scannex

ip.buffer OEM and Lua Manual

1st March 2021

Firmware Version 3.01

ip.buffer OEM and Lua Manual

Date	*Author*	*Release*
2008-05-09	MP	For version 2.00
…		
2014-05-02	MP	For version 2.90
2014-10-23	MP	For version 2.91
2015-10-15	MP	For version 2.92
2016-04-28	MP	For version 2.93
2017-10-03	MP	For version 2.94
2018-03-29	MP	For version 2.95
2021-03-01	MP	For version 3.01

Scannex Electronics Ltd, UK
t: +44(0)1273 715460
f: +44(0)1273 715469

http://www.scannex.co.uk
info@scannex.co.uk

Scannex LLC, USA
t: 1-866-4BUFFER
(1-866-428-3337)

http://www.scannex.com
info@scannex.com

Table of Contents

1. Introduction

The ip.buffer product includes facilities for OEM (Original Equipment Manufacturer) settings and extensive Lua scripting. This manual outlines the technical details for each of the ip.buffer's extensions.

2. OEM Facilities

2.1. Overview

The OEM facilities included in the ip.buffer provide:

- The ability to customize the page footer in all web pages of the ip.buffer.
- The ability to add Lua scripting features that enhance your total solution. These scripts may:
  - analyze the incoming data
  - provide extra features beyond the factory shipped product
  - include extra protocols for devices
  - The script may also be encrypted using a write-only secret so you can safely update the OEM script in the field.
- The ability to provide a vendor certificate and key for the SSL/PKI infrastructure

2.2. Accessing the OEM page

The OEM web page is "loosely" hidden.

- Simply use SEDiscover, or access the ip.buffer in any normal way through a web-browser.
- Change the end of the url to "oem". e.g. "http://192.168.0.235/status.shtm" should be changed to "http://192.168.0.235/oem"
- You should see the OEM web page appear.

2.3. Set OEM Key

The OEM key is a hexadecimal secret that is used to decrypt any encrypted OEM scripts. You will have to decide what is the best balance of security vs inconvenience:

- You can use a single key for all buffers you ship.
  - This is convenient. It means that any scripts you produce can only be executed by ip.buffers that have been pre-loaded with your secret key.
  - A customer cannot obtain an ip.buffer from another source and run your encrypted OEM scripts!
  - However, if the key is leaked then potentially all of your customer base is affected.
- You can use a single key for each of your customers.
  - If you load custom encrypted OEM scripts into the ip.buffers then each of your customers has to have a separately encrypted version.
  - Scripts from your customer A will not run on customer B
  - The supply chain is protected as above.
  - However, if one customer key is leaked, then the others remain protected.

- You can use a different key for each buffer you ship.
  - Each buffer requires a separately encrypted version of your OEM script.
  - However, you have complete control over who runs what script.

> - Above all, PROTECT your OEM key! (The same way you don't leave the keys to the office doors lying around)
> - Don't lose the OEM key! Once written, no-one can read out the key from the ip.buffer!

You can obtain cryptographically random hexadecimal bytes from www.random.org

If you intend to use the OEM key, you should follow the procedures:

- Load the OEM key *before* you ship to the customer (i.e. Don't type it in on-site. The customer's PC may cache the password, or record it).
- If a box needs the OEM key to be updated or added, **use the modem dial-in** and SSL communications.
- Once encrypted, OEM scripts can be emailed safely. You can instruct your customer in the process of uploading the new script through the OEM web-pages. *They can never read the OEM key!* (In fact, nor can anyone else!)

2.4. Script: Edit

If the OEM script is not encrypted, then you (or your customer) can edit the OEM script using this web page.

The OEM script is treated identically to the normal script that is loaded in the “normal” setup pages. However, scripts are loaded in the following order:

1. Internal initialisation scripts & protocols
2. OEM script
3. “Normal” script
4. Configuration

Within the OEM script you have the opportunity to declare functions, routines, and variables that can be referenced within the “normal” script. Since the OEM script can also be encrypted, you can effectively hide any IP (Intellectual Property) that you develop, while still allowing for a great deal of flexibility.

For example, you may write a complex script to provide basic fraud detection algorithms. This script may have taken a lot of time and effort on your part – so you encrypt it. However, you may want to reference values that you would like the user to have control over. Your OEM script can define the functions and the default variable settings, while you can instruct your customers how to add overrides within their “normal” script.

- If a script is encrypted, you cannot see the source code, and a red warning appears on the Script: Edit page.

2.5. Script: Download

If the script is unencrypted, you can download the OEM script to a file. If you left-click on the link you will see the script. If you right-click and choose “Save Target As” within your web-browser then you can save to disk.

This is helpful when developing a script in-house. The downloaded script can then be encrypted using Scannex's script encrypter.

2.6. Script: Upload

If you have a script on disk – either in source code or encrypted – you can upload it to the ip.buffer through this link.

If the file is encrypted, the ip.buffer must have a matching OEM key in order to decrypt and run the script. A mismatch in OEM key will be shown as an error on the status web-page.

2.7. Upload Certificate and Private Key

The ip.buffer ships with three SSL/PKI certificates:

1. A Scannex Root Certificate
2. The “ip.buffer” range certificate
3. The individual device certificate

The lowest certificate – the “individual device certificate” - can be generated from the next certificate up using the setup web-pages.

However, in protecting your supply chain (or in other special circumstances), you may wish to break the chain-of trust with Scannex and establish your own.

For example, you can use a self-signed certificate that is identified with you[1]. You upload the Certificate and the associated Private Key in the ip.buffer web-page. Now, when you generate the “individual device certificate”, it will be signed by your certificate (and not Scannex's).

You can then ship just the Certificate to your customer – and their machines will identify all ip.buffers shipped by you (which have been preloaded with your certificate and key).

Additionally, if you have written software to communicate with the ip.buffer directly then you can check the chain of trust back to you – and again protect your supply chain. For example, if you “FTP pull” the data out with FTP+SSL, then the ip.buffer will provide the certificates when you connect. At this point you can check whether the ip.buffer was genuinely signed by your certificate.

[1] You could also purchase a certificate from Verisign, Thawte, etc – but this is usually “rented” and the certificate expires after one or two years. You would have to update the certificates in the field whenever your purchased certificate expires. However, 99% of user browsers are already configured to accept the commercially generated certificates.

3. Changing the footer on the web-pages

The ip.buffer has a feature where the "scannex" logo and URL link at the foot of each web-page can be customized with basic HTML text (graphics are not supported).

This is how to do it:

- Go to the OEM Script Edit page
- Type in, or add, the following script lines:

```
oem = {}
oem.advert = {}
oem.advert.text = "<font size=5>My Company Text</font>"
oem.advert.url = "http://www.mycompany.com"
```

- Save the script and reboot Lua when asked.

Now every page has a new footer!

You can use any HTML text to "prettify" your text, or provide a telephone number for example[2]. e.g. "`oem.advert.text= "<font size=5>Acme</font><br>For support call 12345678"`

This script "snippet" could be part of a larger script that you encrypt using your OEM key.

[2] There is currently a limit of 1024 (was 128) characters for the oem.advert.text field.

4. Configuration Lua Table

Everything that defines the setup of the ip.buffer is stored in the configuration Lua table. This table is visible when editing the ip.buffer configuration via the web, and when viewing the diagnostic information table in the “Tools” page.

The Lua variable is “c”, and defines the following table tree of information:

4.1. Global Configuration

4.1.1. Base Config: c

`c.firmware='IPBSSL2.30.92 2008-10-20 / i5.0.10'`

Effectively a read-only value. The full text of the firmware version.

`c.version=2.3`

Effectively a read-only value. The version number as a number.

4.1.2. Globals: c.globals

- These options are not normally visible - each needs to be manually input.

`c.globals.hdaplocal='enabled'` 2.82

(Optional) Controls whether the 169.254.x.x additional IP address support is enabled for HDAP/SEDiscover.
'`disabled`' = no local IP address extensions supported
'`window`' = only allow the setting of local IP addresses during the 5 minute window (pressing the front button will open up another 5 minute window)

`c.globals.laxwebpassthru=0` 2.82

(Optional) Set to 1 to allow access to TLS enabled Pass-Through sockets via an unencrypted http:// link.

`c.globals.liverecordcount=10` 2.82

(Optional) Sets the number of live records saved and displayed. Limited between 1 and 200.

`c.globals.onsecondtimeout=10` 2.82

(Optional) Sets the Lua timeout for the "`x.onsecond`" event. Set to '`0`' to disable the timeout completely, allowing arbitrarily long execution.

`c.globals.nosourcepassthru=0` 2.81. Deprecated 2.91

(Optional) Set to '1' to allow pass through connections even when the TCP source is not connected.

`c.globals.passthrusrcoff='!SOURCE NOT CONNECTED!\r\n'` 2.92

(Optional) The message to send to the passthrough socket when the source disconnects (not used for Linked mode).

`c.globals.passthrusrcon='!SOURCE IS CONNECTED!\r\n'` 2.92

(Optional) The message to send to the passthrough socket when the source connects (not used for Linked mode).

`c.globals.passthrutimeout=180` 2.92

(Optional) The number of seconds to allow before closing the pass-through connection when the source is not connected.

`c.globals.sftpauthmethods='publickey,password,keyboard'`

(Optional). Set the allowable SFTP push authentication methods (order unimportant) e.g. to use only publickey, set `c.globals.sftpauthmethods='publickey'`

`c.globals.snmp.delim=' '` 2.76

(Optional). The delimiter character between SNMP trap decoding.

`c.globals.snmp.escape=1` 2.76

(Optional). Set to zero to use the old format with no string escaping

`c.globals.snmp.vardelim='\t'` 2.76

(Optional). The delimiter character between SNMP trap var bind fields.

`c.globals.sourceusesproxy=1` 2.92

(Optional). Set to zero to disable proxy use on TCP active source connections.

`c.globals.tcptimebeforedrain=4`

(Optional). The number of seconds between "draining[3]" the TCP socket of data when using a TCP data delivery with "Stay Connected".
A value of 0 (zero) will send chunks of about 2k at a time, ensuring the data has arrived at the remote end before deleting. However, bandwidth will be severely impacted.
When the storage only has a small chunk of data (e.g. a single call record) it will always be "drained" before erasing.

[3] The ip.buffer TCP stacks has the ability to 'know' how much data is in transit across the network, and what data has been positively acknowledged by the remote TCP stacks. Note that this only indicates that the operating system stacks has the data – the application hasn't necessarily read and processed the data (so it can still lose data if it does not handle the socket handle correctly).

4.1.3. LAN Config: c.network

`c.network.bandwidth=0`
: The bandwidth limit function. In kilo-bytes per second.

`c.network.dhcp=0`
: Whether DHCP or fixed IP. 0=fixed IP, 1=DHCP

`c.network.dns[#]='192.168.0.1'`
: The IP address of up to two DNS servers. (#=1,2)

`c.network.dns.cachesize=10` 2.70
: An optional setting that can be used to change the cache size of the DNS module in the ip.buffer. Allowed: between 1 and 1024. If not present, the value 10 is used internally.

`c.network.extra[#].ip=''`
: The multi-homing IP addresses. (#=1,2)

`c.network.extra[#].subnet='255.255.255.0'`
: The multi-homing subnet (#=1,2)

`c.network.gateway='192.168.0.1'`
: The IP address of the gateway.

`c.network.ip='192.168.0.235'`
: The IP address of the ip.buffer

`c.network.mtu=1500` 2.60
: The MTU value for the Ethernet interface. If not present then 1500 is assumed.

`c.network.name='ipbuffer'`
: The name of the device, as appears on the web-page and in emails.

`c.network.snmptrap='255.255.255.255'`
: The IP address of the device that will receive SNMP traps from the ip.buffer. “0.0.0.0” will stop transmission of all traps. “255.255.255.255” will broadcast to the local LAN segment.
Can also be a list of up to sixteen addresses separated by comman, space, tab, or semicolon[4].

`c.network.snmptrapintf=''` 2.82
: Specify the interface to send SNMP traps over. Options are:
'`ppp`' or '`mod...`' = send over the modem (for nailed-up connections)
'`lo...`' = send over the loop back interface (for collection by a source)
'`et...`' = use Ethernet LAN
If blank, the traps will be routed and sent over the most appropriate interface.

`c.network.subnet='255.255.255.0'`
: The subnet of the ip.buffer.

`c.network.syslog.name=''`
: The IP address or name of the syslog server.

[4] Firmware 2.82+

4.1.4. LAN Proxy Configuration: c.proxy

`c.proxy.httpbasicfirst=0` 2.80

(optional) Whether to default to using Basic HTTP Authentication first. May save an extra HTTP transaction to the proxy server.

`c.proxy.type=''` 2.60

The type of proxy server - "`http`", "`socks5`", or "`socks4a`"

`c.proxy.name=''` 2.60

The name, or dotted IP address of the proxy server.

`c.proxy.port=''` 2.60

The TCP/IP port the proxy server uses. Usually 8080 for HTTP, or 1080 for SOCKS.

`c.proxy.user=''` 2.60

The username – if the proxy server requires authentication.

`c.proxy.pass=''` 2.60

The password – if the proxy server requires authentication.

`c.proxy.noproxy=''` 2.60

The list of addresses (explicit or wildcarded) to not proxy for. Separate with a comma, semicolon, or space. e.g. "`192.168.*, *.scannex.com`"

4.1.5. SNMP Agent Config: c.snmp

`c.snmp.community='public'` 2.70

The community name for all SNMP-related activities[5].

`c.snmp.contact=''`

The name, email address, or number of the person responsible for this device. Reported to the SNMP client through the sys OID group.

`c.snmp.location=''`

The location of this device. Reported to the SNMP client through the sys OID group.

`c.snmp.name=''`

The sysName of this device.
If blank this will return "`ip.buffer-00-02-ae-xx-xx-xx`"

`c.snmp.port=161`

The port for the SNMP agent. Zero will disable the agent completely.

4.1.6. RADIUS Config: c.radius

`c.radius.ftp='L'`

The authentication method for the FTP Server Delivery. "L" = Local only, "R" = RADIUS only, "T" = RADIUS then Local on timeout, "B" = RADIUS and Local.

`c.radius.nasid=''`

The NAS-Identifier that is sent in each RADIUS request.
If blank, this will be the string "`00-02-ae-xx-xx-xx`" (i.e. now an override value)[6]

`c.radius.nasip=''`

The NAS-IP-Address value that is sent in each RADIUS request. If the value is blank then the NAS-IP-Address value is not included in the request.

`c.radius.passthru='L'`

The authentication method for the pass-through sockets. "L" = Local only, "R" = RADIUS only, "T" = RADIUS then Local on timeout, "B" = RADIUS and Local.

`c.radius.servers[#].name=''`

The IP address, or name, of the RADIUS server. If this `c.radius.servers[1].name` is blank then RADIUS is disabled.

`c.radius.servers[#].port=1812`

The UDP port number for the server. Common values are 1645 and 1812.

`c.radius.servers[#].secret='********'`

The shared secret for this server. Note that this value is always write-only, and you will only ever read eight asterisks.

[5] This includes: the SNMP Agent, the SNMP Trap client, SNMP trap reception (only traps matching the community are decoded), and the SNMP trap-query mechanism for collection.

[6] This now allows configurations to be easily duplicated between different ip.buffers. However, be aware that copying a 2.75+ NAS ID value will clear the NAS ID for a pre 2.75 firmware buffer (because earlier versions are not an 'override' but a direct value).

`c.radius.tcp='L'`

The authentication method for the TCP Server Delivery. “L” = Local only, “R” = RADIUS only, “T” = RADIUS then Local on timeout, “B” = RADIUS and Local.

`c.radius.timeout=2`

The timeout (in seconds) for the RADIUS servers. Each RADIUS server is tried twice before trying the next.

`c.radius.web='L'`

The authentication method for the web-server. “L” = Local only, “R” = RADIUS only, “T” = RADIUS then Local on timeout, “B” = RADIUS and Local.

4.1.7. Certificate Global Options: c.certs

`c.certs.allowed=''`
: The list of SHA1 fingerprints that are allowed. Each fingerprint should be separated by a CR/LF ('\r\n'). The fingerprint line may be just a fingerprint, or may include a descriptive name, e.g.
"`collect.scannex.com=8e:52:81:63:7b:06:a6:d4:8b:ef:d1:0a:03:05:be:2d:54:0d:74:88`"

`c.certs.clients=0`
: Whether to verify clients against the approved fingerprint list. 0=ignore, 1=verify.

`c.certs.date=0`
: Whether to check the date validity of server and client certificates. 0=ignore, 1=verify.

`c.certs.ignorecerterrors=0` 2.91
: (Optional) Whether to ignore errors while parsing and checking the signing of the certificate chain. These checks are performed before checking the fingerprint. You SHOULD only include fingerprints for the device.[7]

`c.certs.name=0`
: Whether to check that the address matches the CN (Common Name) field for server certificates. 0=ignore, 1=verify.

`c.certs.servers=0`
: Whether to check server certificates against the approved list of fingerprints. 0=ignore, 1=verify.

`c.certs.source=0` 2.91
: (Optional) By default Source-side certificates (both client & server) are not validated. This is because some devices have weakly protected private keys and can be compromised. However, if you can trust the source as much as the destination, you can set this to "1" to apply the same checks to source.

`c.certs.cbcsplit=1` 2.93
: Whether to perform AES-CBC record splitting 1/n-1 for server operations.

`c.certs.ciphers='normal'` 2.80
: (Optional) Descriptor to restrict cipher suites used by TLS/SSL for server operations. See User Manual

`c.certs.keymin=1024` 2.92
: (Optional) Specify the minimum peer RSA key size. 512/1024/2048

`c.certs.signhash=''` 2.92
: (Optional) Override the signature hashes presented during TLS, and allowed in the peer's TLS certificates.

[7] Do NOT include fingerprints for CAs when ignoring certificate errors, as the ip.buffer will (wrongly) approve a certificate chain that is not correctly linked but includes a fingerprint for a CA.

`c.certs.sslmin=0`
`c.certs.sslmax=3` 2.91

(Optional) The minimum & maximum TLS/SSL version to accept for server operations.
Values:
0 = SSLv3
1 = TLSv1.0
2 = TLSv1.1
3 = TLSv1.2

`c.certs.client.cbcsplit=0` 3.01

Whether to perform AES-CBC record splitting 1/n-1 for client operations.

`c.certs.client.ciphers=''` 3.01

(Optional) Descriptor to restrict cipher suites used by TLS/SSL for client operations.

`c.certs.client.keymin=1024` 3.01

(Optional) For client sockets - Specify the minimum peer RSA key size.
512/1024/2048

`c.certs.client.signhash=''` 3.01

(Optional) For client sockets - Override the signature hashes presented during TLS, and allowed in the peer's TLS certificates.

`c.certs.client.sslmin=2`
`c.certs.client.sslmax=3` 3.01

(Optional) The minimum & maximum TLS/SSL version. *(See c.certs.sslmin & c.certs.sslmax for value.)*

`c.certs.src.cbcsplit=0` 2.93

Whether to perform AES-CBC record splitting 1/n-1

`c.certs.src.ciphers=''` 2.92

(Optional) Descriptor to restrict cipher suites used by TLS/SSL.

`c.certs.src.keymin=512` 2.92

(Optional) For source sockets - Specify the minimum peer RSA key size.
512/1024/2048

`c.certs.src.signhash=''` 2.92

(Optional) For source sockets - Override the signature hashes presented during TLS, and allowed in the peer's TLS certificates.

`c.certs.src.sslmin=0`
`c.certs.src.sslmax=3` 2.93

(Optional) The minimum & maximum TLS/SSL version to accept for source connections. *(See c.certs.sslmin & c.certs.sslmax for value.)*

4.1.8. Battery Config: c.battery

`c.battery.ethernet='a'` 2.80
Ethernet power while on batteries. 'a'=always, 'n'=off

`c.battery.limit=60` 2.80
The maximum time to run on batteries. Specified in minutes.

`c.battery.modem='a'` 2.80
Modem power while on batteries. 'a'=always, 'd'=on demand, 'n'=off

4.1.9. Modem In & Out Config: c.modem

When a PSTN modem is detected, the GPRS-specific settings will not appear. Conversely, if an EDGE or GPRS modem is detected, the PSTN-specific settings will not appear.[8]

`c.modem.alternate=0`
Specifies the order to dial ISP#1 and ISP#2. 0=Try ISP#1 then #2 on fail. 1=Alternate between ISP#1 and ISP#2.

`c.modem.answertime=120`
The maximum time, in seconds, to answer, negotiate, and get a connection.

`c.modem.chap=1`
Whether to use PAP/CHAP or just CHAP when authenticating incoming calls. 1=CHAP only, 0=PAP/CHAP

`c.modem.connect='od'` 2.76
How to connect for out-bound connections.
'od' = On Demand, 'nu' = Nailed-Up (always connected, no firewall).

`c.modem.connecttime=150`
The maximum time in seconds to dial, negotiate, and get a connection.

`c.modem.country='B5'`
The country code of the modem.

`c.modem.dialoutmax=15`
The maximum time to spend online when dialling out - in minutes.

`c.modem.dialtype='T'`
The dial type. “T”=Tone, “P”=Pulse.

`c.modem.exclude.hi='1900'`
The dial-out exclusion end time. 24 hour clock format, just digits.

`c.modem.exclude.lo='1800'`
The dial-out exclusion start time. 24 hour clock format, just digits.

`c.modem.failreset=3` 2.90
The number of failures in a row before resetting a nailed-up connection.

`c.modem.gprs.apn=''`
The GPRS Access Point Name string.

[8] Applies to firmware 2.70+

`c.modem.gprs.band=31`
: The frequency bands to use. +1=GSM-850, +2=P-GSM-900 (primary), +4=E-GSM-900 (extended), +8=DCS-1800, +16=PCS-1900

`c.modem.gprs.delay=0`
: An optional setting. Specifies the number of seconds the ip.buffer will wait before communicating with the GPRS modem after a power-cycle or reset. Allowed values are 0 to 120 seconds.

`c.modem.gprs.pass=''`
: The GPRS password (if required).

`c.modem.gprs.pin=''`
: The SIM PIN number (if required). The value is always hidden and will be shown as “********”. If blank, there is no PIN set.

`c.modem.gprs.rstint=300`
: An optional setting. Specifies the number of seconds to wait for the modem to register before resetting. The default is to reset the modem every 5 minutes while not registered with the local network cell.

`c.modem.gprs.user=''`
: The GPRS username (if required).

`c.modem.holdoff=240`
: The time, in seconds, to hold off between dialling, when dialling fails.

`c.modem.init=''`
: Any extra initialisation commands to send to the modem. When using this feature, exclude the initial “AT”.

`c.modem.interval=60`
: The time to pause, in seconds, after hanging up after a dial-out process.

`c.modem.ip=''`
: The default IP address for the ip.buffer end of the PPP connection. Blank = Computer IP address + 1.

`c.modem.isp[#].pass=''`
: The password to use when authenticating PPP with the ISP. (#=1,2)

`c.modem.isp[#].tel=''`
: The telephone number for the ISP. (#=1,2)

`c.modem.isp[#].user=''`
: The username to use when authenticating PPP with the ISP. (#=1,2)

`c.modem.mtu=1460` 2.60
: The MTU for the PPP interface. If not present then 1460 is assumed.

`c.modem.pass='password'`
: The dial-in PPP password.

`c.modem.recycletime=360` 2.90
: The number of PPP-quiet minutes before resetting a nailed-up connection.

`c.modem.remoteip='192.168.234.1'`
: The IP address for the computer if the computer connects with auto IP enabled.

`c.modem.retry=3`

The number of times to try dialling before entering the “holdoff” time.

`c.modem.ringcount=2`

The number of times to ring before answering the call.

`c.modem.user='user'`

The dial-in PPP username.

4.1.10. Email Service Config: c.smtp[#]

`c.smtp[#].address=''`
The IP address, or name, of the SMTP server.

`c.smtp[#].domain=''`
The domain name to log in with.

`c.smtp[#].intf='L'`
The interface to connect to this SMTP server on. The substring “L” denotes “LAN”, while “M” denotes “Modem”. e.g. “LM” = LAN first, then Modem on fail. e.g. “ML” = Modem first, then LAN on fail. e.g. “M” = modem only.

`c.smtp[#].limit=1024`
The data limit for this SMTP server. Specified in kilobytes.

`c.smtp[#].name='SMTP1'`
The display name of the SMTP service.

`c.smtp[#].pass=''`
The password to use – if authentication is needed for the SMTP server.

`c.smtp[#].port=25`
The TCP/IP port number to connect to the SMTP server on.

`c.smtp[#].ssl=''`
Whether to connect with SSL encryption. “e” = explicit SSL (i.e. Connect plain first then negotiate up), “i” = implicit SSL (i.e. The listening port on the SMTP server must negotiate SSL first). Blank = plain.

`c.smtp[#].user=''`
The username to use – if authentication is needed for the SMTP server.

4.1.11. Time updates: c.sntp

`c.sntp.address='time.nist.gov'`
The name or IP address of the SNTP (Simple Network Time Protocol) server

`c.sntp.dst=''`
e.g. '-1;0;2;0200;-1;0;9;0200'
The Daylight Savings Time values. Consists of semicolon delimited values: Start Week (-1 = last, 1=1st, 2=2nd, 3=3rd, 4=4th); Start Day of Week (0=Sunday, 1=Monday, etc); Start Month (0=Jan, 1=Feb, 2=Mar, etc); Start Time (24hr); End Week; End Day of Week; End Month; End Time. (The example shows “Last Sunday in March at 2am to Last Sunday in October at 2am”.)

`c.sntp.intf='L'`
Which interface(s) and order to try contacting the SNTP server. “L”=LAN, “M”=Modem. e.g. “LM” = try LAN first, then Modem if LAN fails.

`c.sntp.time=0200`
The time of day to contact the SNTP server.

`c.sntp.tz=0`
The timezone (in hours past GMT).

`c.sntp.update=1`
A write-only value. If you add this value to the configuration tree, then the SNTP service will immediately contact the SNTP server and update the time.

`c.sntp.vari=0` 2.70
The variance time, in minutes.

4.1.12. Email Alerts Config: c.alerts

`c.alerts.auth=0`
Whether to send an email on a failed authentication. 0=don't send, 1=send.

`c.alerts.comfort=0`
Whether to send “comfort” email alerts. Specified in minutes. 0=don't send.

`c.alerts.config=0`
Whether to send an email when the ip.buffer is configured through the web-interface. 0=don't send, 1=send.

`c.alerts.eastext=0` 2.92
(Optional) Set to 1 to send the alerts as paragraph text, not table

`c.alerts.emailto=''`
The list of email addresses to email to. Multiple addresses can be separated by a semicolon.

`c.alerts.eskiptable=0` 2.92
(Optional) Set to 1 to skip the information table.

`c.alerts.etitle='<h1>Alerts</h1>'` 2.92
(Optional) Sets the title of the alerts list

`c.alerts.full=75`
: The percentage global memory to send an email alert. Individual channel alerts are found at `c.chnl[#].notify.full`

`c.alerts.http.server=_` 2.93
: The HTTP Cloud Server to use. NULL (_) is the default. 2=Cloud 2, 3=Cloud 3

`c.alerts.method='none'`
: How to send alerts. Either 'smtp' , 'http', or 'none'.

`c.alerts.reboot=0`
: Whether to send an email on reboot and battery switchover. 0=don't send, 1=send.

`c.alerts.sched.days='1234567'`
: Which days of the week to send quiet alerts. 1=Mon, 2=Tue, 3=Wed, 4=Thu, 5=Fri, 6=Sat, 7=Sun. If the day's number is not present, then the email is not sent on that day.

`c.alerts.sched.hi=2359`
: The end time for quiet alerts to be sent. In 24 hour format - digits only.

`c.alerts.sched.lo=0`
: The start time for quiet alerts to be sent. In 24 hour format - digits only.

`c.alerts.sendinfo=1` 2.60
: Whether to send the info dump along with the alert.

`c.alerts.server=2`
: Which SMTP server to use for sending email alerts. 1=SMTP#1, 2=SMTP#2.

`c.alerts.temp.hi=30`
: The high temperature alert, in Celcius (integer only). 0=disabled.

`c.alerts.temp.lo=0`
: The low temperature alert, in Celcius (integer only). 0=disabled.

4.1.13. FTP Server Config: c.ftps

`c.ftps.allow=''` 2.75
: The name, address, wildcard or list to allow (ie. .passive=1) Blank = all devices.

`c.ftps.intf='LM'`
: Which interfaces to allow FTP server connections on. “L”=LAN, “M”=Modem.

`c.ftps.passivehi=50099`
: The high port number to use for FTP PASV transfers.

`c.ftps.passivelo=50000`
: The low port number to use for FTP PASV transfers.

`c.ftps.port=21`
: The TCP/IP port number for the FTP server

`c.ftps.ssl=''`
: Whether to use SSL encryption on the FTP server. “e”=explicit SSL (ie. Start plain, and then negotiate to upgrade), “i”=implicit SSL (ie. Start with SSL on a special port number).

4.1.14. Cloud Server: c.cloud

`c.cloud.delay=300`
The time, in seconds, to wait before trying again after a failure

`c.cloud.intf='L'`
Which interfaces to use for the HTTP Post. “L”=LAN, “M”=Modem.

`c.cloud.manage=''`
An optional value to decide which server should manage the ip.buffer. Set to 'scannex' to enable Scannex's support servers to manage the ip.buffer.

`c.cloud.ring=0`
Whether to perform an update on modem ring. 1 = update, 0 = ignore

`c.cloud.updatedelay=120 (optional)` 2.90
(optional) Sets the number of seconds to delay after rebooting before connecting to the Cloud Server.

`c.cloud.url=''`
The full URL of the central HTTP Cloud Server.
e.g. “http://cloud.ipbuffer.com/upload”

Cloud Server Optional Settings

`c.cloud.httpbasicfirst=0`
Whether to default to using Basic HTTP Authentication first. (Not secure unless over htttps!)

`c.cloud.maxhttpchunk=2000` 2.80
The maximum number of bytes for a chunked POST. Allowable values range between 64 bytes and 8192 bytes.

Additional Cloud Servers

`c.cloud[2].url=''` 2.93
`c.cloud[3].url=''` 2.93
The full URL of the HTTP Cloud Server.
e.g. “http://cloud.ipbuffer.com/upload”

`c.cloud[2].intf='L'` 2.93
`c.cloud[3].intf='L'` 2.93
Which interfaces to use for the HTTP Post. “L”=LAN, “M”=Modem.

4.1.15. Web Server Config: c.web

`c.web.refresh.liveon=1` 2.91
(optional) Decides the default live web page automatic refresh. 1=refresh, 0=stop

`c.web.refresh.live=5000` 2.91
The interval for refreshing the live record web page, in milliseconds.

`c.web.refresh.statuson=1` 2.91
(optional) Decides the default status web page automatic refresh. 1=refresh, 0=stop

`c.web.refresh.status=5000` 2.91
The interval for refreshing the status web page, in milliseconds.

`c.web.setup.basic=0`
Whether to allow Basic Authorization as well as Digest (MD5). 0=Digest only, 1=Digest+Basic.

`c.web.setup.hide=0`
Whether to hide the local passwords. 0=visible, 1=obscured[9].

`c.web.setup.http=1`
Whether to allow HTTP+HTTPS or HTTPS only. 0=HTTPS only, 1=HTTP+HTTPS.

`c.web.setup.lock=0`
Whether to use Scannex “WebLock” for setup pages. 0=Unlocked, 1=WebLock

`c.web.setup.pass='secret'`
The password for setup and tools web pages.

`c.web.setup.user='admin'`
The username for setup and tools web pages.

`c.web.setup.window=0`
Whether to allow a 5 minute window on reboot. 0=Authenticate on reboot, 1=Allow 5 minutes window (where any username/password can be used).

`c.web.status.pass=''` 2.70
The password for the status web page.

`c.web.status.user=''` 2.70
The username for status web page. If this is blank then no authentication is done for the status page[10].

[9] Once obscured, the passwords cannot be un-obscured until they are re-programmed.

[10] If RADIUS authentication is enabled, then the status page is only checked against RADIUS if the username is set in c.web.status.user

4.1.16. Debug Output: c.debug

`c.debug.filter=''` 2.70

(An optional setting) Which debug streams to send. Enter a list of strings: "`modem`", "`ppp`", "`log`", "`user`". If blank, all streams are sent.

`c.debug.ip=''` 2.70

(An optional setting) Enter a dotted IP address (e.g. 255.255.255.255) and the ip.buffer will output UDP debug information to the Scannex debug server.

`c.debug.port=56023` 2.70

The UDP port to send the debug information on.

4.2. User Lua placehold: c.lua

`c.lua={}` 2.95

The c.lua table can be used for storing Lua script settings.

- This is a 'flat' table. You cannot include sub-tables!

e.g.

```
c.lua.mine = 1234
c.lua.another = 'Testing'
```

4.3. Channel Config[11]: c.chnl[#]

`c.chnl[#].name='Channel1'`
The descriptive name of the channel.

4.4. Source Configuration

4.4.1. Source Config: c.chnl[#].src

`c.chnl[#].src.type='com'`
The source type for the channel. Allowed values are: “com” = COM port/serial, “tcp” = TCP, “udp” = UDP, “ftps” = FTP server collection.

4.4.2. Serial Source Config: c.chnl[#].src.com

`c.chnl[#].src.com.autobaud=1`
Whether autobauding is enabled. 0=fixed baud rate, 1=autobaud enabled.

`c.chnl[#].src.com.baud=115200`
The baud rate. 300,600,1200,2400,4800,9600,19200,38400,57600,115200

`c.chnl[#].src.com.loopback=0`
Whether to use the diagnostic loopback hardware. 0=normal, 1=serial loopback.

`c.chnl[#].src.com.passbreak=0` 2.90
The serial break to issue when pass-through connects, in milliseconds. 0=disabled. Maximum=2000

`c.chnl[#].src.com.passhand=''`
Which handshake line(s) to assert only when the pass-through socket is connected. Like .rxflow, the substrings “rts” and “dtr” apply.

`c.chnl[#].src.com.passin=''` 2.92
Which handshake line(s) to check for source being “connected” when using pass-through. Like .txflow, the substrings “cts” and “dsr” apply[12].

`c.chnl[#].src.com.protocol='8N'`
The serial protocol. Comprises the data length in bits, and the parity format. Available values: 7N, 7E, 7O, 8N, 8E, 8O

`c.chnl[#].src.com.rxflow='rts'`
Which handshake line(s) to use for receive flow control. The substring “rts” denotes RTS, while “dtr” denotes DTR. e.g. “rtsdtr” = RTS & DTR flow control.

`c.chnl[#].src.com.rxpin=0`
Which pin to receive from. 0=Auto, 2=DTE, 3=DCE, -1=diagnostics

`c.chnl[#].src.com.txchunk=16`
The number of bytes to queue up for transmission to the serial port. Smaller values increase CPU load but allow the ip.buffer to stop transmitting sooner when the device signals “don't send”.

[11] # is the channel number - 1,2,3,4

[12] If set to “ctsdsr” then both must be unasserted to consider the COM-port not-connected.

`c.chnl[#].src.com.txflow='cts'`

Which handshake line(s) to check for transmission flow control. The substring “cts” denotes the CTS line, while “dsr” denotes DSR. e.g. “ctsdsr” = CTS & DSR flow control.

`c.chnl[#].src.com.txpause=0`

The number of *bits* pause to insert between each transmitted character. A value of 1 will simulate 2-stop-bits, while a value of 10 will introduce a complete character pause (useful for slow-to-respond devices).

4.4.3. TCP Source Config: c.chnl[#].src.tcp

`c.chnl[#].src.tcp.address=''`
The address or name to connect to when using an active connection (ie. .passive=0)

`c.chnl[#].src.tcp.allow=''`
The name, address, wildcard or list to allow when using a passive connection (ie. .passive=1) Blank = all devices.

`c.chnl[#].src.tcp.heartbeat.interval=0`
The interval, in seconds, to send the heartbeat string back to the device. 0=send nothing.

`c.chnl[#].src.tcp.heartbeat.string=''`
The string to send on the heartbeat interval. The same special characters as for .match[x] apply.

`c.chnl[#].src.tcp.match[x]=''`
The match string (x=1,2,3,4). Special characters: #=CR/LF, $=0x00, /hh or {hh} for hex. So, “{E1A2}” will wait for the hex characters 0xe1, 0xa2.

`c.chnl[#].src.tcp.passive=1`
Specifies which direction the TCP/IP socket should connect. 1=Device-to-ip.buffer (passive), 0=ip.buffer-to-device (active).

`c.chnl[#].src.tcp.port=2001`
Which TCP/IP port to connect to, or listen on.

`c.chnl[#].src.tcp.send[x]=''`
The send string (x=1,2,3,4) that is sent after the match is made. The same special characters as for .match[x] apply.

`c.chnl[#].src.tcp.ssl=''`
Whether to use SSL/TLS encryption on the source TCP socket. “i”=implicit SSL/TLS

4.4.4. UDP Source Config: c.chnl[#].src.udp

`c.chnl[#].src.udp.address=''`
The address to allow incoming UDP packets from. Blank = all device.

`c.chnl[#].src.udp.packet='A'`[13]
The packet decoding type. “A”=ASCII+CR/LF, “B”=pure binary, “L”=Length+Binary

`c.chnl[#].src.udp.port=2001`
The UDP/IP port number to listen on. Special consideration is given internally to the pre-defined ports 514 (syslog) and 162 (snmp trap).

`c.chnl[#].src.udp.probe=600`
The SNMP GET probe interval, in seconds.

`c.chnl[#].src.udp.snmp=''`
The list of addresses to SNMP GET probe. The entries will be in the form “address=name” (or “address”) with “\r\n” separating the individual lines.

[13] Version <= 2.75 defaulted to 'B'. Version >= 2.76 defaults to 'A'

4.4.5. FTP Server Source Config: c.chnl[#].src.ftps

`c.chnl[#].src.ftps.markers=1`
Whether to add STOR file markers. 0=no markers, 1={ftp...}, 2={ftp...}+CRLF

`c.chnl[#].src.ftps.pass='password'`
The password for the username.

`c.chnl[#].src.ftps.timeout=0`
The inactivity timeout, in minutes, before disconnecting the FTP user.

`c.chnl[#].src.ftps.user='_channel1'`
The username that links the FTP Server collection to this channel.

4.4.6. Cloud Server Source Config: c.chnl[#].src.cloud

`c.chnl[#].src.cloud.markers=1` 2.92
Whether to add markers. 0=no markers, 1={cloud...}, 2={cloud...}+CRLF

4.4.7. Source Protocol Config: c.chnl[#].src.protocol

The settings for the protocol engine of the channel.

`c.chnl[#].src.protocol.name='ascii'`
The protocol name for the channel. Built in protocols include:
“`alcatel`” = Alcatel TCP port 2533
“`ascii`”=ASCII lines
“`avaya`”=Avaya RSP
“`bcmlive`” = Nortel BCM Live CDR
“`binary`”=binary handling
“`fdcrip`”= Philips FDCR protocol (Sopho iS3000 etc)
“`genericrecs`” = Generic Records
“`intertel`” = Inter-Tel/Mitel Axxess & 5000
“`isdx`”=Realitis/iSDX binary
“`nec`” = NEC NEAX 2400 serial
“`necip`” = NEC NEAX 2000/2400 over TCP
“`nortel`”=Nortel Meridian and Norstar
“`udp`”=Internal UDP ASCII Decoded packets
“`udpbin`”=Internal UDP binary packets
Script-added protocols will have a prefix of “*”. So, a protocol added for the NEC NEAX2400 may be “*neax”

`c.chnl[#].src.protocol.params='xon=1;codes=0;timeout=1000'`
A semicolon-delimited list of parameters for the protocol. The protocol can define extra settings that are displayed within the web-page. These parameters are parsed and populate the “p” table that is passed to each of the protocol function calls. See section 13.1.5.

`c.chnl[#].src.protocol.prefix=''`
The prefix string for time-stamping the records. Note that binary record types (iSDX & binary) will ignore this setting (the web-page disables the entry for binary-based protocols).

4.4.8. Pass-through Config: c.chnl[#].src.pass

The pass-through setting for the channel.

`c.chnl[#].src.pass.actauth=0` 2.92
: Set to '1' to make an active passthru connection perform wait/respond authentication.

`c.chnl[#].src.pass.address=''` 2.92
: The name or IP address to connect to when using client/active mode

`c.chnl[#].src.pass.allow=''`
: The name, IP address, or wildcard IP address, or list to allow connection to the pass-through socket.

`c.chnl[#].src.pass.client='A'`
: Which client/software type. “A”=auto, “T”=Telnet, “R”=raw TCP/IP.

`c.chnl[#].src.pass.intf='L'`
: Which interface(s) to allow pass-through. “L” denotes LAN, while “M” denotes modem.

`c.chnl[#].src.pass.logic=''` 2.92
: (Optional) Overrides the default logic behaviour of connection, closure, and data for the passthru. Options are one or more of the following groups of letters:
  - Connection:
    - `S` = Source first
    - `P` = Passthru first
    - `B` = Both
  - Closure:
    - `C` = Close passthru when source closes
    - `O` = Keep passthru open even when source closes
  - Data:
    - `K` = Keep data in the source while the passthru opens
    - `D` = Discard data from source until passthru completes the open
  - Source Closure:
    - `L` = cLose source when the passthru closes
    - `E` = kEep source open when the passthru closes

`c.chnl[#].src.pass.mode='N'`
: The pass-through mode. “N”=Not stored, “S”=stored, “M”=monitor, “D”=debug.

`c.chnl[#].src.pass.password='password'`
: The password for the pass-through socket. Blank=no password checking.

`c.chnl[#].src.pass.port=2101`
: The TCP/IP port number to listen on.

`c.chnl[#].src.pass.prompt='Password:'` 2.30
: The prompt for the pass-through connection.

`c.chnl[#].src.pass.ssl=''`
: Whether to use implicit SSL. “i”=implicit SSL, blank=plain connection.

`c.chnl[#].src.pass.successmsg='OK\r\n'` 2.30

(Optional) The string to send to the client when the password is correct.

4.4.9. Channel Email Notification Config: c.chnl[#].notify

`c.chnl[#].notify.connect=0`

Whether to send an email alert (and SNMP trap) when this channel's source connects and disconnects. 0=don't send, 1=send.

`c.chnl[#].notify.full=0`

Whether to send an email alert (and SNMP trap) when this channel's memory exceeds this value - specified in megabytes. 0=disabled.

`c.chnl[#].notify.quiet=0`

Whether to send an email alert (and SNMP trap) when this channel is quiet. Specified in minutes. 0=don't send, 1=send.

4.5. Channel Memory Config: c.chnl[#].mem

Channel memory configuration.

`c.chnl[#].mem.enc=0`
: Whether deliveries of this channel should be encrypted with the 40-bit Scannex cipher. 0=plain, 1=encrypted[14].

`c.chnl[#].mem.hi=20`
: The “maximum” value, in megabytes, for this channel.

`c.chnl[#].mem.linear=0`
: Whether old or new data should be erased when full. 0=Erase old data, 1=Stop Storing (dump new data).

`c.chnl[#].mem.lo=0`
: The “reserved” value, in megabytes, for this channel.

4.6. Delivery Configuration

4.6.1. Channel Delivery Config: c.chnl[#].dst

`c.chnl[#].dst.prefix=''`
: The string prefix to output when delivering this channel's data. Applies to all delivery methods.

`c.chnl[#].dst.suffix=''`
: The string suffix to output when delivering this channel's data. Applies to all delivery methods except TCP with “Stay Connected”.

`c.chnl[#].dst.type='ftps'`
: The delivery method to use. Available values are: “`smtp`”=email push, “`ftps`”=FTP Server, “`ftpc`”=FTP Push, “`tcps`”=TCP Server, “`tcpc`”=TCP Push, “`emul`”=Legacy emulation[15], “`none`”=Pass-through only.

[14] On the web-pages this value is replicated on each of the delivery types, rather than located under the “Storage” section.

[15] Only applicable when the “`emulation.connect`” function has been assigned in script. See section 8.11.

4.6.2. Email Delivery Config: c.chnl[#].dst.smtp

`c.chnl[#].dst.smtp.compress=0`
Whether to zlib compress the data. 0=no compression, 1=zlib deflate

`c.chnl[#].dst.smtp.emailto=''`
The semicolon delimited list of email addresses (or one address) to send the data to.

`c.chnl[#].dst.smtp.filename='channel1.dat'`
The filename to use for attaching the data to the email.
Special filenames are '`<body>`' and '`<body>SUBJECTLINETEXT`' that trigger the ip.buffer to send the data within the body and not as an attached file[16].

`c.chnl[#].dst.smtp.inlinetitle='Data'` 2.82
(Optional) Changes the title used when data is embedded within the body of the email (with the `.dst.smtp.filename='<body>'` etc)

`c.chnl[#].dst.smtp.sendinfo=1`
Set to 0 to disable sending the Info attachment along with the email.

`c.chnl[#].dst.smtp.server=1`
Which SMTP service to use. Either 1 or 2.

`c.chnl[#].dst.smtp.skiptable=0` 2.82
(Optional) Set this entry to '`1`' to skip the output of the summary table in the email. Use this along with the special `.dst.smtp.filename='<body>'` or '`<body>SubjectLine`' for sending the data in the main body of the email.

4.6.3. FTP Server Delivery Config: c.chnl[#].dst.ftps

`c.chnl[#].dst.ftps.autodel=0`
Whether to auto-delete the file after download. 0=no delete, 1=delete.

`c.chnl[#].dst.ftps.compress=0`
Whether to zlib compress the data. 0=no compression[17], 1=zlib deflate.

`c.chnl[#].dst.ftps.filename='channel1.dat'`
The filename to show to the FTP client.

`c.chnl[#].dst.ftps.limit=0`
Whether to limit the FTP transfer size, specified in kilobytes. 0=no limit[18].

`c.chnl[#].dst.ftps.pass='password'`
The password for the FTP server for this channel.

`c.chnl[#].dst.ftps.user='channel1'`
The username for the FTP server for this channel.

[16] Firmware 2.82+

[17] The FTP client can still request the filename with a ".zlib" extension. In this case the FTP server will dynamically use zlib deflate.

[18] The FTP client can override, or request, a transfer limit by using the "LIMIT size" command. e.g. "LIMIT 100" will limit the transfer to 100k. (Most FTP clients can "quote" the command direct to the ip.buffer.)

4.6.4. FTP Push Config: c.chnl[#].dst.ftpc

`c.chnl[#].dst.ftpc.address=''`
The name or IP address of the FTP server to push to.

`c.chnl[#].dst.ftpc.compress=0`
Whether to zlib deflate the file. 0=normal, 1=zlib deflate with ".zlib" extension.

`c.chnl[#].dst.ftpc.dir=''`
The directory to push the file into. Blank = home directory. e.g. "/mysite/"[19]

`c.chnl[#].dst.ftpc.filename='channel1.dat'`
The filename to use when pushing.

`c.chnl[#].dst.ftpc.intf='L'`
Which interface(s) and order to try doing the FTP push. "L"=LAN, "M"=Modem. e.g. "LM" = try LAN first, then Modem if LAN fails.

`c.chnl[#].dst.ftpc.limit=0`
The data limit for the FTP push transfer. Specified in kilobytes.

`c.chnl[#].dst.ftpc.mode='rename'` `(moved here 2.80)`
The transfer mode. 'rename' = Tmp File & Rename; 'create' = Overwrite; 'append' = Append.

`c.chnl[#].dst.ftpc.pass='password'`
The password to authenticate with.

`c.chnl[#].dst.ftpc.port=21`
Which TCP/IP port to connect to the server on.

`c.chnl[#].dst.ftpc.ssl=''`
Whether to use SSL for FTP push. Blank=plain, "e"=explicit SSL (start plain and negotiate up), "i"=implicit SSL (server's TCP port should be configured for SSL only), "sftp"=SFTP over SSH.

`c.chnl[#].dst.ftpc.user='username'`
The username to authenticate with.

[19] Use forward slashes "/", not backslashes "\" !!

4.6.5. HTTP Push Config: c.chnl[#].dst.httpc[20]

`c.chnl[#].dst.httpc.filename='channel1.dat'`
The filename to use when pushing.

`c.chnl[#].dst.httpc.server=_` 2.93
The HTTP Cloud Server to use. NULL (_) is the default. 2=Cloud 2, 3=Cloud 3

All other settings are covered under c.cloud global settings (see section 4.1.14)

4.6.6. TCP Server Delivery Config: c.chnl[#].dst.tcps

`c.chnl[#].dst.tcps.allow=''`
The name, IP address, wildcard IP address, or list to allow connection. Blank = any device.

`c.chnl[#].dst.tcps.intf='LM'`
Which interface(s) to allow TCP Server connection. The substring “L” denotes LAN, while “M” denotes Modem.

`c.chnl[#].dst.tcps.password='password'`
The password to request before releasing data. Blank = no password check.

`c.chnl[#].dst.tcps.port=5001`
The TCP/IP port to listen on.

`c.chnl[#].dst.tcps.prompt='Password:'` 2.30
The prompt for the socket.

`c.chnl[#].dst.tcps.realtime=0`
Whether to establish a real-time link. 0=Disconnect/One-shot, 1=Real-time/Stay-connected[21].

`c.chnl[#].dst.tcps.ssl=''`
Whether to use SSL on the TCP/IP link. Blank=plain, “i”=implicit SSL.

`c.chnl[#].dst.tcps.successmsg=''` 2.30
(Optional) The string to send when a correct password is entered. e.g. “OK\r\n”

[20] From 2.30 onwards

[21] One-shot TCP deliveries are an “all-or-nothing” delivery. Data is only deleted from the ip.buffer when the remote end's TCP/IP stacks have ACK'd all the packets. In contrast, the real-time link will read/deliver/delete in 32k chunks.

4.6.7. TCP Push Delivery Config: c.chnl[#].dst.tcpc

`c.chnl[#].dst.tcpc.address=''`
The IP address or name of the TCP server to push to. **Must be filled in!**

`c.chnl[#].dst.tcpc.intf='L'`
Which interface(s) to try TCP pushing, and which order. “L”=LAN, “M”=Modem. e.g. “LM” = try LAN first, then Modem if LAN fails.

`c.chnl[#].dst.tcpc.port=2001`
The TCP/IP port to connect to.

`c.chnl[#].dst.tcpc.realtime=0`
Whether to establish a real-time link. 0=Disconnect/One-shot, 1=Real-time/Stay-connected.

`c.chnl[#].dst.tcpc.ssl=''`
Whether to use SSL. Blank=normal, “i”=implicit SSL.

4.6.8. COM port serial Delivery Config: c.chnl[#].dst.com[22]

`c.chnl[#].dst.com.baud=19200`
The baud rate to transmit.

`c.chnl[#].dst.com.connect=5`
(Optional) When supplied, sets the time in seconds the handshake lines need to be asserted before the connection is considered “connected”.

`c.chnl[#].dst.com.disconnect=10`
(Optional) When supplied, sets the time in seconds the handshake lines must be unasserted for before the connection is considered “disconnected” and closed.

`c.chnl[#].dst.com.port=#`
Which COM port to transmit on.

`c.chnl[#].dst.com.protocol='8N'`
The bit length and parity setting.

`c.chnl[#].dst.com.txchunk=16`
The number of bytes to queue up for transmission to the serial port. Smaller values increase CPU load but allow the ip.buffer to stop transmitting sooner when the device signals “don't send”.

`c.chnl[#].dst.com.txflow='ctsdsr'`
Which handshake line(s) to check for transmission flow control. The substring “cts” denotes the CTS line, while “dsr” denotes DSR. e.g. “ctsdsr” = CTS & DSR flow control.

`c.chnl[#].dst.com.txpause=0`
The number of *bits* pause to insert between each transmitted character. A value of 1 will simulate 2-stop-bits, while a value of 10 will introduce a complete character pause (useful for slow-to-respond devices).

[22] From version 2.40

`c.chnl[#].dst.com.txpin=2`

The pin to transmit on. 0=Auto, 2=DCE (connect straight to a PC), 3=DTE (connect with a null-modem cable to the PC).

4.6.9. Emulation Delivery Config: c.chnl[#].dst.emul[23]

`c.chnl[#].dst.emul.allow=''`

The name, IP address, wildcard IP address, or list to allow connection. Blank = any device.

`c.chnl[#].dst.emul.client='T'`

Which client type. “A”=auto, “T”=Telnet, “R”=raw TCP/IP.

`c.chnl[#].dst.emul.intf='LM'`

Which interface(s) to allow emulation TCP connection. The substring “L” denotes LAN, while “M” denotes Modem.

`c.chnl[#].dst.emul.port=6001`

Which TCP port the emulation will listen on. Often port 23 is required when emulating legacy devices.

`c.chnl[#].dst.emul.ssl=''`

Whether to use SSL on the TCP/IP link. Blank=plain, “i”=implicit SSL. Most legacy devices do not support SSL – this option was added to provide a quick and convenient way of adding strong encryption to a system already using legacy protocols[24].

[23] The emulation configuration set only appears, and can only be programmed, if a valid emulation script has previously been loaded.

[24] It is fairly trivial to add SSL support to an existing PC application. Proxy-style products like “stunnel” can provide SSL without any code changes to the PC application.

4.6.10. Delivery Trigger Config: c.chnl[#].dst.trig

`c.chnl[#].dst.trig.full=0`
The amount, in kilobytes, to trigger a push. 0=ignore.

`c.chnl[#].dst.trig.pause=0`
Perform a push after a pause in incoming data. Specified in seconds. 0=ignore.

`c.chnl[#].dst.trig.retrytime=60`
The time, in seconds, between attempting a retry for a failed push.

`c.chnl[#].dst.trig.ring=0`
Perform a push when the modem rings but doesn't answer. 0=ignore, 1=push.

`c.chnl[#].dst.trig.sched.data=1`
Whether to push only when there is something to deliver. 0=always, 1=when data.

`c.chnl[#].dst.trig.sched.days=''`
The days of the week for “zone A”. 1=Mon, 2=Tue, etc.

`c.chnl[#].dst.trig.sched.hia='1800'`
The end time for “zone A”. 24 hour format, digits-only. If .inta=0 then this is ignored. This is the “...and” time value on the web.

`c.chnl[#].dst.trig.sched.inta=60`
The interval to deliver during “zone A”. Specified in minutes. This is the “Deliver Every” setting on the web. 0=deliver once.

`c.chnl[#].dst.trig.sched.intb=0`
The interval to deliver in “zone B” (ie all times outside of “zone A”). This is the “At all other times deliver every” setting on the web.

`c.chnl[#].dst.trig.sched.loa='0800'`
The start time for “zone A”. 24 hour format, digits-only. This is the “At/Between” value on the web.

`c.chnl[#].dst.trig.vari=0` `2.70`
Sets the variance (in minutes) for the push time.

5. Information Lua Table

Status information from inside the ip.buffer is exposed to the outside world through a Lua table named “i”. Like the “c” table, this Lua table is accessed through the web interface and appears within the diagnostic information page on the “Tools” menu. It is meant to be read-only (although this isn't forced).

The “i” table is also sent as an attachment with email alerts and data deliveries.

The “i” table includes the following possible entries:

5.1. General: i

`i.alive=1037502`
: The number of seconds the buffer has been alive for.

`i.loader='112/123'`
: The hardware loader version (for Scannex diagnostic purposes).

`i.now='2008-05-07 13:31:41'`
: The current date and time in human readable format.

`i.seconds=1210167101`
: The current date and time in standard format – number of seconds since midnight, 1st January 1970

`i.started=1209129598`
: The .seconds value when the buffer was booted.

`i.temp=33`
: The current temperature (in integer degrees Celcius) of the ip.buffer. (Only available on ip.2 and ip.4 devices.)

`i.version=2.82`
: The firmware version, in decimal form.

5.2. Cloud data: i.cloud

`i.cloud.active=1` 2.82

Set to 1 when the ip.buffer is trying to contact the Cloud Server, otherwise NULL.

`i.cloud.connected=1` 2.82

Set to 1 when the ip.buffer is actually connected to the Cloud Server, otherwise NULL.

`i.cloud.waitmodem=1` 2.82

Set to 1 when the ip.buffer is waiting for the modem to connect, otherwise NULL.

5.3. Device data: i.device

`i.device.channels=4`

The number of source channels in the ip.buffer.

`i.device.files=4`

The number of storage files (and delivery channels) in the ip.buffer.

`i.device.serial='00-02-ae-20-00-1a'`

The serial number of the ip.buffer.

`i.device.serialports=4`

The number of physical serial ports in the ip.buffer.

5.4. System info: i.system

`i.system.batteryfitted=1`
Whether the internal NiMH batteries are fitted. 0=not fitted, 1=fitted.

`i.system.batterypower=0`
Whether currently running on battery power. 0=mains, 1=battery.

`i.system.externalpower=0`
Whether the ip.buffer is running from the optional 48VDC input. 0=no, 1=yes.

`i.system.log[#].desc='1:Cold Boot'`
The event number and description. Available entries include:
"`1:Cold Boot`" - the ip.buffer was asked to reboot.
"`2:Power Off`" - the ip.buffer was powered off.
"`3:Power On`" - the ip.buffer was powered on and started running code.
"`4:Trap`" - internal error.
"`5:INTEGRITY exception`" - internal error.
"`6:Lua exception`" - internal error.
"`7:Battery power`" - the ip.buffer switched to running on NiMH battery.
"`8:Mains power`" - the ip.buffer switched to running on mains power.

`i.system.log[#].time='2008-04-24 13:19:21'`
The time of the event log entry.
"#" is between 1 and 32. 1 is the most recent, while 32 is the oldest event.

`i.system.mainspower=1`
Whether the ip.buffer is running from the 7VDC power input. 0=no, 1=yes.

`i.system.phypower=1`
Whether the Ethernet PHY is currently powered.

`i.system.power=1` `2.40`
Whether the ip.buffer is running from enough power. 0=low volts, 1=ok.

`i.system.timeonmains=4799`
How long the ip.buffer has been running on mains, in seconds (and therefore charging NiMH batteries).

5.5. Modem status: i.modem

`i.modem.auth=''`
The PPP authentication mode for the current connection. “PAP” = Password Access Protocol, “CHAP-MD5” = Challenge Handshake Access Protocol.

`i.modem.connect=0`
The time, in seconds, left to connect before failing the connection.

`i.modem.country='B5'`
The reported country code from the modem. Should match the c.modem.country field.

`i.modem.current='Idle'`
The current modem state. Available states include:
“`Detecting`” - the boot-up process of checking the modem.
“`Booting`” - waiting for the rest of the system before finishing modem boot-up
“`Idle`” - idling, waiting for call etc.
“`Reset`” - currently resetting the modem.
“`Ringing`” - the modem line has detected a ring signal.
“`Answering`” - currently answering the call.
“`Negotiating`” - answering the phone line, or dialled, and waiting for PPP negotiation.
“`Online (trusted)`” - on-line to a trusted source (ie. Someone has dialled in).
“`Online (ISP)`” - on-line to the Internet. (ie. Dialled out through an ISP).

`i.modem.fails=0` 2.90
The number of failures to connect.

`i.modem.found='none'`
Which modem type has been detected. Values include: “none” = no modem; “pstn” = standard telephone modem; “gprs” = GPRS mobile network modem; “edge” = EDGE/GPRS modem.

`i.modem.gprs.cell=46777` 2.82
The cell ID of the GPRS connection.

`i.modem.gprs.dns1='192.12.3.5'` 2.82
`i.modem.gprs.dns2='192.12.3.6'`
The IP address of the DNS servers given by the GPRS PPP link

`i.modem.gprs.ip='10.0.0.2'` 2.82
The IP address of the ip.buffer on the GPRS PPP link

`i.modem.gprs.loc=609` 2.82
The location ID of the GPRS connection.

`i.modem.gprs.operator='T-mobile'` 2.82
The cellular operator name

`i.modem.gprs.peer='192.12.3.4'` 2.82
The IP address of the cellular network node on the GPRS PPP link

`i.modem.gprs.pssci='642,13,13,0,0,23,-651547237,3339267,234,30,609,46777,1,15,4,1,1,0,28'` 2.82

The result of AT*PSENGI=0 command, showing current cell info, etc.

`i.modem.gprs.reg=1`

The registration status (present only when using the GPRS modem).
0 = Network unavailable
1 = Registered on a home network
2 = Searching for a network
3 = Network denied (e.g. PIN number required, or not activated with network)
5 = Registered on a roaming network.

`i.modem.gprs.signal=14`

The signal strength (present only when using the GPRS modem).
If greater than 31 then there is no signal.
Otherwise, percentage = ((signal + 1) * 100) / 32

`i.modem.gprs.sim='ERROR'`

The SIM status.
"`OK`" = SIM correct and ready.
"`ERROR`" = SIM not installed, or modem error.
"`SIM PIN`" = SIM requires a PIN number. Enter the correct PIN in the setup.
"`SIM PUK`" = SIM requires a PUK unlock code[25]
Other codes are as returned by the "`AT+CPIN?`" command.

`i.modem.holdoff=0`

The amount of holdoff time left, in seconds.

`i.modem.hungup=0`

The time the modem was last hungup. (In seconds since 1970-01-01.)

`i.modem.id='CX58601'` 2.70

The modem's identifier (either `ATI3`/`ATI1` response).

`i.modem.isgprs=0`

Whether the modem is GPRS. 0 = RJ-modem (PSTN), 1 = GPRS mobile

`i.modem.lastppp=''`

A descriptive string of the highest phase of the PPP negotiation.
"`(blank)`" - PPP not attempted.
"`Authenticating`" - LCP up, starting to authenticate.
"`Acquiring Address (PAP)`" - PAP complete, getting IP address.
"`Acquiring Address (CHAP)`" - CHAP complete, getting IP address.
"`IP4 complete`" - IP stack running
"`Complete`" - PPP link up and running.

`i.modem.online=0`

The time on-line, in seconds.

`i.modem.power=1` 2.80

Whether the modem is currently powered on.

[25] You will need to power down the ip.buffer, remove the SIM, and place in a standard mobile phone to unlock.

`i.modem.quiet=0` 2.90

The number of seconds the nailed up connection has been quiet.

`i.modem.requestlist=''`

The list of channels requiring modem. “`Dst#1`”=channel 1, “`Dst#2`”=channel 2, “`Alert`”=email alert system.

`i.modem.requests=0`

The number of delivery channels requesting use of the modem.

5.6. Flash system info: i.mem

`i.mem.blocks.counts.append=2`
The total number of blocks that have been appended to.

`i.mem.blocks.counts.erase=132448`
The total number of blocks that have been erased.

`i.mem.blocks.counts.read=29`
The total number of pages that have been read from the file-system.

`i.mem.blocks.counts.wear=27463`
The total number of wear-levelling operations.

`i.mem.blocks.counts.write=99`
The total number of pages that have been written to the file-system.

`i.mem.blocks.erased=981`
The number of pre-erased blocks in the file-system.

`i.mem.blocks.fw='a+b+c+d'` 2.90
Shows the firmware block allocations. a=existing, b=uploaded, c=to-erase, d=erased

`i.mem.blocks.size=131072`
The size of a Flash storage block. Usually 128k. (A block is sub-divided into 2k pages.)

`i.mem.blocks.total=989`
The total number of Flash storage blocks that can be used for the system. Some blocks are reserved for firmware, while other blocks may be mapped out (because they were marked as defective[26]).

`i.mem.blocks.totalstore=984`
The number of blocks that are available for use by the channel storage area.

`i.mem.blocks.used=8`
The number of blocks that have been used by storage and static files.

`i.mem.blocks.usedstore=3`
The number of blocks that have been used by storage files only.

`i.mem.blocks.waiting=0`
The number of blocks that are waiting to be erased. When a large file is erased it may take several seconds to erase the blocks.

`i.mem.file[#].blocks=1`
The number of flash blocks that the static file is using. "#" is between 1 and 16.

`i.mem.file[#].size=5181`
The total size, in bytes, of the static file.

`i.mem.store[#].blocks=1`
The total number of blocks that a storage file is using. "#" is the channel number[27].

[26] Most flash devices have some bad blocks. Up to 10% of the blocks may be bad, while blocks can age and die with extensive use.

[27] There may be no bytes in the storage area, but the storage file may still be using a block – there are still blank pages to be used before the block gets erased.

`i.mem.store[#].frozen=0`
: The "frozen" byte size of the storage file. The file is frozen to allow reading while writing can continue.

`i.mem.store[#].lostnew=0`
: The total number of new bytes that have been thrown away (because the file has reached the maximum, or because the file system is full).

`i.mem.store[#].lostold=0`
: The total number of old bytes that have been erased to make room for new data.

`i.mem.store[#].size=40`
: The total number of bytes in the channel storage file.

5.7. Alerts status: i.alerts

`i.alerts.active=1` 2.82
: Set to 1 when the alert system is trying to push, otherwise NULL.

`i.alerts.connected=1` 2.82
: Set to 1 when the alert system is actually connected to the peer, otherwise NULL. Note that the expression `(busy and not waitmodem)` shows that delivery is taking place, as there could be a pause between `waitmodem` clearing and `connected` being set.

`i.alerts.waitmodem=1` 2.82
: Set to 1 when the alert system is waiting for the modem to connect, otherwise NULL.

`i.alerts.last.*`
: The ".this." values are transferred to ".last." when the email completes.

`i.alerts.this.connected=1`
: Whether the email is connected to the server. NULL=no, 1=yes.

`i.alerts.this.end=0`
: The time, in seconds past 1970-01-01, when the email was completed.

`i.alerts.this.error=''`
: The SMTP error string.

`i.alerts.this.intf='L'`
: The interface that the email is using. "L"=LAN, "M"=modem.

`i.alerts.this.remote='0.0.0.0'`
: The remote IP address of the SMTP computer.

`i.alerts.this.ssl=0`
: Whether SSL is being used on this email. 0=plain, 1=SSL/TLS.

`i.alerts.this.start=0`
: The time, in seconds past 1970-01-01, when the email was started.

`i.alerts.this.waitmodem=0`
: Whether the email is waiting for the modem to dial-out. 0=no, 1=yes.

5.8. Time update: i.sntp

`i.sntp.last=1213112298`

The time, in seconds past 1970-01-01, when the time was last synchronised with the SNTP server.

`i.sntp.lastupdate= "2008-06-10 15:38:18"`

The time, in human readable form, when the time was last synchronised with the SNTP server.

5.9. Source status: i.chnl[#].src

- Many of the source parameters appear only when the specific source type is selected and running.

5.9.1. Source

`i.chnl[#].src.connected=1`
Whether the source channel is currently connected. 0=no, 1=yes.

`i.chnl[#].src.data.quiet=1037359`
The number of seconds the channel has been quiet (ie. Not receiving data).

`i.chnl[#].src.protocol.error.code`
In the case of Lua errors, this is the Lua pcall error value. For the "`Protocol`" errors, this is zero.

`i.chnl[#].src.protocol.error.text`
The text of the error. In the case of Lua errors, this is the string that was left on the stack after the pcall (hence the Lua error message). For the "`Protocol`" errors this is the string that was passed to the `source.error` call.

`i.chnl[#].src.protocol.error.which`
Specifies where the error occurred.
"`_setup`" a Lua error in running the protocol.setup function.
"`_connect`" a Lua error in running the protocol.connect function.
"`active_loop`" a Lua error in running the loop while connected.
"`_data`" a Lua error in running the protocol.data function.
"`Protocol`" means the protocol registered an error with the `source.error` function.

`i.chnl[#].src.type='com'`
The currently running source type. When a source type is selected through the c.chnl[#].src.type parameter there is a delay while the source closes down cleanly.

5.9.2. COM/Serial Source

`i.chnl[#].src.com.autobauding=0`
Whether the serial port is autobauding. 0=no, 1=yes.

`i.chnl[#].src.com.autobaudrate=0`
The baud rate detected during the autobauding phase[28].

`i.chnl[#].src.com.cts=0`
The state of the CTS input line. 0=unasserted, 1=asserted.

`i.chnl[#].src.com.detect='0000'` 2.70
The diagnostics information for the COM port. Normally '222x', '333x', or '0000'.

`i.chnl[#].src.com.dsr=0`
The state of the DSR input line. 0=unasserted, 1=asserted.

`i.chnl[#].src.com.dtr=1`
The current state of the DTR output line. 0=unasserted, 1=asserted.

`i.chnl[#].src.com.pdce=0`
Internal debugging information. Shows the total number of times the serial input buffer has been exhausted (and so data would have been lost).

`i.chnl[#].src.com.pdcne=0`
Internal debugging information. Shows the total number of times the serial input buffer has nearly been exhausted (so the ip.buffer has had to signal “stop sending” with the Rx Flow handshake lines).

`i.chnl[#].src.com.protocol='8N'`
The current protocol being used. Changes during autobauding (and indicates the current protocol that is being tested).

`i.chnl[#].src.com.rts=1`
The current state of the RTS output line. 0=unasserted, 1=asserted.

`i.chnl[#].src.com.rxpin='3'`
The receive pin that the serial port is using. 0=not connected, 3=DCE, 2=DTE, 2*=DTE (but receive data detected on pin 3 as well. ie. Y-lead situation).

[28] When autobauding completes, the baud rate and protocol are programmed into the c.chnl[#].src.com values so the ip.buffer can reboot with the correct settings.

5.9.3. TCP Source

`i.chnl[#].src.tcp.busy=1`
: Whether the source TCP socket is currently working, or listening. 0=no, 1=yes.

`i.chnl[#].src.tcp.complete=0`
: Whether the source TCP is connected and has worked through the match/send strings. 0=no, 1=complete.

`i.chnl[#].src.tcp.connected=0`
: Whether the source TCP has connected (but not necessarily worked through the match/send strings). 0=not connected, 1=connected.

`i.chnl[#].src.tcp.last.err=0`
: The last socket error on the source TCP socket. This is an internal TCP/IP stack error number.

`i.chnl[#].src.tcp.last.text='No error'`
: The last socket error on the source TCP socket in human readable form.

`i.chnl[#].src.tcp.peer='0.0.0.0'`
: The remote IP address. (ie the address of the device)

`i.chnl[#].src.tcp.sendmatch=0`
: The match/send sequence number that has been processed. 0=none, 1=completed #1, etc.

`i.chnl[#].src.tcp.ssl=0`
: Whether the socket is TLS/SSL. 0 = no encryption, 1 = SSL/TLS

`i.chnl[#].src.tcp.sslerr=''`
: The last SSL/TLS connection error.

5.9.4. UDP Source

`i.chnl[#].src.udp.busy=0`
: Whether the UDP socket is busy resolving DNS. 0=no, 1=yes

`i.chnl[#].src.udp.last.err=0`
: The last socket error. This is an internal TCP/IP stack error number.

`i.chnl[#].src.udp.last.text='No error'`
: The last socket error in human readable form.

`i.chnl[#].src.udp.peer='0.0.0.0'`
: The IP address of the last received UDP packet.

5.9.5. FTP Server Source

`i.chnl[#].src.ftps.connect=0`
: The time, in seconds past 1970-01-01, when the client connected.

`i.chnl[#].src.ftps.disconnect=0`
: The time, in seconds past 1970-01-01, when the client disconnected.

`i.chnl[#].src.ftps.lastsize=0`
: The last file size, in bytes, that was pushed by the client.

`i.chnl[#].src.ftps.peer=''`
: The IP address of the FTP client, when connected.

`i.chnl[#].src.ftps.storend=0`
: The time, in seconds past 1970-01-01, when the client finished the STOR operation.

`i.chnl[#].src.ftps.storstart=0`
: The time, in seconds past 1970-01-01, when the client started a STOR operation.

5.9.6. Cloud Server Source

`i.chnl[#].src.cloud.lastsize=0` 2.92
: The last file size, in bytes, that was fetched from the Cloud server.

`i.chnl[#].src.cloud.end=0` 2.92
: The time, in seconds past 1970-01-01, when the fetch completed.

`i.chnl[#].src.cloud.start=0` 2.92
: The time, in seconds past 1970-01-01, when the fetched started.

5.9.7. Pass-through

`i.chnl[#].src.pass.busy=1`
: Whether the pass-through socket is currently working, or listening. 0=no, 1=yes.

`i.chnl[#].src.pass.complete=0`
: Whether the pass-through socket has completed the password checking. 0=no, 1=yes.

`i.chnl[#].src.pass.connected=0`
: Whether the pass-through socket is connected (but not necessarily negotiated the password or SSL). 0=no, 1=yes.

`i.chnl[#].src.pass.err=0`
: The last error number for pass-through. This is an internal TCP/IP stack error number.

`i.chnl[#].src.pass.peer='0.0.0.0'`
: The IP address of the connected pass-through peer (ie. The computer end).

`i.chnl[#].src.pass.ssl=0`
: Whether SSL has been negotiated. 0=no, 1=yes

`i.chnl[#].src.pass.sslerr=''`
: The last SSL error string.

`i.chnl[#].src.pass.text='No error'`
: The human readable last error string for the pass-through socket.

5.10. Delivery status: i.chnl[#].dst

`i.chnl[#].dst.active=1` 2.40
Set to 1 when the channel is running a push delivery. Set to 0 when not pushing. Set to NULL if a pull type channel.

`i.chnl[#].dst.alive=1` 2.82
Set to 1 when the channel is running a push or pull delivery, otherwise NULL

`i.chnl[#].dst.connected=1` 2.82
Set to 1 when the delivery is actually connected to the peer, otherwise NULL Note that the expression `(busy and not waitmodem)` shows that delivery is taking place, as there could be a pause between `waitmodem` clearing and `connected` being set.

`i.chnl[#].dst.holdoff=30` 2.40
The hold-off time (in seconds) before pushing again.

`i.chnl[#].dst.last.*`
The ".this." values are transferred to ".last." when the delivery completes.

`i.chnl[#].dst.seq=145` 2.40
The file sequence number.

`i.chnl[#].dst.this.cipher=''` 2.80
The cipher suite name.

`i.chnl[#].dst.this.connected=1`
Whether the delivery is connected. NULL=no, 1=yes.

`i.chnl[#].dst.this.cts=0`
Whether the CTS pin is asserted. 0=no, 1=yes. Only shown for "COM port serial" delivery.

`i.chnl[#].dst.this.dsr=0`
Whether the DSR pin is asserted. 0=no, 1=yes. Only shown for "COM port serial" delivery.

`i.chnl[#].dst.this.end=0`
The time, in seconds past 1970-01-01, when the delivery was completed.

`i.chnl[#].dst.this.error=''`
The error string. Depending on the delivery method, this string can originate from the ip.buffer firmware, or from the remote server (e.g. SMTP authentication errors).

`i.chnl[#].dst.this.frozen=0`
How many bytes were frozen for delivery.

`i.chnl[#].dst.this.intf='L'`
The interface that the delivery is using. "L"=LAN, "M"=modem.

`i.chnl[#].dst.this.remote='0.0.0.0'`
The remote IP address of the computer.

`i.chnl[#].dst.this.ssl=0`
Whether SSL is being used on this delivery. 0=plain, 1=SSL/TLS.

`i.chnl[#].dst.this.start=0`

The time, in seconds past 1970-01-01, when the delivery was started.

`i.chnl[#].dst.this.transferred=0`

The number of bytes that have been transferred to the remote computer.

`i.chnl[#].dst.this.txpin='2'`

Which pin is being used for transmission for “COM port serial”. '0'=not detected, '2'=pin 2, '3'=pin 3, 'Y?'=Y-lead[29], '?'=unknown state.

`i.chnl[#].dst.this.waitmodem=0` 2.40

Whether the delivery method is waiting for the modem to dial-out. 0=no, 1=yes.

`i.chnl[#].dst.trigger=0` 2.40

Whether push delivery is triggered. 0=not, 1=triggered.

`i.chnl[#].dst.type='ftps'`

The current delivery method.

`i.chnl[#].dst.waitmodem=1` 2.40

Set to 1 when the channel is waiting for the modem to connect before starting the delivery, otherwise NULL.

[29] When 'Y?' is shown the ip.buffer has detected a transmit into it on both pin 2 and 3. It cannot transmit back out when this is shown.

6. Diagnostic Lua Table

Some diagnostic data is stored in the Lua table “d”. Like the “c” table, this Lua table is accessed through the web interface and appears within the diagnostic information page on the “Tools” menu.

The “d” table includes the following possible entries:

6.1. Source

`d.chnl[#].src.data.last='N 301 DATA HERE\r\n'`
: The last data received on this channel.

`d.chnl[#].src.protocol.p.*`
: Shows diagnostic information about the protocol. The values that appear are specific to the protocol.

`d.chnl[#].src.protocol.p.n=1`
: The channel number for this protocol.

6.2. Cloud Server

`d.cloud.auth='none'`
: The type of authorization used. 'basic', 'digest' are other values.

`d.cloud.cachei=0`
: The number of requests that have been issued to the Cloud Server.

`d.cloud.cachex=20`
: The maximum number of requests to keep the cache for.

`d.cloud.freqi=0`
: The number of seconds since the last contact.

`d.cloud.freqx=0`
: The minimum contact time.

`d.cloud.infoi=0`
: The number of seconds since sending the info data.

`d.cloud.infox=0`
: The number of seconds to send the info data.

`d.cloud.cached=0`
: Whether the redirection and server features have been cached.

6.2.1. Cloud Server Cache

`d.cloud.cache.name='cloud.ipbuffer.com'`
The name of the server.

`d.cloud.cache.uri='/upload'`
The URI.

`d.cloud.cache.delay=0`
(HTTP contact method.)

`d.cloud.cache.tls=1`
Whether using TLS/SSL security

`d.cloud.cache.port=443`
The TCP port number

`d.cloud.features='zlib-opt-mgr-log'`
The features supported by the Cloud Server script.
Features include:

- zlib = compressed transfers
- opt = use optimal header, rather than body
- mgr = server supports central update management
- log = server would like log files
- tec = server wants ip.buffer to send “Transfer-encoding: chunked”

6.3. Modem

`d.modem.gprs.cell=46777`
`d.modem.gprs.loc=46777`
The mobile cell identifier and location ID

`d.modem.gprs.pssci='642,13,13,0,0,23,-651547237,3339267,234,30,609,46777,1,15,4,1,1,0,28'`
Additional detailed GPRS cell information.

7. Execution (Callback Hook) Lua Table

The Lua table "x" provides a mechanism for hooking in functions for certain operations.

7.1. Timed Hooks

```
x.onsecond = function()
```
2.20

Called every second.
See Section 11 "Executing Lua code every second"

7.2. Alert Hooks

```
x.alerts.onrun = function(channelno, typedest)
```
2.82

```
x.alerts.onpush = function(channelno, typedest, action, reason)
```
2.82

```
x.alerts.onconnect = function(channelno, typedest, peer, modem)
```
2.82

```
x.alerts.ondisconnect = function(channelno, typedest, peer, modem,
successful)
```
2.82

See the notes for `x.chnl[#].dst.on*`
(onprefix & onsuffix are not relevant)

7.3. Execution Hooks

```
x.onrun = function()
```
2.93

Called when Lua starts/reboots. This function has no timeout limit, and can house long-running Lua functions. If the function returns, the function is called again after a one second delay.
Within the loop, use the "`while tools.run() do`" construct, rather than "`while true do`" - this will provide a graceful closure when Lua is rebooted.

e.g.

```
-- Simple SNMP Trap rx that doesn't use a channel
function myservercode()
  local u=socket.new("udp")
  u:open(162)
  while tools.run() do
    local n,rx=socket.select(500,{u})
    if rx[u] then
      local b,ip = u:recv()
      b=snmp.decode(b,ip)
      if b~="" then mem.write(4,"TRAP:"..b.."\r\n") end
    end
  end
  u:close()
end
x.onrun=myservercode
```

7.4. Authentication Hooks

`x.auth.ftp = function(table)` 2.82
`x.auth.tcp = function(table)` 2.82
`x.auth.passthru = function(table)` 2.82
`x.auth.web = function(table)` 2.82

Called every time an authentication is performed for the particular service. Called before any reference to RADIUS, the function is passed a Lua table.
Table entries (shown as "`t.`"):

- `t.service` = 'ftp', 'tcp', 'passthru', 'web', 'ppp'
  - The name of the service that is being authenticated
- `t.channel` = `NULL` or `1-4`
  - The channel number (if appropriate), or NULL
- `t.username` = user name string
- `t.hash` = SHA-1 has of the password (NULL for PPP)
- `t.interface` =
  - 'ppp0s' = PPP
  - 'et0' = Ethernet
  - 'lo0' = Loopback interface
- `t.modem` = `NULL` or `true`
- `t.pppuser` = `NULL` or username (NULL if over LAN)
- `t.peer` = `NULL` or IP address of peer

The return values signal to the ip.buffer what to do with this authentication request:

- `NULL` = continue as before (use RADIUS or Local as programmed)
- `-1` = Timeout simulated. Fall back to local
- `0` = Reject request
- `1-4` = Accept for channel 1 to 4
- `string` = Change fallback to
  - 'R' (RADIUS)
  - 'B' (Both - RADIUS & Local)
  - 'T' (RADIUS and Local on Timeout)
  - 'L' (Local)

> If the Lua function denies a user, the web service will still accept the programmed administration username + password during the 5 minute HDAP/SEDiscover window (so you can always get back in even without RADIUS connected)

`x.auth.ppp = function(table)` 2.82

Called when a PPP user is authenticated.
The table structure is identical to the one described in x.auth.* above.
However, the return values are different for this function:

- `NULL` = continue as before (Lua function has no view on this user)
- `false` = deny user
- `string` = Secret Store keyname to use as the PPP password for the given username.
- `NULL, string` = absolute password to compare for the given username

7.5. Web server Hooks

`x.http.admin = function(table)` 2.90
`x.http.admin.onget`
`x.http.admin.onpost`
`x.http.admin.onhead`

Called whenever the "/ext/admin/" region of the ip.buffer web-server is referenced.

`x.http.user = function(table)` 2.90
`x.http.user.onget`
`x.http.user.onpost`
`x.http.user.onhead`

Called whenever the "/ext/" or "/ext/public/" regions of the ip.buffer web-server is referenced.

7.5.1. Http Hook Table Structure

The ip.buffer fills in the information about the HTTP call:

```
admin=0/1 (whether user or admin)
filename = string
method = string ('HEAD', 'GET', 'POST')
path = string
queryuri = string
uri = string
client = string (IP address)
scheme = string ('http://' or 'https://')
relroot = string (prefix to use for URIs)
headers = table (list of [index]=values)
query = table (list of URL query values)
cgi = table (list of CGI post parameters)
post = table (list of multipart/mime post parameters)
```

7.5.2. Http Hook Return

The web server HTTP hook function can return a result in several formats:

`integer`

HTTP response code. e.g. 404

`string`

Plain HTML. If the string begins with "<html>" then the ip.buffer header + footers are not applied. Otherwise, the text should conform to the same rules as the internal ip.buffer pages.

`string, string`

Content with Content-Type. e.g. 'Hello!', 'plain/text'
This allows returning of images and other type of content.

`table`

Discrete control over the response and HTTP headers:

```
response = integer   // HTTP response code
headers = table      // table of extra HTTP headers
content = string     // e.g. content='text/plain'
body = string        // the main body of response
ipbuffer = boolean   // set to true to add headers & footers
```

7.6. Channel Hooks

7.6.1. Pass-through

`x.chnl[#].pass.rx = function(string, protocol)` 2.50

`x.chnl[#].pass.tx = function(string, protocol)` 2.50

Called every time there is data received and transmitted on a pass-through connection.
See Section 10 “Filtering data for pass-through operations”

`x.chnl[#].pass.onconnect = function(channelno, sourcetype, ipaddress, isondemand, wasopened)` 2.90

Called when the pass-through socket has connected.

`x.chnl[#].pass.ondisconnect = function(channelno, sourcetype)` 2.90

Called when the pass-through socket has disconnected.

7.6.2. Source

`x.chnl[#].src.onrecord = function(textstring, channelnumber, tagstring)`

Called every time there is data to store in a channel.
See Section 9 “Executing Lua every time data is stored”

`x.chnl[#].src.onclosed = function(channelno, sourcetype)` 2.90

Called when the source has closed.

`x.chnl[#].src.onclosing = function(channelno, sourcetype)` 2.90

Called when the source starts closing.

`x.chnl[#].src.onopened = function(channelno, sourcetype)` 2.90

Called when the source has opened.

`x.chnl[#].src.onopening = function(channelno, sourcetype)` 2.90

Called when the source starts opening.

7.6.3. Destination

- The sequence of destination callback hooks is:
  `onrun / [onpush] / onconnect (if success) / ondisconnect` …
  repeat through onpush if multiple interfaces configured.

```
x.chnl[#].dst.onconnect = function(channelno, typedest, peer, modem)
```
2.82

Called when the ip.buffer connects to the remote end (either push or pull).
'`channelno`' is the channel number.
'`typedest`' is the string type of the delivery. e.g. '`tcpc`', '`httpc`'
'`peer`' is the remote IP address (if applicable)
'`modem`' is a boolean whether running over modem.

```
x.chnl[#].dst.ondisconnect = function(channelno, typedest, peer, modem,
successful, moretodo, transferred)
```
2.82

Called when the ip.buffer connects to the remote end (either push or pull).
'`channelno`' is the channel number.
'`typedest`' is the string type of the delivery. e.g. '`tcpc`', '`httpc`'
'`peer`' is the remote IP address (if applicable)
'`modem`' is a boolean whether running over modem.
'`successful`' is a boolean whether transfer was successful.
'`moretodo`' is a boolean whether there's another transfer required.
'`transferred`' is the number of bytes transferred this time.

- onconnect is only called if the peer is reached. In contrast, ondisconnect is always called as the delivery process closes down.

`x.chnl[#].dst.onprefix = function(channelno, typedest, str)` 2.82

Called when the ip.buffer inserts the prefix string into the data stream.
'`channelno`' is the channel number.
'`typedest`' is the string type of the delivery. e.g. '`tcpc`', '`httpc`'
'`str`' is the existing string.
Return a string to replace the prefix.

`x.chnl[#].dst.onpush = function(channelno, typedest, action, reason)` 2.82

Called as a push-delivery is started. This is called after 'onrun' and will be called for each interface chosen for delivery.
'`action`' is a number that denotes what push action is being taken:

- 0 = Finish
- 1 = LAN try
- 2 = LAN skip (reason given)
- 3 = LAN fail (reason given)
- 4 = Modem try
- 5 = Modem skip (reason given)
- 6 = Modem fail (reason given)

'`reason`' is the string reason (if applicable)
Returning a number in the range of 0-6 will change the flow of attempts (use with extreme caution!)

`x.chnl[#].dst.onrun = function(channelno, typedest)` 2.82

Called as the destination delivery task is created for running.
'`channelno`' is the channel number.
'`typedest`' is the string type of the delivery. e.g. '`tcpc`', '`httpc`'

`x.chnl[#].dst.onsuffix = function(channelno, typedest, str)` 2.82

Called when the ip.buffer inserts the suffix string into the data stream.
'`channelno`' is the channel number.
'`typedest`' is the string type of the delivery. e.g. '`tcpc`', '`httpc`'
'`str`' is the existing string.
Return a string to replace the prefix.

7.7. FTP Hooks

```
x.ftp.onlist = function(isnlstboolean, channelnumber, filenamestring,
    filesize, modifiedseconds)
```

Provides an opportunity to change the FTP listing format. Called when a client performs a LIST or NLST on the ip.buffer's FTP server.
If this function returns a string the client is given the string for the given directory entry.

7.8. Cloud server Hook

```
x.cloud.ondata = function(filename, txt, zlws)                    2.92
```

Called when the Cloud server has source data to inject. The default Lua function "CloudOnData" is hooked such that it pulls the channel number from the filename ('.##.'), and whether the file is zlib compressed ('.zlib').
The hook function uses the Lua zlib.inflate, and calls the appropriate sources[#]:pushopen / push / pushclose functions.

Needs to return the two results from the sources[#]:push call.
e.g. `return true -- indicates that the function succeeded`
e.g. `return NULL, 'error string' -- reported back to the Cloud server`

8. Lua Script Extensions

This section documents the additions to the basic Lua core that provide the functionality required for the ip.buffer product.

8.1. Lua core changes

There are some basic internal changes to the Lua core. Namely:

1. Reading from a table that doesn't exist returns NULL, and does not generate a run-time error[30]. Attempting to write to a NULL table will still generate a run-time error.
2. C/C++ style comments are supported with the "//" tag and "/*...*/" tag. This is in addition to the regular Lua comment marker "--"[31].
3. Strings can use the C/C++ hex markers. e.g. "`\xee\ff`" is 0xee + 0xff[32].
4. The os functions "execute", "exit", "getenv", "remove", "rename", "setlocale", and "tmpname" removed.

8.2. Global "Ad-hoc" functions

The global functions are called from an Ad-Hoc script on the web.

`defaults()`
: Completely clears the configuration table and applies the ip.buffer defaults. Does not erase Flash, and does not wipe Certificates or Keys.

`prune()`
: Prunes the configuration table tree – removing any unrecognised entries.

`wipeall()`
: Erases *everything* in the ip.buffer. It will erase all flash storage (as long as there are no delivery tasks in progress) and will erase all configuration, keys, and certificates. It is the program equivalent of holding in the front-button for more than 10-seconds.

[30] Lua's lvm.c modified.

[31] Lua's llex.c llex function modified.

[32] Lua's llex.c read_string function modified.

8.3. alert

The alert module provides an interface into the SNMP/Email alert system. With this module you can trigger pre-registered alarms, and register new ones.

`alert.alarm(id)`
`alert.alarm(id, message)`
`alert.alarm(id, message, immediate)` 2.92

Triggers an alarm with the identifier "id". Preprogrammed alarm IDs include: "Reboot", "Comfort", "Auth", "Mains", "Battery", "User"
Passing immediate as 'true' will send the alert with the specified message now[33].

`alert.clearall()` 2.40

Clears all pending alerts[34].

`alert.register_counter(id, msg, max, period, repeat)`

Registers a counting alert. It enables the easy registration of an alert that will look for "`max`" number of occurrences within the specified "`period`" (in seconds). It calculates the decrement value based on the period and maximum value.
Equivalent to "`register_raw(id, msg, 1, max, max, (period/max), repeat)`"

`alert.register_holdoff(id, msg, time, repeat)`

Registers a hold-off and repeat style alert.
Equivalent to "`register_raw(id, msg, time, time, 1, 1, repeat)`"

`alert.register_oneshot(id, msg)`
`alert.register_oneshot(id, msg, holdoff)`

Registers a "one-shot" style alert. This is just an abbreviated version of the register_raw call.
Equivalent to "`register_raw(id, msg, holdoff, holdoff, 1, 1, 0)`"

`alert.register_raw(id, defaultmsg, inc, max, trig, dec, rep)`

Registers a new alarm (or overwrites the definition for an existing one) with the raw parameters.
`id` = the string name of the alert (to be used in a subsequent call to alert.alarm)
`defaultmsg` = the default message to be used if the call to alert.alarm did not include a message.
`inc` = the amount to increment the internal value each time alert.alarm is called.
`max` = the highest value that the internal value should go to.
`trig` = the trigger point. This is the cross-over point where an alert goes from the "idle" state to the "alarmed" start.
`dec` = the amount to decrement the alert internal value by every second.
`rep` = the time, in seconds, between re-sends of the alert. While in the "alarmed" state, an email will be sent periodically at this interval.
With these parameters you can program alerts that count, alerts that have a "decay" time, alerts that will trigger immediately, etc.

[33] This option is useful for E911 style alerts where several alerts of the same tag ID can be generated, but with different messages. *This option was introduced in v2.92*

[34] Note that some alerts, such as Quiet alerts, will be reasserted every second.

`alert.register_trap(id, specifictrap)` 2.10
`alert.register_trap(id, specifictrap, channel)`

Associates a custom trap with the alert. When the alert triggers, the trap is sent using the message passed to the “alert.alarm” string.

8.4. arp

The arp module provides facilities for manipulating the ARP network table.

It replicates the 'net.arp*' functions.

`arp.clear()` 2.92

(same as net.arpclear)
Clears the ARP table

`arp.del(ipaddress)` 2.92

(same as net.arpdel)
Deletes the ARP entry for the given IP address.

`arp.find(macstring)` 2.92
`arp.find(ipstring)`

(same as net.arpfind)
Finds an ARP entriy for either dotted IP entries, or MAC substrings.
If the parameter is a MAC address or MAC address prefix (e.g. `00:02:ae:12:34:56` or `00:02:ae`) the return value is the IP address.
If the parameter is a dotted IP address (`123.45.67.89` format) the return value is the MAC address.

`arp.table()` 2.92
`arp.table(macstring)`

(same as net.arp)
Returns a table with MAC addresses. The key of the table is the IP address, with the value as the MAC.
When a string is provided, the function returns a table of entries where '`macstring`' exists. '`macstring`' can be a full MAC address (using colons, e.g. 00:02:ae:12:34:56) or a partial MAC prefix (e.g. 00:02:ae) which can be used to locate devices by a particular manufacturer.

8.5. atomic

The atomic module provides for functions that are useful to synchronise multiple threads.

`atomic.execute(function, …)` 2.82

Helper function that encapsulates `atomic.lock()` / function / `atomic.unlock()`

`atomic.lock()` 2.82
`atomic.unlock()`

Equivalent to `tools.lock()` & `tools.unlock()`

`atomic.modify(table, valuename, mask, increment)`

Updates an entry within the table using the mask and increment.
Returns the original value of *table.valuename*
e.g.

```
atomic.modify(mytable,'x',0,1) //increments mytable.x
atomic.modify(mytable,'x',0,-1) // decrements mytable.x
atomic.modify(mytable,'y',-1,123) // sets mytable.y to 1234
```

`atomic.testandset(table, valuename, testvalue, setvalue)` 2.70

Checks whether *table.valuename* is equal to *testvalue*, and if it is it will set it to *setvalue*. This is helpful for providing mutex style synchronisation.
Returns true if the test was equal.
e.g.

```
if atomic.testandset(mytable, 'x', 0, 1) // Acquire the 'mutex'
then
 // do something
 atomic.testandset(mytable, 'x', 1, 0)  // Release the 'mutex'
end
```

8.6. auth

The auth Lua modules provides some useful authentication routines.

`auth.hash(string, [format], [which])` 2.82
`auth.md5(string, [format])`
`auth.sha1(string, [format])`

Pass a string and this function will calculate the hash. The optional format modifier changes the result type:

- 0 = raw binary
- 1 = ASCII hexadecimal
- 2 = colon delimited ASCII hex (default)

The *which* specifier should be either "`md5`" or "`sha1`" (default)

`auth.hashcompare(hash1, hash2)` 2.82

Compares two hashes. Returns true if both the same.

`auth.hashformat(hash, [format])` 2.82

Converts binary or ASCII MD5 or SHA1 hash to binary or formatted ASCII (see 'format' definitions under 8.6 `auth.hash`)

`auth.radius()` 2.82

Returns true if the RADIUS authentication server has been contacted yet.

8.7. bit

The bit module provides useful bit operation functions[35].

`bit.arshift(a,b)` 2.10
: Returns a right shifted arithmetically b places.

`bit.band(w1,...)` 2.10
`bit.band(string)`
: Returns the bitwise AND of the w's, or string

`bit.bnot(a)` 2.10
: Returns one's complement of a.

`bit.bor(w1,...)` 2.10
`bit.bor(string)`
: Returns the bitwise OR of the w's, or string

`bit.bxor(w1,...)` 2.10
`bit.bxor(string)`
: Returns the bitwise XOR of the w's, or string

`bit.cast(a)` 2.10
: Cast a to the internally used integer type.

`bit.lshift(a,b)` 2.10
: Returns a left shifted b places.

`bit.rshift(a,b)` 2.10
: Returns a right shifted b places.

`bit.swap8(a)` 2.10
: Returns a with the nibbles exchanged.

`bit.swap16(a)` 2.10
: Returns a with the LSB and MSB bytes exchanged.

`bit.swap32(a)` 2.10
: Returns a with each of the four bytes reversed.

[35] 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.)

8.8. cloud

The cloud module provides Cloud Server extensions.

`cloud.basicfirst(boolean)`
: Set whether to use basic authentication first.

`cloud.cachelimit(integer)`
: Set the number of requests that the cache should be kept for. Also can be set by the Cloud Server.

`cloud.chunksize(integer)`
: Set the HTTP chunk size in bytes. Also can be set by the Cloud Server.

`cloud.datalimit(integer)`
: Set the maximum amount of data to send in bytes. Also can be set by the Cloud Server.

`cloud.flush()`
: Flushes the cached redirection and server features.

`cloud.frequency(integer)`
: Set the contact minimum time in seconds. Also can be set by the Cloud Server.

`cloud.infotime(integer)`
: Set the info data limit time in seconds. Also can be set by the Cloud Server.

`cloud.send(string)` 2.92
: Triggers sending the files to the Cloud Server.
Use the markers “`log`”, “`info`”, and “`diag`” to choose what to send.
e.g. `cloud.send('diag+log')`

`cloud.senddiag()` 2.92
: Triggers sending the diagnostics dump file to the Cloud Server.

`cloud.sendinfo()` 2.92
: Triggers sending the log file to the Cloud Server.

`cloud.sendlog()`
: Triggers sending the log file to the Cloud Server.

8.9. crc

The crc module provides 8, 16, and 32-bit CRC functions.

`crc.calc(init, polyrev, xor, string)` 2.92

General purpose CRC calculator (handles all sizes up to CRC-32)
"`init`" is the initial value of the CRC
"`polyrev`" is the polynomial value, reversed. Use negative values when the input and output are reflected.
"`xor`" is the value to XOR with at the end.
http://en.wikipedia.org/wiki/Cyclic_redunancy_check "Standards and common use"
http://reveng.sourceforge.net/crc-catalogue
e.g.

- **CRC-32:**
  `crc.calc(0xffffffff, -0xedb88320, 0xffffffff, data)`
- **CRC-16/Modbus:**
  `crc.calc(0xffff, -0xa001, 0, data)`
- **CRC-16:**
  `crc.calc(0xffff, 0x8408, 0, data)`
- **CRC-16/Xmodem:**
  `crc.calc(0, 0x8408, 0, data)`
- **CRC-16/Kermit:**
  `crc.calc(0, -0x8408, 0, data)`
- **CRC-16/DNP:**
  `crc.calc(0, -0xa6bc, 0xffff, data)`

`crc.crc8(string)` 2.10

Calculates the 8-bit CRC of the string.
e.g. `local mycrc=crc.crc8("Testing")`

`crc.crc16(string)` 2.10

Calculates the 16-bit CRC of the string.
e.g. `local mycrc=crc.crc16("Testing")`

`crc.crcxmodem(string)` 2.10

Calculates the 16-bit Xmodem CRC of the string.
e.g. `local mycrc=crc.crcxmodem("Testing")`

`crc.crc32(string)` 2.10

Calculates the 32-bit CRC of the string.
e.g. `local mycrc=crc.crc32("Testing")`

`crc.crc8_init()` 2.10
`crc.crc16_init()`
`crc.crcxmodem_init()`
`crc.crc32_init()`

Returns an initialised value for doing repetitive updates.
e.g. `local mycrc=crc.crc16_init()`

`crc.crc8_update(v,string)` 2.10
`crc.crc16_update(v,string)`
`crc.crcxmodem_update(v,string)`
`crc.crc32_update(v,string)`

Returns the updated CRC, given the starting CRC "v", and the string.
e.g. `mycrc=crc.crc16_update(mycrc,"Testing")`

`crc.crc8_update(v,w1,...)` 2.10
`crc.crc16_update(v,w1,...)`
`crc.crcxmodem_update(v,w1,...)`
`crc.crc32_update(v,w1,...)`

Returns the updated CRC, given the starting CRC "v", and w's... (standard Lua convention - can be a single value, or a comma separated list of values).
e.g. `mycrc=crc.crc16_update(mycrc,1234,4567)`

`crc.crc8_finish(v)` 2.10
`crc.crc16_finish(v)`
`crc.crcxmodem_finish(v)`
`crc.crc32_finish(v)`

Returns the finished CRC value of "v".
e.g. `mycrc=crc.crc16_finish(mycrc)`

8.10. dns

The dns module provides an interface into the DNS resolver of the ip.buffer

`dns.clear(string_name)` 2.92

(same as net.dnsclear)
Clears the DNS table entry 'name', if it exists.

`dns.empty()` 2.92

Empties the DNS cache. Returns the number of entries that could not be removed (so a return value of zero indicates that the cache is completely empty)

`dns.hostbyaddr(string_ipaddress)` 2.70

Performs a revers DNS lookup for the *string_ipaddress*. Returns a name string, or NULL/error/errorstring.

```
e.g. myname = dns.hostbyaddr('217.37.194.105')
```

`dns.hostbyname(string_name)` 2.70

Performs a DNS lookup for the *string_name*. Returns an IP address string, or NULL/error/errorstring.

```
e.g. myaddr = dns.hostbyaddr('mail.scannex.com')
```

`dns.match(string_addr, string_list)` 2.70

Checks whether *string_addr* matches one of the entries in *string_list*. This is the same function now used in the “allow” matches for inbound sockets.

```
e.g. if dns.match('192.168.0.123', '192.168.*.1, 192.168.0.*,
test.scannex.com') then ...
```

`dns.setcache(value)` 2.70

Sets the DNS cache size immediately. The value should be in the range 1 to 1024. See also the `c.network.dns.cachesize` option.

`dns.table()` 2.92

Returns a table of the DNS cache entries. Each table entry has the following fields:

- `ip = '0.0.0.0' -- IP address`
- `name = 'text'  -- Name of the entry`
- `qtype = 1      -- Query type`
- `complete = true/false – Whether the response is complete`

8.11. emulation

The emulation module provides the means to simulate legacy devices for delivery. This module includes a set of helper functions for Lua to make that emulation coding easy. The legacy emulation will not be visible as a destination unless the function "`emulation.connect`" has been assigned in the OEM or user script.

The emulation.connect function should be of the form:

```
function emulation.connect(e)
 e:write("Hello")
end
```

All reading from the emulation socket should be handled in Lua – keystroke handling, timeouts etc. When the function returns, the emulation socket is closed (and if using dial-in through a modem, the line is reset).

Sample emulation scripts are available from Scannex.

8.11.1. Emulation Events

Three events are defined.

`emulation.create(e)` 2.30
: Called when the channel initialises the Lua emulation.

`emulation.connect(e)` 2.30
: Called when the modem or TCP socket connects. This main function should do "the work" of the emulation.

`emulation.destroy(e)` 2.30
: Called when emulation finishes (either closed or aborted). Maximum run time for this function is one second.

8.11.2. Emulation Variables

Within the function "emulation.connect(e)", the table "e" has the functions as well as some channel-specific information. Useful variable entries are as follows:

`e.n`
: The channel number, from 1.

`e.peer`
: The IP address of the remote end (string). If connected through the modem in "dumb" mode, then the peer will be "127.0.0.1" (because the modem core connects internally via TCP/IP to the emulation).

`e.telnet`
: True if the socket is a Telnet socket, False if a raw socket. When true you should perform byte stuffing for the 0xff character in the script (ie. 0xff should be transmitted as 0xff 0xff).

8.11.3. Emulation Terminal Socket Functions

```
emulation.dead(e)
e:dead()
```

Returns true if the emulation “e” terminal socket has closed (ie is dead).

```
emulation.flush(e)
emulation.flush(e, timeout)
e:flush()
e:flush(timeout)
```

Flushes the receive data on emulation “e” for 500ms, or the “timeout”.

```
emulation.read(e)
emulation.read(e, size)
emulation.read(e, size, timeout)
e:read()
e:read(size)
e:read(size, timeout)
```

If no parameters are passed, this reads all the waiting data and returns immediately.
If only a size is included, this will read up to the value and return (almost) immediately.
If a size and timeout are included, this will wait for “timeout” milliseconds and wait for at least “size” bytes to be available. It will return up to “size” bytes as a string.
In all cases, it returns the string and the a boolean value indicating whether a timeout occurred.

```
emulation.sent(e)
e:sent()
```

Returns true if there are no bytes in flight (ie all the bytes have been transferred across the remote TCP/IP stack).

```
emulation.write(e, string)
e:write(string)
```

Writes the string to the legacy socket.
Returns a boolean value – true=sent, false=failed.

8.11.4. Emulation Memory Functions

`emulation.adv(e, count)`
`e:adv(count)`
Advances the memory pointer by "count" bytes. Returns true if successful. Return false if there is nothing more to read from the storage.

`emulation.close(e)`
`e:close()`
Closes the file. Return true if successful.

`emulation.delete(e, amount)` 2.92
`e:erase(amount)`
Deletes *amount* data. Will only delete up to the read pointer (allows for read-ahead, delete-behind processing).

`emulation.erase(e)`
`e:erase()`
Erases data that has been read from the file (if open). Returns true if successful.

`emulation.erasefrozen(e)`
`e:erasefrozen()`
Erases all frozen data in the file. Returns true if successful.

`emulation.freeze(e)`
`e:freeze()`
Freezes (or refreezes) the bytes in the file – making the data available for reading.

`emulation.frozen(e)`
`e:frozen()`
Returns the number of frozen bytes available in the file.

`emulation.get(e, maximum)`
`e:get(maximum)`
Pulls up to "maximum" bytes from the flash storage and returns the string and the length.
Note that this call may not return up to a whole record boundary, but merely up to the next flash page ending. You may need to call this twice, or more, to retrieve a complete record (along with a call to emulation.adv)

- This can validly return zero even when there is more data to read! Use the return of `emulation.adv` to determine when the end of storage has been reached.

See also "`emulation.adv`"

`emulation.open(e, storenumber)`
`e:open(storenumber)`
Opens the file "storenumber" (1= channel 1, etc). Returns true if successful.

`emulation.reset(e)`
`e:reset()`
Resets the file read pointer back to the beginning.

```
emulation.stopsoon(e)
e:stopsoon()
```

Instructs the file system to limit the calls to “emulation.get” so that the read process can stop on the nearest record boundary in the flash pages. Return true if successful.

8.11.5. Emulation General Functions

```
emulation.sleep(e, seconds)
e:sleep(seconds)
```

Goes to sleep for the specified number of seconds.

8.12. key

The key module provides a simple interface into the Private Keys that are normally accessed through the web-interface. These keys are used for the Scannex 40-bit encryption service.

With this module it is possible to build a custom configuration file that includes a full or partial set of configuration parameters, plus calls to the key.set functions. This enables you to have a single file that can be quickly uploaded before the ip.buffer is shipped to your customer. You no longer have to manually key in the secrets through the web-interface.

`key.clear(keyname)`
: Simply clears the requested private key.

`key.empty()`
: Completely empties all local passwords, Scannex encryption secrets, PKI certificates and keys, and RADIUS secrets.

`key.hash(keyname, [format])` 2.82
: Returns SHA-1 hash of the secret, or NULL if the key does not exist. For 'format' see the definition under 8.6 `auth.hash`.

`key.isblank(keyname)` 2.82
: Returns true if the secret is blank for the given keyname.

`key.keys([prefix])` 2.82
: Returns a Lua table with keynames that match the prefix. These can then be iterated.

`key.license(string)` 2.90
`key.license(table)`
: Applies the operation license to the ip.buffer. Passing a table allows a set of several key-codes to be supplied to the ip.buffer - it will pick the one that applies to its serial number.

`key.save()` 2.91
: Saves the secret store to NAND Flash immediately.

`key.scriptlicense(string)` 2.90
`key.scriptlicense(table)`
: Applies the OEM script license to the ip.buffer. Passing a table allows a set of several key-codes to be supplied to the ip.buffer - it will pick the one that applies to its serial number.

`key.set(keyname, hexkey)`
: Sets an encryption private key.
Keyname is the name of the private key entry. "`chnl1`" is for channel1, "`chnl2`" is for channel 2 delivery, etc. "`weblock`" is the WebLock key, and "`oem`" is for the OEM script encryption key.
Hexkey is a hexadecimal string.
e.g. `key.set("chnl1", "010203aaeeffcd")`

8.13. log

The log module allows Lua scripts to inject custom log messages.

`log.clear()` 2.91

Clears the Log file (useful for debugging)

`log.write(id, message)`

Writes a log entry with identifier “id” and “message”.

8.14. mem

The mem module provides for Flash memory operations directly from within Lua. It is currently possible only to write to the Flash. Reading is accomplished directly in the C++ code of the delivery mechanism.

Normally this module is called from within an ad-hoc script via the web-interface. However, it would be possible to erase a channel's memory from within a protocol or detection script.

`mem.resetflags(channel)`

Resets the “`lostold`” and “`lostnew`” counters for the channel.

`mem.resetstats()`

Simply resets the “`i.mem.blocks.counts`” values.

`mem.wipefile(staticfilenumber)`

Erases a static file. Example *staticfilenumbers* include:
1 = configuration file
2 = OEM Lua script file
3 = Lua script file
4 = Private secrets (Scannex secrets, SSL/TLS Certificates and Keys, etc)
Other files are used for keeping other internal information.

```
mem.wipestore(channel)
mem.wipestore() -- wipes all unlocked                    2.92
mem.wipestore(from,to) -- wipes range                    2.92
```

Erases a given channel's storage file. The file cannot be erased if there is a delivery in progress (which would have locked the storage channel). This function, if successful, completely resets the “`lostold`” and “`lostnew`” counters, and the modified date of the file.

`mem.write(channel, string)`

Writes a string to a particular channel's storage file. “channel” is 1, 2, etc.
Returns:

- Success indicator
  - `0` = fail
  - `1` = success
  - `-1` = partial success
- Number of bytes written

8.15. modem

The modem module allows Lua scripts to control the modem (if installed).

```
rxc, txc = modem.bytes( [reset] )
```
3.00

Return the modem byte counters, and optionally clear the values.
This is the number of PPP bytes sent or received to the modem.

```
clr = modem.clearfplmn( [clr] )
```
3.01

Request that the fPLMN list is cleared from the SIM card.
clr = pass false to not clear the list
clr = returned boolean indicates if a clear is pending.

```
rslt = modem.cmd(cmd, [lookfor='\\*'], [firstms=-1],
 [maxtime=1], [retries=3])
```
3.01

Issues a command to the modem.
cmd = AT command (without the 'AT' prefix)
lookfor = string response to look for (use lowercase), or the special strings:
'*' = any text
'\\#' = any numbers
'\\l' = lines of text
firstms = time for first response
maxtime = overall response time
retries = number of retries
rslt = result string, or nil,error
NOTE: can only be called in context of the modem callback functions.

```
modem.debug(ipaddress, portd)
modem.debug(ipaddress, portd, portc)
```
2.70

Enters the modem into a debug pass-through mode. Either one or two TCP sockets can be opened. If one socket is opened, then the [ESC] key will toggle between data and control modes (where power, reset, and DTR can be controlled). If two sockets are opened, then portd specifies the data port, and portc the control port.
Positive port numbers indicate a client operation (i.e. the ip.buffer will make a connection to your listening server). Negative port numbers indicate a server operation (i.e. your PC needs to connect to the ip.buffer)[36].
e.g. `modem.debug('192.168.0.117', 9001, 9002)`
Will open a data socket on port 9001 and a control port on 9002.
e.g. `modem.debug('', -9001)`
Will listen on port 9001 for an incoming connection. Any incoming IP address will be allowed.

```
modem.decided()
```
2.60

Returns true if the modem detection is complete.

```
txt,blnk = modem.fplmn( [clr] )
```
3.01

Read and/or clear the fPLMN list in the SIM card.
clr = optional: set to true to clear the list.
txt = the fPLMN value (before clearing)
blnk = true if the txt contains only 'F'

[36] The ip.buffer will listen for a maximum of 10 minutes before aborting the debug mode.

`modem.hangup()` 2.50

Hangup the modem.
If passed a "true" boolean value this will power-cycle the GPRS modem (if attached).
e.g. "`modem.hangup(true)`"

`modem.isedge()` 2.60

Returns true if a high speed EDGE/GPRS modem has been detected.

`modem.isgprs()` 2.60

Returns true if a GPRS modem has been detected.

`modem.ispstn()` 2.60

Returns true if a PSTN/RJ modem has been detected.

`modem.ismodem()` 2.60

Returns true if a modem is present (either RJ or GPRS)

`modem.powercycle()`

Power-cycles the GPRS modem (if attached).

`modem.poweroverride(`*`value`*`)` 2.80

Override the modem power control. Allowable values:

- `-1` = no override. Power is controlled based on whether running from battery or mains.
- `0` = force power off.
- `1` = force power to 'on demand'. The modem is kept off until the ip.buffer needs to dial out.
- `2` = force power on.

`modem.reset()` 2.50

Resets the modem hold-off timers.

`txt = modem.simrb( fileid, [offset], len )` 3.01

Read SIM FileID binary, using AT+CRSM=176,fileid,offseth,offsetl,len
fileid = the decimal FileID (e.g. 28539) to read from the SIM card
offset = optional offset in bytes
len = length of the file to read

`tbl,len,orig = modem.sims( fileid )` 3.01

Read SIM FileID STATUS, from the AT+CRSM=192,fileid command
Fileid = the decimal FileID (e.g. 28539) to read from the SIM card
tbl = status array as a table. See modem.stot
len = the integer value of the tag '80' (decimal 128)
orig = the original string returned.

`ok = modem.simwb( fileid, [offset], txt )` 3.01

Write SIM FileID binary, using AT+CRSM=214,fileid,offseth,offsetl,length(txt),txt
fileid = the decimal FileID (e.g. 28539) to read from the SIM card
offset = optional offset in bytes
txt = the ASCII-hex data to write

`tbl, len = modem.stot( txt )` 3.01

Convert SIM status to table.
txt = ASCII hex data to convert
tbl = status array as a table. The index is the decimal value of the field. e.g. tbl[128] = '000C' - length
len = the integer value of the tag '80' (decimal 128)

8.15.1. Modem Callbacks 3.01

These are the Lua callbacks that are executed in the context of the modem handler thread. The callbacks allow you to hook into various modem events.

The callbacks have no parameters, and no return value.

Callback Name	When executed
`mdmOn`	Modem about to be powered on
`mdmInit`	Initialisation of modem complete (after the c.modem.init string processed)
`mdmSim`	After the SIM card has been initialised
`mdmReg`	When registered
`mdmOnline`	Internet online
`mdmOffline`	Internet offline
`mdmDereg`	When deregistered
`mdmOff`	Modem about to be powered off

8.16. net

The 'net' module provides some low level network control functions that are useful for complex scripting.

`net.arp()`
`net.arp(macstring)`
Returns a table with MAC addresses. The key of the table is the IP address, with the value as the MAC.
When a string is provided, the function returns a table of entries where '`macstring`' exists. '`macstring`' can be a full MAC address (using colons, e.g. 00:02:ae:12:34:56) or a partial MAC prefix (e.g. 00:02:ae) which can be used to locate devices by a particular manufacturer.

`net.arpclear()`
Clears the ARP table

`net.arpdel(ipaddress)`
Deletes the ARP entry for the given IP address.

`net.arpfind(macstring)`
`net.arpfind(ipstring)`
Finds an ARP entriy for either dotted IP entries, or MAC substrings.
If the parameter is a MAC address or MAC address prefix (e.g. `00:02:ae:12:34:56` or `00:02:ae`) the return value is the IP address.
If the parameter is a dotted IP address (`123.45.67.89` format) the return value is the MAC address.

`net.dnsclear(name)` 2.82
Clears the DNS table entry 'name', if it exists.

`net.firewall(true/false)` 2.92
`net.firewall(number)`
Control the firewall for the Ethernet.
When using the numbers, add the following values to decide which is firewalled:
- 1 = SYN
- 2= ICMP
- 4 = UDP (except DNS response)
- 8 = DNS response (do not use this normally!)

e.g. `net.firewall(2)  -- prevents PING`
Pressing the front button for <10s will remove the Ethernet firewall.

`net.hdap()` 2.91
`net.hdap(dhcp, ip, sn, gw)`
Returns the DHCP, IP-address, Subnet, Gateway values.
Optionally sets one, or more, of the HDAP values.

`net.hostbyname(name, [useppp], [clearfirst])` 2.82
Performs a DNS query for "`name`".
If '`useppp`' is true then the query is sent over the PPP link.
If '`clearfirst`' is true then the DNS cache entry for '`name`' is cleared first.
Returns the IP address of NIL and the error.

`net.lowpower(boolean)` 2.80

Control the low power Ethernet mode. In low power (e.g. net.lowpower(true)) the PHY is powered down when no energy is detected on the line. The PHY is powered up for 2 seconds every 10 seconds to see if energy is restored.

`net.ping(ipstringorname)` 2.82
`net.ping(ipstringorname, delaymilliseconds)`
`net.ping(ipstringorname, delaymilliseconds, useppp)`

Performs a network ping function to the given IP address or name.
The default 'delay' is 2000ms.
Returns three parameters:

- Boolean = whether responded
- String = IP address
- Integer = Maximum ping time in milliseconds

`net.power(boolean)` 2.80

Control the power on the Ethernet PHY. Powering off the Ethernet (e.g. net.power(false)) will save a significant amount of power, but means you cannot communicate with the ip.buffer! Press the front button until the red LED blinks - this will wake up the PHY for 5 minutes.

8.17. nmea

The 'nmea' library was added in v3.00 and provides basic NMEA operations.

`tab,special,ok = nmea.tab(txt)` 3.00

Convert NMEA line to a Lua table.
'`txt`' = ASCII NMEA line
'`tab`' = Table of values
'`special`' = true if this is an AIS '!' record
'`ok`' = true if the checksum is good.

`txt = nmea.txt(tab, [special])` 3.00

Convert a table to NMEA string
'`tab`' = table of values
'`special`' = true if this is an AIS '!' record
'`txt`' = output of NMEA string with checksum

8.18. os

Some helper functions have been added to the standard “os” Lua scope to improve understanding of time related functions.

`os.localtime(...)` 2.82

Returns the local time.
Equivalent to `time.utctolocal(os.time(...))`

`os.utctime(...)` 2.82

Same as os.time(...)

8.19. profiler

The Lua profiler function set provides some useful timing resources.

`profiler.getms()` 2.82
`profiler.getus()` 2.82

Returns the number of milliseconds or microseconds.

`profiler.log(string)` 2.82

Inserts a log entry with timing in microseconds.

`profiler.restart()` 2.82

Resets the timer point for getms & getus

8.20. sebus[37]

The sebus module provides the interface between Lua and the Scannex External Sensors that are connected to the SEbus[38]. The sensors can be read by these functions, and outputs can be controlled.

This module is the foundation for complex alarm detection solutions.

In the following descriptions:

`device` = 1,2,3, or 4

`sensor` = 1 or greater. Where not specified this means the first sensor of that type on the device.

`sebus.adc(device)`
`sebus.adc(device, sensor)`

Reads the Analogue to Digital Converter input. If the ADC specified does not exist this returns NULL, otherwise it returns a positive integer value.

`sebus.count(device)`
`sebus.count(device, sensor)`

Reads the current Count value from the device. If the Count input specified does not exist this returns NULL, otherwise it returns the number of edge transitions on the input since the last call to sebus.count. It is up to the script to keep track of the total count if tracking across multiple calls to sebus.count.

`sebus.dac(device, value)`
`sebus.dac(device, sensor, value)`

Outputs to the Digital to Analogue Converter for the device. Value should be a positive integer.

`sebus.get_name(device)`

Returns the name of the device (as programmed with “set_name”).

[37] From 2.30 onwards

[38] Only available in the ip.2 and ip.4 products.

`sebus.input(device)`
`sebus.input(device, sensor)`

Reads the specified digital input.
Returns:
`NULL` = device not online, or non-existent
`-1` = input has changed to "off"
`0` = input is "off"
`1` = input is "on"
`2` = input has changed to "on"

`sebus.input_invert(device, invert)`
`sebus.input_invert(device, sensor, invert)`

Specifies whether the input should be inverted. Normally the inputs are not inverted - so that closed = "on", open = "off". When inverted, closed = "off" and open = "on". When "sensor" is not specified the first digital input on the device is assumed. Invert should be either `true` = invert, or `false` = non-inverted.

`sebus.interval(seconds)`

Sets the read interval for the SEbus. The default read time is 10 seconds. This function allows the setting of a faster (or slower) read time - with a minimum of 1 second.

`sebus.online(device)`

Checks whether a particular device is online. Additionally it returns information about the device.
Returns: *online, reset, board_id, board_variant, firmware_id*
`online` : `false` = device not online, `true` = device online.
`reset` : `false` = device not reset, `true` = device has been reset since the last call to sebus.online.
`board_id` : the board identifier. Specifies the type of board.
`board_variant` : the board variant. Specifies which fit of board.
`firmware_id` : a firmware identifier for the device.

`sebus.output(device, value)`
`sebus.output(device, sensor, value)`

Sets the output for an SEsensor. The value can be one of the following:
`0` = off
`-1` = on
`1..n` = on for that many seconds. e.g. `sebus.output(1, 1, 10)` will turn the output on for ten seconds.

`sebus.output_invert(device, invert)`
`sebus.output_invert(device, sensor, invert)`

Specifies whether the output should be inverted. Normally the output are not inverted - so that closed = "on", open = "off". When inverted, closed = "off" and open = "on". When "sensor" is not specified the first digital input on the device is assumed. Invert should be either `true` = invert, or `false` = non-inverted.

`sebus.power(on)`

Controls the power to the SEbus. When first booted the SEbus is powered off. To enable all the devices on the SEbus you should enter "`sebus.power(true)`" into the script. It is possible to power off the SEbus to conserve power when running on battery – with the aid of a small script.

`sebus.speed(settle, clocking, interval, readgap)`

Internal function used to adjust the clocking frequency and timing of the SEbus.

`sebus.set_name(device, name)`

Sets a human-readable name for the SEsensor. Shown on the status screen.

`sebus.temperature(device)`
`sebus.temperature(device, sensor)`

Returns the temperature from the SEsensor. The value returned is a signed integer value in degrees Celcius.

8.21. serial

The serial module provides read and control ability for serial handshake lines and pin control.

Note: "port" is between 1 and 4.

`serial.control(port, scriptcontrol)` 2.91

Specifies whether the COM port is under script control. Requesting script control, e.g. `serial.control(1, true)`, will put all outputs (DCE/DTE, RTS, DTR) purely under script control.
Releasing control, e.g. `serial.control(1, false)`, will put the port back under firmware control.

`serial.getcts(port)` 2.80
`serial.getdsr(port)`

Returns the state of the pin for the port number.
Returns:

- Current = whether the pin is current asserted
- Flow = whether pin is being used for flow control

`serial.getdtr(port)` 2.80
`serial.getrts(port)`

Returns the state of the pin for the port number.
Returns:

- Current = whether the pin is current asserted
- Flow = whether pin is being used for flow control
- Manual = the current manual pin state

`serial.getpin(port)` 2.80
`serial.setpin(port, pin)`

The DCE/DTE pin condition.
Pin values:
0 = port is in the "Off" state
1 = DCE mode
2 = DTE mode
3 = Both pins 2 and 3 show detected data
4 = Auto pin detect mode

`serial.setdtr(port, manual)` 2.80
`serial.setdtr(port, manual, flow)`
`serial.setrts(port, manual)`
`serial.setrts(port, manual, flow)`

Set the handshake pin state and whether flow controlled.

`serial.rxbreak(port)` 2.90

Returns the number of break sequences received on the port.
Note that the USART will increment the break sequence count at the beginning of the sequence *and* at the end.

`serial.txbreak(port, milliseconds)` 2.90

Transmit a break to the serial port. Milliseconds between 1 and 2000. Calling with milliseconds=0 will stop any break sequence in progress.

`serial.txing(port)` 2.93

Returns true if the serial port is currently sending.
NOTE: be careful about adding a while-loop. If called in the context of the source the poll may not be called - and the flag may never change (unless you call read etc)

8.22. snmp

The snmp module provides a convenient way to generate custom traps from the ip.buffer. Traps generated this way are sent immediately. If you want to tie in trap generation with the alert system, then see the “alert.register_trap” function.

`snmp.decode(packet, ip, name)` 2.93

Decodes a binary SNMP trap packet, and returns an ASCII representation.

`Snmp.decoderset(delim, vardelim, escape)` 2.93

Sets the SNMP trap decoder delimiter string, variable-delimiter string, and whether to escape the result (boolean).

`snmp.trap(message)` 2.10
`snmp.trap(specifictrap, message)`
`snmp.trap(specifictrap, message, channel)`

Generates a trap, using the “specifictrap” (or 9 if not specified), and the provided message and channel number. Specifictrap numbers that encroach on the ip.buffer's hard-coded traps will be limited (currently to specific trap 68). For this reason, you should choose traps that are safely out of the way – e.g. 10000+.

8.23. socket

The socket module provides access to simple UDP/IP, TCP/IP, and TLS/SSL services for custom Lua functions.

By using "`tostring(socketobject)`" you can acquire information about the socket - its type, and the TCP or UDP socket handle number (to tie up against the Network Tables list).

- Note the use of the colon ':', not dot '.' when calling methods of the socket object.

`socket.list` 2.93

A table that contains the socket objects currently in existence.

`socket.new('udp')` 2.93
`socket.new('tcp_client')`
`socket.new('tcp_server')`

Creates and returns a socket object - either UDP/IP, TCP/IP, or TCP/IP server.
The number of socket objects that can be created is limited.
The objects are tracked in the `socket.list` table.

`socket.select(timeoutms, rxtable, [txtable, [excepttable]])` 2.93

Waits for activity on a set of socket objects, up to "timeoutms" milliseconds.
Returns: `count, rxtable, txtable, excepttable`

8.24. socket.new – all

These functions apply to UDP, TCP, and TCP server objects.

`object:close()` 2.93

Closes the socket object. The object itself is not destroyed until all references have been removed, and the garbage collection occurs.

`object:flush()` 2.93

Flushes, and throws away, any incoming received data.

`object:isopen()` 2.93

Returns true if the object is open.

`object:wait(timeoutms)` 2.93

Waits for data to arrive, up to timeoutms miliseconds.
Returns a boolean value indicating that data has arrived.

8.25. socket.new('tcp'_client)

The tcp socket object handles both client connections, and established incoming connections. Once connected, the TCP object can be upgraded to TLS/SSL.

`object:inflight()` 2.93

Returns the number of bytes waiting to be sent on-the-wire and acknowledged by the remote end.

object:open(*hostname, hostport, [timeoutms, [interface]]*) 2.93

Tries to make a TCP/IP connection to hostname:hostport.
Returns the number of bytes sent[39].

object:send(*data*) 2.93

Sends data down the TCP/IP connection.

object:recv(*[max]*) 2.93

Read data from the socket, up to 'max' bytes.
Returns:

- NIL, errorcode, errorstring
- datastring

object:ssl(*isclient, [checkcerts]*) 2.93

Attempts to upgrade the connection to TLS/SSL.
Returns true if successful.

object:sslclose() 2.93

Tears down the TLS/SSL connection and brings it back to plaintext.

8.26. socket.new('tcp_server')

The svrtcp object handles incoming server connections. No data is actually sent or received on this object - it is just for 'catching' the incoming connections and spawning off a tcp connection object.

object:accept() 2.93

Accepts an incoming connection request and returns a 'tcp_accept' object to handle the connection. (with same methods as the object returned from `socket.new('tcp_client')`)

object:open(*allowlist, localport, [interface]*) 2.93

Opens the TCP server socket. Incoming connections are screened against the allowlist (same symantics as used in the source channel 'allow' fields)

object:setbacklog(*n*) 2.93

Sets the limit for backlog connections to n.

8.27. socket.new('udp')

object:open(*localport*) 2.93

Opens the UDP socket.

object:send(*data, hostname, hostport, [interface]*) 2.93

Sends UDP/IP data to hostname:hostport. If the interface is present, the UDP traffic is routed over that interface - '`ppp`', '`et`', '`lo`' for PPP, Ethernet, and Loopback respectively.

object:recv(*[max]*) 2.93

Read data from the socket, up to 'max' bytes.
Returns:

- NIL, errorcode, errorstring
- datastring, remoteIP, remotePort, localIP, interfacename

[39] Technically the number of bytes written to the TCP-stack output buffer. Use the :inflight() function to query how much is yet to be sent on the wire.

8.28. source

The source module provides an interface into the source of each channel, allowing the writing of custom protocols directly in Lua. See section 13 for more information about the context in which these calls should be made.

`source.close(protocol)`

Closes the source (if applicable). Does nothing when the source is UDP or COM.
"`protocol`" is the protocol table.

`source.debug(protocol, string)`

"`protocol`" is the protocol table.
"`string`" is the message that is sent to the pass-through when it is in Debug mode.

`source.error(protocol, string)`

"`protocol`" is the protocol table.
"`string`" is the error message that is registered. This error becomes available to the web status screen.

`source.event_wait(protocol)`

This is an internal function that is used to glue Lua to the rest of the protocol engine. There is an auxiliary function `source.active_loop` that repeatedly calls this method. You must not call these functions normally! (It is documented here for completeness)

Returns: `quit, data, second`

"`quit`" the protocol engine needs to shut down.
"`data`" there is data. Please call `protocol.data`.
"`second`" it is time to call `protocol.second`.

`source.inject(protocol, function, …)` 2.82

The inject method allows a function to be inserted into the source's protocol stream - for handling direct communication with the connected device. Once the function has been executed it is unchained from the list of injectors.
"`protocol`" is the protocol table.
"`function`" is called, along with any parameters given.
e.g. `sources[1]:inject(function(p) p:write('Hello!') end)`
e.g. `sources[1]:inject(myinjectfn, 'A', 'B', 123)`

`source.injecta(protocol, function, …)` 2.82

As with source.inject, but will queue the injector function even when the source is not currently connected.

`source.injectclear(protocol)` 2.82

Clears the inject function list for the protocol.

`source.push(protocol, txt, timeoutflag)` 2.92

Injects data into the Cloud source[40].

`source.pushopen(protocol, filename, contentlength)` 2.92

Used for a source that is in the 'cloud' mode. Triggers the start of a Cloud transfer, and generates the {cloud begin...} tag.

[40] The x.cloud.ondata function hook links to a function that automatically wraps up the pushopen / zlib.inflate / push / pushclose operations.

`source.pushclose(protocol, filename, totallength)` 2.92

Used for a source that is in the 'cloud' mode. Triggers the end of a Cloud transfer, and generates the {cloud end...} tag.

```
source.read(protocol)
source.read(protocol, bytes)
source.read(protocol, bytes, timeout)
source.read(protocol, bytes, timeout, mask)
source.read(protocol, bytes, timeout, mask, stripcodes)
```

Reads data from the source.

"`protocol`" the protocol to read from. Normally, you would use the object-oriented style "p:read(...)" rather than using the "source." terminology.

"`bytes`" number of bytes to read or wait for. If not present, or zero, then this will read all that is available.

"`timeout`" a timeout, in milliseconds, to wait for the data to arrive.

"`mask`" defines whether to mask D7 for ASCII. 0, or not present mean data is presented as full 8-bit. 1 = top bit is stripped.

"`stripcodes`" defines whether to remove non-printable ASCII characters. If 1 then only codes above 0x20, plus 0x0d (CR), 0x0a (LF), 0x09 (TAB), and 0x0c (FF) are kept.

Returns: `string, timedout`

"`string`" the returned data

"`timedout`" false=did not timeout, true=timedout

```
source.readbytes(protocol, bytes)
source.readbytes(protocol, bytes, timeout)
```

Calls source.read and converts the resulting string into a series of bytes.

"`protocol`" the protocol to read from.

"`bytes`" number of bytes to read or wait for. If not present, or zero, then this will read all that is available.

"`timeout`" a timeout, in milliseconds, to wait for the data to arrive.

Returns: bytes (as a multiple set of value).

e.g. `a,b,c = p:readbytes(3)`

```
source.readline(protocol)
source.readline(protocol, timeout)
source.readline(protocol, timeout, stripcodes)
```

Reads data in from the source and handles the buffering and splitting of data into ASCII lines that are bounded by CR and/or LF.

"`protocol`" is the protocol table.

"`timeout`" a timeout, in milliseconds, to wait for the data to arrive. If timeout is not provided in the call, the value of `protocol.timeout` is used instead.

"`stripcodes`" defines whether to remove non-printable ASCII characters. If 1 then only codes above 0x20, plus 0x0d (CR), 0x0a (LF), 0x09 (TAB), and 0x0c (FF) are kept. If the `stripcodes` parameter is not provided, then the inverted value of `protocol.codes` is used instead (ie. If `protocol.codes=1` then this allows full 8-bit data).

Returns: `string, timedout`

"`string`" the returned line

"`timedout`" false=did not timeout, true=timedout

`source.readuntil(protocol, string, timeout)` 2.82
`source.readuntil(protocol, table, timeout)`

Reads from the source until either the string pattern, or table of string patterns, is found.

Returns:

1. string up to the match
2. whether timed out
3. string that was matched

`source.record(protocol, string)`

Updates the source quiet counter and last-record value.

"`protocol`" is the protocol table.

"`string`" is the data that is stored in the `d.chnl[#].src.data.last` field.

`source.sleep(protocol, milliseconds)` 2.82

Sleeps for the given number of milliseconds. Unlike "tools.sleep" this is not limited to 2000ms (2s), and will return immediately if the source is asked to abort.

`source.splitline(protocol, string)`
`source.splitline(string)`

Performs a split of the line, looking for CR or LF characters. This function is not normally required as the process of reading data and converting it into lines of ASCII text is handled by the function method "`readline`".

"`protocol`" is the protocol table.

"`string`" is the raw text to split.

Returns: `line, remainder`

"`line`" is the first line from the text, or NULL if none found.

"`remainder`" is the remainder of the raw text.

`source.stopping(protocol)` 2.82

Returns true if the source is being asked to terminate. Useful for long loops to exit in a timely manner.

`source.store(protocol, string, tag)`

Stores data with time-stamping.

"`protocol`" is the protocol table.

"`string`" is the data to store.

"`tag`" is the data tag to pass onto the onrecord event.

`source.storebinary(protocol, string, tag)`

Stores data without time-stamping.

"`protocol`" is the protocol table.

"`string`" is the data to store.

"`tag`" is the data tag to pass onto the onrecord event.

`source.timestamp(string)`
`source.timestamp(protocol, string)`

Converts the time-stamp string into a value that represents the current time.

"`protocol`" is the protocol table.

"`string`" is the time-stamp string.

Returns: string.

`source.unwindline(`*`protocol, string`*`)`

Unwinds a chunk of text, putting it back into the 4k internal line buffer. This enables a call to be made to "`readline`", and then undo that action so that the next call to `readline` will pull back the data.
"`protocol`" is the protocol table.
"`string`" is the text (normally including the CR/LF).

`source.write(`*`protocol, string`*`)`

Writes data out to the source. The behaviour depends on the context of the call. If this is called within the context of the channel that owns the protocol, then the call is blocking (it will wait for the data to be transmitted before the function returns). However, if it is a cross-channel write, then the data is queued into a 4k transmit buffer, and the channel's task will transmit it when it is able – effectively the call is non-blocking (and if the 4k buffer overflows, then data is lost)[41].
"`protocol`" is the protocol table.
"`string`" is the text to write to the source.

`source.writebytes(`*`protocol, ...`*`)`

Converts the passed series of byte values into a string and calls source.write.
"`protocol`" the protocol to write to.
"`...`" the series of bytes to write.
e.g. `p:writebytes(0xee, 0xff, 1, 2, 3)`

[41] This resolves problems that can occur if a script writes to a source faster than it can be sent. It prevents the whole ip.buffer from locking up!

8.28.1. sources

To enable cross-channel writing (only)[42], there is a global table called "`sources`". This table enables access to the protocol table for each channel.

For example, to refer to the protocol for channel 1, you can use "`sources[1]`".

So, to write to channel 1: "`source.write(sources[1], "Hello")`", or "`sources[1]:write("Hello")`" would both work.

If you wish to write out through this channel's source then within a protocol script you can use the abbreviated form: "`p:write("Hello")`". However, since the "onrecord" event is not passed the protocol table you must use the expression "`source.write(sources[c], "Hello")`" (where "c" is the channel number that is passed to the onrecord event. See section 9).

[42] All other function methods of "source" are only available within the context of a script running within the channel's task space. For example, within the protocol sections, or in the "onrecord" event.

8.29. string

There are a few extensions to the Lua string class:

`string.iterate(s, func, list, …)`

Used by .findm & .matchm. Iterates through the table of strings 'list' until the 'func' returns something other than NULL.

`string.findm(s, {lookfor, lookfor, …}, …)` 2.90
`string.matchm(s, {lookfor, lookfor, …}, …)` 2.90

The same as string.find & string.match, but allows for optional table of values to look for. Useful when looking for alternative values.
In addition to the usual return values, it also returns the pattern that was matched (as the last in the list of returns)

8.30. struct

The struct module[43] adds C style packing and unpacking (for byte-wide values, not bits).

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

`struct.pack(fmt, d1, d2, ...)` 2.93

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

`struct.unpack(fmt, s, [i])` 2.93

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.size(fmt)` 2.93

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.

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)

[43] Added in firmware 2.93

- "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

Examples

1. To match:

```
struct Str {
 char b;
 int i[4];
};
```

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

2. 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.

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

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

8.31. thread

The thread module adds the ability to run concurrent Lua functions in separate INTEGRITY tasks.

It is similar to the x.onrun function, but allows dynamic creation of the threads.

`n, max = thread.count()` 2.95

Returns the current number of running threads, and the maximum allowed.

`thread.list` 2.95

A list of currently running threads.
Use pairs to iterate:

```
local idx,trd
for idx,trd in pairs(thread.list)
do
  -- idx is the internal C++ reference userdata
  -- trd is the thread object (see the 'thread.new Object'
    section)
end
```

`object = thread.new(function, [name], [timeout])` 2.95

Returns the a new thread object. The Lua 'function' will be the entry point of the thread, with the optional Lua timeout value (in milliseconds).
Both the thread object and 'name' are passed to the 'function':

```
function ThreadOne(athread, aname)
  while athread:run()
  do
    -- do something long...
  end
end

local mythread = thread.new(ThreadOne, 'SampleThread')
```

8.32. thread.new Object

`object:die()` 2.95

Requests that the thread object dies.

`object:run()` 2.95

Returns true if the thread should continue running.

`object:running()` 2.95

Returns true if the thread is running.

`object:wait([seconds])` 2.95

Waits until the thread object has completed.
Returns true if the thread has completed before the timeout.

8.33. time

The time module adds some useful time related functions. See also the regular Lua functions “os.date” and “os.time”.

`time.calcdst(number)` 2.50

Pass the epoch time (seconds since 1st Jan 1970), and this returns the time and whether the time is DST.

`time.dayofweek(number)` 2.82

Returns the day of the week for the given date+time.
1=Sunday.
NOTE: Does not work with days prior to 1970. This Lua call works for all dates:
`os.date('*t',os.time({year=1960,month=1,day=1}))['wday']`

`time.localtoutc(number)` 2.82

Converts a local time to UTC time.
Returns: UTC time, whether local time on DST

`time.set(number, isdst)` 2.50

Pass the local epoch time (seconds since 1st Jan 1970), whether DST is active. Sets the current time.

`time.setlocal(number)` 2.50

Pass the local epoch time (seconds since 1st Jan 1970). Sets real time clock. Also returns the time and whether currently in DST as return parameters.

`time.setutc(number)` 2.50

Pass the UTC epoch time (seconds since 1st Jan 1970). Sets real time clock. Also returns the adjusted time (for time zone and DST) and whether currently in DST as return parameters.

`time.utctolocal(number)` 2.82

Converts a UTC time to local time.
Returns: local time, whether on DST

8.34. tools

The tools module provides a few useful routines (often needed for legacy emulation).

`tools.abortdst(chnl)` 2.92
`tools.abortpass(chnl)` 2.92
`tools.abortsrc(chnl)` 2.92

Aborts the destination/passthru/source for the given channel number.

`tools.ascii(string)` 2.50
`tools.ascii(string, boolean)`
`tools.ascii(string, string)`
`tools.ascii(string, number)`

Converts 8-bit data into ASCII 7-bit data by stripping bit 7 (or AND-ing with 0x7f).
If the 2nd parameter is a boolean then "false" means remove control code (except CR/LF/FF/BS/TAB), and "true" means leave control codes.
If the 2nd parameter is a string then it should contain a list of control codes to keep.
e.g. `s = tools.ascii(s, '\r\n')`
If the 2nd parameter is a number then it is taken as a 32-bit field. If the corresponding bit is set then the control character is kept.
e.g. `s = tools.ascii(s, 0x00002400) // bits 10 and 13 (CR & LF)`

`tools.batteryoff()` 2.70
`tools.batteryoff(seconds)`

Powers off when running on batteries. If an integer is passed as a parameter, the power off will occur after that number of seconds. Calling with a time of -1 seconds will abort the battery shutdown timer.

`tools.bool(v)`

Converts v to a pure boolean value where `false`, `nil`, `0` (zero), and " " (blank string) are all considered false. Anything else is considered true.

`tools.checkforupdates()` 2.50

Triggers the HTTP update process.

`tools.clearmenu()` 2.90

Clears the custom menus and reverts to the built-in menus.

`tools.debug(string)`

Sends the string out to the UDP debug port (if enabled)

`tools.disablecrypto()` 2.90
`tools.disablecrypto(string)`

Disables crypto facilities (for testing)

`tools.disablefeatures(value)`

Disables firmware features (for testing)

`tools.fixending(string)` 2.75
`tools.fixending(string, ending)`
`tools.fixending(string, ending, search)`

Fixes the line endings. By default this changes all CR, CRLF, LF to CR+LF.
The 2nd parameter specifies what line ending to use - to override the CR+LF. For example, to replace with just CR use "`tools.fixending(text, '\r')`"

The 3rd parameter specifies what line endings characters to replace. By default this is '\r\n'. For example, to change all combinations of CR, LF, FF, use "`tools.fixending(text, '\r\n', '\r\n\f')`"

`tools.gc([what, [data]])` 2.93

Calls the Lua garbage collector. See lua_gc in the Lua 5.1 manual.

`tools.get(string)`

Looks up the Lua value and returns it.
e.g. `myvar=tools.get("c.network.name")`
Note that tables can be returned as well.
e.g. `mytab=tools.get("c.network")`

`tools.getfield([table,] string)` 2.82

Safely accesses the offset into the optional table (or root if not present) based on the string. Will return NULL if the value does not exist.

`tools.gethwid()` 2.90

Returns the hardware ID information:
`BoardID, ProductID, SSL, Features`

`tools.getmanuf()` 2.90

Returns the manufacturing string.

`tools.istls()` 2.60

Returns true if using the SSL/TLS enabled firmware.

`tools.ledduty(percentage)` 2.80

Sets the LED duty cycle. A percentage of zero will put the LEDs back to their default duty cycle.
When running on the battery the LEDs are limited to a maximum of 25%.
A low value, e.g. 5, will dim the LEDs and save about 6mA on an ip.1

`tools.lock()`

Runs Lua exclusively from this point. It provides a way of guaranteeing atomic operations (like updating several table entries in one go).
Note that tools.unlock() must be called within 1 second, otherwise the running Lua thread will be terminated!

`tools.neterrstr(integer)` 2.70

Returns a string message from a network error number.

`tools.reboot('Lua')` 2.50

```
tools.reboot('ip.buffer')      // Reboots in 5 seconds
tools.reboot('ip.buffer', 0)   // Cancels reboot
tools.reboot('ip.buffer', -1) // Returns the time to reboot
tools.reboot('ip.buffer', 60) // Reboots in one minute
```

Reboots either the Lua core, or the ip.buffer (i.e. a cold boot).
For the 'ip.buffer' calls, the function always returns the old reboot counter. Passing a positive integer will only initiate the reboot if there is not a reboot currently pending.

`tools.run()` 2.93

Returns true if Lua should still run. Use this instead of "`while true do`", e.g. "`while tools.run() do`"

`tools.set(string, value)`

Sets a Lua entry to the given value.
e.g. `tools.set("c.network.name", "MyBoxName")`

`tools.setfield([table,] string)`

Safely write the offset into the optional table (or root if not present) based on the string. Will create the table tree structure if it isn't already there.

`tools.setmenu(integer, string, string)` 2.90

Apply a custom menu. The first integer is the index (0 to 7), followed by the URL and the menu text.
Do not include the leading slash,
e.g. `tools.setmenu(0, 'setup/setup.shtm', 'setup')`

Creating an index menu of zero will also replace the main index page.

`tools.sleep(ms)` 2.70

Sleeps for the given number of milliseconds. The value is limited to 2000ms (i.e. 2seconds)
e.g. `tools.sleep(250)`

`tools.unlock()`

Removes the exclusive Lua lock. Note you can only call this after tools.lock()

`tools.unquote(string)`

Returns a correct version of the "string", replacing all the "\" tokens with their correct values.
e.g. `s=tools.unquote("My\tString\r\n")`

8.35. trigger

The trigger module provides an interface to trigger or cancel a push delivery on a channel.

`trigger.cancel(channel)`
: Cancels a push on the given channel.

`trigger.cancelall()` 2.40
: Cancels a push on all channels.

`trigger.push(channel)`
: Triggers a push on the given channel.

8.36. twi

The 'twi' library provides core I2C hardware interface operations, and was added in firmware v3.00.

```
a1,... = twi.qy()                                                    3.00
ok = twi.qy(devf,[devt])
```

Query TWI/I2C bus.
'devf' = device address to start from
'devt' = device address to finish
'a1' = device addresses
'ok' = if found

```
res = twi.rd( dev, num )                                             3.00
res = twi.rd( {dev,addr}, num )
res = twi.rd( {dev=dev, addr=addr, alen=alen, slow=slow}, num )
```

Read from TWI/I2C bus.
'dev' = device address
'addr' = address inside device
'alen' = number of bytes for the address (optional)
'slow' = slow setting (optional)
'num' = Number of bytes to read

```
res = twi.wr( dev, txt )                                             3.00
res = twi.wr( {dev,addr}, txt )
res = twi.wr( {dev=dev, addr=addr, alen=alen, slow=slow}, txt )
```

Write to TWI/I2C bus.
'dev' = device address
'addr' = address inside device
'alen' = number of bytes for the address (optional)
'slow' = slow setting (optional)
'txt' = Data to send

8.37. udp

The udp module[44] provides a simple, single socket, UDP mechanism designed primarily for broadcasting data on the local LAN.

***NOTE:** Currently the UDP module provides a single socket only. This interface may change in a later firmware version.*

`udp.open([localport])` 2.93

Opens the UDP socket. If localport is not present then an ephemeral port is used for the local side.

`udp.send(host, remoteport, text, [interface])` 2.93

Sends a string to the host on UDP port remoteport.
Interface can be “modem”, “ppp”, “eth” or “loop”

`udp.waiting()` 2.93

Returns the number of bytes waiting to be read on the UDP socket.

`udp.read([max])` 2.93

Returns *text*, *senderip*, *destip*

`udp.flush()` 2.93

Throws away any received data.

`udp.close()` 2.93

Closes the UDP socket.

[44] Added in firmware 2.93

8.38. util

The 'util' library was added in firmware v3.00 and provides some useful Lua operations.

`t = util.add(t1, t2, ...)` 3.00

Add Lua tables. Combines two, or more, tables into one table. Uses Lua 'pairs'.

`t = util.addi(t1, t2, ...)` 3.00

Add Lua tables. Combines two, or more, tables into one table. Uses Lua 'ipairs'.

`u = util.au16(a, [be])` 3.00

Convert ASCII to UTF-16.
'a' = ASCII text
'be' = set to true for Big-Endian encoding (default)
'u' = UTF-16 encoded text

`u = util.au8(a)` 3.00

Convert ASCII to UTF-8. Simple conversion of 0x00-0x7f and 0x80-0xff into double character sequences.

`t,rtxt = util.csvt(txt, [opt])` 3.00

Convert CSV text to a Lua table.
'txt' = ASCII line to parse
'opt' = optional table of parameters: 'opt.s' = separator, 'opt.q' = quote character
't' = table of values
'rtxt' = remaining text (for incomplete quote etc)

`t1,t2 = util.cuti(t, idx)` 3.00

Cut an indexed table in two.
't' = table to cut
'idx' = index of first entry to be in t2
't1' = first half
't2' = second half

`v = util.dct(t)` 3.00

Deep copy a table. Any changes made to 'v' will be independent of 't'
't' = table to deep copy.
'v' = resulting copy of table.

`idx,found = util.find(t, v, [func])` 3.00

Find in a table using binary search.
't' = table of values
'v' = value to find
'func' = optional comparison function
'idx' = index (effectively the insertion point)
'found' = whether found in the list

`n = util.iter(FOO, start, [depth])` 3.00

Iterate a table, calling a function for each object.
'FOO' = function to call. Prototype is `n=FOO(o)`
'start' = starting object or table
'depth' = optional table depth (default = 3)

'n' = count, or total
NOTE: the function FOO can return a value that's summed to form the total n.

```
t = util.remi(t, idx, [last])
```
3.00

Remove from an indexed table.
't' = table to cut
'idx' = entry to remove, or first entry to remove
'last' = optional last entry to remove

```
t = util.tia(ta, tb)
```
3.00

Append two indexed tables.
'ta' = First table
'tb' = Second table
't' = resultant table

```
t = util.tio(ta, tb)
```
3.00

'OR' together indexed tables. If any indexed entry of 'ta' is nil, the value of 'tb' is used.
'ta' = First table
'tb' = Second table
't' = resultant table

```
u = util.u16u8(uu, [be])
```
3.00

Convert UTF-16 to UTF-8
'uu' = UTF-16 encoded text
'be' = set to true for Big-Endian encoding (default)
'u' = UTF-8 encoded text.

```
a,ok = util.u8a(u, [d])
```
3.00

Convert UTF-8 to ASCII.
'u' = UTF-8 encoded text
'd' = Text to replace undefined characters (default = '?')
'a' = ASCII text
'ok' = true if the decoding was ok

```
uu,ok = util.u8u16(u, [be])
```
3.00

Convert UTF-8 to UTF-16
'u' = UTF-8 encoded text
'be' = set to true for Big-Endian encoding (default)
'uu' = UTF-16 text
'ok' = true if the decoding was ok

8.39. zlib

The zlib module provides facilities for compressing and decompressing strings.

`zlib.check(txt)` 2.92

Checks whether txt is a zlib stream (with 2 byte header). Returns true/false.

`zlib.deflate(txt)` 2.92
`zlib.deflate(txt, isdeflate)` 2.92
`zlib.deflate(txt, windowsize)` 2.92
`zlib.deflate(txt, windowsize, level)` 2.92

Compresses the string txt.
If 'isdeflate' is true then uses the header-less raw stream format (compatible with C# DeflateStream).
Alternatively 'windowsize' specifies the dictionary window size. The default value of 15 produces a zlib-header output. A value of -15 produces a raw stream. Adding 16 (e.g. 31) outputs a gzip format header. See the zlib documenation for initdeflate2.

`zlib.inflate(txt)` 2.92
`zlib.inflate(txt, isdeflate)` 2.92
`zlib.inflate(txt, windowsize)` 2.92

Decompresses the string txt.
If 'isdeflate' is true this assumes that txt is a header-less raw stream – as output by C#'s DeflateStream class.
Alternatively, 'windowsize' specifies the type of stream – see zlib.deflate options.

8.40. Other useful functions

These functions are provided for general use inside emulations, or protocols.

`applycrlf(string)` 2.10

If the supplied “string” has no CR or LF, then the function appends CRLF.

`asciihex(string)` 2.10

Returns a human readable HEX version of the supplied string.

`stripcodes(string, keep)` 2.10

Returns a string where all control codes, except those listed in “keep” have been stripped. e.g.

```
s = stripcodes(s, "\r\n\t") -- remove everything but CR, LF and TAB
```

9. Executing Lua every time data is stored

Whenever a channel's protocol calls `source.store` or `source.storebinary`, the Lua core will allow an "`onrecord`" event for that channel. This gives the opportunity to analyze, modified, and split the data.

The "`onrecord`" hook is made by assigning:

`x.chnl[#].src.onrecord`

(where # is the channel number).

> - Once the hook is assigned it becomes the responsibility of the onrecord function to store the data!

Lua will pass three parameters to the onrecord function:

1. The string to store
2. The channel number
3. The data tag (as passed to `source.store*`)

e.g.

```
x.chnl[1].src.onrecord = function(txt, c, tag)
  -- txt is the string
  -- c is the channel number
  -- tag is the tag string
  mem.write(c, txt) -- perform a simple write
end

x.chnl[2].src.onrecord = function(txt, c, tag)
  -- Read from channel 2 and write out of channel 3
  sources[3]:write(tag..':'..txt)
end
```

Of course, standard Lua applies. So you can define a named function and link to this directly. For example, you may wish to provide your customer with a complex script that does some special functions. You can define the function inside your encrypted OEM script, and allow the user to reference this in their script:

```
-- Inside the OEM script
function our_special_function(txt, c, tag)
  -- Hidden functionality can go here...
  mem.write(c, tag..':'..txt)
end

-- Inside the OEM script OR the normal script
x.chnl[1].src.onrecord = our_special_function
```

This way the source of "our_special_function" can be hidden by encryption, yet the customer can make use of that code. Obviously they will need to be instructed about the name of the function etc.

10. Filtering data for pass-through operations

When the source pass-through TCP socket is connected, there are now Lua functions that allow filtering of the received and transmitted data. The use of these functions can allow for stripping of control characters, for masking digits or other operations.

The two function hooks are:

```
x.chnl[#].pass.rx
x.chnl[#].pass.tx
```

(where # is the channel number).

The “rx” function is called whenever data arrives at the source port. Filtering here will alter what the TCP client sees.

The “tx” function is called whenever the TCP client sends data to be transmitted to the source port. Before the data is sent to the port this filter function is called.

- Be aware that these functions do not see “records” or “lines” of data. Arbitrary sized chunks of data are sent through these filters.

In both cases, Lua will pass two parameters to the filter function:

1. The string to filter
2. The protocol table[45]

For example, in order to strip the top bit of data for incoming data, apply this script:

```
function striprx(s,p)
 return tools.ascii(s)
end

x.chnl[1].pass.rx = striprx
```

[45] The protocol table has a number of fields. e.g. “p.n” is the channel number

11. Executing Lua code every second

It is possible to execute a Lua function repeatedly with a one second delay between calls[46].

e.g.

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

x.onsecond = myonsecond
```

- Note: don't use this routine to count seconds – make use of the “i.now” & “i.seconds” values. The ip.buffer will call the attached function, then pause for 1 second before looping.

[46] The function is called after Lua has been configured on bootup.

12. Calling code after boot completes

Occasionally it is necessary to call some fix-up Lua code once all the boot process completes. For example, the OEM Lua script is called first and may need to hook things up based on User-script settings or configuration settings.

`postboot.add(function)` 2.90

The function is called after boot completes.
No parameters are expected.

- Uses “postboot.list” to hold a list of functions.
- “postboot.run” is called by the ip.buffer when bootup completes – to iterate through the list of registered functions in the “postboot.list” table.

13. Writing new protocol scripts

The ip.buffer has a very complex and full mechanism to allow for new protocols to be added to the device – without requiring Scannex to update the firmware! This section of the manual outlines how you extend the protocol list.

In the web-page for the source the user can select the required protocol from a drop down list. This list of protocols is built from the Scannex-defined protocols (which are hard-coded in firmware[47]), and user-installed protocols.

13.1. Writing the protocol

13.1.1. Where to put the protocol code

It depends on your business model as to where you place the code. The easiest place is in the normal script area. However, any code written to this area is visible to anyone with the admin password – including your customers and your competitors.

In order to hide any protocols you write, you will need to assign an OEM Private Key to each ip.buffer you ship, and encrypt the OEM script[48].

13.1.2. Beginning the definition

Begin your protocol definition with the following Lua lines:

```
protocols.myprotocol = {}
protocols.myprotocol.desc = 'My protocol description'
```

The “myprotocol” should be something reasonably descriptive. For example, you may choose to use “necneax” for the NEC NEAX2400 protocol. So the above lines would become:

```
protocols.necneax={}
protocols.necneax.desc= 'NEC NEAX2400 Serial'
```

The following sections deal with each parameter that should be defined for the protocol.

13.1.3. Binary style protocol

If the protocol is binary in nature (such as the Realitis/iSDX binary, or the “Binary” protocol) then define the following entry:

```
protocols.myprotocol.binary = 1
```

This entry will disable the “Time Stamp” field in the web page.

13.1.4. Bidirectional protocol

If the protocol requires full bidirectional access to the source to work, then define the following entry:

```
protocols.myprotocol.bidi = 1
```

This is only required for protocols that need to have a full “conversation” with the connected device. For example, the Avaya RSP protocol is currently the only Scannex-supplied protocol that has this entry.

[47] Even though they are hard-coded, the Scannex supplied protocols are still written in 100% Lua.

[48] The OEM script is kept encrypted inside the ip.buffer. It is decrypted when run.

Protocols marked as bidirectional will prevent the pass-through from connecting in anything except “Monitor” and “Debug” mode. In addition, the protocol will not interface with source types that are unidirectional (ie. UDP and FTP).

13.1.5. Parameters for the web page

If the protocol has additional parameters that require user input, then define the following entry[49]:

```
protocols.myprotocols.params = 'string'
```

The actual value of 'string' is complex. You can define drop-down combo box entries and edit-box entries for the web. Each parameter is separated by the bar character '|'.

Drop down combo-box

To define a drop down combo-box, use the following method:

```
S;var;label;hint;default;choice1=text1;choice2=text2...
```

The “`var`” is the variable name. The user's choice will be loaded into the protocol table as `p.var` and can be referenced within all of the protocol events (setup, connect, data, second).

The “`label`” is the text that appears on the web-page for that parameter.

The “`hint`” is the text that appears on the far right side of the web-page.

The “`default`” is the default value for the variable.

The list of “`choice1=text1`” define the variable definition and the text that is displayed in the combo box.

As an example, the “ASCII lines” built-in protocol defines whether to send XON characters to the source. The string for that option would be:

```
S;xon;XON;Software flow controlled?;0;0=None;1=Send
```

The variable will be defined as `xon=0` for “None”, and `xon=1` for “Send”.

Edit box

An normal edit box is defined with the following method:

```
T;var;label;hint;default;size
```

The “`var`”, “`label`”, “`hint`”, and “`default`” are as above.

The “`size`” defines how many characters wide the edit box is.

For example, the “ASCII lines” protocol requires a timeout value to be set:

```
T;timeout;Timeout;Data timeout (ms);1000;5
```

The variable will be “`timeout`” and is a maximum of 5 characters wide.

Complete example

The list of parameters can be combined to produce a full set of options.

For example, the “ASCII lines” protocol would be:

[49] If no parameters are required, then define a blank string, or leave out this entry.

```
protocols.ascii.params="S;xon;XON;Is the device software flow
    controlled?;0;0=None;1=Send|S;codes;Allow;What to
    save;0;0=ASCII only;1=ASCII + codes|T;timeout;Timeout;Data
    timeout (ms);1000;5;”
```

This defines three parameters:

1. A combo-box for xon
2. A combo-box for codes
3. An edit-box for timeout

- Do not define a var that clashes with any of the “source” methods (e.g. “write”), or a var called “n”! Doing this will prevent the protocol from working completely!

13.1.6. Protocol functions

All protocol function calls pass a single parameter – the protocol table.

This protocol table is an object that you can call all the source function methods on. Additionally, it contains all of the parameters (as defined by the user), and a field “`n`” that specifies the channel number.

13.1.7. Action on setup

Sometimes the protocol requires a particular action when setup. This call is made when a channel starts firing up the protocol.

```
function protocols.myprotocol.setup(p)
  -- do something
  return true --reload the configuration
  return false --just continue
end
```

It is illegal to attempt to read or write to the source – it is not connected yet. However, you can choose to initialise variables that are required by the protocol. Additionally the protocol's setup function can make changes to the source parameters before continuing.

For example, a protocol may choose to force the TCP port number for the channel:

```
protocols.myprotocol.setup(p)
 if c.chnl[p.n].src.tcp.port ~= 23
 then
  c.chnl[p.n].src.tcp.port = 23
  return true
 end
 return false
end
```

The function will check the port number and force it to 23 if different. Note that the change will be stored and will effectively overwrite the user's input.

13.1.8. Action on connect

Like the .setup function, the connect function can be used to setup various working variables for the protocol (like timeout values and clearing buffers). It is called whenever the source enters the "connected" state[50].

Unlike the .setup function hook you can read and write to the source.

```
function protocols.myprotocol.connect(p)
 p:write('Hello') --send a string
 p:readline(1000) --wait 1second for a response
end
```

Additionally, the protocol can choose to close the source and issue an error:

```
function protocols.myprotocol.connect(p)
 p:write('Hello') --send a string
 s,t = p:readline(1000) --wait 1second for a response
 if t
 then
  p:error('No response')
  p:close()
 end
 p:write('Now we are running')
end
```

This functionality is required for the Avaya RSP, for example, where on connection there is an initial conversation to establish the link.

13.1.9. Action every second

Many protocols require timeouts to be kept, and other actions. The Lua protocol framework allows for a function hook that is called every second, *while there is no incoming data*.

You can perform any action within the second event hook.

```
function protocols.ascii.second(p)
 if (p.xontime>=30)
 then
  if (p.xon == 1) then p:writebytes(0x11) end
  p.xontime=0
 end
 p.xontime=(p.xontime or 0) + 1
end
```

The above function is from the supplied "ASCII lines" protocol. It keeps track of the idle count with the "p.xontime" counter. When that exceeds 30, the protocol will send an XON character – only if the user has configured it and the p.xon variable is therefore 1.

Some protocols may perform keep-alive checks with the other end, and decide to close the connection if there is a serious timeout (as is the case with the Avaya RSP protocol).

[50] COM and UDP sources will "connect" immediately. A TCP source will call this when the socket connects and the match/send have completed. An FTP Server source will call this when the FTP client logs in with the username/password that match the source channel.

13.1.10. Handling incoming data

Whenever data arrives at the source, the Lua protocol framework will make a call to the .data function hook of the protocol. It is the responsibility of this function to read data from the source, respond if necessary, and store any data that is considered valid.

As a very simple example:

```
function protocols.ascii.data(p)
    local s=p:readline()
    p:store(s,"ascii")
end
```

This function simply calls the source.readline function, pulling the data into a local variable “s”. That string is then stored with the tag “ascii” - which involves the optional time stamping.

- Local keyword: it is vitally important that you use the local keyword when defining variables that are used within the scope of a single function. If you do not, then the variables will be **global** and can be read, modified, and corrupted by other channels that are using the same protocol or variable name.

Some protocols will require assembling a “frame” of data and perhaps isolating the data. The NEC NEAX2400 serial protocol is an example that is worth analysing:

```
function protocols.nec.data(p)
 local r,s
 p._buff = (p._buff or "")..p:read()
 p.time=0
 if string.len(p._buff)>2048 then p._buff=string.sub(p._buff,-
    2048,-1) end
 while true
 do
   _,r,s=string.find(p._buff,"\002(.-)\003")
   if not (s) then break end
   if (s) then p:store(s.."\r\n","cdr nec") end
   p._buff=string.sub(p._buff,r+1,-1)
 end
end
```

This code builds a buffer that is local to the protocol (stored in p._buff[51]). The buffer length is then limited to 2k. Finally, in a “while true” loop the Lua script searches for pattern captures between the 0x02 and 0x03 characters. Any matching captures are stored in the “s” local variable and are then stored with a CR/LF suffix to the flash.

(There is some other code that uses the .second function to clear the buffer – just in case the protocol is selected and we are being sent non-NEC data!)

13.2. Testing the protocol

It is often beneficial to setup a small Lua test file on a PC to get the basic script working with predefined values. Obviously this may be a lot harder for bidirectional protocols like

[51] Any variables that begin with an underscore will not appear in the diagnostic dump output.

the Avaya RSP. However, by scripting on a PC you save a lot of time by weeding out the most obvious errors and problems.

When trying to debug more complex protocols with a connected device, it is helpful to temporarily insert source.debug (p:debug) commands and use the Pass-Through connection in Debug mode. In this mode the pass-through socket will be sent event information, debug information, and all data that is received and transmitted by the protocol.

It is also worth noting that the pass-through socket in "Monitor" mode is helpful in showing that the protocol is functioning. If the protocol stalls for any reason, then the pass-through socket will stop showing data. The pass-through socket is only fed data when the protocol requests data from the source.

13.3. Sets of protocols

There is nothing to stop you defining a whole set of protocols within your script or OEM script.

Just continue defining new protocols within the script:

```
protocols.myprotocol = {}
protocols.myprotocol.desc = 'My protocol'
...

protocols.myotherprotocol = {}
protocols.myotherprotocol.desc = 'Another protocol'
...

-- etc
```