rt.buffer Reference Manual

Scannex Electronics Limited

2018-10-17

Table of Contents

Data	Author	Release
2017-10-11	MP	For firmware 1.00
2017-10-27	MP	For firmware 1.01
2017-11-10	MP	For firmware 1.02
2017-12-06	MP	For firmware 1.03
2018-01-26	MP	For firmware 1.10
2018-03-08	MP	For firmware 1.11
2018-06-13	MP	For firmware 1.30 (documentation format change)
2018-10-17	MP	For firmware 2.00

Introduction

See also the "rt.buffer Getting Started Manual".

Lua 5.1.5 has been chosen, along with numerous patches and extensions that allow it to run in a very resource-constrained environment.

Lua in the rt.buffer has about 80k bytes of memory available to it, and this has required using short names for functions and variables (because even though code is compiled into p-code, the tables and functions are still referred to by their string name).

A combination of event-driven (aka "Reactor Pattern") programming and loop programming allows for a flexible combination of ultra-low-power operation, and ease of programming for sequential transmit/receive style operations.

A real-time-operating system (RTOS) provides multi-tasking support, and Lua is called from three different tasks in the RTOS.

1. `Main Lua Loop` Task
2. `Event Callback` Task
3. `Modem/3G Delivery` Task

Think of the 'Event Callback` Task as an "interrupt handler" in traditional coding.

It is important to be aware that the Event Callback Task must NEVER perform long-running operations. If you try, they will be aborted. Instead, callbacks issued from within this task should signal to the Main Lua Loop Task (or otherwise queue up requests in a Lua table).

There are also callbacks that occur within the Modem/3G Delivery Task — and these can take a reasonable time (minutes). However, the task can still choose to abort the running Lua callback.

Most of the system architecture has been designed to be completely flexible. Consequently, the process of designing a suitable Lua-based App may take some time and skill to be successful.

This document aims to provide a reference for the many Lua library extensions that have been written to complement the Lua core.

Useful Bookmarks

These links may be helpful:

- USB and Terminal
  - Terminal Commands
  - BootLoader Terminal Commands
- `c.` and `i.` values
  - Configuration Variables
  - Information Variables
- Scheduling
  - Job Schedule Strings
  - Schedule Examples
- LED Sequences

IoT Operations

Update Mechanism

The update mechanism works by pulling an `update.txt` file from the IoT server, and parsing the lines of the ASCII file.

When completed, the `update.txt` file will be renamed on the server as `update.YYYYMMDDHHMMSS.txt` (using the rt.buffer’s UTC time that the update was completed).

The commands and available options are listed below.

update.txt Keywords

> ℹ If the `c.iot_gz=0`, then the following transfers will not use gzip compression, and will not use the .gz extension.

`cellinfo`

Perform a cellular survey and post the results to the IoT server.

The file will be named `cellinfo.{i.rt_sn}.txt.gz`

`diag`

Post a diagnostics dump file onto the IoT server.

The diag file will be named `diag.{i.rt_sn}.txt.gz`

`COMMAND`

`COMMAND:Lua expression`

Execute a Lua command. This is syntactically the same as the USB command `lx`. However, the results of the Lua command will be thrown away.

Complex expressions can be executed, with a line length up to 128 characters.

```
lx job.utc(os.time()+6000, function() iot.go('user') end)
```

If an App has utility functions, for example to make configuration changes to the connected device, you can call these too. They will be executed from the context of the modem task.

```
lx CallMyFunction(10.34, 34.56)
```

getfile SOURCE,TARGET

SOURCE:The full path and name of the file on the rt.buffer

TARGET:The (path and) filename of where to save the file on the IoT server

Post a file from the flash file system onto the IoT server.

Listing 1. gz will be added to the target filename, if appropriate.

```
getfile /Logs/system.log,/temp/system.log
```

setfile SOURCE,TARGET

SOURCE: Source filename to ask from the IoT server

TARGET: The full path of where to save the file on the rt.buffer

Pull a file from the IoT server and save into the flash file system.

The source file cannot be gzip or zlib compressed. The rt.buffer does not have the RAM to decompress data.

```
setfile /temp/config.mine.txt,/Config/config.txt
```

If updating the /Config/config.txt file be **very** careful you do not change the IoT settings, or break the cellular settings… otherwise you could lose control of the rt.buffer!

NAME.blf

NAME: Filename (and path) on the IoT server to pull

Pulls a firmware image from the IoT server and burns.

```
../rt.buffer.Lua.1.02.blf
```

c.KEY = VALUE

c.KEY: Config parameter

VALUE: Config value

Make a configuration change

User Process

The user process is typically triggered by the magnet.

The rt.buffer will connect to the IoT server and repeatedly send the config and info sets. This should enable a back-end process to present the data to a live web-page, so any engineer can view the current information on a phone, tablet or PC.

The process will pause, by default 15s, and send again. This will continue for a default time of 10 minutes. The settings `c.user_ins` & `c.user_onm` override the values.

Configuration Options

The process will post the data to a file named `"user.{i.rt_sn}.txt.gz"` in the folder specified by
`c.iot_url`
`c.iot_user`.

If the `c.iot_user` value is not present, then `"Update/{c.site_name}"` is used (same logic as the `c.iot_upd` value).

The function `"doUser"` will be called on each loop, allowing the Lua App to include additional values (like live ADC readings) in the information tree.

Stopping the User Process

The back-end process can abort the user mechanism by leaving a file named: `"user.{i.rt_sn}.stop"`

When the rt.buffer detects this file, it deletes it and quits the User Process.

Lua Modifications

This section documents the additions to the standard Lua core that provide the functionality required for the {rpbuffer} product.

Lua Compiler and Virtual Machine changes

There are some basic internal changes to the Lua core, including:

1. Reading from a table that doesn't exist returns NIL, and does not generate a run-time error. Attempting to write to a NIL table will still generate a run-time error.
   - `lvm.c` modified
2. C/C++ style comments are supported with the `//` tag and `/…/` tag. This is in addition to the regular Lua comment marker `--`.
   - `llex.c:llex` function modified
3. Strings can use the C/C++ hex markers. e.g. `'\xee\ff'` is 0xee + 0xff.
   - `llex.c:read_string` modified
4. The os functions `execute`, `exit`, `getenv`, `setlocale`, and `tmpname` removed.

Lua "Basic Functions"

Fully included.

Coroutine Manipulation - `coroutine`

Fully included

Modules - `package`

NOT included. The memory requirements were too large.

However, you can still use Lua's `loadfile` and `loadstring`.

String Manipulation - `string`

Fully included.

Table Manipulation - `table`

Fully included.

Mathematical Functions - `math`

Fully included, using double-precision floating point internally.

Input and Output Facilities - `io`

Mostly included.

Not included

- `io.popen`
- `io.tmpfile`
- `file:setvbuf`

Operating System Facilities - `os`

Mostly included.

Not included

- `os.execute`
- `os.getenv`
- `os.tmpname`

Added

`ok = os.move(fromfile, destfile)`

Parameters

- `fromfile`: Filename of existing file
- `destfile`: Filename of target

Returns

- `ok`: True if successful

Description

Move a file

Debug Library - `debug`

NOT included.

`boot.lua`

If the file `/Lua/boot.lua` exists, the rt.buffer will execute this file before loading the application Lua file.

Warning: Be careful what you code you put in this file!

Lua Library Extensions

There is a large set of Lua library extensions embedded in the firmware. The libraries and their functions are constantly being improved and added to.

Many of the libraries have default behaviour that can be overridden using tables, variables, and function overrides.

Library `alm` : Alarm Library

The alarm library allows for tracking counts on 32 distinct alarms. The alarms can be updated with bitfields, or individually.

```
alm.clr(n)
alm.clr( {bitfield} )
```

Parameters

- `n`: Alarm number, 1-32
- `bitfield`: The bitfield number is placed in the first element of a table

Returns

None

Description

Clear the Counts.

```
aa, at = alm.get(n)
ab, at = alm.get( {bitfield} )
```

Parameters

- `n`: Alarm number, 1-32
- `bitfield`: The bitfield number is placed in the first element of a table

Returns

- `aa`: Alarm active (true if active)
- `ab`: Alarm bits that are active
- `at`: Alarm table of values

Description

Returns a table with the alarm parameters and counts for each alarm specified.

```
local ab, at = alm.get( {0x3} )
-- ab = 1
-- at = {1={co=2,cf=0,lo=2,lf=1,alm=true}, 2={co=0,cf=0,lo=2,lf=1} }
```

`alm.set(n, limiton, limitoff)` `alm.set( {bitfield}, limiton, limitoff)`

Parameters

- `n`: Alarm number, 1-32
- `limiton`: Number of sequential 'ons' before alarmed
- `limitoff`: Number of sequential 'offs' before inactive.
- `bitfield`: The bitfield number is placed in the first element of a table

Returns

None

Description

Set the parameters for an individual alarm 'n', or a range of alarms '{bitfield}'

```
alm.set( {0xff}, 10, 1) -- set alarms 1-8
alm.set( 10, 2, 2) -- set alarm 2 only
```

`txt=alm.txt(n)`

Parameters

- `n`: alarm number

Returns

- `txt`: text

Description

Return the text for the alarm, as set in c.alm_txt##

e.g. alm.txt(2) will return the value from *c.alm_txt02*

alm.upd(n, ison)
alm.upd({bitfield, bitvalues})

Parameters

- n: Alarm number, 1-32
- ison: Boolean indicating whether on or off
- bitfield: The bitfield number is placed in the first element of a table
- bitvalues: A bitfield representation of the alarms covered by bitfield, where 0=off, 1=on

Returns

None

Description

Update the alarm counters, and trigger the alarm callbacks as required.

```
alm.upd(2, true)
alm.upd( {0xff, 0x5a} )
```

Alarm Table `at`

As returned by alm.get(...)

- `.co` = Counts on
- `.cf` = Counts off
- `.lo` = Limit on
- `.lf` = Limit off
- `.alm` = true if in the alarmed state

Alarm Callbacks

There are two types of callbacks made from the library:

`function onAlm##(state)`

Parameters

- `state`: true if alarmed, false if inactive

Returns

None expected

Description

Callback for an individual alarm, where `##` is the two-digit decimal number of the alarm:

```
function onAlm02(state)
 if state
 then -- do something
 else -- do something else
 end
end
```

`function onAlm(n, state)`

Parameters

- `n`: Alarm number, 1-32
- `state`: true if alarmed, false if inactive

Returns

None expected

Description

A callback for all alarms. For each alarm, the onAlm## is called first, followed by the onAlm callback (allowing a mix of specific and general handling).

> You can use the function `alm.txt(n)` within the callback to pick out a name for the alarm.

Library app : Application Utilities

The app library provides some helpful functions related to the Lua app.

Memory efficient dynamic blocks

You can add special commented markers within the Lua app file, and dynamically retrieve these at runtime.

The tags are in the form:

```
--[[TAG]text--]]--

--[[TAG]
multi-line text
is here
--]]--
```

rv,⋯ = app.eval(tag) v2.00

Parameters

- tag: Tag name

Returns

- rv: Lua return values...

Description

Find the tag and evaluates, equivalent to `return app.txt(tag)`

`rv,… = app.exec(tag)` v2.00

Parameters

- `tag`: Tag name

Returns

- `rv`: Lua return values...

Description

Find the `tag` and execute, like Lua's `dostring`.

> 💡 Useful for dynamic function definitions.

`tb = app.lines(tag)` v2.00

Parameters

- `tag`: Tag name

Returns

- `tb`: Table of strings

Description

Find the `tag` and return as a table of strings.

> 💡 Useful for internationalisation strings.

`txt = app.txt(tag)` v2.00

Parameters

- `tag`: Tag name

Returns

- `txt`: Text block, if found.

Description

Find the `tag` and return as a raw string.

Library `bit` : Logical Bit Functions

The bit module provides useful bit operation functions.

Bit functions by Reuben Thomas <rrt@sc3d.org>
http://luaforge.net/projects/bitlib
(Extensions for bit-wise string handling and nibble/byte swap added by Scannex.)

```
v = bit.ar(a,b)
v = bit.arshift(a, b)
```

Parameters

- `a`: Value to shift
- `b`: Number of bits to shift

Returns

- `v`: Result

Description

Returns `a` right shifted arithmetically `b` places.

```
v = bit.band(v1, ⋯)
v = bit.band(txt)
```

Parameters

- `v1`: Value 1, etc
- `txt`: String

Returns

- `v`: Result

Description

Returns the bitwise AND of the `v's, or string

```
v = bit.bnot(a)
```

Parameters

- a: Value

Returns

- v: Result

Description

Returns one's complement of a.

```
v = bit.bor(w1, …)
v = bit.bor(string)
```

Parameters

- v1: Value 1, etc
- txt: String

Returns

- v: Result

Description

Returns the bitwise OR of the `v's, or string

```
v = bit.bxor(w1, …)
v = bit.bxor(string)
```

Parameters

- v1: Value 1, etc
- txt: String

Returns

- v: Result

Description

Returns the bitwise XOR of the `v's, or string

```
v = bit.cast(a)
```

Parameters

- `a`: Value

Returns

- `v`: Result

Description

Cast `a` to the internally used integer type.

```
v = bit.ls(a, b)
v = bit.lshift(a, b)
```

Parameters

- `a`: Value to shift
- `b`: Number of bits to shift

Returns

- `v`: Result

Description

Returns `a` left shifted `b` places.

```
v = bit.rs(a, b)
v = bit.rshift(a, b)
```

Parameters

- `a`: Value to shift
- `b`: Number of bits to shift

Returns

- `v`: Result

Description

Returns `a` right shifted `b` places.

`v = bit.swap16(a)`

Parameters

- `a`: Value

Returns

- `v`: Result

Description

Returns `a` with the LSB and MSB bytes exchanged.

`v = bit.swap32(a)`

Parameters

- `a`: Value

Returns

- `v`: Result

Description

Returns `a` with each of the four bytes reversed.

`v = bit.swap8(a)`

Parameters

- `a`: Value

Returns

- `v`: Result

Description

Returns `a` with the nibbles exchanged.

Library `cfg` : Configuration

The `cfg` Library provides helpful routines for reading and writing values into the configuration tree.

`v = cfg.get(key, [t], [d])` v1.00

Parameters

- `key`: Config key name
- `t`: (optional) table for replacing the # characters in key
- `d`: (optional) default value

Returns

- `v`: Result

Description

Reads a value from the config tree.

The `'c.'` prefix is not required, and will be ignored if supplied.

If `key` contains hash (#) characters then each hash is replaced with a value from the table `t`.

If `d` is supplied, and the config tree does not have the key named, then v=d.

`v` will be NIL, a string, or a number (i.e. its natural type).

```
local t = cfg.get('adc_c', 0.001)
local t = cfg.get('d_#_#', {1, 'm'}) -- returns value of c.d_1_m
```

`v = cfg.getb(key, [t], [d])` v1.00

Description

As cfg.get, but converts to boolean (or nil).

`v = cfg.getn(key, [t], [d])` v1.00

Description

As cfg.get, but converts to number (or nil).

`v = cfg.gets(key, [t], [d])` v1.00

Description

As cfg.get, but converts to string (or nil).

`ok = cfg.set(key, [t], v)` v1.00

Parameters

- `key`: Config key name
- `t`: (optional) table for replacing the # characters in key
- `v`: Value to set

Returns

- `ok`: True if succeeded

Description

Sets `c.key`=v

If `key` contains hash (#) characters, then each hash is replaced with a value from the table `t`.

```
cfg.set('d_#_#', {1, 'm'}, 'Testing')
cfg.set('adc_c', 1.234)
```

`v = cfg.table(key, [t, [d]])` v1.00

Parameters

- `key`: Config key name
- `t`: (optional) table for replacing the # characters in key
- `d`: (optional) default table

Returns

- `v`: Resulting table, or NIL

Description

Reads a config sub-tree as a Lua table.

If `key` contains hash (#) characters, then each hash is replaced with a value from the table `t`.

If `d` is supplied, then this forms the starting point of the `v` result. The table `d` is "deep-copied" to v, and then any values within the config sub-tree will replace the entries within `v` - effectively combining the two.

```
local t = cfg.table('') -- gets the whole tree
local t = cfg.table('adc', {}, {m=1,c=0})
```

Library `coro` : Coroutine Extensions

Lua includes a mechanism to run 'coroutines' - also known as cooperative multi-tasking. The rt.buffer firmware extends that library to make it simple to run scheduled tasks.

Any of the `onXXX` callback functions and the UI functions absolutely require that code runs very quickly. The `coro` library provides a very convenient way to queue longer running tasks and execute them in the context of the Lua Loop task.

Up to 16 coroutine functions can be queued at any one time.

Depending on how the functions are written, they will execute strictly sequentially, or can cooperatively multitask (if you use the `coro.w()` or `coro.yld()` functions).

ℹ Added in firmware v2.00

❗ Lua library functions that delay or wait, such as the serial routines or `rt.ms`, will block **all** coroutines from running during the delay. Once the Lua library function returns then it is possible to call `coro.w` or `coro.yld` to allow other coroutines to run.

```
ok = coro.add([dup], func, [user])
ok = coro.add([id], func, [user])
```

Parameters

- `dup`: True to allow multiple instances of func
- `id`: Identifier for the function
- `func`: Lua function
- `user`: User value (string, number, or table) that is passed to `func`

Returns

- `ok`: True if added to the queue

Description

Adds a Lua function, using the supplied id (or the function itself).

By default, the function will not be queued again if it is already pending or running. e.g. `coro.add(MyFuncName)` will only allow a single run of `MyFuncName` at a time.

The `id` is useful to associate the same function with different activities. So, the id could be a string, or a number. You could use the id for a process name, e.g. `smp`, or `prog` - so that one function can be queued for each particular activity.

The `user` value can be used to queue up parameters for the function. So, for example, the function may reconfigure a connected device, and the `user` parameter may be a table of values that are used to reconfigure. This approach allows a single function to perform variable actions.

```
v = coro.id()
```

Parameters

None

Returns

- `v`: RTOS task and Lua coroutine identifier

Description

Helpful function that can differentiate different Lua tasks and RTOS tasks.

The locking functions rely on this id value.

```
n = coro.kill()
n = coro.kill(id)
```

Parameters

- `id`: The specific `id` of the coroutine to kill.

Returns

- `n`: Number of coroutines that were terminated

Description

Abort all coroutines, or the specified routine.

`n = coro.n()`

Parameters

None

Returns

- `n`: Number of queued coroutines

Description

Return the number of coroutines currently queued.

`coro.yld()`
`coro.w(ms)`

Parameters

- `ms`: Milliseconds to wait

Returns

None

Description

Yields the current coroutine, allowing the next coroutine task to execute immediately. If `ms` is provided, the current coroutine will not resume until the delay has expired.

If there are no other pending coroutines, the `coro.yld` will return immediately and execution will continue.

> The actual delay may be *longer* than the `ms` value.

> `coro.yld` and `coro.w` are identical.

```
local onErr = coro.except(func, v)
local onKill = coro.except(func, v)
```

Parameters

- `func`: Lua function to call
- `v`: Parameter to pass

Returns

- `onErr`: Exception for error handling
- `onKill`: Exception for `coro.kill` on this coroutine

Description

Creates a special Lua exception handler that **must** be saved local.

The exception handler gives your code a chance to tidy up on exceptions.

If there is a Lua error, the `onErr` will be called after the firmware error handling. If `coro.kill` causes this routine to abort, then `onKill` will be called.

```
function OnMyExcept(cause, func, user)
  -- Cleanup
end

function MyCoro(v)
 local onErr = coro.except(OnMyExcept, v)
 local onKill = onErr
 -- do stuff
end

coro.add(MyCoro)
```

Library `crc` : CRC Calculations

CRC calculations for CRC-8 (c8), CRC-16 (c16), CRC-32 (c32), and CRC-Xmodem (xm), and custom CRC values.

```
cv = crc.c8(txt) / crc.c16(txt) / crc.c32(txt) / crc.xm(txt)
```

Parameters

- `txt`: String to CRC

Returns

- `cv`: CRC value

Description

Perform CRC-8, CRC-16, CRC-32, or CRC-Xmodem on a string.

Short-hand for using crc.new / crc.upd / crc.fin

```
local v = crc.c32('Testing')
```

```
ct = crc.new(size)
ct = crc.new(txt)
ct = crc.new(ci, ref, polyrev, xor)
```

Parameters

- `size`: Number of bits = 8, 16, 32
- `txt`: Name of crc, e.g. 'xmodem', 'crc32'
- `ci`: Initial CRC value
- `ref`: Boolean value. True to reflect the bits
- `polyrev`: Bit-reversed polynomial
- `xor`: Final XOR value

Returns

- `ct`: CRC table structure

Description

Initialise a new CRC object table.

```
--[[ Sample CRC Custom Values
CRC-32: crc.new(0xffffffff, false, 0xedb88320, 0xffffffff)
CRC-16/Modbus:crc.new(0xffff, false, 0xa001, 0)
CRC-16/SDI-12:crc.new(0, false, 0xa001, 0)
CRC-16: crc.new(0xffff, true, 0x8408, 0)
CRC-16/Xmodem:crc.new(0, true, 0x8408, 0)
CRC-16/Kermit:crc.new(0, false, 0x8408, 0)
CRC-16/DNP: crc.new(0, false, 0xa6bc, 0xffff)
]]--
```

`cv = ct:fin()`

Parameters

None

Returns

- `cv`: CRC value

Description

Finish CRC and return CRC value.

```
local c = crc.new('xmodem')
c:upd('Testing')
local v = c:fin()
```

`ct:upd(txt)` `ct:upd(fhandle, [length])`

Parameters

- `txt`: String to CRC
- `fhandle`: File handle
- `length`: Optional number of bytes (otherwise does all)

Returns

None

Description

Update CRC.

```
local c = crc.new(32)
c:upd('Testing')
```

Library `dpc` : Digital Pulse Counters

`pc1,pc2 = dpc.dis()`

Parameters

None

Returns

- `pc1`: Pulse count 1 before disabled
- `pc2`: Pulse count 2 before disabled

Description

Disable pulse counter input.

`l1, l2 = dpc.lvl()` v1.00

Parameters

None

Returns

- `l1`: True if Pulse 1 is at 0V/GND
- `l2`: True if Pulse 2 is at 0V/GND

Description

Return the instantaneous level of the digital pulse inputs.

`pc1,pc2 = dpc.rst()`

Parameters

None

Returns

- `pc1`: Pulse count 1 before reset
- `pc2`: Pulse count 2 before reset

Description

Reset digital pulse counts (and enables).

`pc1,pc2 = dpc.val()`

Parameters

None

Returns

- `pc1`: Pulse count 1
- `pc2`: Pulse count 2

Description

Query the pulse count values.

Library `evt` : Event Library

`evt.en(evts)`
`evt.dis(evts)`

Parameters

- `evts`: Event name strings

Returns

None

Description

Enable or disable event callbacks.

A group of the events are always enabled, and cannot be disabled.

```
evt.en('onDevRx,onPL1')
```

`evt.lsig()`

Parameters

None

Returns

None

Description

Signal the event loop.

It is important that the main Lua Loop Task is asleep for most of its life. The evt.lsig() can be called from an event callback function to wake up a sleeping loop process.

`ok = evt.lwait([timeout])`

Parameters

- `timeout`: Number of milliseconds to wait. If NIL, then waits forever.
- `ok`: True if signalled. False if timed-out

Description

Wait for the signal from the event callbacks.

It is important for power-consumption that the main Lua Loop Task is asleep for most of its life. The evt.lsig() can be called from an event callback function to wake up a sleeping loop process that has called evt.lwait()

`evt.sig(evts)`

Parameters

- `evts`: Event name strings

Returns

None

Description

Trigger a callback (for testing and development).

Event Callback Functions

Event callbacks are run in a separate Lua Events Task inside the rt.buffer. To make use of a callback, you must declare a function within the App, and enable the named function with evt.en(...)

The callbacks are passed three parameters:

```
function eventcallbackname(name, idx)
```

Parameters

- `name`: Event name, e.g. "onSec"
- `idx`: Event index number, 1-32

Returns

None expected

Event Callback Names

- For time-based callback events, use the job library.
- Always enabled
  - `onMag` magnet is closed (default is iot._onMag)
  - `onMagOff` magnet is released
  - `onConfig` configuration change occurred
  - `onEngCon` engineer port is connected or disconnected
  - `onDevCon` device port is connected or disconnected
  - `onUnload` called when the script is being unloaded v1.00
- Require enabling
  - `onEngCts` Engineer CTS line changed
  - `onDevCts` Device CTS line changed
  - `onEngRx` data is being received
  - `onDevRx` data is being received
  - `onPL1` digital pulse input 1 is low
  - `onPL2` digital pulse input 2 is low
  - `onPH1` digital pulse input 1 is high
  - `onPH2` digital pulse input 2 is high

> Note that some of the events (like the digital pulse events) will dramatically increase power consumption if used with high-frequency devices.

i.e. do not enable the pulse events unless they will occur relatively infrequently (e.g. door closure contacts, etc)

Event Callback Examples

```
function onMag()
 -- Magnet triggered!
 iot._onMag() -- call the original default
 -- Do some other things
end

function onDevCon(n,idx,v)
 if ser.dev():okr()
 then
 -- connected!
 else
 -- disconnected :-(
 end
end

evt.en('onMag,onDevCon') -- enable these two (onMag already on)
```

Library `fc` : File Controls

`name = fc.add(filename, ext1, [ext2…])`

Parameters

- `filename`: Base filename
- `ext1`: Extension to add
- `ext2`: …More extensions (if required)

Returns

- `name`: Full filename

Description

Add extension(s) to filename (dots not required)

`nb = fc.append(fhandle, filename, [txt])`

Parameters

- `fhandle`: File handle
- `filename`: Filename of existing file to append to fhandle
- `txt`: optional string to add after the file is appended

Returns

- `nb`: Number of bytes added to fhandle

Description

Writes the contents of filename to the filehandle (at the current position).

`nb = fc.cache([blocks, [mode]])` v1.00

Parameters

- `blocks`: Number of blocks (2112byte) cache to assign
- `mode`: Cache mode string:
  `'r'` = Read,
  `'w'` = Write-back,
  `'t'` = Write-Thru

Returns

- `nb`: Number of bytes allocated to the cache

Description

Sets up the file system cache.

> `'w'` is the mode usually required where data is lazy written to the volume.

The default is a single block of write-back cache to speed up writing.
i.e. `fc.cache(1, 'w')`

> ⚠ Memory is taken from the Lua memory area.

`fc.chkupd()` v1.00

Parameters

None

Returns

None

Description

Triggers a check on the config.txt and Lua App files.

`fc.clean([blocks, [sectors]])` v1.00

Parameters

- `blocks`: =2 (optional) Number of flash blocks to clean
- `sectors`: =0 (optional) Number of sectors

Returns

None

Description

Prepare the flash file system for faster writing. The call does internal housekeeping to free up the requested number of blocks/sectors.

> This is helpful when you expect a large file write to take place that must occur without stopping.

```
fc.clean(2, 0) -- make space for about 200k
```

`ok = fc.copy(fname, tname)` v1.00

Parameters

- `fname`: Source file name
- `tname`: Target file name

Returns

- `ok`: True if copied

Description

Copies a file.

`fc,fs = fc.del([compare], dirpattern, [isdir], [UTCfrom, [UTCto]], [function])`

Parameters

compare: dirpattern: isdir: UTCfrom: UTCto:

- `function`: Same structure as fc.find

Returns

- `fc`: filecount
- `fs`: filesize (if isdir==false)

Description

Deletes selected files from the directory.

`ext,name = fc.ext(filename)`

Parameters

- `filename`: Full filename string

Returns

- `ext`: Extension (without the dot)
- `name`: Filename

Description

Remove first extension from filename (without the dot).

n = fc.find([compare], dirpattern, [isdir], [UTCfrom, [UTCto]], [function])

Parameters

- compare: How to compare the files '=' first file that matches (default) '==' last file that matches '!' first file that does not match '!!' last file that does not match
- dirpattern: Filename and pattern, e.g. '/Send/*.txt'
- isdir: boolean. Set to true to find directories rather than files. (Files are default) UTCfrom:
- UTCto: Also adds file's modified time to comparison, such that:
  UTCfrom ≤ filemodtime < UTCto
- function: optional Lua function that's called for each found file (to allow for moving, table creation, etc)

Returns

- n: Filename of found file

Description

Finds the file or directory in directory that conforms to the wildcard pattern.

ok = fc.lv(fhandle, luavnames, [txt])

Parameters

- fhandle: File handle
- luavnames: Lua value names
- txt: optional string to append after the list

Returns

- ok: True if evaluated some Lua values

Description

Outputs the Lua value trees to the filehandle.

(Same syntax as the USB LV command)

If txt is assigned, also writes this afterwards.

```
fc.lv(fh, 'c,i', '\r\n')
```

`fn = fc.mdf(dir, name)`

Parameters

- `dir`: Directory path
- `name`: Filename

Returns

- `fn`: Full directory + filename

Description

Join directory + filename.

`ok,rc = fc.mkd(dir)`

Parameters

- `dir`: Directory to create

Returns

- `ok`: True if success
- `rc`: Internal result code

Description

Create a directory, and all sublevels if required.

`ok,rc = fc.rmd(dir)`

Parameters

- `dir`: Directory to remove

Returns

- `ok`: True if success
- `rc`: Internal result code

Description

Remove a directory and all contents (can take a long time to run if there are lots of contents).

`dir,name = fc.sdf(filename)`

Parameters

- `filename`: Full filename

Returns

- `dir`: Directory path
- `name`: Just filename and extensions

Description

Separate directory and filename.

`fc.sfwm(safe)` v1.00
`fc.sfwm(mode)`

Parameters

- `safe`: Boolean.
  true = safe and slow
  false = medium
- `mode`: Mode string. `'fast'`, `'med'`, `'safe'`

Returns

None

Description

Scannex debug facility to set the file volume s file write mode.

The default mode is safe .

Do not use this unless necessary! Using non-safe file system could corrupt data!

`fc.sync()` v1.00

Parameters

None

Returns

None

Description

Scannex debug facility to synchronise and flush the NAND file system.

`utc = fc.utc(filename)`

Parameters

- `filename`: Filename of existing file

Returns

- `utc`: Integer UTC file creation time.

Description

Returns the creation date of the filename or directory.

```
local s = utc.txt( fc.utc('/Config/config.txt') )
```

`ok = fc.wc(filename, pattern)`

Parameters

- `filename`: Filename to compare against
- `pattern`: Wildcard pattern

Returns

- `ok`: true if filename matches the pattern

Description

Does a wildcard compare.

Library `float` : Floating Point Utilities

The float library provides conversion between Lua's numbers and IEEE-754 binary representations. Both single precision (32-bit) and double precision (64-bit) numbers are supported, in integer and ASCII formats.

> ⚠ Be aware that converting from single to double, and vice-versa, can introduce apparent 'errors'. There are helpful articles on the Internet that highlight the technical reasons why this happens. Lua is double-precision (64-bit) internally.

`iv = float.d2v(fp)`

Parameters

- `fp`: Floating point that needs converting

Returns

- `iv`: Integer 64-bit IEEE-754

`ah = float.d2x(fp)`

Parameters

- `fp`: Floating point that needs converting

Returns

- `ah`: ASCII-Hex representation of 64-bit IEEE-754

Description

Convert from Lua number to 64-bit float

iv = float.f2v(fv)

Parameters

- `fv`: Floating point that needs converting

Returns

- `iv`: Integer 32-bit IEEE-754

Description

Convert from Lua number to 32-bit integer

ah = float.f2x(fv)

Parameters

- `fv`: Floating point that needs converting

Returns

- `ah`: ASCII-Hex representation of 32-bit IEEE-754

Description

Convert from Lua number to 32-bit ASCII-hex float

mpco = float.mpc(tx, ty) v2.00

Parameters

- `tx`: Table of x-values
- `ty`: Table of y-values

Returns

- `mpco`: Multi-Point-Conversion object

Description

Creates a new MPC object that can be called to perform a conversion.

See: [mpco:y]

Stores the `tx` and `ty` in a memory-efficient C structure in the Lua RAM area.

Use `float.rf` or `float.rd` to load raw IEEE values from a file.

`polyo = float.poly(m0,m1,m2⋯)` v2.00

Parameters

- `m0,m1,m2⋯`: polynomial multiplers

Returns

- `polyo`: Polynomial-Conversion object

Description

Creates a new polynomial object that can be called to perform a conversion. You can use any number of values.

e.g. `float.poly(5, 0.5, 0.1)` = `5 + 0.5x + 0.1x^2`

See: [polyo:y]

ℹ Stores the values in a memory-efficient C structure in the Lua RAM area.

`t = float.rf(fh, n)` v2.00
`t = float.rd(fh, n)` v2.00

Parameters

- `fh`: File handle
- `n`: Number of values to read

Returns

- `t`: Table of floating numbers

Description

Reads raw IEEE single float 4-byte (`rf`) or double float 8-byte (`rd') into a table.

```
fp = float.v2d(iv)
```

Parameters

- `iv`: Integer 64-bit IEEE-754 value

Returns

- `fp`: Floating point value

```
fp = float.v2f(iv)
```

Parameters

- `iv`: Integer 32-bit IEEE-754 value

Returns

- `fp`: Floating point value

Description

Convert from 32-bit integer to Lua number

```
n = float.wf(fh, t) v2.00
n = float.wd(fh, t) v2.00
```

Parameters

- `fh`: File handle
- `t`: Table of numbers

Returns

- `n`: Count of values written

Description

Writes raw IEEE single float 4-byte (`wf`) or double float 8-byte (`wd') into a file.

fp = float.x2d(ah)

Parameters

- ah: ASCII-Hex 64-bit string

Returns

- fp: Floating point value

Description

Convert from 64-bit ASCII-hex float to Lua number

fp = float.x2f(ah)

Parameters

- ah: ASCII-Hex 32-bit string

Returns

- fp: Floating point value

Description

Convert from 32-bit ASCII-hex float to Lua number

yv = mpco:y(xv) v2.00

Parameters

- mpco: Multi-Point-Conversion Object (returned from [float.mpc])
- xv: X-value to convert

Returns

- yv: Y-value conversion

Description

Finds the pair of X-values that xv is within, and performs linear slope conversion to find yv.

ℹ Returns nil if xv is out-of-bounds.

ℹ Uses the tx and ty tables passed to [float.mpc].

```
local m = float.mpc( {10,20,30}, {100,200,300} )
local y = m:y(15)  -- > 150
local y = m:y(1)   -- > nil
local y = m:y(301) -- > nil
```

yv = polyo:y(xv) v2.00

Parameters

- polyo: Polynomial-Conversion Object (returned from [float.poly])
- xv: X-value to convert

Returns

- yv: Y-value conversion

Description

Performs the n-order polynomial converion of xv

```
local p = float.poly(5, 1, 0.5, 0.01) -- 5 + x + 0.5x^2 + 0.01x^3
local y = p:y(123)  -- > 26301.17
```

Library hash : Hash Calculations

Performs MD5, SHA1, and SHA256 hash functions of text or file contents.

```
hv = hash.md5(txt)
hv = hash.sha1(txt)
hv = hash.sha256(txt)
```

Description

Calculate hash of direct string

```
ht = hash.new(method)
```

Parameters

- method: Which hash = 'sha1', 'sha224', 'sha256', 'md5'

Returns

- ht: Hash table structure

```
ha = hash.txt(hv, [gap])
```

Parameters

- hv: Hash Value, as returned from ht:fin()
- gap: optional gap string. Default is a colon ':'

Returns

- ha: ASCII representation

Description

Convert hash value into a human-readable format.

`hv = hash.val(ha)`

Parameters

- `ha`: ASCII representation of hash (non-hex digits are ignored)

Returns

- `hv`: Hash Value.

Description

Converts ASCII hex to string

`hv = ht:fin()`

Description

Produces hash value.

> 💡 Use hash.txt(hv) to convert to ASCII

`ht:upd(txt)`
`ht:upd(fileobject, [length])`

Description

Updates the hash with text or file contents

Hash Table Structure

The hash table is returned from a call to hash.new.

> ⚠ There should not be any real need to use the table directly in an app.

- `m` = Method (1, 5, 224, 256)
- `c` = Context for hash (opaque internal value)
- `b` =Number of bytes processed

```lua
local h = hash.new('sha1')
h:upd('Testing')
local v = h.b -- should be 7
```

Library `hydrins` : HydrINS Support

`res, txt = hydrins.cmd(cp, cmdtxt, [timeout])`

Parameters

- `cp`: Com Port Object
- `cmdtxt`: The command to send (# is not required)
- `timeout`: Timeout in milliseconds

Returns

- `res`: True if ACK'd by the HydrINS
- `txt`: Response string

Description

Send a command to HydrINS.

```
...
if hydrins.wake(dp, 10, 20)
then
 hydrins.cmd(dp, '086;966')
end
```

`ht = hydrins.parse(txt)`
`ht = hydrins.parse(txt, [prm])` v2.00

Parameters

- `txt`: The received HydrINS data. MUST include the 6-byte binary header. The "W" wakeup character is optional.
- `prm`: The #086 value. When present, parses HydrINS v1 format.

Returns

- `ht`: HydrINS table as described

Description

Parse a HydrINS line into a table.

`res = hydrins.run(cp)`

Parameters

- `cp`: Com Port Object

Returns

- `res`: True if ACK'd

Description

Issues the #028 command to put into run mode.

Short-hand for:

```
hydrins.run(cp, '028')
```

`ht = hydrins.rx(cp, [timeout])`
`ht = hydrins.rx(cp, [timeout], [prm])` v2.00

Parameters

- `cp`: Com Port Object (as returned by ser.dev())
- `timeout`: Optional timeout, in milliseconds
- `prm`: The #086 value. When present, parses HydrINS v1 format.

Returns

- `ht`: HydrINS table of values, or NIL if not received

Description

Read and parse a HydrINS record from the Com Port.

`res = hydrins.wake(cp, [delaysec, [trysec]])`

Parameters

- `cp`: Com Port Object
- `delaysec`: Number of seconds to delay before starting (default 0)
- `trysec`: Number of seconds to keep trying (default 130s)

Returns

- `res`: True if successfully woke up the HydrINS into command mode and received an ACK.

Description

Try and wake up the HydrINS.

You can make use of the ht.cycle value to work out a suitable combination of delaysec + trysec:

```
local ht = hydrins.rx(dp)
...
if ht
then
 if hydrins.wake(dp, ht.cycle-2, 4) -- 2s before next cycle
end
```

Hydrins Table `ht`

- `ht.ver` = Hydrins version (1, 2 or 2.1)
- `ht.params` = Parameters 16-bit field
- `ht.alarms` = Alarms 16-bit field
- `ht.cycle` = Cycle time, in seconds
- `ht.tests` = Self tests 16-bit field (for Hydrins 2.1)
- `ht.water` = True if in water
- `ht.air` = True if in air
- `ht.PULSES` = HydrINS pulse count value
- `ht.VPAV` / `.VPAVu`
- `ht.SPV` / `.SPVu`
- `ht.VMAV` / `.VAMVu`
- `ht.SMV` / `.SMVu`
- `ht.FAV` / `.FAVu`
- `ht.SVG` / `.SVGu`
- `ht.TOTp` / `.TOTpu`
- `ht.TOTm` / `.TOTmu`
- `ht.TOTn` / `.TOTnu`
  - For each, there is a value and optional units. i.e. `ht.TOTm` is the value, and `ht.TOTmu` is the unit string.
- `ht.B1p` = Battery 1 percentage
- `ht.B1v` = Battery 1 voltage (if present)
- `ht.B2p` = Battery 2 percentage (if present)
- `ht.B2v` = Battery 2 voltage (if present)
- `ht.BU` = Batteries used (if present)
- `ht.BF` = Batteries fitted (if present)
- `ht.TEMP` / `ht.TEMPu` = temperature & units (for Hydrins 2.1 if enabled)
- `ht.NAMUR` = (for Hydrins 2.1 if enabled)

Library `iot` : Internet Of Things

The iot library provides utilities for sending data via FTP, and FTPS to the central IOT server.

- FTP default port is 21;
- FTPS (**implicit** TLS/SSL) default port is 990.
- *You can override the port number with the standard :xxx in the URL.*

`txt = iot.abort([reasons])`

Parameters

- `reasons`: String containing text (as in iot.go)

Returns

- `txt`: String representation of reasons that are still outstanding.

Description

Cancel the modem requests.

Use with CAUTION. You may prevent the rt.buffer from ever connecting to the IoT server!

`iot.aup()`

Parameters

None

Returns

None

Description

Abort the User Process (if active).

ok = iot.cert(cafile, [certfile, keyfile]) v1.00
ok = iot.cert([use])

Parameters

- cafile: Local filename of the CA.pem file
- certfile: (optional) Local filename of the client certificate
- keyfile: (optional) Local filename of the matching client private key
- use: Passing 'false' will erase the CA, cert, and key

Returns

- ok: If successful

Description

Copies the local file to the modem for use as a CA certificate check, and optionally uploads the client certificate and key.

(Can only be called within the context of the modem task.)

```
iot.cert('/config/ca.pem')
iot.cert('/config/ca.pem', '/config/client.pem', '/config/client.key')
iot.cert(false) -- stop using CA/Cert/Key
```

iot.cho()

Parameters

None

Returns

None

Description

Clear hold off.

❗ Use this sparingly to avoid over-use of modem.

`ok = iot.diag([target], [opts])`

Parameters

- `target`: Optional target filename
- `opts`: String containing options of what is to be sent:
  `'info'`= Information tree
  `'config'`= Configuration tree
  `'app'` = Lua App source code

Returns

- `ok`: Whether ok

Description

Sends the diagnostics dump to the IoT server.

```
iot.diag() -- everything
iot.diag('my-file.diag')
iot.diag(_, 'config app') -- not info
```

`ok = iot.dorun()`

Parameters

None

Returns

- `ok`: Whether modem can still remain online

Description

Check whether we should still be online, or whether we are about to hangup.

`txt = iot.flg()`

Parameters

None

Returns

- `txt`: String containing the outstanding reasons for connecting to IoT

Description

Returns a string compatible with iot.go values.

`ht = iot.gho()`

Parameters

none

Returns

- `ht`: The hold-off time in seconds

Description

Return the current hold off time, in seconds

`iot.go([reasons])`

Parameters

- `reasons`: optional reasons for triggering IoT
  `'info'` = perform cellular survey v1.01
  `'data'` = send data
  `'update'` = perform update process
  `'time'` = synchronise time
  `'ntp'` = (same as 'time')
  `'user'` = perform user actions (like magnet)
  `'term'` = connect terminal OTA v1.10
  `'toa'` = connect terminal OTA v1.10
  `'test'` = test delivery
  `'sms'` = SMS transmission v1.10
  `'data time update'` = default options

Returns

None

Description

Trigger connect and send.

❗ The survey results are saved in `/Logs/cellinfo.txt`. This process may take between 2 and 5 minutes (perhaps longer if there is no SIM installed).

ℹ The modem may be in a 'hold-off' period, but will connect when that period expires.

`iot.hum([kill])`

Parameters

- `kill`: true to perform nasty, immediate, power kill.

Returns

None

Description

Hang up the modem. Without the 'kill' option, the modem handler will let all tasks close down gracefully.

With the `kill` option, the modem power is cut immediately, and it can take a few minutes for the modem handler to timeout and clean up.

`stop = iot.isf(name)`

Parameters

- `name`: Root name of the file to look for

Returns

- `stop`: True if there is a '.stop'

Description

Check whether there’s a request to stop from the IoT server.

If found, the file is deleted from the server.

```
local stp = iot.isf('mine') -- looks for 'mine.{i.rt_sn}.stop'
```

fc,fs = iot.send(srcptn, [options, [arch]])

Parameters

- srcptn: File path pattern (wildcarded) for choosing what to send
- options: optional overrides for the transfers
  '' = send plain
  'tfr' = use temp-file-rename process
  'gz' = use .gz compression
  'zlib' = use .zlib stream compression
  _ (NIL/not present) = send using tfr + gz
- arch: Whether to archive the file when sent (default = true)

Returns

- fc: Count of files sent
- fs: Number of bytes sent

Description

Transfer files from the rt.buffer directory area that match the filename pattern.

Copies according to source ASCII-sort name.

- MUST only be called within the context of the modem process, within the iotData callback function.

```
iot.send('/Send/*.mine.txt')
iot.send('/Send/*.more.dat', '') -- no compression, direct
```

`iot.set(uri)`

Parameters

- `uri`: Absolute or relative URL for the server

Returns

None

Description

Sets the address relative to the previous call to iot.set (i.e. appends this path to the previous full URL).

```
iot.set('ftp://user:pass@ftp.scannex.com:21/target')
-- or --
iot.set('/NewDirectory/Here')
```

Uses the scheme and server details from the previous call to iot.set

```
iot.set('Relative/Path/Here')
```

This function makes use of [rt.exp] to provide for Lua variable expansion:

```
-- include the Site Name and Serial Number:
iot.set('Relative/Path/{c.site_name}-{i.rt_sn}')
```

See [rt.exp] for details on the expansion mechanism.

`iot.sho(value)`

Parameters

- `value`: The hold off time required in seconds.

Returns

None

Description

Set modem hold off. Only applies if the current hold off is less that value.

Default Functions

function iot._test

Description

Default IoT Test function. Effectively:

```
function iot._data()
 iot.set(c.iot_data)
 iot.send('/Config/*.tst')
end
```

function iot._onJob11

Description

Default onJob11 function. Effectively:

```
function iot._onJob11()
 iot.go('update time')
end
```

function iot._onJob12

Description

Default onJob12 function. Effectively:

```
function iot._onJob12()
 smp.cut()
 iot.go('data')
end
```

function iot._data

Description

Default IoT Data function. Effectively:

```
function iot._data()
 iot.set(c.iot_data)
 iot.send('/Send/*')
end
```

function iot._user

Description

Default IoT User function.

function iot._onMag

Description

Default Magnet event handler. Effectively:

```
function iot._onMag()
 iot.go('user update')
 iot.cho()
end
```

IoT Callbacks

function iotData()

Description

Called when the rt.buffer should send data to the central server.

If this function is not defined, then all files within the /Send directory are sent.

- Within this callback you should make use of iot.set and iot.send (and not much else).

```
function iotData()
 iot.send('/Send/*.dat')
 iot.set('Another/Path')
 iot.send('/Send/*.csv')
end
```

function iotTest()

Description

Called when the rt.buffer should send test file(s) to the central server.

If this function is not defined, then all *.tst files in /Config directory are sent, and not archived.

function iotUpd()

Description

Called within the context of the Update mechanism.

This Lua callback occurs *after* the firmware has performed its processes.

function iotUser()

Description

Called when the rt.buffer should perform the user interaction (e.g. when the magnet is triggered).

The default function repeatedly sends a diagnostic dump to 'user.SERIAL.txt' in the c.iot_user directory (which is the c.iot_url by default). It checks for a 'user.SERIAL.stop' file - if this exists it will delete the file on the server and quit the loop. Otherwise, it will pause 15s, and loop.

The pause can be set with c.user_ins, and the total loop time set with *c.user_onm*.

Something like this:

```
function iotUser()
 local et = (c.user_onm or 2) * 60 + i.rt_alv
 local iv = (c.user_ins or 15)
 while (i.rt_alv < et) and iot.dorun()
 do
  iot.diag( rt.exp('user.{i.rt_sn}.txt'), 'config info' )
  if iot.isf('user') then break end
  rt.ms(1000 * iv)
 end
end
```

Library job : Job Scheduler

The job scheduler provides an efficient mechanism to handle repeat events like sampling, triggering delivery etc.

There are 12 jobs, and each job can have up to 8 time slots (so that different times of the day, and/or days of the week, can have different frequencies).

When the job needs running, the Lua Event Task will execute onJob#. e.g. Job 1 will execute the Lua callback function `onJob1`.

`Job12` is reserved for the Data Delivery schedule, and defaults to 23:00:00 each day.

`Job11` is reserved for the Update mechanism schedule, and defaults to 23:00:00 each day.

Job Schedule Strings

The rt.buffer has a flexible job scheduler that is used throughout the firmware. There are up to twelve jobs that can be used for different purposes (e.g. deliver data; contact update server). Each job can have eight different time slots (e.g. for different times throughout the week).

A schedule is programmed with a string that has the following structure:

- The primary value is the interval (default is 24 hours)
    - `00:00:15` = every 15 seconds
- The `@` specifies a time range (default start time is 00:00:00, and default end time is 24:00:00)
    - `@23:00` = at (or from) 11 pm
    - `@6:00-23:30` = between 6:00am (inclusive) and 11:30pm.
- The `#` specifies the day bit-field
    - Where: 1 = Sunday, 2 = Monday, 4 = Tuesday, 8 = Wednesday, 16 = Thursday, 32 = Friday, 64 = Saturday
    - `#62` = weekdays
    - `#65` = weekends
    - `#d` = weekdays, Mon-Fri
    - `#e` = weekends, Sat+Sun
    - `#a` = all days (#127)
- A comma (,) separates the entries

The end time is *exclusive*. So, for example, `2:00@12:00-18:00` will fire at 12:00, 14:00, and 16:00 only. This method allows for clarity when changing the interval, say to 0:00:01.

Schedule Examples

- `00:00:30@8-18, 00:05:00`
  - every 30s between 8am and 6pm, and 5 minutes outside those times
- `00:00:30@8-18#2,00:00:45@8-18#60,00:05:00`
  - every 30s between 8am and 6pm on Monday; every 45s between 8am and 6pm on Tue-Fri, and 5 minutes otherwise.
- `4@8-18,12`
  - every 4 hours between 8am and 8pm, and every 12hrs otherwise (00:00:00 & 12:00:00)
- `@12:30`
  - an event that occurs each day at 12:30pm

Scheduled Jobs

`job.set(jobn, txt, [variance])`

Parameters

- `jobn`: Job number, 1-12
- `txt`: The job schedule string
- `variance`: Set to true to apply delivery variance to the schedule.

Returns

None

Description

Sets the parameters for a job. Not all parameters are required for an entry, and you may have up to 8 time slots defined for each job. Spaces are optional, as are leading zeroes.

`txt,var = job.txt(jobn)`

Parameters

- `jobn`: Job number, 1-12

Returns

- `txt`: Job schedule string (long-hand format)
- `var`: Whether variance applied

Description

Returns the textual representation and whether variance is applied for the job.

`var = job.var(jobn, [variance])`

Parameters

- `jobn`: Job number, 1-12
- `variance`: If present, sets the variance flag for the job

Returns

- `var`: Whether variance is applied on this schedule.

Description

Sets or queries the variance flag for a job.

Scheduled Job Callbacks

function onJob##(jobn, utc, utcj)

Parameters

- jobn: The job number, 1-12
- utc: The actual UTC time value. Note that this may be slightly behind the current UTC time if there has been a backlog of jobs.
- utcj: The UTC the job sees (if variance is applied)

Returns

None

Description

This is the function prototype of the job callback, i.e. for onJob1 etc.

The function name should match the job number:

```
function onJob1(n, u, j)
 -- n = 1
 -- u + j = time
 -- do stuff here...
end
```

However, with Lua, the following is also possible:

```
function Foo(n,u,j)
 -- stuff
end

onJob1 = Foo
onJob2 = Foo
```

UTC Based Jobs

The UTC based jobs are useful for one-time events that need to occur at a certain time, such as triggering a rendezvous to base at a certain time.

There are 16 UTC slots that can be used.

`job.utc(jobu, utc)`

Parameters

- `jobu`: Job number, 1-16
- `utc`: Time the job should run

Returns

None

Description

Sets the Job UTC slot to the UTC time provided

Calls the function defined for the job number:

```
onJobU##( ) ①
```

① where `'##'` is the jobu number

```
function onJobU5()
-- do something
end

job.utc(5, utc.cvt('2017-09-01 16:00'))
```

jobu = job.utc(utc, function)

Parameters

- `utc`: Time the job should run
- `function`: Lua function that is called at utc

Returns

- `jobu`: The job number assigned, or NIL if no free jobs

Description

Find a spare UTC job slot, and glue the supplied function to the onJobUxx event

e.g. Assuming slot 5 is free, the function will be linked to onJobU5. On execution, the onJobU5 function is unlinked, and therefore the function originally provided may be garbage-collected (i.e. if it was an unnamed function).

```
job.utc(os.time()+120, function() --[[ do stuff ]]-- end)
job.utc(os.time()+120, MySpecialFunction)
```

utc,tmp = job.utc(jobu)

Parameters

- `jobu`: Job number, 1-16

Returns

- `utc`: The time the job is scheduled for, or NIL if blank.
- `tmp`: True if the job is a temporary one (see [job.utc](utc, function))

Description

Query Job UTC slot number.

UTC Based Job Callbacks

function onJobU##()

Parameters

None

Returns

None

Description

Called when the UTC job runs.

e.g.

```
function onJobU4()
 -- do something
end
```

The name of the function must match the UTC job number, 1-16

Library kvc : Key Value Container

The kvc library can be used to save Lua RAM for sets of values. It uses the same internals as the c configuration tree.

nb = kvco:_size v2.00

Parameters

None

Returns

- nb: Number of bytes.

Description

The total number of bytes in the data area of the kvco object.

> ℹ Not a function, but a pseudo variable.

nb = kvco:_free v2.00

Parameters

None

Returns

- nb: Number of bytes.

Description

Returns the number of bytes free in the data area of the kvco object.

> ℹ Not a function, but a pseudo variable.

ni = kvco:_freei v2.00

Parameters

None

Returns

- ni: Number of index entries free

Description

Returns the number of index entries free in the kvco object.

> Not a function, but a pseudo variable.

nb = kvco:_sizei v2.00

Parameters

None

Returns

- nb: Number of bytes.

Description

Returns the number of bytes used by the index area of the kvco object.

> Not a function, but a pseudo variable.

ok = kvco:_load(filename,[clear]) v2.00

Parameters

- filename: Name of the text file to load from
- clear: Whether to clear the kvco before loading (default=true)

Returns

- ok: If successful

Description

Loads the file into the kvco object.

ok = kvco:_save(filename) v2.00

Parameters

- filename: Name of the text file to save to

Returns

- ok: If successful

Description

Saves the contents of the kvco object into the text file.

kvco = util.kvc(nb,ni) v2.00

Parameters

- nb: Number of bytes for the data
- ni: Number of index entries

Returns

- kvco: KVC object

Description

Return a KVC object. RAM is taken from the Lua memory for the KVC object, the data area, and the index block.

RAM will be recovered during garbage collection when the kvco object becomes unused.

```lua
local k = util.kvc(2048,64)
k.my_value=1.234
k.my_setting='Test'
```

Library modbus : MODBUS functions

The MODBUS library provides raw frame transmission and reception, as well as a higher-level command processor that can frame and de-frame data.

Only supports MODBUS RTU on serial - either RS-232 or RS-485 (with an adapter).

rt,txt = modbus.cmd(cp, adr, fnc, [tfmt, ttab, [rfmt, ms]])

Parameters

- cp: COM Port object
- adr: Slave address
- fnc: Function number
- tfmt: Transmit format string
- ttab: Transmit data table
- rfmt: Receive format string
- ms: Receive time out

Returns

- rt: Table of values (as specified by rfmt)
- txt: Remaining string (if any)

Description

Send a MODBUS RTU request, and read the reply.

The indexed table 'ttab' is used, along with tfmt, to create a data string that is passed to modbus.tx, The reply formatting is optional.

For example to write to register 1000:

```
modbus.cmd(dp, 1, 16, 'WWBW', {1000, 1, 2, 7878})
-- register 1000 will have 2 bytes written = 7878
```

Modbus.rx is called, and any reply is pulled apart using the rfmt specifiers to create a list of results.

For example, the Read Holding Registers (0x03) function will return a byte-count, followed by n × 16-bit values (rfmt = 'BWWW' for 3 values):

Use modbus.rhr for Read Holding Registers. This example provided for instruction only.

```
t = modbus.cmd(dp, 1, 3, 'WW', {2, 3}, 'BWWW')
if t
then
 -- t[1] = Number of bytes
 -- t[2] = register 0002
 -- t[3] = register 0003
 -- t[4] = register 0004
 reg1, reg2, reg3 = unpack(t)
end
```

Note the use of Lua's unpack(t) to turn a table into a list of values.

`tab = modbus.fs2t(tfmt, txt)` v1.00

Parameters

- `tfmt`: Format string
- `txt`: Binary block of ModBus data to unpack

Returns

- `tab`: Lua table of values

Description

Utility function (used by modbus.cmd etc) to unpack a binary block received by ModBus and convert to a table of values.

`txt = modbus.ft2s(tfmt, ttab)` v1.00

Parameters

- `tfmt`: Format string
- `ttab`: Table of values to pack

Returns

- `txt`: ModBus packed block to send

Description

Utility function (used by modbus.cmd etc) to pack a set of values into a binary string block ready to send.

`modbus.rdi(cp, adr, fc, nc, [ms])` v1.30

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `fc`: First coil to read
- `nc`: Number of coils to read
- `ms`: Receive time out

Returns

- `tab`: Table of booleans (one for each input)
- `err`: ModBus error

Description

Read discrete inputs (function 02)

`tab = modbus.rhr(cp, adr, regs, regc, [rfmt, [ms]])` v1.00

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `regs`: First register to read
- `regc`: Count of registers to read
- `rfmt`: Receive format string
- `ms`: Receive time out

Returns

- `tab`: Table of values

Description

Wrapper function for modbus.cmd that performs a ModBus **Read Holding Registers** (3_{dec} / 0x03) function.

```
local t = modbus.rhr(dp,1,28,6,'W+') -- read DVP readings
```

`tab = modbus.rir(cp, adr, regs, regc, [rfmt, [ms]])` v1.30

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `regs`: First register to read
- `regc`: Count of registers to read
- `rfmt`: Receive format string
- `ms`: Receive time out

Returns

- `tab`: Table of values

Description

Read Input Registers (function 04).

ℹ Same format as `modbus.rhr` function 03.

`tab, err = modbus.rmc(cp, adr, fc, nc, [ms])` v1.30

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `fc`: First coil to read
- `nc`: Number of coils to read
- `ms`: Receive time out

Returns

- `tab`: Table of booleans (one for each coil)
- `err`: ModBus error

Description

Read multiple coils (function 01)

`dt,err = modbus.rsid(cp, adr, fmt, [ms])` v1.30

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `fmt`: In format string
- `ms`: Receive time out

Returns

- `dt`: Data table of results (or string if `fmt` not defined)
- `err`: ModBus error

Description

Read Server ID (function 17/0x11)

`ok,adr,fnc,data = modbus.rx(cp, [ms])`

Parameters

- `cp`: COM Port object
- `ms`: Optional timeout (default 250ms)

Returns

- `ok`: Whether the frame response is CRC'd ok
- `adr`: Address sent by the slave
- `fnc`: Function number sent by the slave
- `data`: String of remaining data (not including the CRC)

Description

Receive a MODBUS RTU frame.

`modbus.set(cp, bps, [rs485])`

Parameters

- `cp`: COM Port object, e.g. from ser.dev()
- `bps`: Baud and Protocol string
- `rs485`: true or '485' = RS-485, false or '232'=RS-232

Returns

None

Description

Sets the baud-rate and protocol for the MODBUS.

```
local dp=ser.dev()
modbus.set(dp, '19200e8')
```

`modbus.tx(cp, adr, fnc, d1, d2  )`
`modbus.tx(cp, adr, fnc, str)`

Parameters

- `cp`: COM Port object
- `adr`: Slave address 1-127
- `fnc`: Function number
- `d1,d2`: Data bytes
- `str`: String

Returns

None

Description

Sends a CRC’d MODBUS RTU frame on the COM port 'cp'.

```
modbus.tx(dp, 1,3, 0,1, 0,2) -- read two registers from reg 0001
```

`tab,txt = modbus.wmr(cp, adr, regs, tfmt, ttab, [ms])`

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `regs`: First register to read
- `tfmt`: Transmit format string
- `ttab`: Transmit data table
- `ms`: Receive time out

Returns

- `tab`: Table of values returned (should be just start register and count)
- `txt`: Remaining string (if any)

Description

Wrapper function for modbus.cmd that performs a ModBus **Write Multiple Registers** (16_{dec} / 0x10) function.

```
modbus.wmr(dp,2,19,'W',{5}) -- sets OverFLO average length=5
```

`rt,txt=modbus.wsr(cp, adr, rno, rv, [ms])` v1.30

Parameters

- `cp`: COM Port object
- `adr`: Slave address
- `rno`: Register Number
- `rv`: Register Value
- `ms`: Receive time out

Returns

- `rt`: Table of values (like `modbus.cmd`)
- `txt`: Remaining text, if any (like `modbus.cmd`)

Description

Write Single Register (function 06)

modbus.cmd `tfmt` & `rfmt` Format Strings

The format strings tell the command processor how to package data, and how to parse it.

- `B` = 8-bit unsigned byte
- `b` = 8-bit signed byte v1.30
- `C` = 8-bit character v1.30
- `W` = 16-bit unsigned word
- `w` = 16-bit signed byte v1.30
- `D` = 32-bit double unsigned word v1.00
- `d` = 32-bit double signed word v1.30
- `F` = 32-bit IEEE-754 floating point value v1.00
- `S` = string
- `SN` = string with length 'N' v1.30
- `+` = keep using the last parameter until no more data (only applicable at the end of the string) v1.00

Library `nmea` : NMEA Functions

The NMEA library provides firmware functions that convert from ASCII to Lua table, and from Lua table to ASCII.

`tab, special, ok = nmea.tab(txt)`

Parameters

- `txt`: ASCII NMEA line

Returns

- `tab`: Lua table of values
- `special`: True if an AIS '!' record
- `ok`: True if the checksum is good

Description

Convert a line of NMEA data to a table.

```lua
local tb = nmea.tab('$GPAAM,A,A,0.10,N,WPTNME*32')

--[[
 tb[1] = 'GPAAM'
 tb[2] = 'A'
 tb[3] = 'A'
 tb[4] = 0.10
 tb[5] = 'N'
 tb[6] = 'WPTNME'
]]--
```

`txt = nmea.txt(tab, [special])`

Parameters

- `tab`: Table of values
- `special`: True if this is an AIS '!' record

Returns

- `txt`: ASCII output, with checksum and CRLF

Description

Convert Lua table to NMEA string.

```lua
local tx = nmea.txt( {'PQRS', 12.34, 'TEST'} )
```

Library `out` : Digital Outputs

The digital output library allows direct control of the four logic outputs on the expansion header HD3.

Additional plug-in PCBs can provide for signalling open-collector outputs, high-current drive outputs, etc - depending on the application.

`out.init([def])` v1.10

Parameters

- `def`: Default values

Returns

None

Description

Call this once in the App to initialise the outputs.

If included, 'def' will set the startup state of the outputs. The default value is 0.

`out.off(n)` v1.10

Parameters

- `n`: Which output to turn off (1-4)

Returns

None

Turn OFF a selected output.

`out.on(n)` v1.10

Parameters

- `n`: Which output to turn on (1-4)

Returns

None

Turn ON a selected output.

`mp,dp,ep = out.pwo()` v2.00
`out.pwo(v)` v2.00
`out.pwo(n,en)` v2.00

Parameters

- `v`: Bitfield to set both outputs (D0=main, D1=daughter, D2=engineer)
- `n`: Which output to turn off (1=main,2=daughter,4=engineer)
- `en`: True to enable the output

Returns

- `mp`: Mainboard Power status
- `dp`: Daughterboard Power status
- `ep`: Engineer Power status

Description

Either queries the two power outputs, or controls the power outputs.

Mainboard power output is 3.6V

Daughterboard power output is 12V with the RS485/switchout convertor PCB.

Engineer power output is 3.3V (with a nominal 0.3V Schottky diode drop), but will rise to 5V when the USB is connected (because it's the same pin!).

Overrides the cp:pwo() function! Use *either* out.pwo or cp:pwo

`out.set(v)` v1.10
`out.set(n, ison)`

Parameters

- `v`: Binary value to output (0-15)
- `n`: Which output to change (1-4)
- `ison`: Boolean value, true/false

Returns

None

Description

Set all the outputs, or control an individual output.

```
out.set(2, true) -- turn output #2 ON
out.set(1, false) -- turn output #1 off
```

Library `pwr` : Power Control

`pwr.apo(n)`

Parameters

- `n`: 0 = Power off
  1 = Power on

Returns

None

Description

Control power for power outputs for ADC reference.

> ! The rt.adc command may override the ADC reference power output.

`pwr.boot()`

Description

Reboot the rt.buffer.

`pc = pwr.dpc([mA, [V]])` v1.00

Parameters

- `mA`: Current in milliamps
- `V`: Optional voltage measured at (3.6V default)

Returns

- `pc`: Power consumption in milli-coulombs (@3.6V)

Description

Set or get the Device port power consumption.

To get the current value, pass no parameters.

`pc = pwr.epc([mA, [V]])` v1.00

Parameters

- `mA`: Current in milliamps
- `V`: Optional voltage measured at (3.3V default)

Returns

- `pc`: Power consumption in milli-coulombs (@3.3V)

Description

Set or get the Engineer port power consumption.

To get the current value, pass no parameters.

`sa,sl,cr,cw = pwr.gpm()`

Parameters

None

Returns

- `sa`: Seconds alive
- `sl`: Seconds in ultra-low power
- `cr`: CPU cycles running
- `cw`: CPU cycles waiting

Description

Get power manager metrics. Can be used to calculate power consumption:

- Effective CPU freq.cf = (cr + cw) ķ (sa - sl)
- CPU run timert = sa x cr / (cr + cw)
- Idle percentageip = (sa - rt) / sa x 100

`pwr.off()`

Description

Enter deep sleep / off mode.

Wakes up on USB or Engineer Serial port re-connection.

`pt,cpu = pwr.rsn()`

Parameters

None

Returns

- `pt`: Table of {"section"=count} indicating which power functions are in use.
- `cpu`: CPU frequency, in Hz

`bc = pwr.sbc([ac, [v]])`

Parameters

- `ac`: Battery capacity <200 = A/hr
  >200 = Coulombs
- `v`: Voltage of battery (default is 7.2V)

Returns

- `bc`: Battery coulombs

Description

Set battery capacity for power consumption estimation.

To read the capacity, pass no parameters

```
pwr.sbc(26, 7.2) -- 26Ahr @ 7.2V = 93600C
pwr.sbc(5, 6) -- 5Ahr @ 6V = 15000C (@7.2V)
local c = pwr.sbc() -- read the value
```

`pwr.scu(c)` v1.00

Parameters

- `c`: Coulombs used

Returns

None

Description

Sets the number of coulombs used. Useful if replacing with a half-full battery.

`mv,sv,ws,ta = pwr.tmv()` v2.00
 `m1,s1,m2,s2,mE,sE,ta = pwr.tmv(true)` v2.00

Parameters

None

Returns

- `mv`: Minimum voltage (mV)
- `sv`: Start voltage (mV)
- `ws`: Which source. `1`, `2`, or `E`
- `m1,s1`: Minimum and start for Battery 1
- `m2,s2`: Minimum and start for Battery 2
- `mE,sE`: Minimum and start for External power
- `ta`: Time alive of measurement

Description

Returns the Transient Minimum Voltage.

> ℹ This value is obtained each time the modem is powered up. If no measurement has happened, `pwr.tmv` will return `0,0,'?'`

`pwr.work(ms)`

Parameters

- `ms`: Time to perform hard work (0-2000) milliseconds.
  0 = back to normal (low) power

Returns

None

Description

Runs at 96MHz for a time, before falling back to low power, 12MHz, running.

Development Functions

These functions are for internal testing at Scannex, and are not guaranteed to remain constant between firmware versions.

```
pwr._bv(boost, unboost)
pwr._bv() v1.03
```

Description

Set the boost millisecond timers.

Use boost=0 to disable auto-boost Pass no parameters to set the defaults.

```
pwr._iv(min, max)
pwr._iv() v1.03
```

Description

Set the idle timers for ultra-low power mode.

Pass no parameters to set the defaults.

Library `rt` : rt.buffer Utility Functions

`rt.print(…)` v1.30
`print(…)` v1.30

Parameters

(As for Lua's print function)

Description

Print to the USB terminal, the TOA terminal, or USB log. TIP: This is useful for debugging Lua functions when you execute the function from the USB terminal with `lx`.

`uv,rs = rt.adc([phases], [m, [c]])`

Parameters

- `phases`: Choice of ADC phase (to save power).
  `''` = do all phases (takes about 170ms)
  `'L'` = lock and power up
  `'l'` = lock, but don't change power
  `'S'` = Start fresh sample
  `'s'` = Start sample
  `'R'` = Read sample
  `'U'` = Un-power and unlock
  `'u'` = Unlock only
- `m`: Slope for the ADC value
- `c`: Offset for the ADC value

Returns

- `uv`: Microvolts.
  Range ±0-1500000uV (±0 to 1.5V)
  (3.0V is the reference voltage)
- `rs`: Raw Sample Value

Description

Get pressure reading in microvolts, optionally calculating *y=mx+c*

ℹ The ADC is LTC2485, 24-bit delta-sigma, that has V-REF = 3.0V.

```
local p = rt.adc('LSRU') -- does the default
local p = rt.adc('lSRU') -- measures without powering up
local p = rt.adc(0.001, 500) -- apply m & c
```

`i. = rt.atom(function, args  )`

Parameters

- `function`: Lua function to execute
- `args`: Arguments that are passed to 'function'

Returns

Varies (depending on the function)

Description

Execute the function atomically. Syntactically the same as pcall, but wraps with a mutex so that data writes and file renames can occur atomically.

`str = rt.bin(txt)` v1.00

Parameters

- `txt`: Human-readable ASCII Hex to convert to binary string. Can include separators between hex values

Returns

- `str`: Binary string

Description

Convert a string of ASCII-hex into a binary represenation.

`b = rt.bool(v, [dflt])` v1.01

Parameters

- `v`: Value to check
- `dflt`: (optional) default value

Returns

- `b`: Boolean result

Description

Returns true or false based on the value of v.

- If v is a Lua boolean type, then b = v.
- If v is a Lua number, then b = (v != 0)
- If v is a Lua string, then a non-case-sensitive comparison occurs:
    - b = true if v is 'y', 't', or starts with "true" or "yes"
    - b = false if v is 'n', 'f', or starts with "false" or "no"
    - b = dflt or false if v has any other value.

- If v is NIL then b = dflt or false

`i. = rt.call(function, args   )`

Description

Non-atomic version of rt.atom

The same as Lua's pcall, just presented under the rt group.

`v = rt.cyc()` v1.03

Parameters

None

Returns

- `v`: Cortex cycle counter

Description

Returns the 32-bit CPU cycle counter.

`v = rt.dct(t)` v1.00

Parameters

- `t`: Table to deep copy

Returns

- `v`: Resulting copy of table

Description

Deep-copies a Lua table. Any changes made to `v` will therefore be independent of the original table `t`.

ℹ using the Lua standard   v=t   will only result in a single table in memory. Any changes to   v   will affect   t   as well. That   s why a   deep copy   is needed to produce a replica of the original table that is independent.

Only the Lua types: LUA_TNIL, LUA_TBOOLEAN, LUA_TNUMBER, LUA_TSTRING, and LUA_TLIGHTUSERDATA are handled.

`ok = rt.dorun()`

Parameters

None

Returns

- `ok`: True if still keep running.

Description

Check whether Lua can still run in this task.

> Only useful within the context of Lua Loop Task and Modem Task. (i.e. do not use within the Lua Event Task - it will time out!)

`txt = rt.exp(txt)`

Parameters

- `txt`: String to expand

Returns

- `txt`: Expanded string

Description

Expand string using {..} to evaluate Lua values.

This function forms a key part of the iot.set method.

```lua
local s = rt.exp('Data-{c.site_name}-{i.rt_sn}.csv')
```

Because the text within the {..} is evaluated and executed within Lua, you can even use expressions:

```lua
local s = rt.exp('Data-{i.rt_sn * 12345 + 34.56}')
```

`txt = rt.hex(str, [gap])` v1.00

Parameters

- `str`: Binary string to convert
- `gap`: (optional) Gap character

Returns

- `txt`: Human readable ASCII hex string

Description

Convert a binary string into human readable ASCII hex.

`rt.led(en)` v1.00

Parameters

- `en`: True to enable the 5/4 LED pulse

Returns

None

Description

Scannex debug facility to flash LED in the 5/4 pulse sequence.

`rt.log(module, txt)`

Parameters

- `module`: Module string
- `txt`: Text to log

Returns

None

Description

Log to the /Logs/system.log file.

`ms,ma,mr = rt.mag([ms])` v2.00.0144

Parameters

- `ms`: Time the magnet has to be active (milliseconds)

Returns

- `ms`: Time the magnet has to be active
- `ma`: True if the magnet is considered on (i.e. onMag has been called)
- `mr`: True if the magnet IO line is active (i.e. the pre-timer value)

Description

Controls the delay on the magnet before `onMag` is called

`res = rt.ms(value)`
`ms = rt.ms()` v1.00

Parameters

- `value`: Delay time in milliseconds

Returns

- `res`: Reason - 0=OK, 1=TaskTimeOut, 2=LuaAbort
- `ms`: System millisecond counter

Description

Delay number of milliseconds. The value is limited to half the Lua timeout value for the calling task.

Or, with no parameters, returns the system time in milliseconds.

The actual time will be in the range of 'value-1 < actual ⇐ value'. For small delays use `rt.us`

`fv = rt.msf(offset)` v2.00

Parameters

- `offset`: Offset time in milliseconds

Returns

- `fv`: Future value, in milliseconds

Description

Calculate an end time. Use with `rt.msx` to determine when expired.

`ok = rt.msx(fv)` v2.00

Parameters

- `fv`: Future value - the result from `rt.msf`

Returns

- `ok`: True if expired

Description

Determine when a timeout has expired.

`b1,b2,ext = rt.mv([id, [smp]])`

Parameters

- `id`: (Optional) ID. 0 = all three values
  1 = Battery 1
  2 = Battery 2
  3 = External
- `smp`: (Optional) Set to false to prevent sampling and just read the last reading.

Returns

- `b1`: Batt1 Voltage (mV)
- `b2`: Batt2 Voltage (mV)
- `ext`: External Voltage (mV)

Description

Get millivolt readings

`rt.setblf(filename)` v1.20

Parameters

- `filename`: name of file to be loaded into the BLF area

Description

Load a file into the BLF area, ready for the boot-loader to burn. Use `pwr.boot()` to reboot in code.

`c = rt.temp()` v1.00

Parameters

None

Returns

- `c`: Temperature in degrees Celcius

Description

Returns the temperature from the LTC2485. Note, this can take 160ms to sample.

`rt.us(value)` v2.00

Parameters

- `value`: Delay time in microseconds (1 to 5000)

Returns

None

Description

Delay number of microseconds.

Library sdi12 : SDI-12 Protocol

The SDI-12 protocol requires a hardware adapter (either internal or external) to provide the SDI-12 signal levels.

The library provides a set of low-level protocol commands, as well as a pair of high-level function (sdi12.go and sdi12.done) that handle the complete protocol as a state-machine.

Ensure you call sdi12.set to setup the Comport before calling other SDI-12 library calls.

Example:

```
dp = ser.dev()
sdi12.set(dp)
dp:pwo(true)

VRS = {{'2I'}, {'2CC'} } -- Get ID and measurements

function QueryExample()
 sdi12.gp(dp, VRS)
 while not sdi12.done(t)
 do
  rt.ms(250)
 end
end

QueryExample()
print(VRS[1].v, VRS[2].v[1])
```

res = sdi12.ack(cp, dev)

Parameters

- cp: Comport object
- dev: Device number

Returns

- res: Result - false or address

Description

Use the ACK active command. Returns false if the device is not there, else returns the device number.

```lua
local dn = sdi12.ack(dp, '?') -- Query a single device
```

sdi12.brk(cp)

Parameters

- cp: Comport object

Returns

None

Description

Outputs a 12ms break sequence.

`res = sdi12.chg(cp, dev, devto)`

Parameters

- `cp`: Comport object
- `dev`: Device number
- `devto`: Device address to change to

Returns

- `res`: false = failed; else address of device

Description

Attempt to change the device address

`ok,txt,crc = sdi12.crc(txt)`

Parameters

- `txt`: String to CRC check

Returns

- `ok`: True if the CRC matches
- `txt`: The text without the CRC
- `crc`: The CRC string

Description

Check, or generate a CRC for SDI-12

```lua
local _,txt,crc = sdi12.crc('1Testing---') -- calculate CRC
local ok = sdi12.crc('1TestingDU}') -- verify
```

`ok = sdi12.done(sdit)`

Parameters

- `sdit`: SDI-12 table structure

Returns

- `ok`: True if the state-machine is complete

Description

Polls the SDI-12 protocol state machine to work through the commands in the table.

`sdit = sdi12.go(cp, sdit)`

Parameters

- `cp`: Comport object
- `sdit`: SDI-12 command table

Returns

- `sdit`: The table, ready to be passed to `sdi12.done`

Description

Initialises the SDI-12 protocol table and starts the process.

`idt = sdi12.id(cp, dev)`

Parameters

- `cp`: Comport object
- `dev`: Device number

Returns

- `idt`: ID table idt[0]: Address idt[1]: SDI Version idt[2]: Company (8 characters) idt[3]: Model (6 characters) idt[4]: Version (3 characters) idt[5]: Serial number

Description

Queries the ID of the device.

sdi12.mrk(cp)

Parameters

- cp: Comport object

Returns

None

Description

Waits the required 8.33ms.

ok = sdi12.set(cp, [mb])

Parameters

- cp: Comport object
- mb: Mainboard. Set to false if using internal daughter board.

Returns

- ok: True if ok

Description

Configures the Comport object to 1200E7 and configures the RTS for direction.

txt = sdi12.t(cp, cmd)

Parameters

- cp: Comport object
- cmd: Command string (the '!' is optional)

Returns

- txt: Reply (or false if no reply)

Description

Performs a transparent SDI-12 command.

```
local dp = ser.dev()
local txt = sdi12.t(dp, '2!')
```

SDI-12 Table Structure

The table starts as an indexed set of tables with commands:

```
sdit = { {'1C5'}, {'2C1'}, {'2C2'}, {'3CC'} }
```

Once `sdi12.go` and `sdi12.done` have completed, there will be additional fields within the table:

- `sdit.c` = Comport object
- `sdit.[#].t` = Millisecond alive time when ready. False if the device did not respond
- `sdit.[#].n` = Number of samples (if a measurement command)
- `sdit.[#].v` = Table of results, or a string result

Library ser : Serial Ports

cp = ser.dev()

Parameters

None

Returns

- cp: Com Port object

Description

Return a COM port userdata value for the device serial port.

```
local dp = ser.dev()
dp:tx('Hello!')
```

cp = ser.eng([try])

Parameters

- try: Set to true to try and grab the engineer port. If it is in use by the terminal, then cp will be nil.

Returns

- cp: Com Port object

Description

Return a COM port userdata value for the egnineer serial port.

! This implicitly disables the terminal function, unless try is true.

```
local dp = ser.eng()
dp:tx('Hello Engineer!')
```

```
b = ser.term([en]) v1.00
b = ser.term(prot)
```

Parameters

- `en`: (optional) boolean value to control serial terminal
- `prot`: (optional) Enable terminal with the given protocol

Returns

- `b`: Boolean result. True if serial terminal enabled.

Description

Controls the engineer serial port to terminal logic.

When Lua reboots, the engineer serial port defaults to be used by the terminal.

You must have RX/TX and RTS/CTS connected to work as a terminal! The terminal will not start until RTS is asserted. If RTS is unasserted for 10 seconds the rt.buffer assumes the terminal has disconnected and will de-energise its RS232 port.

To use the engineer serial port for other uses, call:

```
ser.term(false) -- let Lua use the port
ep = ser.eng() -- also lets Lua use the engineer port
--...and the serial port can then be used within Lua.
```

Library `ser` | `cp` : Comport Objects

These functions work on the object returned from ser.dev() and ser.eng().

For the sake of documentation, these are shown to refer to 'cp'

```
cp:brk(ms)
```

Parameters

- `ms`: Number of milliseconds for the break sequence

Returns

None.

Description

Send break on the line.

```
nb,ti = cp:cap(file, [ms]) v1.00
nb,ti = cp:cap(fh, [ms])
nb,ti = cp:cap(file/fh, tab)
```

Parameters

- `file`: Filename to overwrite and create
- `fh`: File handle to write to
- `ms`: Time out to close in milliseconds (default 1000)
- `tab`: Parameter table.
  tab.ms = timeout, in milliseconds
  tab.ft = first byte timeout (default is 2x .ms)
  tab.nb = number of bytes to capture
  tab.hw = set to true to do hardwork at 96MHz

Returns

- `nb`: Number of bytes captured
- `ti`: Timed out flag

Description

Captures serial bytes directly to a file. Allows faster capturing than doing this directly in Lua.

❗ The write will occur from the the file handle current position.

`ok,txt,err = cp:cmd([txt], [tab])`

Parameters

- `txt`: (`'\r\n'`) String to send
- `tab`: Table of values:
  `.pt` = (0) pause time in seconds
  `.ss` = (\r\n) send string (overridden by txt)
  `.wt` = (5000) wait time, in milliseconds
  `.dt` = data time, in milliseconds (default is .wt)
  `.ls` = (>) look for string
  `.es` = (NIL) error string
  `.nt` = (5) number of times to try
  `.tt` = (120) try time, in seconds (.nt must be 0)
  `.re` = (false) Ignore echo of send string

Returns

- `ok`: True if we got the device’s attention
- `txt`: The text returned (without the prompt)
- `err`: If the error string was found, this return value has the string

Description

Send a command string to the device until it responds, or times out.

❗ Text is limited to 2048 bytes.

```lua
local dp = ser.dev()
local ok,txt = dp:cmd('$\r\n', {ls='>', wt=1500, nt=10})

local dp = ser.dev()
local ok = dp:cmd('$\r\n') -- wait for '>' prompt with defaults

local dp = ser.dev()
local t = {ss='$\r\n$\r\n', ls='ERROR\r\n>', tt=240}
local ok,txt = dp:cmd(t)
```

`value = cp:cts()`

Parameters

None

Returns

- `value`: True if the CTS line is asserted.

Description

Return the CTS status

`nb = cp:fifo([size])`

Parameters

- `size`: Number of bytes for the Serial Port DMA buffer + (min 128)

Returns

- `nb`: Number of bytes actually reserved

Description

Change the buffering for the serial port. The size defaults to 1024 bytes on Lua reboot.

Memory is taken from the Lua RAM area, split into 4 parts. Two parts at a time are queued with the hardware DMA controller. Therefore, the buffer time for a given FIFO size will = size / 2 x character-time. e.g. at 115200 baud, with cp:fifo(4096) will give a time of 4096/2 x 87us = 178ms.

`ok = cp:lck([ms])` v2.00

Parameters

- `ms`: Time to keep the lock enabled (default = 10000ms, max=30000)

Returns

- `ok`: True if we have locked the COM port.

Description

Lock the COM port. The lock keeps track of the number of times `cp:lck` was called.

```
dp = ser.dev()

if dp:lck(2000)
then
  -- do stuff short than 2000ms
  dp:ulck()
else
  -- do something else
  -- someone else is using it!
end
```

ok = cp:okr()

Parameters

None

Returns

- ok: true if there is a valid signal level on the serial port.

Description

Check whether the MAX-chip is detecting a valid signal.

```
local dp = ser.dev()
if dp:okr()
then
 dp:tx('You are connected :-)')
end
```

`ison = cp:pwo(ison)`

Parameters

- `ison`: true = enable power output
  false = turn off power output

Returns

- `ison`: True if the COM port power output is on

Description

Set power output on the COM port.

```
local dp=ser.dev()
dp:pwo(true) -- enable the 3.6V output
```

`cp:pwrx(ison, [ms])`

Parameters

- `ison`: true = enable receiving
  false = disable receiving
- `ms`: Optional time to keep the RX power enabled. Zero disables the timer. (ison is ignored)

Returns

None

Description

Set the receive power. Enabling receiving will keep the CPU alive. You MUST disable both pwtx and pwrx for both ports to enter low power sleep mode.

This is only needed if pwtx is not called, but you need to receive on a USART

cp:pwtx(ison, [ms])

Parameters

- ison: true = enable transmission
  false = disables transmission
- ms: Optional time to keep the TX power enabled. Zero disables the timer. (ison is ignored)

Returns

None.

Description

Set the transmit power.

! Enabling transmission on the serial chips requires about 8mA. Only enable the transmit power when absolutely necessary!

```
local dp=ser.dev()
dp:pwtx(0, 250) -- keep TX on for 250ms
-- OR --
dp:pwtx(true) -- use TX
...
dp:pwtx(false) -- finished
```

mode = cp:rs([mode]) v1.00
mode = cp:rs([mode], [main]) v1.10

Parameters

- mode: 232 or 485
- main: true or false

Returns

- mode: Current mode - 232 or 485

Description

Set the RS232 or RS485 mode of the Device COM Port.

The 'main' value defines whether the main PCB electronics are used for serial communications, or whether the add-on PCB is used.

```
local dp = ser.dev()
dp:rs(485, false) -- use add-on board in RS485 mode
dp:rs(485, true) -- use the main PCB in RS485 mode (inline adapter)
dp:rs(232, true) -- default = RS232 on the main PCB
```

`cp:rts(value)`

Parameters

- `value`: Boolean or integer value

Returns

None

Description

Set the RTS line. 0/false unasserts; 1/true asserts.

> If value is false, this overrides receive flow control. i.e. RTS flow control only operates when value = true.

`txt,b,to = cp:rx([ms, [nbytes]])`

Parameters

- `ms`: Time to wait, in milliseconds. Default = 5ms
- `nbytes`: Maximum number of bytes to read. Default = 2k. Maximum = 8k

Returns

- `txt`: String received
- `b`: Number of bytes in txt
- `to`: true if timed-out

Description

Wait for receive data.

`nb = cp:rxe([ms])` v1.00

Parameters

- `ms`: Time to wait, in milliseconds. Default = 100ms

Returns

- `nb`: Number of bytes flushed

Description

Flush receive data.

`cp:rxf(ison)`

Parameters

- `ison`: True if RTS should unassert when receive buffers are full.

Returns

None

Description

Set RX flow control (uses RTS line).

> ❗ If ison is true, this also calls :pwtx(true) and :rts(true) Therefore, power will be consumed all the time that :rxf(true) is enabled because the RS232 chip has to be enabled to assert RTS! And you will see the DeviceTx enabled in the PMM terminal command.

`cp:rxto(bits)`

Parameters

- `bits`: Number of data bits to consider a receive timeout for the internal DMA controls

Returns

None

Description

Set RX time out in bits

`txt,b,ti,fs = cp:rxu(rxut, [ms])`

Parameters

- `rxut`: Receive Until Table
- `ms`: Timeout in milliseconds

Returns

- `txt`: The received data
- `b`: Number of bytes in txt
- `ti`: true if timed-out
- `fs`: Find state:
  0=didn’t find start
  -1=found start but no end
  1=found end (OK)

Description

Receive Until.

See the following information the `'rxut'` table structure.

`b = cp:rxw()`

Parameters

None

Returns

- `b`: Number of bytes waiting

Description

Returns number of bytes waiting on port

`cp:set(txt)`
`cp:set(baud, [parity, [databits, [stopbits]]])`

Parameters

- `txt`: Baud rate and protocol string
  e.g. `'19200n8'`
  stop bits are defined with a suffix:
  '.' = 1, e.g. '19200n8.' ',' = 1.5, e.g. '19200n8,' ':' = 2, e.g. '19200n8:'
- `baud`: Baud rate, from 300-115200
- `parity`: Parity, 'n', 'e', 'o', 'm', 's'
- `stopbits`: 1, 1.5, 2

Returns

None

Description

Set the protocol.

```
local dp=ser.dev()
dp:set('300n8:')
dp:set(300, 'n', 8, 2) -- same as above
```

b,to,txt = cp:tx(txt, [ms])

Parameters

- txt: The text to send
- ms: Timeout for sending

Returns

- b: Number of bytes sent
- to: true if timed-out
- txt: Remaining part of txt that is yet to be sent

Description

Send data.

> If you don’t see data being output, check whether you have transmit flow control and the CTS line status, and whether cp:pwtx has been enabled.

```
local dp=ser.dev()
dp:pwtx(1)
dp:tx('Welcome!')
dp:wtx(100)
dp:pwtx(0)
```

cp:txf(ison)

Parameters

- ison: True if CTS should be asserted before data is sent

Returns

None

Description

Set TX flow control (uses CTS line).

`cp:txtg(bits)`

Parameters

- `bits`: Number of data bits to pause between each transmitted byte

Returns

None

Description

Set TX time guard in bits. The time-guard slows down transmission of data for the benefit of slow devices.

`ok = cp:ulck()`
`cp:ulck(true)` v2.00

Parameters

- `true`: This form will unlock all counts

Returns

- `ok`: True if we still owned the COM port (i.e. false indicates the COM port timer has expired, etc)

Description

Unlock the COM port (before the timeout completes). You will need to call `cp:ulck` the same number of times that `cp:lck` was called in order to unlock before the timeout.

`to = cp:wtx(ms)`

Parameters

- `ms`: Number of milliseconds to wait

Returns

- `to`: true if timed out before data was sent

Description

Wait for transmission to complete.

Receive Until Table `rxut`

As used by the cp:rxu method.

- `st = 0`
  - start timeout, in milliseconds, for flushing. If > 0 then the function will wait for a pause until beginning the scanning.
- `ss = ''`
  - start string or table-of-strings
- `bn = 0`
  - minimum number of bytes (the number of bytes before the et/es are started)
- `bx = 256`
  - maximum number of bytes (256 is the largest)
- `et = 0`
  - end timeout in milliseconds.
- `es = ''`
  - end string or table-of-strings.

```
local txt,to = dp:rxu( {ss='begin'} )

local txt,to = dp:rxu( {ss={'this', 'that'} } ) -- waits for this or that
```

Useful Examples

```
NMEA= {ss='$',es='\r\n',bn=4}
ASCII= {es='\n'} or {es='\r\n'} or {es=\{'\r','\n'}}
HydrINS= {ss='W',es='\r\n',bn=8}
```

Library `smp` : Sample Data Store

The sample data store library provides a convenient way to handle periodic writes of sample data. The writing is handled 'atomically'. When a sample store reaches a certain size (e.g. 128k), or when it s time to send what has been saved, the sample store is "cut" and the file moved to the /Send directory.

The sample data store is also suitable for collecting bulk data from a device, and choosing to either save or abandon the data (e.g. on a communication drop-out).

By default, the sample data store works with a single file. However, it is possible to use the table structures defined to save to multiple sample stores for example where there are multiple connected devices, or where selected data needs to be extracted to a secondary file.

Structure of `ftable`

- `.n = 'AppName.data.txt'`
  - Filename of the current file (".tmp" is added automatically)
  - Default is 'AppName.data.txt'
- `.f = nil`
  - Current file object (NIL if closed)
- `.r`
  - String or Function for renaming when cutting the file.
  - If this is nil (or not present), then the default function will use the form "YYYYMMDD-HHMMSS-SerialNo-SiteName-data.txt"
- `.h`
  - String or Function for the header of the file
  - `function(ftable, file)` return nothing, do the writes within the function
- `.t`
  - String or Function for the tail of the file
  - function(ftable, file) return nothing, do the writes within the function
- `.bx = 131072`
  - Maximum number of bytes for the file
  - 0 means unlimited (not recommended)
  - Default is 128k (131072 bytes)

Sample Data Store Functions

ov = smp.cnt([v])

Parameters

- v: New value for the i.smp.cnt

Returns

- ov: The old value

Description

The i.smp.cnt value is 'hidden' in C++. The value will be incremented and applied to i.smp.cnt on the next call to smp.save.

txt = smp.csv(valuetable, [inctime])

Parameters

- valuetable: table of values to turn into CSV
- inctime: optional boolean. Set to false to not include the time as the first value.

Returns

- txt: returned string in CSV format

Description

Creates a CSV line with UTC as the first field in the form

"YYYY-MM-DD HH:MM:SS" (default timeformat = '%Y-%m-%d %H:%M:%S')

Strings are quoted with quote (default double quotes)

The line ends with lineend (default [CR][LF])

The CSV values are controlled by a global table "csv".

```
local t = {1, 2, 3, i.cell_csq}

smp.save( smp.csv(t) )
```

smp.cut([ftable], [killfile])

Parameters

- `ftable`: optional. If not present, uses global table `"ft"`
- `killfile`: optional boolean. If true, the cut file will be discarded.

Returns

None

Description

Split off the file and rename.

Use the killfile option when downloading bulk data from a device and the process failed.

```
function onJob2() -- do this on a schedule
 smp.cut()
end
```

ftable = smp.save([ftable], txt)

Parameters

- `ftable`: optional. If not present, uses global table `"ft"`
- `txt`: text to save to the sample file

Returns

- `ftable`: table that was passed (for convenience)

Description

Atomically saves a single block of data to the data store

Also flashes LED to indicate data being stored

```
local t = 'this is a sample\r\n'

smp.save(t)
```

Prebuilt Function Methods

These functions save you coding them in pure Lua.

smp.r_ft

Description

Default rename function

Renames the file according to YYYYMMDDHHMMSS-serial-site_name-filename

(Uses the fc.utc and utc.fn formatting functions to use the file’s creation time)

```
ft = {r=smp.r_ft}
```

smp.t_md5

Description

Tail function that adds CR/LF+MD5 of the file on close

```
ft = {t=smp.t_md5}
```

smp.t_crc32

Description

Tail function that adds CR/LF+MD5 of the file on close

```
ft = {t=smp.t_crc32}

ft = {r=smp.r_ft, t=smp.t_md5}    two options
```

CSV Formatting Table `csv`

The global table 'csv' defines the formatting for the smp.csv function.

- `csv.s = ','`
  - Separator string override.
- `csv.q = '  '`
  - Quote character for strings.
- `csv.tf = '%Y-%m-%d %H:%M:%S'`
  - Time format. If this is blank (i.e. ""), the time is NOT placed at the beginning of the line. You'll need to add the time yourself into the table. Use standard C strftime format strings.
- `csv.le = '\r\n'`
  - Line ending string

Library `sms` : SMS Text Messaging

The SMS library allows for the transmission of 8-bit binary, or 16-bit UCS2 encoded PDU SMS messages, and the reception of 7-bit GSM, 8-bit, and 16-bit UCS2 encoded PDU SMS messages.

Up to 16 SMS messages are queued up while the modem is offline, and the messages are sent when the modem connects. Incoming messages are handled asynchronously, and the rt.buffer can be programmed to remain connected to the network for a minimum time - to allow for delivery of the SMS messages.

`tv = sms.dec(PDU)` v1.10

Parameters

- `PDU`: Binary PDU block

Returns

- `tv`: Table of values
  `.num` = originating number
  `.utc` = UTC time
  `.tzm` = Time zone minute offset
  `.pid` = Protocol identifier
  `.dcs` = data coding scheme (0=7-bit GSM, 4=8-bit, 8=UCS2)
  `.msg` = message ASCII
  `.ucs` = raw UCS2 (if encoding is UCS2)

Description

Decodes a PDU frame into usable fields.

PDU = sms.enc(msg, [number], [options]) v1.10

Parameters

- msg: Message string (max 140 octets)
- number: (optional) target phone number. If not present, then c.cell_sms is used
- options: (optional) table of optional values
  .mti = Message Type. 0x01 or 0x11 are acceptable
  .mr = Message Reference
  .vp = Validity period. e.g. 0xaa = 4-days (See 3GPP TS 123 040 section 9.2.3.12.1)
  .pid = Protocol identifier
  .ucs = true (sets UCS2 encoding, and msg is assumed to be two-byte Unicode)

Returns

- PDU: Binary output

Description

Encodes a simple ASCII or binary message into a PDU frame suitable for sending.

fc,mc = sms.free() v1.10

Parameters

None

Returns

- fc: Number of free messages available
- mc: Number of messages currently queued for transmission.

Description

Returns the message queue counters.

`sms.go([DoItNow])` v1.10

Parameters

- `DoItNow`: (optional) whether to connect immediately.

Returns

None

Description

Tells the modem handler that we need to perform SMS transmission on the next connection phase.

DoItNow is assumed to be true (i.e. clear hold off and dial immediately), so 'sms.go(false)' will just arm the modem handler for the next phase.

`v = sms.rxt([secs])` v1.10

Parameters

- `secs`: Number of seconds to stay online (minimum). Set to 0 to disable the receive wait process.

Returns

- `v`: Current receive time

Description

Sets, or reads, the receive time. This is the minimum number of seconds the modem will remain registered before powering off. The delay is required to provide time for the cellular base-station to send the messages to the rt.buffer.

`ok = sms.snd(PDU, [DoItNow])` v1.10

Parameters

- `PDU`: SMS PDU frame. Use sms.enc to encode
- `DoItNow`: Whether to transmit immediately. Pass the value false to simply queue for later

Returns

- `ok`: True if the message was queued.

Description

Queue the PDU frame and connect the modem.

`ok = sms.txt(msg, [number], [DoItNow])` v1.10

Parameters

- `msg`: ASCII message
- `number`: (optional) target phone number. If not present, uses c.cell_sms
- `DoItNow`: Whether to transmit immediately.

Returns

- `ok`: True if the message was queued

Description

Combined function, similar to sms.snd(sms.enc(msg, number, DoItNow)).

However, msg can be longer than 140 characters - this function will queue up multiple PDUs.

SMS Received callback

The SMS callback occurs asynchronously at any time while the modem is registered (i.e. not just during the SMS transmission phase).

The service message center may require the rt.buffer to be connected and registered for some time (e.g. at least 30seconds) in order to send any queued SMS messages. Use the 'sms.rxt' function to enable minimum-waiting-times.

function onSMS(smsc, pdu) v1.10

Parameters

- smsc: The Service Message Center number
- pdu: The binary PDU received.

Description

You should use 'sms.dec' to decode the received PDU frame and do something meaninful with them.

e.g.

```
sms.rxt(30) -- set the minimum online time
function onSMS(nu, pdu)
 local t = sms.dec(pdu) -- decode the frame into a table
 local s = t.num .. ' : ' .. t.msg -- make a message
 usb.log('SMS:'..s)
end
```

Library `struct` : Structure Handling

The struct module adds C style packing and unpacking (for byte-wide values, not bits).

> ℹ See:
> http://www.inf.puc-rio.br/~roberto/struct/
> https://github.com/dubiousjim/luafiveq/blob/master/src/struct.c
> https://github.com/dubiousjim/luafiveq/blob/master/BENEFITS-LUA

`txt = struct.pack(fmt, d1, d2, …)`

Parameters

- `fmt`: Structure Format string
- `d1`: Data
- `d2`: ...more data

Returns

- `txt`: Packed data

Description

Returns a string containing the values d1, d2, etc. packed according to the format string fmt.

`nb = struct.size(fmt)`

Parameters

- `fmt`: Structure Format string

Returns

- `nb`: Number of bytes required

Description

Returns the size of a string formatted according to the format string fmt. The format string should contain neither the option s nor the option c0.

`d1,d2,⋯ = struct.unpack(fmt, txt, [idx])`

Parameters

- `fmt`: Structure Format string
- `txt`: Packed data
- `idx`: Start of reading (default is 1)

Returns

- `d1`: Data 1
- `d2`: Data 2 etc

Description

Returns the values packed in string s according to the format strong fmt. An optional i marks where in s to start reading (default is 1). After the read values, this function also returns the index in s where it stopped reading, which is also where you should start to read the rest of the string.

Struct format strings

Here are the formatting codes. Initially endianness is set to native and alignment is set to none (!1).

- `">"` use big endian
- `"<"` use little endian
- `"!"` use machine’s native alignment
- `"!n"` set the current alignment to n (a power of 2)
- `" "` ignored
- `"x"` padding zero byte with no corresponding Lua value
- `"xn"` padding n bytes
- `"Xn"` padding n align (default to current or native, whichever is smaller)
- `"b/B"` a signed/unsigned char/byte
- `"h/H"` a signed/unsigned short (native size)
- `"l/L"` a signed/unsigned long (native size)
- `"i/I"` a signed/unsigned int (native size)
- `"in/In"` a signed/unsigned int with n bytes (a power of 2)
- `"f"` a float (native size)
- `"d"` a double (native size)
- `"s"` a zero-terminated string
- `"cn"` a sequence of exactly n chars corresponding to a single Lua string. An absent n means 1. The string supplied for packing must have at least n characters; extra characters are ignored.
- `"c0"` this is like "cn", except that the n is given by other means: When packing, n is the actual length of the supplied string; when unpacking, n is the value of the previous unpacked value (which must be a number). In that case, this previous value is not returned.
- `"("` stop capturing values
- `")"` start capturing values
- `"="` current offset

Struct Examples

To match a C structure

```
/* C demo structure */
struct Str {
 char b;
 int i[4];
};
```

in Linux/gcc/x86 (little-endian, max align 4), use "<!4biiii"

To pack and unpack Pascal-style strings

```
sp = struct.pack("Bc0", string.len(s), s)
```

```
s = struct.unpack("Bc0", sp)
```

In the latter command, the length (read by the element "B") is not returned.

To pack a string in a fixed-width field with 10 characters padded with blanks

```
x = struct.pack("c10", s .. string.rep(" ", 10))
```

Library twi : TWI Hardware Interface

```
a1,a2··· = twi.qy()
ok = twi.qy(addf,[addt])
```

Parameters

- addf: Device address to start
- addt: Device address to finish

Returns

- a1: Addresses
- ok: If found

Description

Queries the TWI bus.

```
res = twi.rd(add,txt)
res = twi.rd(add,num,[v])
```

Parameters

- add: Device address
- txt: String to send during the read phase
- num: Number of bytes to read
- v: Output value while reading (default = 0xff)

Returns

- res: String bytes read. False if fails.

Description

Reads from the TWI bus.

HINT: Use the Lua `string.bytes(···)` to pull the reply apart.

```
res = twi.wr(add,txt)
res = twi.wr(add,b1,b2,b3)
```

Parameters

- `add`: Device address
- `txt`: String to write b#: Bytes to send

Returns

- `res`: String bytes read. False if fails.

Description

Writes to the TWI device.

You can mix and match the string and byte parameters.

Library `usb` : USB functions

`usb.log([sub], txt)`

Parameters

- `sub`: Optional sub section string.
- `txt`: Text to output to USB log stream

Returns

None

Description

Output a string for the USB log. Useful for debugging Lua Apps.

`n,vp = usb.on()`

Parameters

None

Returns

- `cn`: USB Connected (at the protocol level)
- `vp`: Voltage present on the USB port

Description

Returns information about the USB port.

Library utc : Time Functions

txt = utc.fmt(format, [t])

Parameters

- format: C strftime format string
- t: UTC time value (integer). Default is now.

Returns

- txt: Formatted time string.

Description

Convert time into formatted text.

txt = utc.fn([t])

Parameters

- t: UTC time value (integer). Default is now.

Returns

- txt: String of time in form 'YYYYMMDDHHMMSS'

Description

Returns a time string that is suitable for filenames.

utc.job(…)

Description

Convenience function   same as job.utc

`ok = utc.ok()`

Parameters

none

Returns

- `ok`: true if the time is sensible.

Description

Returns true if the time is sensible (i.e. > 2016-01-01 00:00:00)

Use this to determine when we've got a good time before continuing with data collection.

`utc.set([utc], [tzm])`

Parameters

- `utc`: UTC time integer, or string
- `tzm`: optional time zone offset, in minutes

Returns

None

Description

Sets the real time clock.

```
utc.set('2017-01-01 10:00:00') -- just the time
utc.set(_, 120) -- just the Timezone (+2hr)
utc.set(1234567, 120) -- integer UTC and timezone
```

`wd,spm,ds = utc.split(t)`

Parameters

- `t`: Time value (integer) or string

Returns

- `wd`: UTC time at 00:00:00 on this day
- `spm`: Seconds past midnight
- `ds`: Days since 1970-01-01

Description

Split the UTC time value into components.

`spm = utc.spm(t)`

Parameters

- `t`: Time value (integer) or string (Default is now.)

Returns

- `spm`: Seconds past midnight

Description

Calculate the seconds-past-midnight. i.e. the UTC time value with the day-portion subtracted.

`txt = utc.txt([t])`

Parameters

- `t`: UTC time value (integer). Default is now.

Returns

- `txt`: String of time in form 'YYYY-MM-DD HH:MM:SS'

Description

Convert time into text.

`tzom, tzos = utc.tz()`

Parameters

None

Returns

- `tzom`: Time zone offset in minutes
- `tzos`: Time zone offset in seconds

Description

Returns the time-zone offset (as retrieved by the 3G modem).

`t = utc.val(txt)`

Parameters

- `txt`: string time to convert

Returns

- `t`: UTC time value

Description

Convert text to either full date or seconds-past-midnight

```
local v = utc.val('08:00')
local v = utc.val('2017-06-03 12:15:00')
```

`wd,ds = utc.wds(t)`

Parameters

- `t`: Time value (integer) or string

Returns

- `wd`: UTC time at 00:00:00 on this day
- `ds`: Days since 1970-01-01

Description

Calculate the whole-day portion of the UTC time value.

Library `util` : Utility functions

`t = util.add(t1, t2, …)`

Parameters

- `t1`: Table
- `t2`: Table …

Returns

- `t`: combined table

Description

Combines two or more tables into a single table. If keys in t2 are present in t1, then t2 takes precedence.

Effectively uses Lua's "pairs" on each table in turn. Lua equivalent for each table:

```
--[[ Equivalent Lua
t = {}
for k,v in pairs(t1) do t[k]=v
]]--
```

`t = util.addi(tv1, tv2, …)`

Parameters

- `tv1`: Table or value
- `tv2`: Table or value …

Returns

- `t`: Combined table

Description

Combines two or more tables (or values) into a single table.

Effectively uses Lua's "ipairs" on each table in turn. Lua equivalent for each table:

```
--[[ Equivalent Lua
t = {}
for k,v in ipairs(tv1) do table.insert(t, v)
]]--
```

Example:

```
local t1 = {1,2,3}
local t2 = {5,6,7}
local t3 = util.addi(t1, 4, t2, 8, 9)
-- t3 = {1,2,3,4,5,6,7,8,9}
```

cs = util.codesize(MYFUNC) v2.00
cs = util.codesize(nil, true) v2.00
cs = util.codesize(nil, function(cs,fn) print(fn,cs) end) v2.00

Parameters

- `MYFUNC`: Function name to profile
- `cs`: Number of bytes (for callback)
- `fn`: Function name (for callback)

Returns

- `cs`: Number of bytes

Description

Calculate the RAM impact of a function, or all top-level functions.

The format `util.codesize(nil, true)` will print the individual function sizes to the console. Providing a function for the second parameter allows customisation of the results.

t1,t2 = util.cuti(t, idx)

Parameters

- `t`: Table that needs cutting in two
- `idx`: Index of first entry to be in second half

Returns

- `t1`: First half
- `t2`: Second half

Description

Cuts a table in half at the 'idx' point.

If idx=1, then t1 will be empty, and t2===t

Idx can be negative to indicate "n-th point from the end", so if idx=-1, then t1==t, and t2 will be empty.

```
local t1 = {1,2,3,4,5,6,7,8,9}
local t2,t3 = util.cuti(t1, 6)
-- t2 = {1,2,3,4,5}
-- t3 = {6,7,8,9}
```

`v = util.dct(t)` v2.00

Parameters

- `t`: Table to deep copy

Returns

- `v`: Resulting copy of table

Description

Deep-copies a Lua table. Any changes made to `v` will therefore be independent of the original table `t`.

using the Lua standard v=t will only result in a single table in memory. Any changes to v will affect t as well. That s why a deep copy is needed to produce a replica of the original table that is independent.

Only the Lua types: LUA_TNIL, LUA_TBOOLEAN, LUA_TNUMBER, LUA_TSTRING, and LUA_TLIGHTUSERDATA are handled.

`idx,found = util.find(t, v, [func])` v2.00

Parameters

- `t`: Table of values
- `v`: Value to find
- `func`: Optional comparison function

Returns

- `idx`: Index (effectively the insertion point)
- `found`: Whether found in the list

Description

Binary search through the ordered table.

Uses the first entry of the table to determine whether string or number.

```
local idx,found = util.find( {1,2,3,4,5}, 3) -- > 3, true
local idx,found = util.find( {1,2,3,4,5}, 3.1) -- > 3,false
local idx,found = util.find( {'apple','orange','pear'}, 'orang') -- > 2,false
local idx,found = util.find( {'apple','orange','pear'}, 'oranges') -- > 3,false
local idx,found = util.find( {'apple','orange','pear'}, 'oranges',
  function compare(a,b)  ①
    if a<b
    then return -1
    elseif a==b
    then return 0
    else return 1
    end) -- > 3,false
```

① Comparison function

n = util.iter(FOO,start,[depth]) v2.00

Parameters

- FOO: Function to call. n = FOO(o) is the prototype.
- start: Starting object or table
- depth: (optional) table depth. Default = 3

Returns

- n: Count or total

Description

The function FOO can return a value that is summed to form the total n.

kvco = util.kvc(nb,ni) v2.00

Parameters

- nb: Number of bytes for the data
- ni: Number of index entries

Returns

- kvco: KVC object

Description

Return a KVC object.

See util.kvc

t = util.remi(t, idx, [last])

Parameters

- t: Table that needs cutting in two
- idx: Entry to remove, or first entry to remove
- last: Optional last entry to remove

Returns

- t: Resulting table

Description

Returns a table that is 't' minus the entry at idx, or minus entries between idx and last (inclusive)

```
local t1 = {1,2,3,4,5,6,7,8,9}
local t2 = util.remi(t1, 6)
-- t2 = {1,2,3,4,5,7,8,9} (#6 removed)
local t3 = util.remi(t1, 6, -2)
-- t3 = {1,2,3,4,5,9} (#6-8 removed)
```

ty = util.tia(ta, tb) v2.00

Parameters

- ta: Table A
- tb: Table B

Returns

- ty: Resultant table

Description

Append two indexed tables.

```
t = rt.tia({1,2,3},{'A','B'})
---> t = {1,2,3,'A','B'}
```

ty = util.tio(ta, tb) v2.00

Parameters

- ta: Table A
- tb: Table B

Returns

- ty: Resultant table

Description

OR together two indexed tables.

Useful for applying defaults to a table.

```
t = util.tio({1,_,3},{'A','B','C'})
---> t = {1,'B',3}
```

LCD and User Interface Library Extensions

Library `key` : Keyboard Library

The keyboard library provides scripting support for up to 8 push-to-make switches that are connected to an IO-expander on the I2C bus.

```
bf,td = key.get()
v,td = key.get(keyno)
```

Parameters

- `keyno`: Which key to check (1-8)

Returns

- `v`: (boolean) whether the key is pressed (if keyno is provided)
- `bf`: (number) bit-field of the pressed keys (D0 = key #1)
- `td`: Time keys have been pressed (milliseconds)

Description

Get the state of an individual key, or all keys as a bitfield.

```
v = key.ign(bf)
v = key.ign(keyno, ign)
```

Parameters

- `bf`: Number bitfield. If a bit is set then the key is ignored.
- `keyno`: Key number (1-8)
- `inv`: Whether to ignore (boolean)

Returns

- `v`: Bitfield of ignore field.

Description

Set which keys to ignore.

`ok = key.inj(keyno)` v2.00

Parameters

- `keyno`: Key number 1-8

Returns

- `ok`: True if injected

Description

Injects a keystroke into the keyboard buffer.

`v = key.inv(bf)`
`v = key.inv(keyno, inv)`

Parameters

- `bf`: Number bitfield. If a bit is set then the key is inverted.
- `keyno`: Key number (1-8)
- `inv`: Whether to invert (boolean)

Returns

- `v`: Bitfield of inversion field.

Description

Set which keys to invert.

`key.kre(keyno, en)`

Parameters

- `keyno`: Key number 1-8
- `en`: Whether to act on key repeats (true=repeat)

Returns

None

Description

Set the Key Repeat Enable. If an individual key has repeat enabled, then there will be a callback at the repeat interval all the time the key is kept pressed.

`rt,it = key.kri(rt,[it])`

Parameters

- `rt`: Repeat time (milliseconds)
- `it`: Idle time (milliseconds)

Returns

- `rt`: Repeat time (milliseconds) Default = 400ms
- `it`: Idle time (milliseconds) Default = 10000ms (10sec)

Description

Set the Key Repeat Idle time and repeat interval time.

`ok = key.ok()`

Parameters

None

Returns

- `ok`: True if the keyboard hardware was found.

Description

Check whether the keyboard is connected.

Key callback functions

`function onKey#(keyno,isdown,t)`

Parameters

- `keyno`: Key number 1-8.
- `isdown`: True if the key is pressed.
- `t`: Time, in milliseconds, that the key has been held down for.

Returns

None

Description

Callback for an individual key. Called after the `onKey` general callback.

```
function onKey2(kn,id,td)
 -- do something
end
```

`function onKey(keyno,isdown,t)`

Parameters

- `keyno`: Key number 1-8. Or -1 for the Idle callback.
- `isdown`: True if the key is pressed.
- `t`: Time, in milliseconds, that the key has been held down for.

Returns

None

Description

Called when any key is pressed. This function is called first.

When the keyboard is idle, the `onKey` function is called with keyno=0 (allowing for a single function to handle all events). The `onKeyIdle` callback (if defined) is called after this function.

`function onKeyIdle()`

Parameters

None

Returns

None

Description

Called when the keyboard has been idle.

See `key.kri` - Idle Time.

Library `lcd` : LCD Library

The LCD library was included in firmware v2.00. None of the LCD functions are in previous releases.

The `lcd` library provides low-level functions for displaying text, icons, and graphics.

Two LCD boards are supported (both from Electronic Assembly http://www.lcd-module.com):

- EA DOGXL-160
  - 160x104 monochome
- EA DOGL-128 or EA DOGM-128
  - 128x64 monochrome

The LCD board is an optional attachment that also includes support for 8 keyboard switches.

See also the `key` library.

The LCD library will not function unless the keyboard library detects the keyboard chip!

The coordinate system has (0,0) as the top-left pixel.

The LCD library is a modified and improved version of the U8G2 library.

Hardware control

`ison = lcd.act()`

Parameters

None

Returns

- `ison`: true if the LCD is currently on

Description

Find out if the LCD is powered.

`lcd.c(col)`

Parameters

- `col`: Colour - 0=WHITE, 1=BLACK

Returns

None

Description

Set the current drawing colour.

`lcd.ers()`

Parameters

None

Returns

None

Description

Erases the LCD buffer.

`lcd.go([ison])`

Parameters

- `ison`: Whether to power on the LCD too (default=true)

Returns

None

Description

Display the current LCD buffer.

`w,h = lcd.gss()`

Parameters

None

Returns

- `w`: Width in pixels
- `h`: Height in pixels

Description

Get the LCD screen size dimensions.

> ℹ X-coordinates will be from `0` to `(w-1)`; Y-coordinates will be from `0` to `(h-1)`

`v = lcd.idle(v)`

Parameters

- `v`: Idle time, in milliseconds

Returns

- `v`: Idle time, in milliseconds

Description

Set, or get, the LCD idle timeout.

> ℹ Limited to >1500ms and <120000ms (2 minutes)

`ok = lcd.init(width, [rot], [eng])`

Parameters

- `width`: Pixel width of LCD, either 160 or 128 (default=128)
- `rot`: Optional rotation 0, 90, 180, 270 (default=0)
- `eng`: True to route LCD to engineer port

Description

Initialise the LCD hardware.

> RAM for the LCD buffer is taken from the Lua working RAM. The DOGXL160 requires 2k, the DOGL128 requires 1k.

`lcd.off()`

Parameters

None

Returns

None

Description

Turns off the LCD.

`ok = lcd.ok()`

Parameters

None

Returns

- `ok`: true if the LCD hardware was found

Description

Check whether the LCD and keyboard is connected.

`lcd.on(ison)`

Parameters

- `ison`: true if enabled

Returns

None

Description

Turn the LCD on, or off.

> ❗ The LCD consumes about 700uA while on.

`lcd.save(filename)`

Parameters

- `filename`: File to save the bitmap to.

Returns

None

Description

Save an NetPBM file `P4` style bitmap of the current LCD frame buffer.

> ℹ Use the extension `.pbm` to be compatible with Photoshop, etc.

`lcd.sdc(c)`

Parameters

- `c`: Contrast, 0-255

Returns

None

Description

Set the display contrast.

Drawing functions

`lcd.box(ft, x,y, w,h, [r])`

Parameters

- `ft`: Fill type:
    - `'f'` = filled
    - `'h'` = hollow
    - `'r'` = rounded hollow
- `x,y`: Starting point
- `w`: Width in pixels
- `h`: Height in pixels
- `r`: Radius, in pixels

Returns

None

Description

Draw a box, or a block.

`lcd.cir(ft, seg, x,y, rx, [ry])`

Parameters

- `ft`: Fill type
    - `'f'` = filled
    - `'h'` = hollow
- `seg`: Which segments to draw:
    - `'*'` or nil = all
    - `'L'` = upper left
    - `'R'` = upper right
    - `'l'` = lower left
    - `'r'` = lower right
- `x,y`: Centre
- `rx`: Radius for width
- `rh`: Radius for height (=rx if not present)

Returns

None

Description

Draw a circle, or ellipse, or segment.

`lcd.l(x1,y1, x2,y2)`

Parameters

- `x1,y1`: Starting point
- `x2,y2`: End coordinate

Returns

None

Description

Draws a line from (x1,y1) to (x2,y2) in the current drawing colour.

`lcd.lh(x, y, w)`

Parameters

- `x,y`: Starting point
- `w`: Width of line, in pixels

Returns

None

Description

Draws a horizontal line in the current drawing colour.

`lcd.lv(x, y, h)`

Parameters

- `x,y`: Starting point
- `h`: Height of line, in pixels

Returns

None

Description

Draws a vertical line in the current drawing colour.

`lcd.p(x, y)`

Parameters

- `x`: X coordinate. 0-159
- `y`: Y coordinate. 0-103

Returns

None

Description

Set a pixel to the current drawing colour.

`ok,x,y,w,h = lcd.pbm(x,y, filename, [invert])`

Parameters

- `x,y`: Position (top left corner)
- `filename`: Name of the PBM file
- `invert`: Whether to invert drawing

Returns

- `ok`: True if loaded and drawn
- `x,y`: New position (bottom right corner)
- `w,h`: Width and height of the bitmap

Description

Draw a NetPBM `P4` binary bitmap.

ℹ Photoshop refers to this as "Portable Bit Map", and saves as a `.pbm` file.

`lcd.tri(ft, x1,y1, x2,y2, x3,y3)`

Parameters

- `ft`: Fill type
  - `'f'` = filled
  - `'h'` = hollow
- `x1,y1`: Start corner
- `x2,y2`: Middle corner
- `x3,y3`: End corner

Returns

None

Description

Draw a triangle.

`lcd.xbm(x,y, w,h, txt)`

Parameters

- `x,y`: Position
- `w`: Bitmap width
- `h`: Bitmap heigh
- `txt`: XBM encoded string

Returns

None

Description

Draw a bitmap on the LCD frame buffer.

Font and Text functions

`fa,fd = lcd.f(name)`

Parameters

- `name`: Font name string

Returns

- `fa`: Font ascent
- `fd`: Font descent

Description

Choose a font to write with.

`lcd.fr(rot)`

Parameters

- `rot`: Rotation angle, 0, 90, 180, 270

Returns

None

Description

Set the font rotation angle.

`lcd.ft(tr)`

Parameters

- `ft`: Whether transparent font. 1=transparent; 0=opaque

Returns

None

Description

Set the font drawing method.

`fa,fd = lcd.gad()`

Parameters

None

Returns

- `fa`: Font ascent
- `fd`: Font descent

Description

Get the current font's ascent and descent metrics.

`lcd.gly(x,y, gn)`

Parameters

- `x,y`: Coordinate
- `gn`: Glyph number

Returns

None

Description

Draw a Unicode glyph from the current font.

`w,fa,fd = lcd.gtw(txt)`

Parameters

- `txt`: UTF-8 text to measure

Returns

- `w`: Width in pixels
- `fa`: Font ascent
- `fd`: Font descent

Description

Measure the width of a UTF-8 string in the currently selected font.

`x,y,w,h = lcd.t(x,y, txt)`

Parameters

- `x,y`: Coordinate for the baseline
- `txt`: UTF-8 text to print

Returns

- `x`: Updated x position
- `y`: Updated y position
- `w`: Width of text
- `h`: Height of text

Description

Draw the `txt` at the position `x,y` with the currently selected font.

You can throw away all, or some, of the return values.

```
local x = 10
x = lcd.t(x, 10, 'Hello ')
x = lcd.t(x, 10, 'world!')

local y,_
y = 10
_,y = lcd.t(10, y, 'Line 1')
_,y = lcd.t(10, y, 'Line 2')
```

Font names

Single size

- `"gt"` Glasstown (modified u8g2_font_glasstown_nbp_t_all)

X11 fonts

- `"4x6"`
- `"5x8"`
- `"6x10"`
- `"6x12"`
- `"7x13"`
- `"7x13b"` Bold
- `"7x13e"` Emphasised
- `"9x15"`
- `"10x20"`

ProFont

- `"pf10"`
- `"pf11"`
- `"pf12"`
- `"pf15"`
- `"pf17"`
- `"pf22"`
- `"pf29"`

Icon and special fonts

- `"batt6"`
- `"cell6"`
- `"i1"` OpenIconic x 1
- `"i2"` OpenIconic x 2
- `"i4"` OpenIconic x 4
- `"sk7"` Soft key icons
- `"uf"` Unifont symbols

Library `ui` : User Interface LCD Library

The LCD library was included in firmware v2.00. None of the LCD functions are in previous releases.

The `ui` library provides high level user interface functions for keyboard and LCD operation.

See also: `lcd` and `key` libraries.

The UI library is designed to be extremely memory efficient. The LCD frame buffer will take just over 2k (for an EA-DOG160XL LCD). Only the currently viewed UI element will be loaded into Lua's RAM - with everything else existing in comments within the Lua App file.

The UI will enter power save after a brief period, and can be instantly woken with a press of a key.

The UI is called from the context of the LuaEvents task. Consequently, ensure any functions do not block - otherwise you will hold up the rest of the system. (Best to trigger flags that are then executed within the Lua Loop task.)

Boilerplate code

The code here shows the smallest UI code required. Additional information is provided in the following chapters.

```
--[[Mmain1] ①
Nsample
[Main menu]
Welcome!
Alive {i.pwr_alv} sec
]]--

--[[Nsample] ②
my_value
[Choose a value]
Something {2} units
99
-99
]]--

lcd.init(160) ③
ui.go()
ui.push('Mmain')
```

① The UI element for page 1 of `Mmain`

② The UI element for a number editor `Nsample`, with a 2 decimal-place value between -99 and +99

③ The initialisation sequence for an EA-DOG160XL display

UI Element : Menu

The prefix M is for menu. The UI core will try and access Mname1 to begin with, and pressing the DOWN button will sequence through the digits until no more are available in the file.

```
--[[Mname1] ①
Next     ②
  (OR)
/execfunc ②
  (OR)
.         ③
[Title]     ④
CommentText ⑤
Label {%.1f,my_value} ⑥
Label-{c.lua_app} {{%.1f,my_value}} Units ⑦
Label {*%.1f,my_value} Units ⑧
/funcname ⑨
]]-- ⑩
```

① The first menu item for level name

② The UI element to push, or Lua function to call, when SET is pressed.

③ Use a dot when there is no Next or function, and there is no [Title]

④ (optional) Title string

⑤ A line that is shown as a comment

⑥ A line that is shown with a label and right-justified small number

⑦ A line with label, formatted value, and units. Note the double { to allow rt.exp expansion of Label and Units

⑧ The * denotes a line with big formatted value.

⑨ A Lua function hook for the line

⑩ The end of the UI element.

Example

Menu with: * static text on page 1 that leads to a number editor * static text on page 2 that leads to a list picker * dynamic text on page 3 showing two values

```
--[[Mmain1]
Nvalue
[Edit value]
Press SET/RIGHT to edit the value.
]]--

--[[Mmain2]
Litems
[Choose list]
Press SET/RIGHT to choose from the list
]]--

--[[Mmain3]
.
[Display only]
Alive {*%.0f,i.pwr_alv}
Serial {i.rt_sn}
]]--
```

UI Element : Values with editing

A UI menu that has no `Next` or `/execfunc` can be used to have in-place-editing values, saving several separate pages and glue-code.

> If you accidentally assign a `Next` or `/execfunc` then the UI will never enter the edit mode. (Even if you have assigned the in-place editing write;dps;min;max values)

```
--[[Mname2] ①
.          ②
[Title]    ③
Label {%.1f,my_value;my_value;1;-10} ④
Label {%.1f,my_value;/WriteFunc;1;-100;200} Units ⑤
Label {*%.1f,my_value} Units ⑥
Jump {;>Ltest} ⑦
Call {;/CallFunc} ⑧
]]-- ⑨
```

① The second menu item for level `name`

② Use a dot when there is no `Next` or function, and there is no `[Title]`

③ (optional) Title string

④ Edit value: write to `my_value`; 1 decimal place; minimum=-10

⑤ Edit value: write with function `WriteFunc(VALUE)`; 1 decimal place; minimum=-100; maximum=200

⑥ View only big value.

⑦ A label that will jump to the UI element `Ltest`

⑧ Lua call to `CallFunc(uiname, uilevel)`

⑨ The end of the UI element.

> Using negative decimal places will increment/decrement in tens, hundreds, etc. e.g. -3 will use 1000 as a delta.

UI Element: Number Editor

The prefix `N` is for number-editor. The number editor can show an optional title, and specify optional maximum and minimum values.

```
--[[Nname] ①
Read ②
  (OR)
Read,Write ②
  (OR)
/readfunc,/writefunc ②
[Title]    ③
Label {dp} Units ④
Maximum ⑤
Minimum ⑥
]]-- ⑦
```

① The UI element name

② Either read; read+write variables. Can also be a Lua function name prefixed with `/`

③ (optional) Title string

④ Label, decimal places, and Units

⑤ (optional) Maximum value

⑥ (optional) Minimum value

⑦ End of the UI element.

Example 1

Number editor that reads `my_val` and shows to 1 decimal place. When saved, the variable `my_val` is updated:

```
--[[None]
my_val
[Edit my_val]
Value {1} cm
]]--
```

Example 2

Number editor that reads `my_val` and shows to 1 decimal place. When saved, the variable `other_val` is updated:

```
--[[Ntwo]
my_val,other_val
[Edit my_val, save other_val]
Value {1} cm
]]--
```

Example 3

Number editor that uses a function `getmyval` to obtain the value. When saved, the function `setmyval`is called to store the value:

```
--[[Nthree]
/getmyval,/setmyval
[Edit with max + min]
Value {1} cm
100
0
]]--

function getmyval()
 return my_val
end

function setmyval(v)
 my_val = v
end
```

UI Element: List Picker

The prefix `L` is for the list-picker.

```
--[[Lname] ①
Read ②
  (OR)
Read,Write ②
  (OR)
/readfunc,/writefunc ②
[Title]    ③
/listfunc  ④
  (OR)
item1 ④
item2
item3
etc
]]-- ⑤
```

① The UI element name

② Either read; read+write variables. Can also be a Lua function name prefixed with `/`

③ (optional) Title string

④ Either a Lua function name for a software list, or the list of items

⑤ End of the UI element.

Example1

List picker that reads and writes to the variable `yn_answer`:

```
--[[Lyesno]
yn_answer
[Decide]
No
Yes
]]--
```

Example 2

List picker that uses functions `getyn` and `setyn` to read and write the values. Also uses a Lua function to generate the list contents - in this example the list is taken from a Lua table.

```
--[[Lsoft]
/getyn,/setyn
[Decide]
/ynlist
]]--

thelist = {"No", "Yes"}

function getyn()
  return yn_answer
end

function setyn(v)
  yn_answer = v
end

function ynlist(n)
  if not n then return #thelist end
  return thelist[n]
end
```

UI Element: Settings and Hooks

The UI elements can be expanded with additional modifiers. Place these before the `[Title]` value.

`:hook:funcname`

Parameters

- `funcname`: Name of a Lua function

Description

Add a draw hook that is called at the end of each LCD update.

You can use this to draw a bitmap for the selected list index, etc.

Example draw hook:

```
function MyDrawHook(name,menuidx,value)
  --name    = UI name, e.g. 'Mtest1'
  --menuidx = UI index
  --value   = value in the UI.
  --          Menu or list index, or Number
end

--[[Mname1] ①
/dothis    ②
:hook:MyDrawHook  ③
:skey:GO!         ④
:skms:2000        ⑤
[Title]   ⑥
Label {%.1f,my_value}
Label {%.1f,my_value} Units
L-{c.somelabel} {{*%.1f,my_value}} {c.someunits}
]]-- ⑧
```

① The UI name

② Lua function call

③ Hook to display

④ The SET button will show `GO!`

⑤ You must hold the SET button down for 2 seconds

`:skey:text`

Parameters

- `text`: String to use for the `SET` key

Description

Override the key text. (See the example in `:hook:funcname`)

`:skms:time`

Parameters

- `time`: The time delay, in milliseconds

Description

Add a delay to the `SET` key. (See the example in `:hook:funcname`)

`ui._bar`

Description

Default `uiBar` handler for the UI.

`ui._key`

Description

Default `onKey` handler for the UI. This has been made available so the default handler can be intercepted and overridden.

`ui._cell`

Description

Default `uiCell` handler for the UI. Draws a large timer and big icons for the status of the cellular modem.

`ui.box(title, [y])`

Parameters

- `title`: Title text
- `y`: Top y-coordinate

Description

Draws the optional title box, and main box (according to the UI preferences set).

`ui.go()`

Parameters

None

Description

Start the UI. Applies the defaults, hooks the `onKey` callback, and begins the UI.

`ui.key(kn,txt,[tfon])`

Parameters

- `kn`: Key number, 1 to 4
- `txt`: String to display
- `tfon`: True if a text font should be used

Returns

None

Description

Draw a soft-key (useful in the display hook callback)

`name = ui.pop()`

Parameters

None

Returns

- `name`: Element name

Description

Pops a UI element off the stack.

`ui.popa([all])`

Parameters

- `all`: True to pop everything, false to pop everything except the top-level

Description

Pops all UI elements off the stack.

`ui.push(name)`

Parameters

- `name`: Element name

Description

Push a UI element onto the stack. The element is loaded from either the current Lua App, or to filename set with `ui.set.i8n`.

`ui.ssk(txt)`

Parameters

- `txt`: String to use for the `SET` key

Returns

None

Description

Set the text or icon for the `SET` key.

`name,val,index = ui.top()`

Parameters

None

Returns

- `name`: Name of the UI element
- `val`: Value of the stack
- `index`: Stack pointer for the element

Description

Accesses the top element on the UI stack.

`txt = ui.txt()`

Parameters

None

Returns

- `txt`: The text loaded by `app.txt`

Description

Provides access to the full text within the UI element.

`y = ui.val(y, label, value, units, [big])`

Parameters

- `y`: Y-position to draw the line
- `label`: The label text
- `value`: The value text
- `units`: The units text
- `big`: True to show large value text

Description

Draws a single row of label/value/units.

The formatting is different depending on the contents of `value` and `units`.

`ui.yes()`

Parameters

None

Returns

None

Description

Show the UI confirmation 'tick'.

Preference settings

The preferences can control the look and feel of the UI.

`tc,bc,kc = ui.set.c(tc,bc,kc)`

Parameters

- `tc`: Title colour
- `bc`: User box colour
- `kc`: Softkey colour

Returns

- `tc`: Title colour
- `bc`: User box colour
- `kc`: Softkey colour

Description

Control the colour of UI elements.

`fn = ui.set.i18n(filename)`

Parameters

- `filename`: Filename to load UI elements from

Returns

- `fn`: Current filename

Description

Set the internationalisation file. By default the UI elements are loaded from the current Lua App file.

> ❗ Make sure the chosen `filename` contains **all** the elements referred to in the App!

`v = ui.set.idle([n])`

Parameters

- `n`: Number of seconds the UI should show when idle

Returns

- `v`: Current value

Description

Sets, or gets, the current UI idle time.

Use a time of -1 to show the UI always.

```
ui.set.idle(-1) -- Show always
```

`ui.set.km(bk,uk,dk,sk)`

Parameters

- `bk`: Back key number (1)
- `uk`: Up key number (2)
- `dk`: Down key number (3)
- `sk`: Set key number (4)

Description

Override the Key Mapping. The soft-keys will be rearranged to match the key assignments.

Key 1 is on the left of the display; Key 4 is on the far right of the display.

```
ui.set.km(4,3,2,1) -- reverse
```

```
ui.set.km(1,3,2,4) -- swap UP / DOWN
```

`kr = ui.set.kr(kr)`

Parameters

- `kr`: Key repeat time

Returns

- `kr`: Key repeat time

Description

Set the Key Repeat value.

`tr,br,kr = ui.set.r(tr,br,kr)`

Parameters

- `tr`: Title round value (pixels)
- `br`: User box round value (pixels)
- `kr`: Softkey round value (pixels)

Returns

- `tr`: Title round value (pixels)
- `br`: User box round value (pixels)
- `kr`: Softkey round value (pixels)

Description

Control whether UI elements have rounded corners.

`ui.set.ui(xt, ft)`

Parameters

- `xt`: Table of X-positions
- `xt.t`: X-pos for Text
- `xt.l`: X-pos for Labels
- `xt.u`: X-pos for Units
- `xt.vl`: X-pos for left justified Values (no units)
- `xt.vr`: X-pos for right justified Values (with units)
- `ft`: Table of Font names
- `ft.t`: Font for Text
- `ft.l`: Font for Labels
- `ft.u`: Font for Units
- `ft.s`: Font for Value small size
- `ft.m`: Font for Value medium size
- `ft.b`: Font for Value big size

Description

Overrides one, or more, of the X-positions, and/or the fonts used for UI components.

❗ Apply settings **after** calling `ui.go`

UI Callbacks

`h = uiBar()`

Parameters

None

Returns

- `h`: Height of the status bar (the default `ui._bar` returns a height of 8 pixels)

Description

An override callback for the top line status bar.

`show = uiCell()`

Parameters

None

Returns

- `show`: Return `false` to prevent the call to `lcd.go`. Any other value will show the contents of the LCD buffer.

Description

An override callback for the cellular status LCD display.

The default `uiCell` function should be sufficient for most requirements.

> ℹ You do not have to call `lcd.go` - this is handled in the firmware.

`uiSmp()`

Parameters

None

Returns

None

Description

An override callback for the sample LCD display.

This callback is invoked if `smp.save` is called - allowing you to display important values from the last saved sample.

> You do not have to call `lcd.go` - this is handled in the firmware.

Configuration Variables

The rt.buffer stores the configuration variables inside an efficient C++ `key=value` memory structure that has about 6k of space.

The contents of this store are saved and read from the file `/Config/config.txt`

The `config.txt` file is updated 500ms after configuration changes have stopped happening. Conversely, editing config.txt through the USB HID interface will trigger a reload of the config.txt into RAM.

However, to make life much easier inside Lua, the values can be queried and modified directly as a Lua global table, called `c`.

Within the C++ structure, the keys are organised in ASCII order, with the underscore (_) character representing the branches of a tree. This method enables simple 'wildcard' style erasing and querying of sets of values, and is also compatible with Lua's syntax.

```
c.my_value = 123
c.my_other = 'A string'
c.my_more = 'A complex \r\n string\twith\x00controlcodes' ①

c.mynot = 'Not the same tree'
c.my_other=_ -- erases just one key-value
c.my=_ -- erases all c.my* key-values. But c.mynot remains
```

① See https://en.wikipedia.org/wiki/Escape_sequences_in_C for examples of escaped characters. e.g. '\t' = TAB, '\r' = CR, '\x00' = NULL, etc

A handful of configuration parameters are fixed, and used within the firmware. The Lua App can choose to make use of additional configuration parameters as needed.

The following sections list the firmware-fixed configuration settings.

Config c.alm : Alarm

```
c.alm_txt01 = ''
c.alm_txt02 = '' …
```

Description

Text names for the alarms 01-32.

ℹ (Developers) Can be retrieved in Lua with `alm.txt(…)` function.

Config `c.arc` : Archive Parameters

The `/Archive/` folder saves copies of the files that are sent. When the file count is exceeded, or the total bytes is exceeded, the oldest files are removed from the folder.

- The folder is flat - any files that do not begin with `/Send/` are mapped so that the `/` characters are replaced by `^` symbols.
  e.g. `/MyFolder/myfile.txt` is archived as `/Archive/`$^{\text{MyFolder}}$`myfile.txt`

If you create sub-folders within `/Archive/` then these folders will not be scanned, nor curated (i.e. they will stay there and not be pruned).

`c.arc_fc = 32` v1.00

Description

Maximum number of files to keep in the /Archive/ directory.

`c.arc_kb = 4096` v1.00

Description

Maximum number of kilo-bytes to keep in the /Archive/ directory.

Config `c.cell` : Cellular

`c.cell_apn = 'Internet'`

Description

APN name for the Internet connection

`c.cell_auth = -1`

Description

(optional) Override the APN authentication method

- `-1` = Auto (depending on `c.cell_user` + `c.cell_pass`)
- `0` = None
- `1` = PAP only
- `2` = CHAP only
- `3` = PAP/CHAP

`c.cell_hof = 10`

Description

(optional) hold off failure time, in minutes.

`c.cell_hos = 2`

Description

(optional) hold off success time, in minutes.

```
c.cell_mto = 15
```

Description

(optional) Maximum time online, minutes.

```
c.cell_pass = ''
```

Description

(optional) Password for the Internet connection

```
c.cell_pin = ''
```

Description

(optional) PIN number for the SIM card

```
c.cell_sms = ''
```

Description

(optional) Default SMS target number. Should be in international format (i.e. starting with "+" and the country code)

```
c.cell_user = ''
```

Description

(optional) Username for the Internet connection

Config `c.iot` : IoT Parameters

`c.iot_data = ''`

Description

(optional) Data process URL or path.

Uses Lua expansions.

`c.iot_gz = 1`

Description

(optional) Use GZIP compression and naming.

`c.iot_job = '@23:00'`

Description

(optional) Job string for Data delivery (and Update).

> (Developers) Sets both `onJob11` for Update, and `onJob12` for Data delivery, unless `c.iot_ujob` is set in which case `c.iot_job` does just data.

`c.iot_ntp = 'time.apple.com'`

Description

(optional) NTP server name.

`c.iot_tfr = 1`

Description

(optional) Use Temp File and Rename for FTP.

Set to 0 to disable this method and send the file directly.

> ℹ `c.iot_tfr=1` requires the user FTP account support renaming. However, this approach is **STRONGLY** recommended as it ensures files are completely transferred. Just make sure your back-end processes ignore `*.tmp` files (It is also a good idea to purge very old *.tmp files, e.g. over 1 month old.)

`c.iot_ujob = _`

Description

(optional) Update job string. If not present, then *c.iot_job* sets both update and data job strings.

`c.iot_upd = 'Update/{c.site_name}'`

Description

(optional) Update URL or path for the update mechanism, relative to c.iot_url (although can be an absolute URL too!)

Uses Lua expansions - see `rt.exp`.

`c.iot_url = ''`

Description

(required) Base URL for IOT.

> ℹ This uses Lua expansions.

> ⚠ Required!

```
c.iot_url = 'ftp://mp:pm@ftp.scannex.com/Upload/{c.site_name}-{i.rt_sn}'
```

`c.iot_user = ''`

Description

(optional) User process URL or path.

Uses Lua expansions.

`c.iot_var = 3600`

Description

Server variance (seconds).

Jobs that are scheduled can be skewed by a combination of c.iot_var and the serial number of the rt.buffer. This ensures that your server does not get 'hit' by many rt.buffer devices all at the same time, but are spread over time.

Config `c.lua` : Lua Configuration

`c.lua_app = ''`

Description

The App name to run.

> ℹ This configuration value is **required** for the rt.buffer to work. And the corresponding `.lua` app must already be in the `/Lua` folder.

Config `c.site` : Site Deployment

```
c.site_loc = ''
```

Description

Optional Site location details. Can be used by the App.

```
c.site_name = ''
```

Description

(optional) rt.buffer Site Name.

Used by the smp library for naming the files.

Config `c.term` : Terminal Controls

`c.term_epw = 1` v1.00

Description

(optional) Whether Engineer Serial Port needs a password.

If there is a dedicated MCU connected to the Engineer Serial Port, then this can be set to 0 so that no password is required (but it will be required from the USB still if `c.term_pass` is set).

`c.term_pass = ''`

Description

(optional) Terminal password.

Do NOT forget this password - there is no backdoor.

Config `c.tls` : Security Settings

`c.tls_csc = 0xffff` v1.00

Description

(optional) The modem cipher suite code for IoT connections

Recommend either `0x0035` or `0x002f` as a balance between security and server compatibility.

- `0x003D` = TLS_RSA_WITH_AES_256_CBC_SHA256 (Only possible when `c.tls_ver=3` since AES-256/SHA-256 is only available in TLSv1.2)
- `0x0035` = TLS_RSA_WITH_AES_256_CBC_SHA1
- `0x002F` = TLS_RSA_WITH_AES_128_CBC_SHA1
- `0x0005` = TLS_RSA_WITH_RC4_128_SHA1
- `0x0004` = TLS_RSA_WITH_RC4_128_MD5
- `0x000A` = TLS_RSA_WITH_3DES_EDE_CBC_SHA1
- `0xFFFF` = Support all ciphersuites above (default)

`c.tls_ilt = 1` v1.00

Description

(optional) Ignore Local Time [1: Checking the time against the certificate requires the local rt.buffer time be valid too!]

- 0 = Care about time checks for certificates
- 1 = Ignore time checks

`c.tls_ver = 3` v1.00

Description

(optional) TLS Version

- 0 = SSLv3 (not recommended)
- 1 = TLSv1.0 (not recommended)
- 2 = TLSv1.1
- 3 = TLSv1.2

Config `c.toa` : Terminal Over the Air settings

The Terminal Over the Air mechanism connects a TCP socket (or TCP+SSL) to an Internet server and provides a full link to the rt.buffer's terminal interface.

`c.toa_ito = 120` v1.10

Description

The idle timeout, in seconds, for the terminal.

If nothing is received from the server connection in this time, the rt.buffer will close the connection.

`c.toa_key = NIL` v1.10

Description

The shared secret key for HMAC-SHA-256 mutual authentication between the rt.buffer and the server.

This is for the link-level, so any terminal password will still be requested after the link has been established.

`c.toa_pass = NIL` v1.10

Description

The password override for the TOA terminal.

If this field is not present the c.term_pass password is used. This setting allows you to keep the passwords separate.

`c.toa_try = 300` v1.10

Description

The number of seconds to keep trying to make a TCP socket connection to the server. There is a 15 second pause between successive attempts.)

`c.toa_url = ''` v1.10

Description

The URL details for the TOA link.

TCP and TCPS (SSL) links are currently handled.

```
c.toa_url='tcp://matt@terminal.scannex.com:12345/Path/To/Resource'
```

This example will use a plain-text TCP client connection to `terminal.scannex.com` on TCP port 12345.

Once connected, a CONNECT header will be sent to the server with the `/Path/To/Resource` and `user: matt` fields.

Config `c.user` : User IoT settings

The user process is typically triggered by the magnet.

```
c.user_ins = 15
```

Description

The interval, in seconds, for the default User mode (by default: how many seconds to wait after sending the diagnostics data before sending again).

```
c.user_onm = 2
```

Description

The online time, in minutes, for the default User mode (by default: how many minutes to keep pushing the diagnostics data).

Information Variables

Like the Configuration table, the global table `i` provides a window into some internal C++ values (without consuming Lua memory).

However, there are two 'real' Lua tables available too - that can be used for passing back application-specific information. = Info `i.app.XXXXX` : Application

For developers.

This is also a real Lua table that can be used by the application to hand back information through the USB console, or through the diagnostic dumps, etc.

The `i.app` table can contain additional tables within it, and these will be handled correctly.

```
-- within App
i.app.mine = \{} -- blank table

-- later in the App
i.app.mine.v = 123
i.app.mine.too = 'Hello'
```

```
-- Within the USB terminal or via Update.txt:
lv i.app.mine
```

Info i.cell : Cellular Information

Modem activity

i.cell_pha v1.10 removed

Description

The current, or last, phase of the cellular activity. (If i.cell_sta is 'off' then this is the last phase).

- _ / NIL = idle
- 'update' = update process
- 'ntp' = time sync
- 'pass' = pass thru
- 'toa' = Terminal Over Air
- 'user' = User Process
- 'data' = Data Upload
- 'loop' = Loopback control
- 'hold' = Hold-off

i.cell_pwr

Description

Total number of seconds the modem has been powered.

`i.cell_sta`

Description

String containing the state of the modem:

- `'off'` = powered off
- `'pwr'` = powering on
- `'reg'` = registered
- `'online'` = PDP context
- `'fail'` = serious error
- `'?'` = unknown error

Modem timings

Cellular timings are split into the "Current" details, and the "Last" details - so you can see what the last transfer status was.

`i.cell_c_XXXX` has the current timings; `i.cell_l_XXXX` has the last timings.

`i.cell_X_csq` v1.03

Description

The signal quality indicator.

- 0-1 = nothing
- 2-9 = marginal
- 10-14 = ok
- 15-19 = good
- 20-31 = excellent
- 99 = unknown

i.cell_X_log v1.10

Description

Brief textual log of the modem activity and errors.

When the modem is powered up, the string starts with a colon ":".

Each job is listed. Any errors are listed with an exclamation mark "!" followed by the error.

Additional jobs are delimited by a comma "," and when the job list is complete the string is terminated with a full-stop "."

```
i.cell_c_log = ':ntp,update,test!FTP(625/530)!Connect.'
```

Shows that NTP time sync worked; the update process worked; but the test phase failed with an FTP 530 error (not logged in) - which failed the connection phase.

i.cell_X_od v1.03

Description

Online Duration. How many seconds the rt.buffer was online for.

Being online only implies an IP address anddoes not guarantee access to the wide Internet.

i.cell_X_pd v1.03

Description

Power Duration, in seconds. i.e. how many seconds the modem was powered up.

i.cell_X_pu v1.03

Description

Power UTC. The UTC time the modem was successfully powered up. (Power was applied about 5 seconds earlier.)

i.cell_X_rs v1.03

Description

Registration Seconds. How many seconds it took to get registered on the cellular network.

i.cell_X_sc v1.03

Description

Server Connected. Whether the IoT server was connected.

- 0=no server(s) connection
- 1=at least one server connection made

Info `i.fw` : Firmware Information

`i.fw_blv`

Description

BootLoader version

`i.fw_date`

Description

Firmware date

`i.fw_desc`

Description

Firmware name

`i.fw_ver`

Description

Firmware version

Info `i.iot` : IoT Information

`i.iot_var`

Description

The actual number of seconds this rt.buffer will have variance-enabled schedules delayed.

Info `i.lua` : Lua Information

`i.lua_run`

Description

The seconds-alive value when the Lua App was started.

Zero means there is no Lua loop running - either onLoop has quit, or has not been defined.

Info `i.rt` : rt.buffer Information

`i.rt_cbf`

Description

Number of configuration store bytes free.

`i.rt_sn`

Description

Serial number string

`i.rt_utc`

Description

The current time.

Info `i.smp.XXXX` : Samples

This is a real Lua table that is used for the activities of the Lua App's Sample (smp) library. By default, any record that is saved by the Lua App will update these info entries.

These parameters are only updated when the default smp table (ft) is used. When other tables are used - e.g. when handling multiple sample files - these info variables are not updated.

`i.smp.cnt`

Description

The number of times that smp.save has been called.

This counter is reset if the rt.buffer is cold-booted, watchdog restarted, or Lua reboots.

`i.smp.txt`

Description

The last text that was written with smp.save

`i.smp.utc`

Description

The UTC time that smp.save was last called

Info `i.pwr` : Power Information

`i.pwr_alv`

Description

Seconds alive.

`i.pwr_ccb`

Description

Battery capacity, in C.

`i.pwr_ccu`

Description

Estimated charge used

`i.pwr_pct` v1.00

Description

Estimated percentage power remaining (0-99%)

`i.pwr_slp`

Description

Number of seconds spent asleep (in ultra-low power mode).

`i.pwr_tob`

Description

Number of seconds running on battery.

Terminal Commands

These commands apply to the `rt.buffer>` prompt that is available through rtbTool via USB, serial, or TCP Terminal-Over-the-Air.

Terminal: Internal Debug Commands

These commands are not guaranteed to remain, and the format may change. They are primarily for Scannex development and testing.

`led`

Description

Show LED status.

`mv`

Description

Show the millivolt readings for power, etc.

`ninv`

Description

Show the sense inputs on the COM ports.

`pmm`

Description

Show power manager modes.

`wde`
`wde0`

Description

Show watchdog entries.

`wde0` will reset the maximum counters.

```
work-
work+
```

Description

Decrement or increment the counter for hard-work.

If the counter is >0 then the CPU runs at 96MHz and will not enter low power mode.

Terminal: External Devices

adc

Description

Show the ADC readings.

pcr

Description

Pulse Counter Reset.

pcv

Description

Show Pulse Counter Values.

pt[@]d [STOP]

Parameters

- `@`: Include the '@' symbol to pass the characters directly to the terminal. Without the '@' then control characters are escaped in Javascript/Lua/C style.
- `STOP`: Optional Lua expression to alter the stop sequence. Normally the [ESC] key quits the pass through mode.

Description

Enter pass-through mode for the Device COM port.

Use the `pt@d` format when using software to communicate directly through rtbAPI.dll or USB-HID control.

Without the `@` symbol e.g. a NULL will be escaped in ASCII as '\x00', an ACK character as '\x06' etc. See https://en.wikipedia.org/wiki/Escape_sequences_in_C - but `\r`, `\n`, and `\t` are sent as real characters CR, LF, TAB.

Although the STOP can be a string sequence, the terminal is not fully buffered. If you are using a keyboard, then just use a single character!

```
ptd
pt@d
ptd '!'
ptd '\x1b'
```

pt[@]e [STOP]

Parameters

(as for ptd above)

Description

Enter pass-through mode for the Engineer COM port.

Terminal: General Functions

boot
reboot

Description

Reboot the rt.buffer immediately. If USB is connected, the rt.buffer will remain in the Boot Loader for 5 seconds (unless you type a boot loader command).`

deepsleep

Description

Turn 'off' the rt.buffer.

> You can only wake up by reconnecting the Engineer serial port or USB for more than 1 second.

info

Description

Show information about the firmware.

temp

Description

Show temperature.

```
time
time YYYY-MM-DD HH:MM:SS [TZ]
```

Parameters

- `YYYY`: Year
- `MM`: Month
- `DD`: Day
- `HH`: Hour
- `MM`: Minute
- `SS`: Second
- `TZ`: (Optional)timezone offset, in minutes

Description

Show the current time information, or set and show the time.

Terminal: Lua Core

`lboot`

Description

Reboot Lua.

If the modem is active, the actual reboot will occur once Lua is unlocked.

`lgc`
`lgc0`

Description

Garbage collect Lua memory and show stats.

`lgc0` will show and reset the statistics.

`lm`
`lm0`

Description

Show Lua memory stats.

`lm0` will show and reset the statistics.

`lv LIST`
`?LIST` v1.00

Parameters

- `LIST`: comma separated list of values

Description

Show Lua value tree(s), or specific values.

=CMD v1.00 NOTE: `=CMD` is effectively the short-cut for `lx return CMD`

```
lx iot.go()
lx return iot.flg()
/return rt.ms()
```

`c.KEY = VALUE`

Parameters

- `KEY`: Configuration key name
- `VALUE`: Value in Lua format.

Description

Set a configuration value.

```
c.iot_url='ftp://ftp.scannex.com/Path/To/Success'
```

Can also be used to wipe out a setting, or a tree of settings, using the Lua 'nil':

```
c.iot_url=nil
c.iot=nil
```

`lf FILE`

Parameters

- `FILE`: Filename to run

Description

Execute a Lua file.

ℹ If FILE does not include a `/` character, then `/Lua/` is assumed.

ℹ If FILE does not include a `.` character, then `.lua` is assumed.

Terminal: NAND Flash

`nfcheck`

Description

Check the flash disk.

`nfdel FILE`

Parameters

- `FILE`: Full path and filename

Description

Delete a file from flash disk.

`nfdir PATH`

Parameters

- `PATH`: Path of the folder to view.

Description

Show the folder listing.

ℹ The path separator character is forward-slash `/` (Like UNIX/Linux, not Windows!)

`nfformat`

Description

Erase EVERYTHING.

There is no way to undo this command! You WILL lose everything in the rt.buffer

nfinfo

Description

Show information about the flash drive.

nfmkdir PATH

Parameters

- PATH: Full path

Description

Make a folder, and intermediaries.

```
nfmkdir /Temp/Dir/Here
```

nfmove SOURCE TARGET

Parameters

- SOURCE: Source path + filename
- TARGET: Target path + filename

Description

Moves a file or folder.

nfrename SOURCE TARGET

Parameters

- SOURCE: Source path + filename
- TARGET: Target filename (no path!)

Description

Rename a file.

`nfrmdir PATH`

Parameters

- `PATH`: Full path

Description

Remove a folder. Will fail if the folder is not empty

`nftype FILE`

Parameters

- `FILE`: Full filename

Description

Shows the contents of a file.

`nfwipedir PATH`

Parameters

- `PATH`: Full path

Description

Wipe a folder, including all its contents.

> There is no way to undo this! May take a long while to execute, especially if there are lots of files.

nfwrite FILE

Parameters

- FILE: Path and filename

Description

Write to a file. (Used internally by the rtbTool GUI app and rtbAPI tools)

BootLoader Terminal Commands

These USB-HID terminal commands apply to the `bootloader>` prompt.

```
boot
reboot
```

Description

Reboot the rt.buffer.

```
deepsleep
```

Description

Turn 'off' the rt.buffer.

You can only wake up by reconnecting the Engineer serial port or USB for more than 1 second.

```
eraseall
```

Description

Erase the whole NAND flash - erasing all Lua Apps, settings, and stored data.

The firmware and bootloader are not affected.

> ⚠ You will lose everything that was stored. Each block of NAND flash is physically erased. This is useful if the data was sensitive and the device needs to be shipped.

`erasefw`

Description

Erase the current application firmware. Only the bootloader will remain.

This will not affect any stored NAND flash data.

`info`

Description

Show information about the rt.buffer, bootloader, and application firmware.

`run`

Description

Leave the bootloader and run the install firmware.

LED Sequences

The single, ultra-bright, amber LED gives an indication of the state of the rt.buffer. LED sequences consist of pairs of flashes.

Do not stare directly into the LED - it is *very* bright (especially if the top lid is off)!

In the list below, e.g. "12" = one flash + pause + two flashes

- Status
  - 11 = Alive (every 10 seconds)
    - If the rt.buffer is in ultra-low power mode the LED will be very dim, otherwise you’ll see the '11' sequence but at normal brightness.
  - 12 = Data has been stored
- Engineer Port
  - 22 = Engineer USB connected
  - 23 = Engineer USB in use
  - 25 = Magnet triggered
- Modem
  - 31 = Modem powering up
  - 32 = Modem registered
  - 33 = Modem online
- Boot Loader
  - 41 = Boot Loader active
  - 42 = USB running
  - 44 = Upgrading firmware
  - 45 = No Application BLF
- Faults
  - 54 = Debug
  - 55 = Fault

Example Lua App

To illustrate how to start coding a Lua App, consider an app that logs digital pulse counts.

> This example uses the v2.00+ `coro` coroutine library.

The Main Sample

```
function GetSample()
 local p1,p2 = dpc.rst()
 local t = {p1,p2}
 smp.save( smp.csv(t) )
end
```

At the moment, the function will not be called.

So, we need to glue in a job schedule, and define a job:

```
function onJob1()
 coro.add(GetSample)
end

job.set(1, '00:01')
```

Keeping Tally

Let's now extend the app to keep a total, where: `total = total + p1 - p2`

```
pt = 0 -- the total ①

function GetSample()
 local p1,p2 = dpc.rst()
 pt = pt + p1 - p2 ②
 local t = {p1,p2,pt} ③
 smp.save( smp.csv(t) )
end
```

① `pt` has the running total

② The value is updated

③ We create a temporary table to convert to CSV

Now, you can see the temporary table `t` also has the value of `pt` logged.

Linking the Schedule into the Configuration

Let’s make the job setting more flexible by linking into the config table.

You’ll see we’ll make use of Lua’s "or" - that takes the default if there is no config value defined (i.e. it’s NIL):

```
job.set(1, c.myapp_job or '00:01')
```

Updating the Schedule Immediately

At the moment, the schedule is only assigned when Lua reboots (or the rt.buffer starts up). To apply the job settings whenever the configuration changes, we need to put the job.set function inside an onConfig handler function:

```
function onConfig()
 job.set(1, c.myapp_job or '00:01')
end
```

Adding Diagnostic Information

To illustrate making life easier to debug, we can feed back some information back into the i.app table. Let’s just add the `t` table by adding the highlighted line:

```
local t = {p1,p2,pct}
i.app.t = t ①
smp.save( smp.csv(t) )
```

① The whole table is saved for examination in `i.app.t`

Adding Pressure Readings

It’s not difficult to add the pressure reading as well:

```
function GetSample()
 local p1,p2 = dpc.rst()
 pt = pt + p1 - p2
 local pv = rt.adc() ①
 local t = {p1,p2,pt,pv} ②
 i.app.t = t
 smp.save( smp.csv(t) )
end
```

① Reads the ADC value into `pv`

② Adds it to the table for conversion to CSV

Adding ADC Scaling

Now, let's add `y=mx+c` scaling, by linking the `rt.adc` command into the values: `c.adc_m` and `c.adc_c`

```
local pv = rt.adc(c.adc_m or 1, c.adc_c or 0)
```

Again, you can see the default values being used.

Adding file headers and footers

Here, we can add a header that consists of all c & i values, with a blank CR/LF. In addition, the `ft` assignment makes use of the built-in MD5 hash generator to add an ASCII hash at the end of the file:

```
function Hdr(tb,h,nm)
 fc.lv(h, 'c,i', '\r\n')
end
ft = {h=Hdr,f=smp.t_md5}
```

Full Listing

```
--------------------------
-- Demo App
-- Logs: time,count1,count2,total,adc
--------------------------
pt = 0 -- the total

function Hdr(tb,h,nm)
 fc.lv(h, 'c,i', '\r\n')
end
ft = {h=Hdr,f=smp.t_md5}

function GetSample()
 local p1,p2 = dpc.rst()
 pt = pt + p1 - p2
 local pv = rt.adc(c.adc_m or 1, c.adc_c or 0)
 local t = {p1,p2,pt,pv}
 i.app.t = t
 smp.save( smp.csv(t) )
end

function onJob1()
 coro.add(GetSample)
end

function onConfig()
 job.set(1, c.myapp_job or '00:01')
end
```

Configuration settings can be controlled through:

```
c.adc_m = 1 -- Slope
c.adc_c = 0 -- Offset
c.myapp_job = '00:01' -- Job string for logging
```

Software Licenses

Operating System

- Segger embOS
- Segger emFile
- Segger emUSBD

Portions:

```
(c) 2014 - 2016 SEGGER Microcontroller GmbH & Co. KG
www.segger.com Support: support@segger.com
```

Cortex Support Libraries (CMSIS)

```
 Copyright (c) 2009 - 2015 ARM LIMITED

 All rights reserved.

 Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
 - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
 - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
 - Neither the name of ARM nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
 INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL COPYRIGHT HOLDERS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
 OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Atmel Files

Portions of Atmel’s ASF have been used:

```
Copyright (c) 2012-2015 Atmel Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

3. The name of Atmel may not be used to endorse or promote products derived
from this software without specific prior written permission.

4. This software may only be redistributed and used in connection with an
Atmel microcontroller product.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Lua License

Lua is licensed under the terms of the MIT license reproduced below.

This means that Lua is free software and can be used for both academic and commercial purposes at absolutely no cost.

For details and rationale, see http://www.lua.org/license.html .

```
Copyright (C) 1994-2008 Lua.org, PUC-Rio.

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.
```

(end of COPYRIGHT)

Libstruct Library

eLua additions

Some portions of the eLua project have been used:

```
Copyright (c) 2007, 2008, 2009, 2010, 2011 Dado Sutter and Bogdan Marinescu

eLua is Open Source and is freely distributed under the MIT licence.

The Lua part of eLua is licensed under the Lua licensing terms, which
you can find at http://www.lua.org/license.html.

The "pack" module is adapted from the "lpack" module by Luiz Henrique de
Figueiredo and it's placed in the public domain.

The "bit" module is adapted from the "bitlib" library by Reuben Thomas,
distributed under a MIT license.

The multiple memory allocator (dlmalloc) is written by Doug Lea and is
placed on the public domain.

Manufacturer provided CPU support libraries are licensed under their own
terms, check src/platform/*platform-name* for details.
```

The rest of the eLua code is licensed under MIT, listed below: The MIT License

```
Copyright (c) 2007, 2008, 2009, 2010 Dado Sutter and Bogdan Marinescu

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.
```

zlib Compression Library

```
zlib.h -- interface of the 'zlib' general purpose compression library
 version 1.2.11, January 15th, 2017

Copyright (C) 1995-2017 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu

The data format used by the zlib library is described by RFCs (Request for
Comments) 1950 to 1952 in the files http://tools.ietf.org/html/rfc1950
(zlib format), rfc1951 (deflate format) and rfc1952 (gzip format).
```

Crypto Hash Functions

Portions of mbedTLS have been used:

U8G2 LCD Library

https://github.com/olikraus/u8g2

```
The U8g2lib code (http://code.google.com/p/u8g2/) is licensed under the terms of
the new-bsd license (two-clause bsd license).
See also: http://www.opensource.org/licenses/bsd-license.php

The repository and optionally the releases contain icons, which are
derived from the WPZOOM Developer Icon Set:
http://www.wpzoom.com/wpzoom/new-freebie-wpzoom-developer-icon-set-154-free-icons/
WPZOOM Developer Icon Set by WPZOOM is licensed under a Creative Commons
Attribution-ShareAlike 3.0 Unported License.

Fonts are licensed under different conditions.
See
  https://github.com/olikraus/u8g2/wiki/fntgrp
for detailed information on the licensing conditions for each font.

============ X11 Fonts COUR, HELV, NCEN, TIM, SYMB ============

For fonts derived from the following files, the license below applies.
COURB08.BDF COURB10.BDF COURB12.BDF COURB14.BDF COURB18.BDF
COURB24.BDF COURR08.BDF COURR10.BDF COURR12.BDF COURR14.BDF
COURR18.BDF COURR24.BDF HELVB08.BDF HELVB10.BDF HELVB12.BDF HELVB14.BDF
HELVB18.BDF HELVB24.BDF HELVR08.BDF HELVR10.BDF HELVR12.BDF HELVR14.BDF
HELVR18.BDF HELVR24.BDF NCENB08.BDF NCENB10.BDF NCENB12.BDF
NCENB14.BDF NCENB18.BDF NCENB24.BDF NCENR08.BDF NCENR10.BDF
NCENR12.BDF NCENR14.BDF NCENR18.BDF NCENR24.BDF SYMB08.BDF SYMB10.BDF
SYMB12.BDF SYMB14.BDF SYMB18.BDF SYMB24.BDF TIMB08.BDF TIMB10.BDF
TIMB12.BDF TIMB14.BDF TIMB18.BDF TIMB24.BDF TIMR08.BDF TIMR10.BDF
TIMR12.BDF TIMR14.BDF TIMR18.BDF TIMR24.BDF

Copyright 1984-1989, 1994 Adobe Systems Incorporated.
Copyright 1988, 1994 Digital Equipment Corporation.

Adobe is a trademark of Adobe Systems Incorporated which may be
registered in certain jurisdictions.
Permission to use these trademarks is hereby granted only in
association with the images described in this file.

Permission to use, copy, modify, distribute and sell this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notices appear in all
copies and that both those copyright notices and this permission
notice appear in supporting documentation, and that the names of
```

```
Adobe Systems and Digital Equipment Corporation not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  Adobe Systems and
Digital Equipment Corporation make no representations about the
suitability of this software for any purpose.  It is provided "as
is" without express or implied warranty.


============ BSD License for U8g2lib Code ============

Universal 8bit Graphics Library (http://code.google.com/p/u8g2/)

Copyright (c) 2016, olikraus@gmail.com
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list
  of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this
  list of conditions and the following disclaimer in the documentation and/or other
  materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Index