SMART WEIGHING SOLUTIONS

400 Series
Lua
Reference Manual

Table of Contents

1. Overview and Setup

1.1. Overview

This document covers connection of the Lua module to an R400 indicator, setup of the environment on the PC and an introduction to Lua scripting and the usage of Rinstrum Lua libraries. It can be read in conjunction with the following Rinstrum documents that are available from www.rinstrum.com.

- Rinstrum Application Package and API Reference (L000-517)
- Rinstrum Environment Setup (L001-506)
- Rinstrum Lua Quick Start Manual (L001-601)
- Rinstrum Linux Commands (L001-602)
- Rinstrum Lua Commands (L001-603)

1.2. M4223 Features

1.2.1. Ethernet port

Allows for remote connections to the module.

1.2.2. USB port

Full USB Host which is compatible with USB hub use. This allows for the connection of keyboards and other event devices like barcode readers, usb printers, usb serial devices and usb storage devices.

Note that USB storage devices need to be formatted with an NTFS file system.

1.2.3. Embedded Linux

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

Local file editing is supported with the use of vi.

1.2.4. Web interface

The module includes a web interface that allows for firmware to be easily upgraded. This is covered in detail in 1.8 Web Interface.

1.2.5. 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.

More details can be found in 6. System Libraries.

1.2.6. Online and offline capabilities

The M4223 provides powerful networking support but it is perfectly suited for standalone offline applications as it requires no services from outside for its operation.

1.2.7. 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.

1.3. Logical Architecture

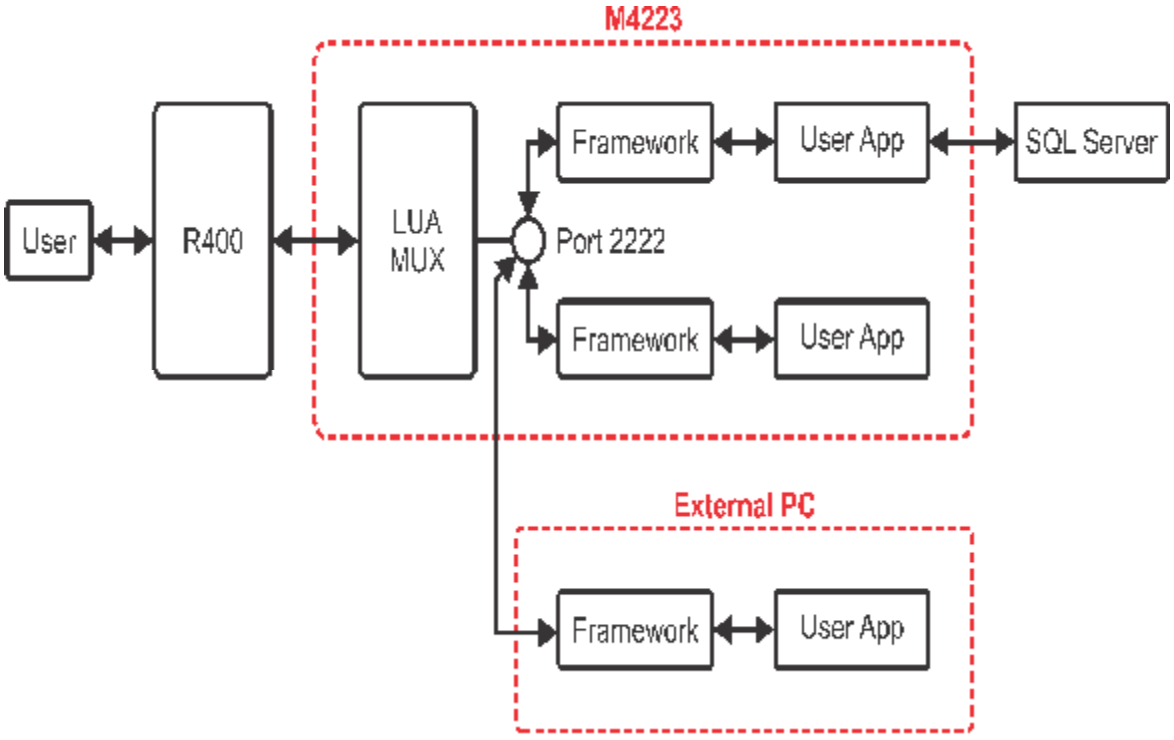


Figure 1: Logical Architecture with Framework (rinLib and rinSystem)

1.4. Physical Architecture

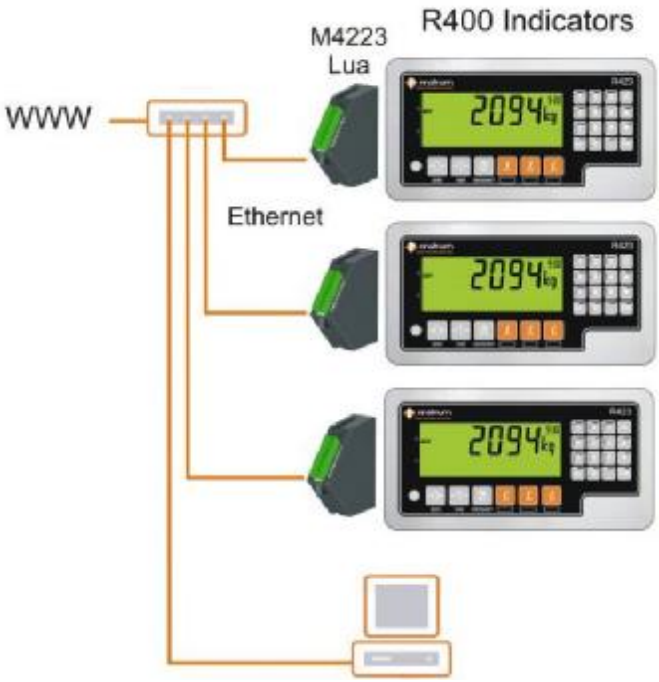


Figure 2: Physical Architecture

1.5. Connect M4223 to R420 or R423

- Disconnect power to the indicator
- Plug the M4223 into the back of the indicator and tighten the screws to secure the module.
- Plug an Ethernet cable into the M4223 to connect to your local network (the module is shipped expecting to receive an IP address from your local network DHCP, see Reference Manual for static IP address setup, this is simply configured in the R400 indicator)
- Turn on indicator

1.6. Verify Module Connection on the Indicator and establish its IP address

- Bring up the Acc (Accessory) menu by holding the 0 key on the alpha numeric keypad.
- Use the arrow keys to navigate until TYPE displays M4223
- 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.
- Press +/- once more so the IP is displayed (referred to as <IP> from here).
    - Use the "." on the alpha numeric keypad to scroll through the IP address if it is more than 9 characters.
    - If the IP does not change from 0.0.0.0 within at least a minute after start-up, this indicates the module is not getting an IP address. This may be because the Ethernet cable is not plugged in properly, or the network is not configured properly.

1.7. Remote Interface

1.7.1. 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

1.8. Web Interface

1.8.1. 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

1.8.2. Logging in to the Web Interface

- Get the IP of the M4223 using 1.7.1 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

1.8.3. 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 a .opk or .rpk file)
- Press the 'Upload' button

2. Lua

2.1. 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

2.2. 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, varNum < 100)   -- 123  456 true false

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

2.3. Function Arguments and Returns

Lua has simple ways of handling overflow of parameters

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"
```

2.4. Standard Libraries

Lua comes with a number of standard libraries included.

These include the core language interpreter as well as math, string, table, OS, and IO libraries.

2.5. Advanced Concepts

2.5.1. Tables

In lua, a table is an associative array that holds sets of key/value pairs. This is the only 'container' type in lua but with lua tables it is possible to create all of the common data types used in other languages.

The closest real world analogy is to think of the table as a bag of Christmas gifts with each gift in the bag being labelled. The label is the 'key' and the contents of the gift is the associated 'value'. With this structure it is possible to store all sorts of different information within the same construct including functions and other tables themselves.

Table Example

```
t = {}                          -- Initialise the table
t["age"] = 35                   -- store ('age' = 35) in table t
print("age = " .. t["age"])     -- age = 35
print("age = " .. t.age)        -- age = 35  equivalent syntax

--t.5 = 1                       -- This line is not allowed, and will error
t[5] = 1                        -- This works though as keys can be numbers as
print(t[5])                     -- well as strings.  Output is 1

for key,value in pairs(t) do    -- This will print all the key value pairs.
    print(key,value)            -- Note that the order is not defined.
end

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

t.address = {}                 -- Tables can also contain other tables, which
t.address.street = 'High St'   -- can be accessed and traversed as above.

-- 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 is 5
t.extra = 'test'
print (#table)                   -- Length of the table is still 5 even though
                                 -- there are 6 (key,value) pairs stored within.
                                 -- The # operator only works with items that
                                 -- have numeric ordered keys.

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

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

for key,value in ipairs(t) do    -- Print only the ordered items in the table.
    print(key,value)             -- This are printed in consecutive order
end
```

2.5.2. 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. Instrument API

3.1. 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.

3.2. 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.

3.3. rinApp

rinApp creates all the application framework. It loads in the lower level libraries required to implement communications sockets so that typically 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.run() run the application

rinApp.isRunning() returns **true** while the application is running.

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

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 instructs the application to exit

debug, info, warn, error, fatal set the debug level to determine what types of messages are logged.

setUserTerminal() lodge your own function to handle user commands entered from the telnet terminal.

3.3.1. 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.

3.3.2. Status Change Events

By default, rinApp sets up its own set of streaming registers to keep track of instrument status.

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

setStatusCallback(),setIOCallback(),setSETPCallback()

Register a function to be called on the change of a particular status bit, Input/Output or Setpoint. The callback function gets given the status bit and the current state.

anyStatusSet(), allStatusSet(), waitStatus()

anyIOSet(), allIOSet(), waitIO()

anySETPSet(), allSETPSet(), waitSETP()

Routines to check if any particular status bits are set; check if all specified status bits are set; or to wait for a particular combination of status bits.

3.3.3. Real Time Clock

rinApp automatically sets up and monitors the built in instrument real time clock.

RTCreadDate(), RTCwriteDate(), RTCreadTime(), RTCwriteTime()

Read and write the real time clock data and time

RTCdate(), RTCtime()

Return formatted date and time data

3.3.4. Keyboard Events

Instrument key events are first sent to the lua application for processing. Key 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:

A callback function can be linked to a single key or to groups of keys (eg all function keys or all number keys). There are four types of key events: `short`, `long`, `repeat` 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.

setKeyCallback() to register a callback on a particular key and event

setKeyGroupCallback() to register a callback for a particular key group and event

The keyboard library maintains an underlying idle timer to detect the case where an operator walks away from the instrument without completing the interface task.

setIdleCallback() to register a callback function to be called if the idle timer is triggered. Typically the original request would be aborted by the callback and the instrument returned to its idle state.

sendKey(), sendIOKey() sends an artificial key scan code to the instrument to be processed exactly as if the physical key or external input was pressed.

3.3.5. 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.

Use the **setIdleCallback()** provision to register a callback to manage the case that a dialogue routine is left running indefinitely. Also **dialogRunning()** returns true if any of the dialogue routines is actively waiting for user input.

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
sEdit()	Prompt user to enter string using keypad SMS style
editReg()	Trigger the instrument to run the local editing process for a built in register parameter.
askOK()	Prompt user to press OK or CANCEL
selectOption()	Prompt user to select from a list of options
selectConfig()	Prompt user to select from a list of options where each options includes a label and a value.
selectFromOptions	Prompt user to select one or more items from the list of options. Returns the list of items selected.

3.3.6. User Menus

createMenu() called to create a menu from a list of supplied elements. Menu elements have a unique reference identity which defaults to the operator prompt and can be any of the following:

- existing instrument register settings,
- integer, number or string variables
- passcodes
- option list
- sub menu
- generic function calback

run() run the menu, prompting the user for navigation input and data input as required. Menu items are enabled and disabled dynamically during the run process.

getValue(), setValue() called to get or set a particular menu item setting.

toCSV(), fromCSV() convert menu settings to and from a CSV table to enable the settings to be saved to file.

3.3.7. 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.

Note that the Lua control takes precedence to the instrument control. It is therefore possible to use the enableOutput and disableOutput routines to coordinate between realtime instrument control of an output and Lua oversight of the application flow.

3.3.8. 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

3.3.9. 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.

3.3.10. LCD Control

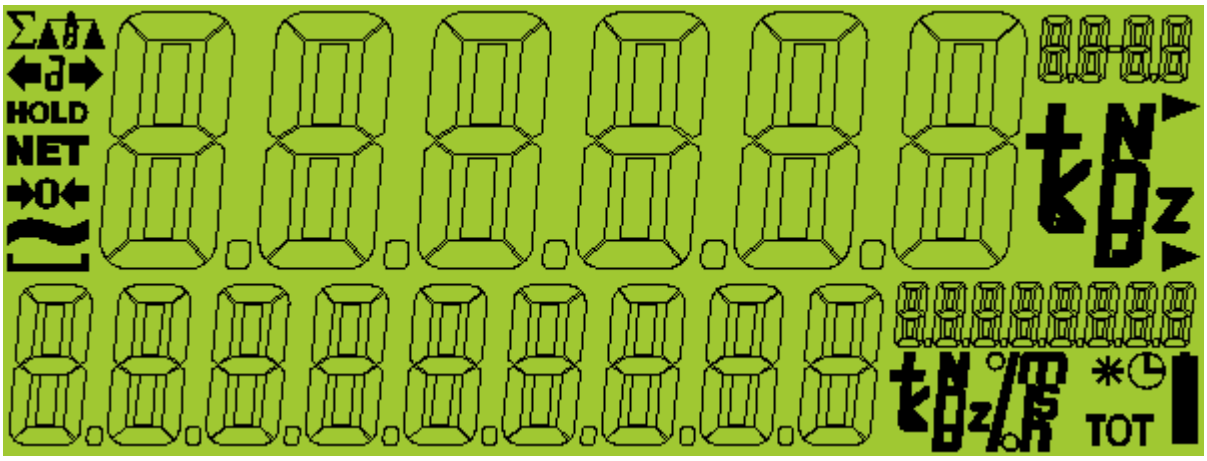
The instrument LCD is divided into 4 areas

`'topLeft',` `'topRight',`

`'bottomLeft'` `'bottomRight'`

write() writes text to the specified screen location. Write() automatically formats long text messages into a series of short messages that are presented one after the other. It is possible to control how fast the messages are presented and whether or not to wait for the message to complete before returning.

writeAuto() instructs the instrument to automatically update the specified screen location from data sourced from a particular register



In addition to the 4 main text areas there are two units field (top and bottom) and a variety of annunciators.

writeTopUnits() sets the units at the top of the LCD. Note that automatic updates configured for 'topLeft' also control the top units field.

writeBottomUnits() sets the units at the bottom of the LCD. Note that automatic updates configured for 'bottomLeft' also control the bottom units field.

setAnnunciators(), clearAnnunciators() set or clear particular annunciators. Again many of the annunciators are automatically controlled depending on the configuration of the automatic updates (eg motion, centre-of-zero, and NET)

3.4. 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.

4. rinCMD Network Protocol

The entire programmability of the R400 instrumentation is built on the foundation of the rinCMD protocol interface. It is not usually necessary to interact with the instrument at this low level typically as most services are wrapped up in the higher layers of the libraries. However for advanced users the raw instrument interface is available for direct control over the instrument operation.

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 ";").

4.1.1. 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.

All of the common register addresses are already declared in the library so you can use names like '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

To receive data, `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. `unbindRegister` removes a registers bound callback function.

5. Lua Libraries

The M4223 comes preloaded with the following libraries:

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. System 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.

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 console or a file accorinding to the settings in rinApp.ini. Configuration settings are entered into the rinApp.ini file as follows:

```
logger = console  -- options are console, file
level = INFO      -- INFO,DEBUG,WARN,ERROR,FATAL
timestamp = on    -- date/time stamping (on, off)
[file]
filename=debug.log  -- filename to use if logger = file
```

print(prompt, …)

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

debug(prompt, …), info(prompt,…), warn(prompt, …), error(prompt, …), fatal(prompt, …)

log a debug message at a particular level.

6.2. rinSystem

Core to the Rinstrum application framework is the use of sockets and timers. All of the high level libraries use these facilities extensively. It is possible to directly use these same services in your own applications to extend the capabilities of the framework.

6.2.1. Sockets

createServerSocket() Establish a server ready for an external process to make connections to.

addSocket(), removeSocket() add or remove a sockets.

readSocket(), writeSocket() Read and write messages to a particular socket.

6.2.2. Timers

Timers are processed by the frameworks event handler and provide a powerful way to control the application flow.

addTimer() register a function to be called with specified parameters at a particular time or times in the future.
removeTimer() remove the timer function.
addEvent() essentially a once off timer triggered immediately.

6.3. 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.

6.4. rinINI

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

6.5. 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. Traditional 'Hello World'

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"
local dwi = rinApp.addK400("K401") – connect to the instrument

-- start the application framework
rinApp.init()

-- Write "Hello World" to the LCD screen.
dwi.write('bottomLeft','Hello World')
dwi.getKey()-- Wait for the user to press a key on the DWI

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

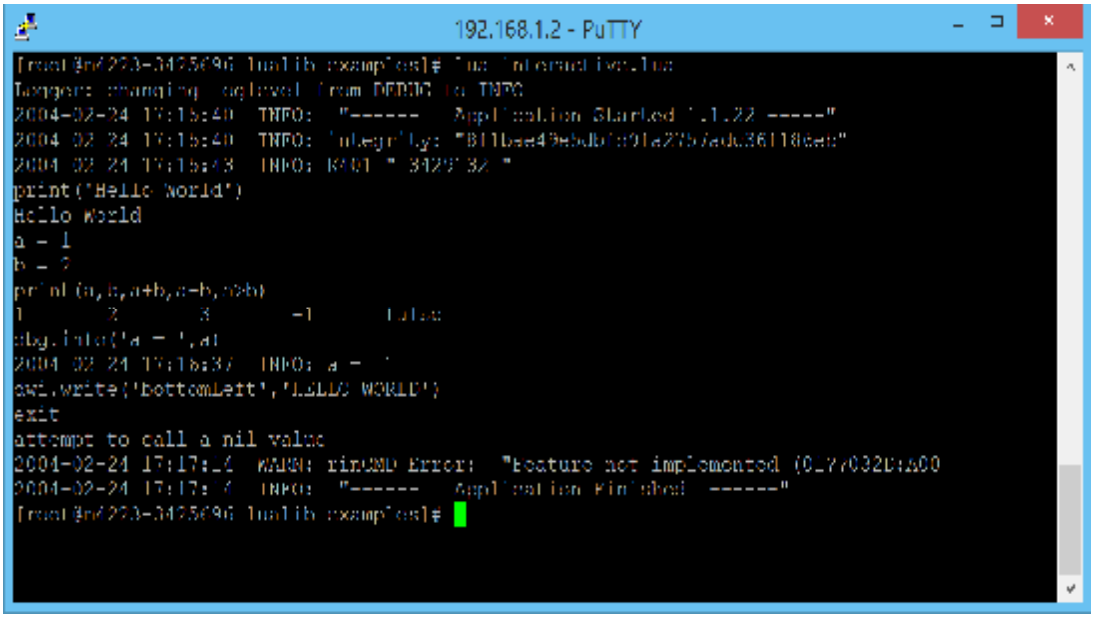
7.2. Interacting with Lua directly

Since Lua is an interpretive scripting language it is possible to experiment with the system directly.

To do this simple type in 'lua interactive.lua' from the lualib_examples directory.

Once running you can explore the language directly just by typing in commands directly into the telnet sessions. Interactive.lua automatically loads the framework for you so you can access all of the library functions directly.

It is possible to copy and paste instructions from a text file using the right-click paste function in PuTTY.



7.3. Multiple-Device Control

The multi-device application shows how to control two instruments from the one application.

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")                       --connect to local device
local dwiB = rinApp.addK400("K401","172.17.1.139", 2222) --connect to second device

-- start the application framework
rinApp.init()

dwiA.write('bottomLeft', 'Hello DWI-A')
dwiB.write('bottomLeft', 'Hello DWI-B')

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


-- Clean up the devices
rinApp.cleanup()
```

8. Developer Environment

8.1. Environment Setup

8.1.1. Windows

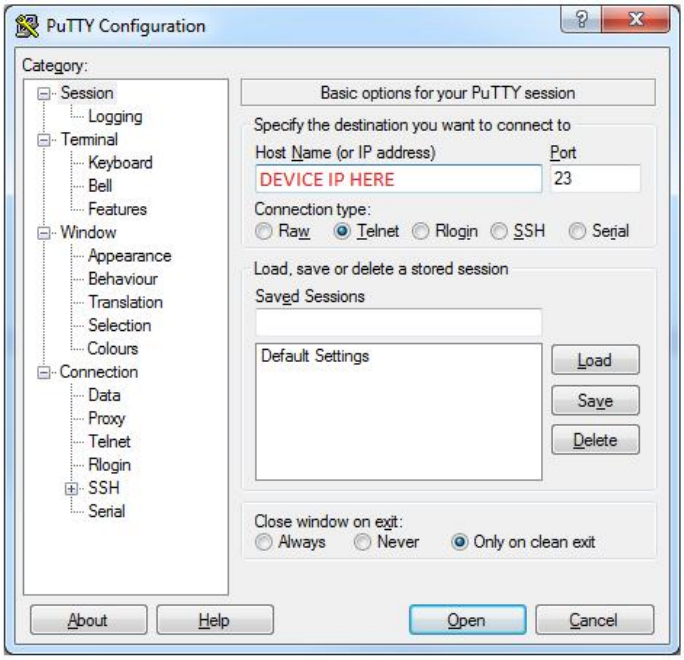
To develop on Windows for the M4223, any Telnet client, FTP client and text editor can be used. The files are pulled off the device using FTP, modified with the text editor, and pushed back using FTP. The Telnet service is used to log into the M4223 to run and debug the applications.

Installer

Rinstrum provides an installer (L001-506) that includes Notepad++, winSCP, and PuTTY. This is available as a download from the Rinstrum website www.rinstrum.com

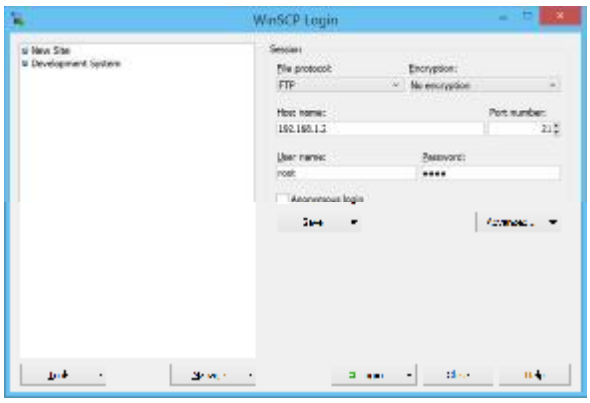
Putty Setup

Run PuTTY and open a Telnet session (for the IP address see Section 1.5). This gives a Telnet connection to the device, as shown in Section 1.6.

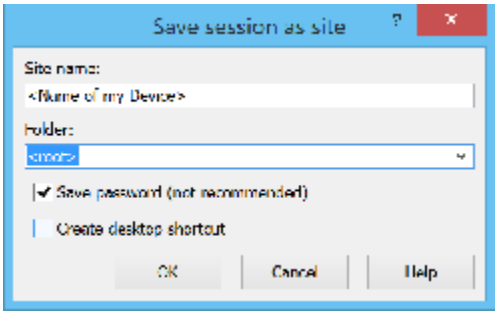


..

winSCP setup

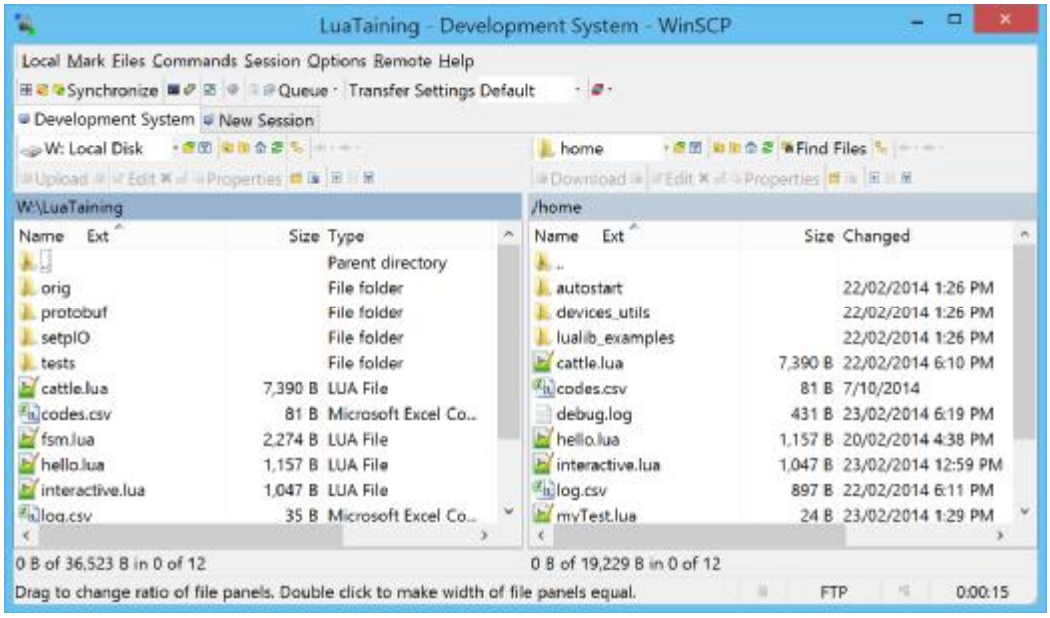


Click on New Site, select FTP and enter the IP address of your instrument. User name and password are root/root.



Save settings for later use using a convenient name.

Now you can navigate to the location of your application files on the PC (left pane) and to the location of your application files on the M4223 (right pane). Synchronise by simply dragging the files in the direction you want them to go.

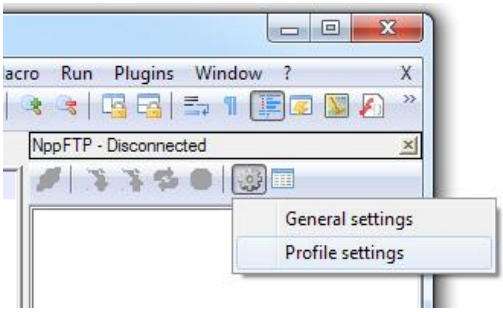


..

Using Notepad++ FTP plugin

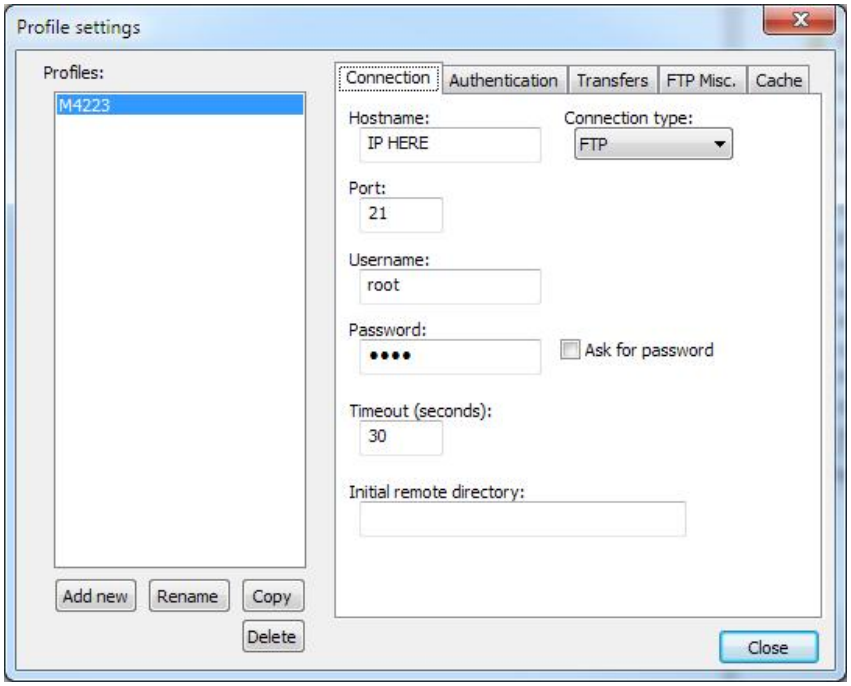
Notepad++'s NppFTP plugin can be used to allow easy editing of files directly on the M4223.

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 **Error! Reference source not found. Error! Reference source not found.**), and the username and password (see 1.7.1 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 navigated 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.

If is not possible with the NPP plug-in to copy files from the M4223 to your windows PC. This still needs to be done using a separate FTP client like winSCP.

8.1.2. 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.