annunciator groups.

setBitsTopAnnuns()

clrBitsTopAnnuns() **writeTopUnits()**



setBitsBotAnnuns()

clrBitsBotAnnuns()

writeBotUnits()

Automatic Updates

Alternatively these areas can be set to display particular register data automatically.

setAutoTopAnnun(), setAutoTopLeft(), setAutoTopRight(), setAutoTopUnits(), setAutoBotUnits(),

are used to link a particular register address with a particular display area.

5. Lua Libraries

The M4223 comes with a variety of Lua libraries that have already been configured for the device.

5.1. LuaBitOp 1.02

Provides bitwise operations to Lua scripts such as 'or', 'not', 'and', 'xor', etc.

Further reading: http://bitop.luajit.org/

5.2. LuaSocket 2.0.2

Provides a socket interface so that Lua scripts can connect to other machines.

Supports TCP, UDP and Unix sockets, as well as providing special support for HTTP, FTP and SMTP connections.

Further reading: http://w3.impa.br/~diego/software/luasocket/

5.3. LuaLogging 1.2.0

Provides an API to structured, levelled logging of data. Data can be logged at a DEBUG, INFO, WARNING, ERROR or FATAL level, and the output can be configured to filter output below a set level.

This filtered output can be displayed to console, file system, email, socket and SQL.

Further reading: http://www.keplerproject.org/lualogging/

5.4. LuaPosix 5.1.23

Provides a POSIX binding (including curses) to C API's.

Further reading: https://github.com/luaposix/luaposix

5.5. LuaFileSystem 1.6.2

Provides a method for interacting with the underlying directory structure and file attributes of the file system.

Further reading: http://keplerproject.github.io/luafilesystem/

5.6. Penlight 1.0.2

Provides alternate data types and functionality for Lua.

Further reading: http://stevedonovan.github.io/Penlight

5.7. LDoc 1.2.0

Provides HTML documentation based on commented code.

Can be called on the device using 'ldoc' command, and can be used for generating the code documentation (e.g. `ldoc -d src`)

Further reading: https://github.com/stevedonovan/LDoc

5.8. LuaSQL 2.1.1

Provides access to databases using SQL interfaces.

Currently only supports MySQL, but will be upgraded in the future to allow for MSSQL connections over ODBC.

Further reading: http://www.keplerproject.org/luasql/

6. Rinstrum Lua Libraries

6.1. rinDebug

This module wraps around LuaLogging and provides a clean way of serialising and printing variables and tables. Variables are converted to strings, and tables are recursively expanded to show all the data they contain before they are logged.

A copy of rinDebug is automatically loaded and configured by rinApp with the debug level set by the command line parameter.

Data can be logged with an identifier, which can be used to easily find the logged data in the log file, and a level (DEBUG, INFO, WARN, ERROR, FATAL) which can be used to control the verbosity of the debugging.

The debugger will output all messages which are greater than or equal to the level the debugger is started with. For example, if the debugger is started at INFO level (the default), INFO, WARN, ERROR and FATAL log messages will be displayed but DEBUG level messages will not be.

Data can be logged to a variety of locations such as console, a file or an SQL database depending on the settings. The method for doing this can be found in the LuaLogging documentation.

The logger type can be changed in the rinDebug file, and the level can be set by calling `rinDebug.configureDebug`.

printVar(name, v, level)

This is the main debug function called with an optional name to be logged along with the contents of variable v at a particular debug level.

6.1.1. rinCMD Network Protocol

The entire programmability of the R400 instrumentation is built on the foundation of the rinCMD protocol interface which provides various functions for a comprehensive list of register settings.

Following is a brief overview of the protocol that the various libraries use to construct the programming interface.

The network protocol uses ASCII characters with a single master POLL / RESPONSE message structure. All information and services are provided by registers each of which has its own register address.

The basic message format is as follows:

ADDR	CMD	REG	:DATA	8

By convention the LUA libraries assume that there is only one instrument connected to any given socket so all commands are sent out with the broadcast address.

ADDR

ADDR is a two character hexadecimal field corresponding with the following:

ADDR	Field Name	Description
80_H	ADDR_RESP	'0' for messages sent from the master (POLL). '1' for messages received at the master (RESPONSE)
40_H	ADDR_ERR	Set to indicate that the data in this message is an error code and not a normal response.
20_H	ADDR_REPLY	Set by the master to indicate that a reply to this message is required by any slave that it is addressed to. If not set, the instrumet should silently perform the command.
00_H .. $1F_H$	Indicator Address	Valid instrument addresses are 01_H to $1F_H$ (1 .. 31). 00_H is the broadcast address. All slaves must process broadcast commands. When replying to broadcasts, slaves reply with their own address in this field.

CMD is a two character hexadecimal field:

CMD	Command	Description
05_H	CMD_RDLIT	Read register contents in a 'human readable' format
11_H	CMD_RDFINALHEX	Read register contents in a hexadecimal data format
16_H	CMD_RDFINALDEC	Same as Read Final except numbers are decimal.
12_H	CMD_WRFINALHEX	Write the DATA field to the register.
17_H	CMD_WRFINALDEC	Same as Write Final except numbers are decimal.
10_H	CMD_EX	Execute function defined by the register. Uses parameters supplied in the DATA field.

REG	is a four character hexadecimal field that defines the address of the Register specified in the message.
: DATA	carries the information for the message. Some messages require no DATA (eg Read Commands) so the field is optional. When a DATA field is used a ':' (COLON) character is used to separate the header (ADDR CMD REG) and DATA information.
8	is the message termination (CR LF or ";").

6.1.2. Register Access

At the lowest level it is possible to directly manipulate the R400 instrument using rinCMD commands. There are a number of functions provided to make this convenient. The source code for these is contained in the rinCon.lua file.

All of the common register addresses are already declared in the library so you can use names like REG_GROSS in your code rather than the actual constant vale of 0x0026 (40 decimal). This makes your code more readable and easier to maintain.

If you need to use a register that is not already declared in the library it is a simple matter of looking up the R400 reference manual appendix or using the Viewer software or .RIS files to determine the address which can then be declared in your own application.

`send(cmd,reg,)` is useful for sending a message to a connected device, and takes arguments for the command, register and data. The default behaviour is

`preConfigureMsg()` is useful for wrapping up the send function and the arguments into a single function that can be called easily .

To receive data, `rinCon.bindRegister` provides a way of binding the register on a received message to a callback function. This means that whenever the device sends up a message associated with a bound register, the bound function is called with the data as an argument. `rinCon.unbindRegister` removes a registers bound callback function.

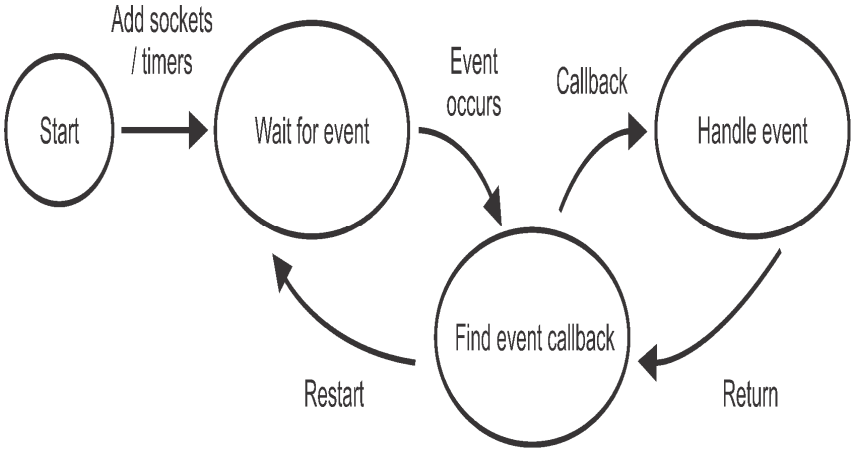
6.2. K400

This module builds on top of rinCMD and provides all of the instrument services exported by rinAPP.

This includes data streaming (on change or at a set frequency), key interception (allowing the program to take control of the R400's inputs), status monitoring, LCD control, analogue I/O control, and set point control, and is designed to operate as part of rinSystem.

6.3. rinSystem

This module provides socket handling and framework for user applications. Sockets (which are compatible with LuaSocket) and a corresponding callback function can be registered with the framework, and when data is received on a socket the callback function will be run. Timers with corresponding callback functions can also be added, and can be set to be repeating or single use.



The program flow is single threaded, which means that any event handling should not take a significant amount of time or other items in the system will be delayed.

rinAPP loads and configures rinSystem linking the sockets and timer services with K400 instrument services to create the application framework.

6.3.1. Sockets

rinSystem allows for LuaSocket-compatible sockets to be registered with the framework (using `System.sockets.addSocket`) along with a callback function that will run when the socket receives data.

To support terminal I/O as well as network sockets, the IOSocket Library provides LuaSocket-compatible socket for standard I/O that can be registered with the framework (using `System.IOsockets.makeSocket`).

6.3.2. Timers

The timer module allows for a callback to be run at certain intervals when added with `System.timers.addTimer`. These are only approximate timers though, and are suited for timing operator interfaces and broad process timing. These timers are not suited to real-time applications, and have an accuracy of approximately 10ms, but accuracy is not guaranteed.

6.4. rinRIS

This module can read a RIS file (which contains Rinstrum instrument settings) and send the configuration to the R400. This is useful for quickly and easily configuring the device for different scripts.

RIS files are created by the view400 and save400 utilities and are a convenient way to establish the default operating parameters for an application.

6.5. rinCSV

This module offers functions for creating a multi-table database stored and recalled in CSV format.

There is a separate .CSV file created for each table.

6.6. rinINI

This module provides services for saving and restoring table settings in a table to .INI configuration files.

6.7. Updates

As well as being provided with the release, the Lua Library has been released on Github. Github is a collaborative code sharing website that hosts source code that is version controlled with Git.

Github will always have the latest version of the library, and will have a history of all stable releases made by Rinstrum.

The libraries are available at https://github.com/rinstrum/LUA-LIB.

For more information on Git: http://git-scm.com/

7. Example Applications

7.1. Single-Device Control

The hello application outlines how rinApp can be used to write a simple script.

hello.lua

```
--------------------------------------------------------------------------------
-- Hello
--
-- Traditional Hello World example
--
-- Configures a rinApp application, displays 'Hello World' on screen and waits
-- for a key press before exit
--------------------------------------------------------------------------------

-- Require the rinApp module
local rinApp = require "rinApp"

-- Add control of an DWI at the given IP and port
local DWI = rinApp.addK400("K401")

-- Write "Hello world" to the LCD screen.
DWI.writeBotLeft("Hello")
DWI.writeBotRight("World")

-- Wait for the user to press a key on the DWI
DWI.getKey()

-- Clean-up the application and exit
rinApp.cleanup()
os.exit()
```

7.2. Basic Functionality

The testDialog application demonstrates how to display data to the R400 and get user input.

testDialog.lua

```
--------------------------------------------------------------------------------
-- testDialog
--
-- Example of how to use various library dialog functions
--------------------------------------------------------------------------------

local rinApp = require "rinApp"
local DWI = rinApp.addK400("K401")

--------------------------------------------------------------------------------
-- Put a message on LCD and remove after 2 second delay
DWI.writeBotLeft("DIALOG")
DWI.writeBotRight("TEST")
DWI.delay(2000)
DWI.writeBotLeft("")
DWI.writeBotRight("")

--------------------------------------------------------------------------------
-- Prompt user to enter the number of times to sound buzzer, validate,
-- and then buzz after 0.5 second delay
local val = DWI.edit('BUZZ',2)
if DWI.askOK('OK?',val) == DWI.KEY_OK then   -- confirm buzz amount
   DWI.delay(500)
   DWI.buzz(val)
end

--------------------------------------------------------------------------------
-- Prompt user to select from a list of options. Options list will loop.
-- (e.g. if user presses 'up' key when option is large, loop back to small.
local sel = DWI.selectOption('SELECT',{'SMALL','MEDIUM','LARGE'},'SMALL',true)
DWI.delay(10)
-- show selected option (on device and console) and wait until key pressed
DWI.writeBotLeft(sel)
DWI.writeBotRight('SELECTED')
rinApp.dbg.printVar('Selected value', sel, rinApp.dbg.INFO)
DWI.getKey()

--------------------------------------------------------------------------------

rinApp.cleanup()  -- shutdown application resources
os.exit()
```

7.3. Advanced Functionality

The marquee application shows a more complex application that uses a timer to slide the user's text across the screen. A suggestion for improving that will help the understanding timers has been included.

marquee.lua

```
--------------------------------------------------------------------------------
-- Marquee
--
-- Allows for marquee messages to be displayed across the screen
--
-- POSSIBLE EXERCISE:
-- Write a keyboard callback that allows dynamic editing of the scrolling speed
-- when the up and down keys are pressed
--
-- Hint: you will have to stop and start the slide timer
--------------------------------------------------------------------------------

local rinApp = require "rinApp"
local DWI = rinApp.addK400("K401")


local msg = ''
--------------------------------------------------------------------------------
-- This is a timer callback that moves a message across the screen
local function slide()

    -- Check if message is finished
    if msg == false then
        return
    end

    -- If there's nothing left to move, clear the screen
    -- and write the msg to false so we know we're done
    if msg == '' then
        DWI.writeBotLeft('')
        msg = false

    -- If there's something left to write, write a substring of 9 characters
    -- to the device and remove a character from the message
    else
        DWI.writeBotLeft(string.format('%-s',string.upper(string.sub(msg,1,9))))
        msg = string.sub(msg,2)
    end
end

-- Start a time that will call slide
-- The timer has a 400ms delay between iterations, which can be easily altered
-- The timer has a 100ms delay before it starts for the first time
local slider = rinApp.system.timers.addTimer(400, 100, slide)


-- Format the string for slide
local function showMarquee (s)
   msg = '         ' ..  s
end




--------------------------------------------------------------------------------
```

```
-- Key handler
local function handleKey(key, state)
    showMarquee(string.format("%s Pressed ", key))
    if key == DWI.KEY_CANCEL and state == 'long' then
        rinApp.running = false
    end
    return true     -- key handled so don't send back to instrument
end

DWI.setKeyGroupCallback(DWI.keyGroup.all, handleKey)

-- Print a message
showMarquee("This is a very long message for a small LCD screen")

-- Loop and print key presses to the screen
-- If abort is pressed, break the loop
while rinApp.running do
   system.handleEvents()
end

-- Clean up and exit
rinApp.system.timers.removeTimer(slider)
rinApp.cleanup()
```

7.4. Multiple-Device Control

The multi-device application shows a way of remotely controlling of multiple R400s by adding two DWI connections to the system.

multi-device.lua

```
--------------------------------------------------------------------------------
-- multi-device
--
-- Demonstrates how the libraries can control multiple devices
--
-- Displays 'hello' to two instruments and closes when a button is pressed on
-- a certain instrument.
--------------------------------------------------------------------------------
local rinApp = require "rinApp"

local DWIa = rinApp.addK400("K401")
local DWIb = rinApp.addK400("K401","172.17.1.139", 2222)

DWIa.writeBotLeft("Hello")
DWIa.writeBotRight("A")

DWIb.writeBotLeft("Hello")
DWIb.writeBotRight("B")

-- wait for keypress from DWIa
DWIa.getKey()

-- Clean up the devices
rinApp.cleanup()

os.exit()
```

8. Rinstrum Packages

8.1. Luamux – L001.5xx.502

Luamux is an addition to the M4223 that allows the device to simultaneously serve multiple clients, which was previously not possible with the M4221. It provides an interface to the R400 in either TCP mode (default), or in UDP mode.

The configuration file can be found in `/etc/Rinstrum/muxconfig.lua`.

To change the external connection type, the table's `tcpEConfig` and `udpEConfig` can be set with the appropriate settings, and then `_M.ext` can be set to point to the desired configuration table.

The logging method and level can also be set in this file. By default the system will log to a temporary file, but it can be configured to log to a USB stick or an SQL database. Instructions on how to configure this can be found on the LuaLogging website.

8.2. Time Sync – L001.5xx.504

To maintain effective timestamps on data logging, the M4223 syncs its clock with the R400 on boot up. This process is invisible to the user.

8.3. Automatic Script Starting – L001.5xx.505

To assist users in automating their R400s, an automatic script service has been provided. In `/home/autostart/run.lua`, users have access to a Lua script that will be run on start-up. This script can be used for a variety of functions, and examples are given in the file for how to start other Lua programs and how to backup log files.

Run.lua is not appropriate for concurrently starting Lua applications that will run over a period of time. To do this, create another Lua file within the /home/autostart folder and the automatic script starter will detect and run it alongside run.lua.

Any script that is placed within the `/home/autostart` folder will have its standard output and error redirected to a file stored in `/var/log` and will be appended with the redirection type. For example, `run.lua` will create `/var/log/run.lua.out` and `/var/log/run.lua.err`.

9. Developer Environment

9.1. Environment Setup

9.1.1. Windows

To develop on Windows for the M4223, any FTP client and text editor can be used. The files can then be pulled off the device, modified, and pushed back up.

The recommended method is to use Notepad++ with the NppFTP plugin. This can be combined with PuTTY (for getting a terminal interface), and FileZilla if a more robust FTP client is needed for transferring files.

To assist in setting this up, a Ninite installer containing Notepad++, PuTTY and FileZilla is available.

Once this has been set up, the user can copy the Rinstrum libraries to the device (e.g. copy to `/home`) and begin developing.

- **Installer**

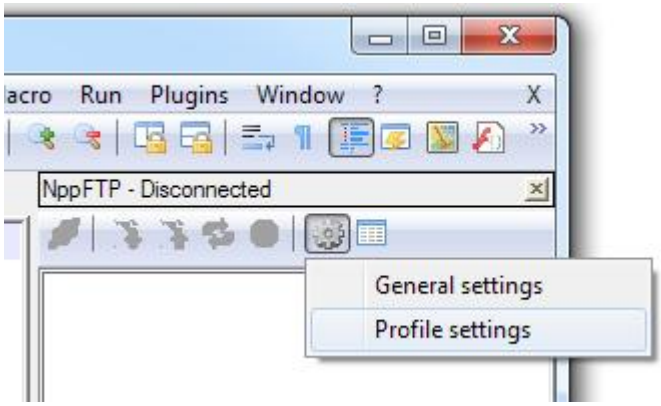
  When developing on Windows, it is recommended that you have (at least) Notepad++, FileZilla, and Putty. An installer has been included with the libraries that will silently install this software on your system.

  If you would like to configure the installation, it is recommended that you download the applications individually and install them.

- **Notepad++ Setup**

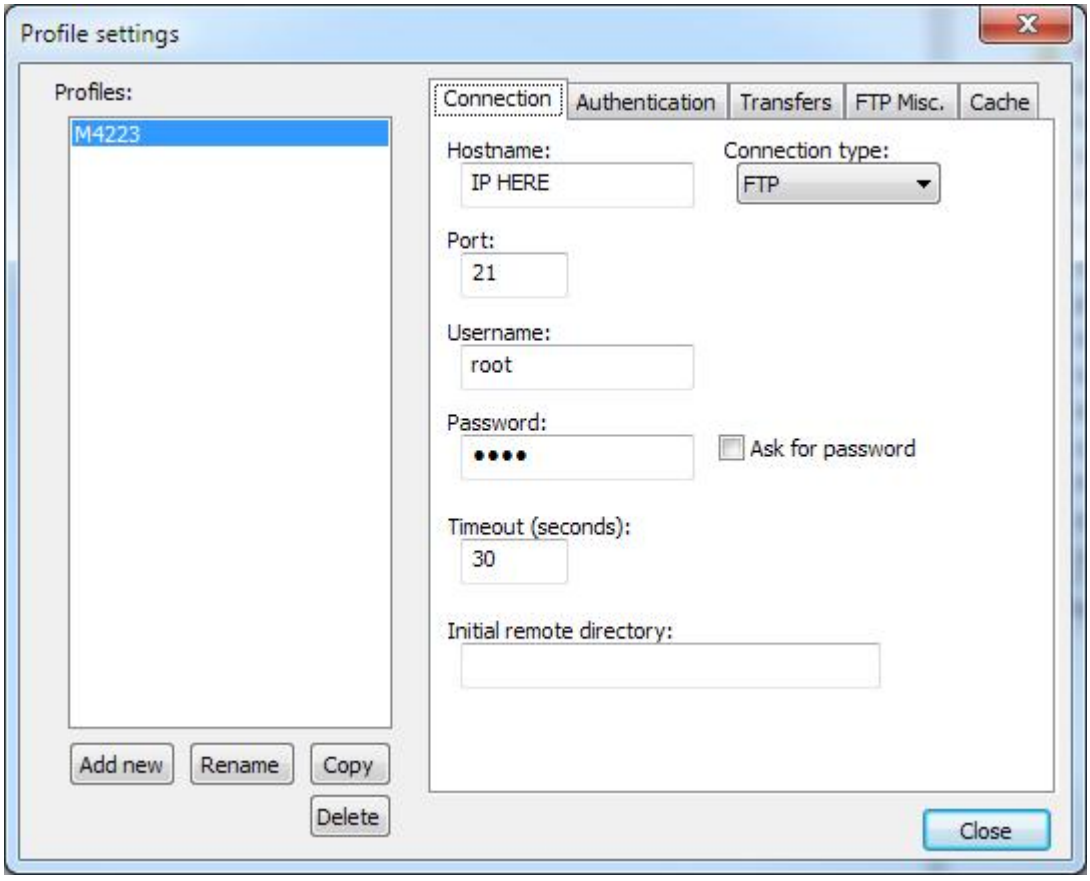
  Notepad++'s NppFTP plugin can be used to allow easy editing of files stored on the M4223.

Once NppFTP has been brought up, the profile can be configured for the M4223 you are using.



Once the profile window is open, press 'Add new' and name the device 'M4223', or similar.

Only the information on the first page needs to be set, specifically the device IP address in Hostname (see 2.5 Verify the Connection), and the username and password (see 2.6.1 Logging In to the Remote Interface).

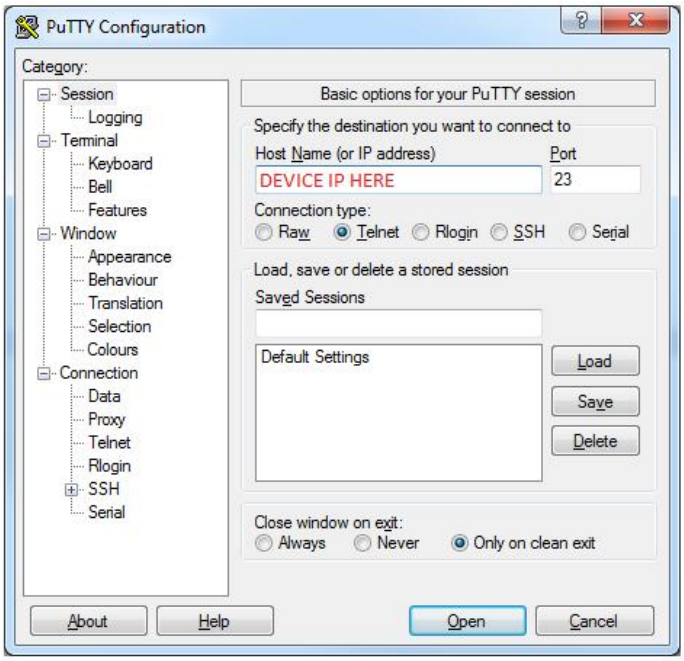


Once this has been done, press close, and click on the connection button to form a connection to the module.

Files can now be navigates to in the side bar, and can be opened in Notepad++ by double clicking on them. When they are saved, they will be written back to the module.

♦ **Putty Setup**

Once putty has been downloaded, it can be configured as below (for the IP address see 2.5 Verify the Connection). This gives a Telnet connection to the device, as shown in 2.6 Remote Interface



♦ **FileZilla Setup**

FileZilla can be set up by entering the IP address (for the IP address see 2.5 Verify the Connection) and pressing the Quickconnect button.

Once this has been done files can be dragged and dropped from users computer to the module.

9.1.2. Windows (Eclipse)

It is possible to set up Eclipse in windows to develop applications, but this is not recommended as it is more difficult to run/debug code for a Linux system on the Windows environment.

9.1.3. Linux

To develop on Linux for the M4223, Eclipse with the Lua Development Tools software (http://www.eclipse.org/koneki/ldt/) is recommended, and FileZilla is recommended for transferring files to the device. The device can be accessed using Telnet via the shell.

This method of development is recommended for experienced users who will be making complex scripts.

Instructions on how to set this up are available from

9.2. Library Usage

9.2.1. PC

The library can be used on a pc by requiring rinApp.lua. This can be done by writing the new Lua script within the same folder that rinApp.lua is stored in, or by changing the require statement to the remote location of rinApp.lua.

This can also be done by modifying the package.path variable in your Lua script.

9.2.2. Module

The library can be used on the module the same way it can be used on the PC, but if the libraries are being used for multiple applications it is possible to save space by copying the contents of '`/src`' to '`/usr/local/share/lua/5.1`'.

After this has been done, the contents of '`/test`' can be executed from anywhere on the module.