RFcreations
...a different perspective

Moreph HDT
User Manual
V1.01

Tim Newton

Last Updated: 27th January 2026

1 Contents

1 CONTENTS 2

2 HARDWARE CONNECTIVITY 9

2.1 Power and control 9

2.2 IUT HCI 9

2.3 Single antenna port 10

2.4 Off-air measurements 10

2.5 Multiple antenna ports 10

2.6 Logic analyser 12

2.7 Multiple IUTs 14

2.8 External reference clock 15

3 LAUNCHING THE HDT APPLICATION 16

3.1 Launching UI 16

3.2 Launching from the API 17

4 ANATOMY OF UI 18

5 MODES OF OPERATION 19

6 TOOLBAR 20

6.1 Open 20

6.2 Save 20

6.3 Export IQ data 20

6.4 Export constellation 21

6.5 Export in-band emissions 21

6.6 Export summary 21

6.7 Export test results 22

6.8 Capture plot 22

6.9 Screen capture 22
6.10 Zoom in 23
6.11 Zoom out 23
6.12 Pan 23
6.13 Pan horizontally 23
6.14 Pan vertically 23
6.15 Connect IUT 23
6.16 Query IUT 24
6.17 Reset 24
6.18 Run/Stop 25
6.19 Continuous operation 25
6.20 Discard 26
6.21 About 26
6.22 Help 26
6.23 IUT selection 26
7 STATUS BAR 27
7.1 Overview 27
7.2 Friendly name 27
7.3 Unit serial number 27
7.4 Overload indicator 27
7.5 Number of IUTs 27
7.6 Switch box detection 27
7.7 Error message 28
7.8 Signal analyser progress bar 28
7.9 Signal analyser trigger status 28
7.10 Data and time 28
8 CONNECTING TO A IUT 29

8.1 IUT control menu 29
8.2 Mandatory items 30
8.3 Optional items 30
8.4 Establishing a connection 31
8.5 Setting supported features 32
8.6 Sending additional HCI commands 35
8.7 Disconnecting 36
9 IXIT 37
9.1 Overview 37
9.2 HDT IXIT parameters 37
10 SCRIPTED MODE 39
10.1 Overview 39
10.2 Creating a script 39
10.3 Gain control test 42
10.4 Setting the run mode 43
10.5 Running a script 44
10.6 Examining results 44
10.7 Debugging test scripts 46
10.8 Running test scripts from the API 47
11 SIGNAL GENERATOR MODE 49
11.1 Overview 49
11.2 Signal selection 49
11.3 Common signal parameters 51
11.4 HDT parameters 53
11.4.1 General 53
11.4.2 Long training sequence (LTS) 55

11.4.3 Control header 55
11.4.4 Payload header 56
11.4.5 Rx portion 57
11.4.6 Tx portion 58
11.4.7 Extended header portion 59
11.5 Signal distortions 60
11.6 Starting and stopping the signal generator 62
11.7 Measurement of BLER with signal generator 63
11.8 Saving and restoring settings 63
11.9 GPIO 64
12 SIGNAL ANALYSER 65
12.1 Overview 65
12.2 Demodulation 66
12.3 Time domain quantities 67
12.3.1 General 67
12.3.2 IQ data 67
12.3.3 Constellation data 67
12.3.4 Power data 68
12.3.5 Background labelling 68
12.3.6 Navigation 68
12.3.7 GPIO 68
12.3.8 Delta cursors 69
12.4 Packet contents 70
12.5 Spectral measurements 73
12.5.1 In-band emission plots 73
12.5.2 Spectrogram 74
12.5.3 Coupled measurements 74

12.6 Constellation diagram 75
12.7 Measured quantities 76
12.8 Starting and stopping the signal analyser 78
12.9 Controlling a IUT 81
12.10 Saving and restoring settings 84
12.11 Exporting data 84
13 GPIO / SWITCH BOX 86
14 CABLE LOSS 88
15 LOG WINDOW 90
15.1 Overview 90
15.2 Raw octets 90
15.3 HCI decode 91
15.4 Informational messages 91
15.5 Error messages 91
15.6 Test script messages 91
15.7 Navigating within the log window 92
15.8 Exporting log window contents 92
16 CALIBRATION 93
17 HEALTH 94
18 C API 95
18.1 Overview 95
18.2 API server 95
18.3 Error handling 95
18.4 Connecting to a moreph 96
18.5 Making the UI visible 96

18.6 Connecting to a IUT 97
18.7 Set cable loss 97
18.8 Setting operating mode 98
18.9 Loading a test script 98
18.10 Creating a test script 99
18.11 Running a test script 99
18.12 Reading the results of a test script 100
18.13 Saving a test script file 101
18.14 Building an HDT signal 101
18.15 Building a CW signal 102
18.16 Adding AWGN 103
18.17 Running the signal generator 103
18.18 Configuring the signal analyser 104
18.19 Running the signal analyser 104
18.20 Reading back measurements 105
18.21 Accessing in-band emissions data 105
18.22 Accessing constellation data 106
18.23 Saving IQ data 107
18.24 Disconnecting 107
18.25 Resetting moreph hardware 108

19 PYTHON API 109
19.1 Overview 109
19.2 API server 109
19.3 Reseting moreph hardware 109

20 STANDALONE MODE 110

21 SUPPORTED GRAPHICS FORMATS 111

2 Hardware connectivity

2.1 Power and control

The moreph is powered by a 12V mains adapter. This should be connected to the barrel jack on the rear panel.

The moreph can be controlled by a host computer over either USB or Ethernet. These connectors are on the rear panel of the moreph.



moreph control over USB
To host computer

To 12V mains adapter

Power and control connections on rear of moreph

2.2 IUT HCI

Although HDT analysis and generation can be executed without the moreph having control of the IUT, most operations require an HCI interface to the IUT.

The HDT software expects the IUT to expose a comport to which HCI commands can be sent and from which HCI responses and events can be received using the H4 protocol. The HDT software provides a number of controls to limit the rate at which HCI commands are sent. All HCI traffic can be recorded in the “Log” window.

2.3 Single antenna port

For IUTs with a single antenna port a connection should be made direct to the Tx/Rx port on the moreph. It is advisable to minimise the loss in this connection.



Single IUT connected to Tx/Rx port of moreph

2.4 Off-air measurements

It is possible to analyse HDT signals captured off-air when in signal analyser mode. To do so, connect an antenna to the “Monitor In” port. Select the input port as “Monitor In” as described in Running the signal analyser.

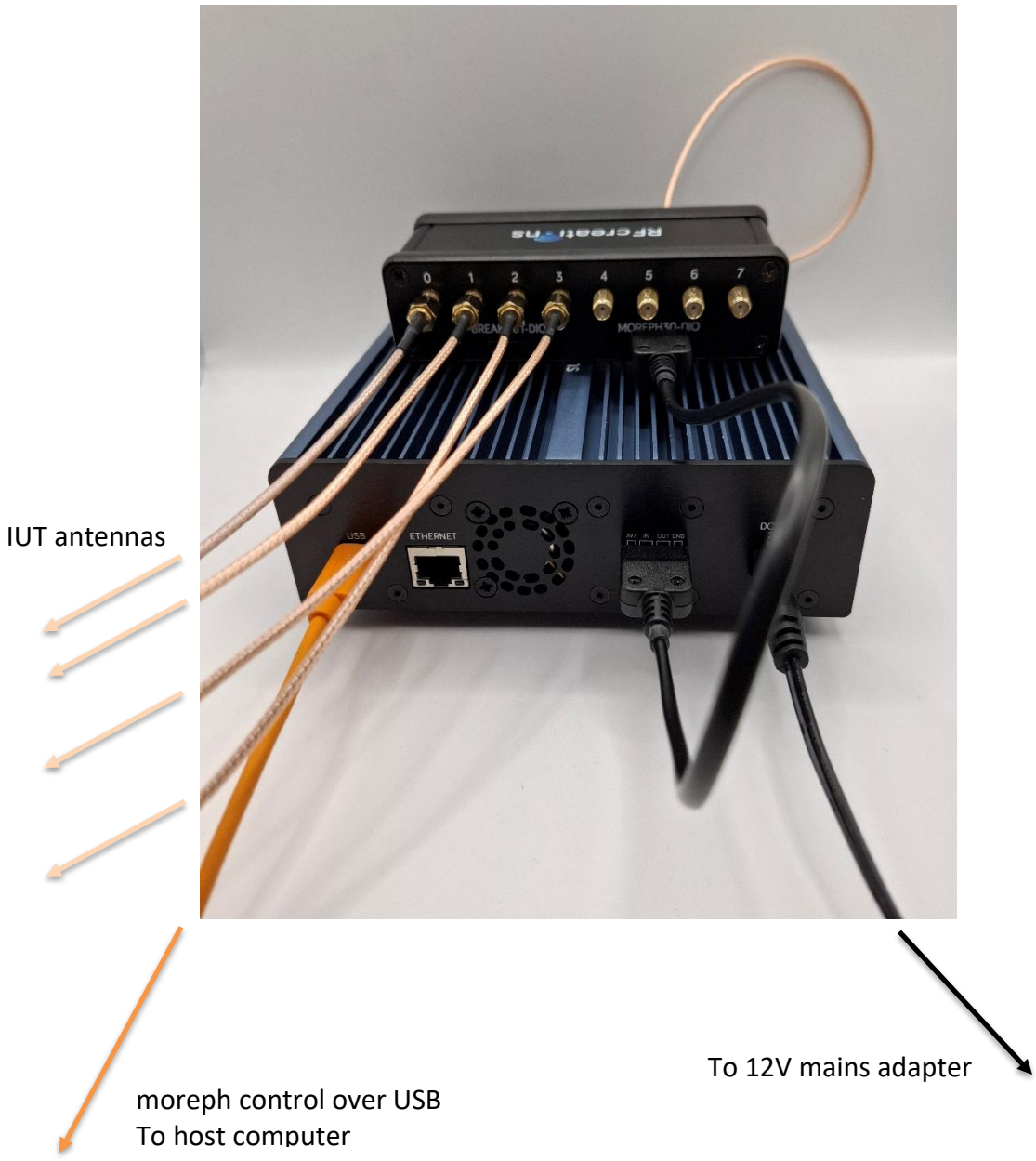
2.5 Multiple antenna ports

For IUTs with multiple antenna ports connections should be made to the 8-way switch box. The lowest numbered antenna port on the IUT should be connected to port “0” on the 8-way switch box, the next to port “1”, etc. The “COMMON” port on the 8-way switch box should be connected to the Tx/Rx port on the moreph. It is advisable to minimise the loss in these connections.

The ”MOREPH-30 DIO” port on the 8-way switch box should be connected to the digital IO port on the rear panel of the moreph.

The 8-way switch box consists of a 1:8 power splitter. On each of the 8 branches are two non-reflective switches. When a branch is disabled, these switches terminate both the internal branch of the power splitter and the associated SMA connector in 50Ω.

The majority of the Bluetooth defined test cases only access one antenna port on the IUT.



8-way RF switch box with cabling to 4 IUT antennas and control to moreph



Common RF connection between 8-way RF switch box and moreph

2.6 Logic analyser

A logic probe can be used to monitor signals generated by the IUT and plot these alongside IQ data when operating in signal analyser mode. The moreph will also output its timing signals via the logic probe when operating in signal generator mode. The logic probe has a bandwidth of 22MHz.

The logic probe should be connected to the digital IO port on the rear panel of the moreph.

The IO voltage for the logic signals can either be provided by the moreph or applied externally. See section GPIO / Switch box. If the moreph provides the IO voltage, then it is fixed at 3v3. If the IO voltage is applied externally, then it must be in the range 0v8 to 3v3.

The red cable on the logic probe either provides the 3v3 IO voltage from the moreph or accepts the IO voltage applied by the IUT. The black cable on the logic probe should be connected to the IUT ground.

The signals output by the moreph are:

First HDT signal	GPIO 8
Second HDT signal	GPIO 9
Third HDT signal	GPIO 10
Fourth HDT signal	GPIO 11
First CW signal	GPIO 13
Second CW signal	GPIO 14
AWGN	GPIO 15

These signals go high once the associated signal has ramped to full output power. The signals go low at the start of the power ramp down.



moreph control over USB
To host computer

To 12V mains adapter

Logic analyser connected to rear of moreph via control cable



Logic analyser daisy chained via 8-way RF switch box to moreph

2.7 Multiple IUTs

It is possible to have multiple IUTs connected to the moreph simultaneously. Only one IUT can be tested at any one time. All the IUTs must operate with the same UART configuration and have the same supported channel sounding features.

The multiple IUTs must be connected via the 8 way RF switch box. The first IUT has its antenna ports connected to ports 0 to (N-1) on the 8 way RF switch box, where N is the number of antenna ports supported by the IUT. The second IUT has its antenna ports connected to ports N to 2N-1 on the 8 way RF switch box.

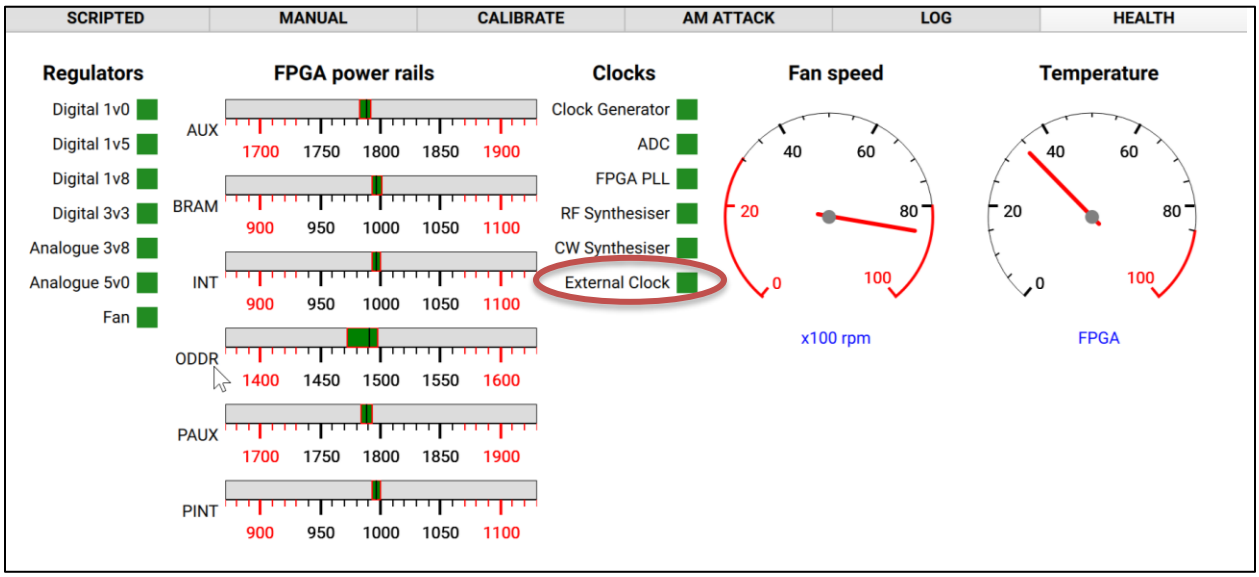
2.8 External reference clock

The moreph can accept an external 10MHz reference clock on the “EXT CLK IN” port on the front panel. If this clock is present prior to launching an application on the moreph, then the moreph will automatically lock its internal clocks to the external reference.

The specification for the reference clock input port is:

Connector type	SMA
Minimum input signal	-10dBm
Maximum input signal	+20dBm
Frequency	10 MHz
Pullability	30ppm
Impedance	50Ω
Coupling	AC
Maximum DC voltage	50V

If the moreph successfully locks to an external reference clock, then this will be indicated under the “Health” tab.



3 Launching the HDT application

3.1 Launching UI

In order to communicate with the moreph, it is necessary to attach it to a host computer via USB or Ethernet (or both). An Ethernet connection is only possible if the host computer and moreph reside on the same subnet. The moreph IP address can be changed by connecting it to a host computer via USB and using the Application Loader.

To launch the channel sounding UI it is first necessary to run Application Loader. This should result in the following screen being displayed:

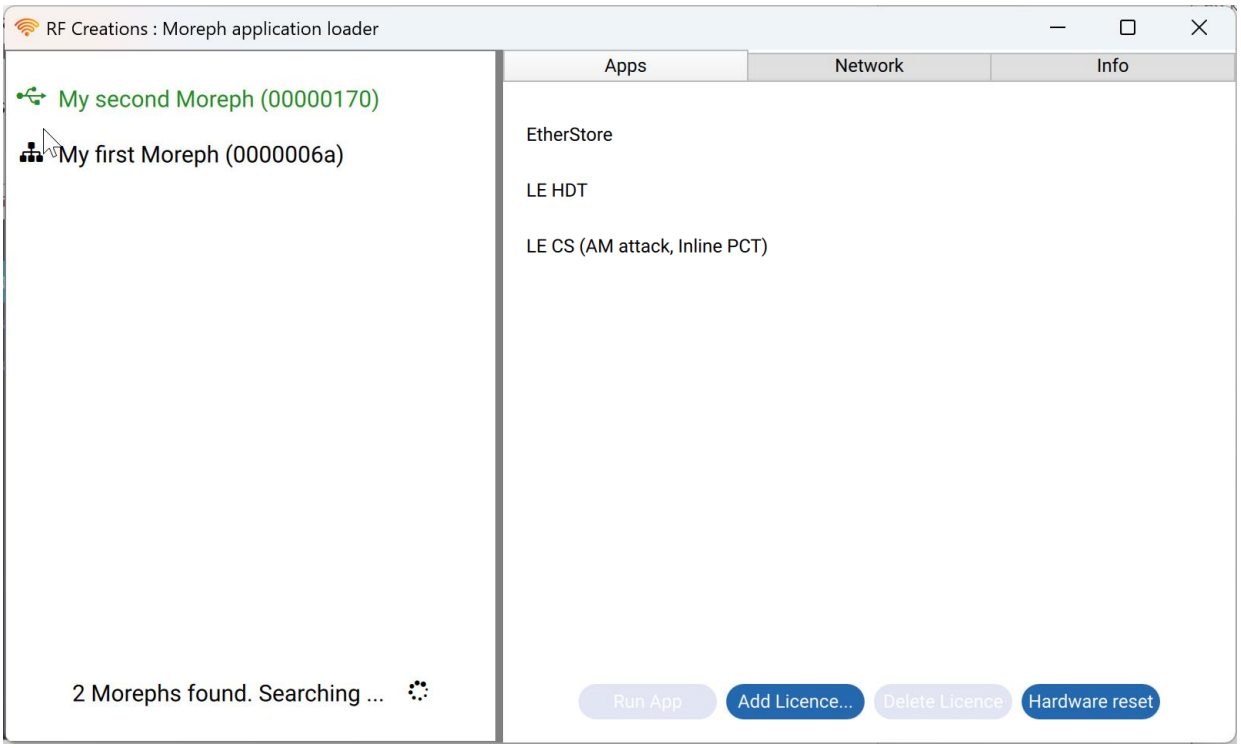


Figure 1: Application Loader main screen.

This screen indicates that the following moreph devices have been discovered:

1. Serial number 00000170 connected via USB (highlighted)
2. Serial number 0000006A connected via Ethernet

The right-hand side of the window has three tabs:

1. *Apps*. Shows which applications are licensed to run on this unit. It also permits the loading of new licence keys.
2. *Network*. Shows the current network settings and permits these to be modified.
3. *Info*. Provides more information about the unit and permits the unit's friendly name to be modified. It also provides a means of updating the firmware on the unit.

To launch the channel sounding UI open the "*Apps"* tab and then either:

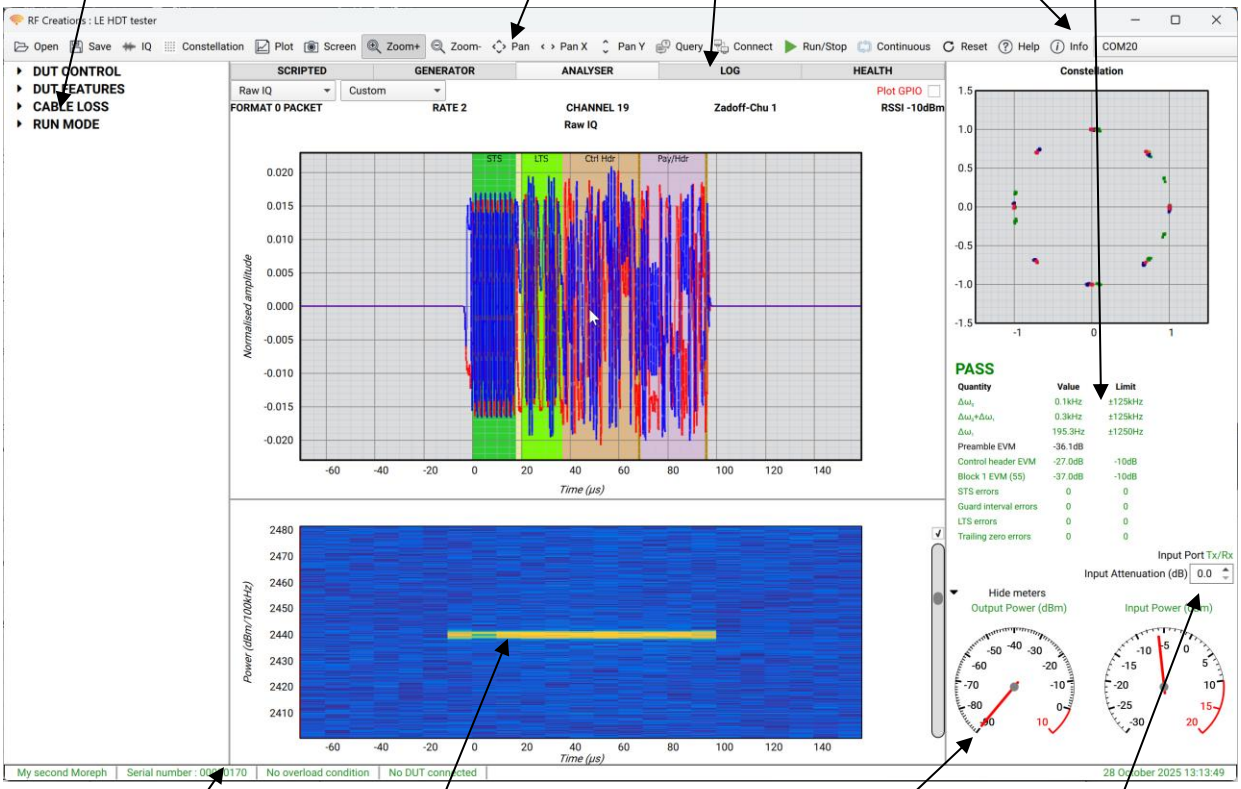
1. Double click on the *LE HDT* application
2. Highlight the *LE HDT* application and then click the "*Run App*" button.

On launching, the searching cursor should stall, the fan on the moreph unit will start to spin and after a few seconds the HDT UI will load.

3.2 Launching from the API

Details of how to launch the HDT application from the API are contained in section C API. When running with the API it is possible to make the UI visible.

4 Anatomy of UI



5 Modes of operation

There are three modes of operation:

1. Scripted
   In this mode of operation the user can construct a list of test cases based on the Bluetooth test specifications. The test cases in the list can be executed and the results exported. Test lists may be saved and restored.
   Scripted mode requires an HCI connection to the IUT.
2. Signal generator
   In this mode of operation the user can construct up to 5 separate signals to be generated by the moreph. These can be a combination of arbitrary HDT packets, CW, AWGN and out-of-band CW blocker. If a IUT is connected, then the BLER will also be displayed.
3. Signal analyser
   In this mode either conducted signals on the “Tx/Rx port” or off-air signals from the “Monitor In” port are analysed. The signal analyser reports carrier offset, drift, EVM, ACP and average power. If a IUT is connected, then control over the signals transmitted by the IUT is possible.

To select the operating mode of the unit from the UI, use the tabs immediately below the toolbar.

To select the operating mode of the unit from the API use:

```
RF_setOperatingMode( opMode_t mode );
```

Options for `mode` are:

`RF_scripted`	Scripted mode
`RF_siggen`	Signal generator mode
`RF_sigana`	Signal analyser mode

6 Toolbar

6.1 Open



Mode	Action	API
Scripted	Loads a test script	`RF_loadTestScript( char* filename )`
Signal generator	Loads saved settings	`RF_loadSigGen( char* filename )`
Signal analyser	Loads saved settings	`RF_loadSigAna( char* filename )`

6.2 Save



Mode	Action	API
Scripted	Saves current test script	`RF_saveTestScript( char* filename )`
Signal generator	Saves settings	`RF_saveSigGen( char* filename )`
Signal analyser	Saves settings	`RF_saveSigAna( char* filename )`
Log	Exports log as HTML	`RF_saveLog( char* filename )`

Note: In the UI the save icon has a different meaning when the log window is being viewed. The API can save the contents of the log window irrespective of the current operating mode.

6.3 Export IQ data



Mode	Action	API
Scripted	*Not available*	
Signal generator	*Not available*	
Signal analyser	Saves IQ data	`RF_saveIQ( char* filename )`

The IQ data consists of pairs of real*4 sampled at 128MSps. These are exported as a binary file.

This icon is only visible after an HDT packets has been successfully received.

6.4 Export constellation



Mode	Action	API
Scripted	*Not available*	
Signal generator	*Not available*	
Signal analyser	Saves constellation data	`RF_saveConstellation( char* filename)`

The constellation data consists of pairs of real*4 sampled at the symbol rate. These are exported as a binary file. See Demodulation for a description of the demodulation process.

This icon is only visible after an HDT packets has been successfully received.

6.5 Export in-band emissions



Mode	Action	API
Scripted	*Not available*	
Signal generator	*Not available*	
Signal analyser	Saves constellation data	`RF_saveInBandEmissions(char*filename)`

The in-band emissions data are saved to a csv file. Both the 1MHz and 100kHz bin data are saved.

This icon is only visible after an HDT packets has been successfully received.

6.6 Export summary



Mode	Action	API
Scripted	*Not available*	
Signal generator	*Not available*	
Signal analyser	Saves signal analyser results	`RF_saveSummary( char* filename )`

The contents of the summary window are exported as a .csv file.

This icon is only visible after an HDT packets has been successfully received.

6.7 Export test results



Mode	Action	API
Scripted	Saves results of scripted tests	`RF_saveResults( char* filename )`
Signal generator	*Not available*	
Signal analyser	*Not available*	

The results of the last executed test script are exported as html. The export includes both the tabular results as well as any graphics which were produced.

This icon is only visible once a test script has been executed.

6.8 Capture plot



Mode	Action	API
Scripted	Saves current displayed graph	`RF_saveGraph( char* filename )`
Signal generator	Saves transmission graph	`RF_saveGraph( char* filename )`
Signal analyser	Saves upper graph	`RF_saveGraph( char* filename )`

The format in which the graph is saved is determined by the filename extension.

6.9 Screen capture



Mode	Action	API
Scripted	Performs a screenshot	`RF_screenshot( char* filename )`
Signal generator	Performs a screenshot	`RF_screenshot( char* filename )`
Signal analyser	Performs a screenshot	`RF_screenshot( char* filename )`

The format in which the screenshot is saved is determined by the filename extension.

6.10 Zoom in



Activates the cross-hair cursor which permits zooming within in the graphics area. Depress the left mouse button whilst dragging the cursor to select the area to be displayed. Clicking the right mouse button within the graphics area will give a list of additional zoom options.

6.11 Zoom out



Zooms out within the graphics area. Clicking the right mouse button within the graphics area will give a list of additional zoom options.

6.12 Pan



Pans within the graphics area. Hold down the left mouse button and drag to pan anywhere within the graphics display.

6.13 Pan horizontally



Pans along the x-axis within the graphics area. Hold down the left mouse button and drag horizontally. This is particularly useful for examining long waveforms.

6.14 Pan vertically



Pans along the y-axis within the graphics area. Hold down the left mouse button and drag vertically.

6.15 Connect IUT



When the icon is shown in outline no IUT is connected. On clicking the icon, the moreph will try to communicate with the IUT(s) using the H4 protocol and the parameters selected in the “IUT control” menu.

Corresponding API call is `RF_connectIUT( char *comport )`.



When the icon is shown filled one or more IUT(s) are connected via H4. On clicking the icon the IUT(s) will be disconnected.

Corresponding API call is `RF_disconnectIUT( void )`.

6.16 Query IUT



Clicking the query icon will cause the moreph to query the IUT supported features over HCI. If no IUT is currently connected, then an attempt will be made to connect to the IUT(s) selected in the “IUT control” menu before querying the supported features. If more than one IUT is connected, then a single query will be made to the currently selected IUT. Once the query has been completed, the “IUT features” menu will be expanded to show the supported features.

Corresponding API call is `RF_queryIUT( void )`.

6.17 Reset



Action depends on mode. In scripted mode the test script is cleared. In signal generator mode, all parameters related to signal generation are reset to their default values, apart from cable loss settings. In signal analyser mode, all parameters related to signal analysis are reset to their default values

Mode	Action	API
Scripted	Clears test script	`RF_resetRF ( char* filename )`
Signal generator	Sets parameters to default values	`RF_resetRF ( char* filename )`
Signal analyser	Sets parameters to default values	`RF_resetRF ( char* filename )`

6.18 Run/Stop



Mode	Action	API
Scripted	Execute the current test script	`RF_run( void )`
Signal generator	Commence signal generation	`RF_run( void )`
Signal analyser	Commence signal analysis	`RF_run( void )`

If no IUT is connected and signal generation includes signals with a finite number of repetitions, then signal generation will cease once all repetitions have been generated. If a IUT is connected and signal generation includes signals with a finite number of repetitions, then once all repetitions have been generated the BLER will be displayed and the signal generator automatically restarted.

Signal analysis will cease as soon as the first HDT signal has been successfully analysed.



Mode	Action	API
Scripted	Use cancel from progress window	`RF_abort( void )`
Signal generator	Aborts signal generation	`RF_abort( void )`
Signal analyser	Aborts signal analysis	`RF_abort( void )`

6.19 Continuous operation



Mode	Action	API
Scripted	*Not available*	
Signal generator	*Not available*	
Signal analyser	Commence continuous signal analysis	`RF_continuous( void )`

In the continuous mode of operation, the signal analyser will continue to analyse HDT packets until a stop condition is reached.

6.20 Discard



Mode	Action	API
Scripted	Discards test results	`RF_discardResults( void )`
Signal generator	*Not available*	
Signal analyser	*Not available*	
Log window	Clears log window contents	`RF_clearLog( void )`

Note that the impact of this icon is different if the log window is the currently selected tab.

6.21 About



Displays version information.

6.22 Help



Displays online documentation in the default web browser.

6.23 IUT selection



Shows the currently selected IUT. When more than one IUT is connected, a drop-down menu is provided to select which IUT to use.

There is no corresponding API call since the API is only able to connect to a single IUT at any one time.

7 Status bar

7.1 Overview

The status bar at the bottom of the window is divided into seven areas:

1. Friendly name of the moreph
2. Serial number of the moreph
3. Overload indicator
4. Number of IUTs connected
5. Switch box detection
6. Error message
7. Signal analyser progress bar (signal analyser mode only)
8. Signal analyser trigger status (signal analyser mode only
9. Current data and time

7.2 Friendly name

Displays the moreph friendly name. This may be modified using the “Info” tab on the Application Loader.

7.3 Unit serial number

Displays the moreph serial number as an 8 digit hexadecimal value.

7.4 Overload indicator

The overload indicator will turn red when an overload condition occurs on either the transmitter output or the selected receiver input port. The text of the message will indicate where the overload condition is occurring.

7.5 Number of IUTs

The text indicates the number of IUTs which are currently connected to the moreph.

7.6 Switch box detection

The text indicates whether an 8-way RF switch box connected to the moreph has been automatically detected.

7.7 Error message

The error message reflects the last error detected by the channel sounding application running on the moreph. This message is cleared when either the “Run” or “Discard” buttons are pressed on the toolbar.

7.8 Signal analyser progress bar

This is only displayed in signal analyser mode once either the “Run” or “Continuous” buttons have been clicked.

Once a suitable HDT packet has been detected for processing by the signal analyser, a progress bar will be shown. The length of the bar gives an approximate indication of the fraction of the packet which has been analysed.

7.9 Signal analyser trigger status

This is only displayed in signal analyser mode once either the “Run” or “Continuous” buttons have been clicked.

Whilst the signal analyser is wait for a suitable HDT signal to analyser, the trigger status is shown as “Trigger?”. Once a suitable HDT packets has been received, the trigger status is changed to “Triggered”.

7.10 Data and time

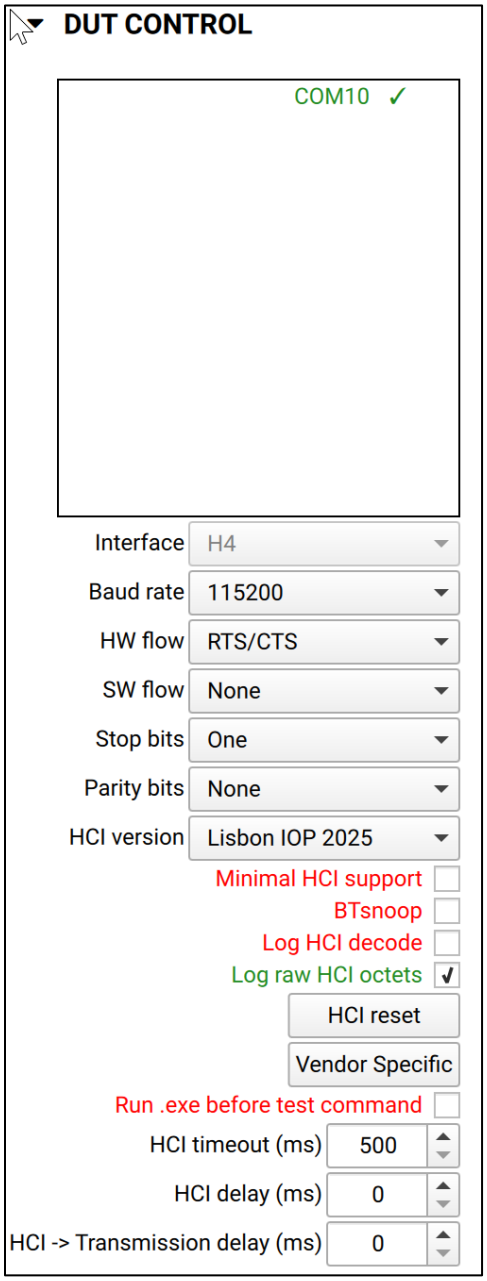
The current data and time are displayed at the bottom right of the status bar.

8 Connecting to a IUT

8.1 IUT control menu

Open the IUT control menu.

Select the comport(s) to which the IUT(s) are attached. More than one IUT can be connected simultaneously. If more the one IUT is connected, they must be connected via the 8-way RF switch box. Although more than one IUT can be connected simultaneously, only one IUT can be tested at a time. The IUT to be tested is selected from the drop-down menu on the toolbar.



8.2 Mandatory items

It is mandatory to set the following items:

Item	API	Options
Baud rate	`RF_setBaudRate( uint32_t )`	
Hardware flow control	`RF_setHWflow( hwFlow_t )`	`RF_HW_none` `RF_HW_RTS_CTS`
Software flow control	`RF_setSWflow( swFlow_t )`	`RF_SW_none` `RF_SW_XON_XOFF`
Number of stop bits	`RF_setStopBits( uint32_t )`	`1` `2`

Number of parity bits	`RF_setParity( parity_t )`	`RF_PARITY_none`
		`RF_PARITY_even`
		`RF_PARITY_odd`

HCI version	`RF_setHCIversion( char * )`	`"Lisbon IOP 2025"`

Currently only H4 protocol is supported. Contact RFcreations if H5 or BCSP is required.

If "Minimal HCI support" is checked, then the moreph will only issue HCI LE test commands and HCI LE test end commands. No event filters will be set.

8.3 Optional items

The following optional items impact the H4 protocol:

Item	Description	API
HCI timeout	The maximum interval between the moreph sending an HCI command and the IUT responding.	`RF_setHCItimeout( uint32_t ms )`
HCI delay	The delay inserted prior to issuing an HCI command.	`RF_setHCIdelay( uint32_t ms )`
HCI -> Transmission delay	The delay between the moreph issuing an HCI test command and the moreph sending the first HDT packet.	`RF_setHCItxDelay(uint32_t ms )`

The following optional items alter what is displayed in the log window:

Item	Description	API
Log HCI decode	When checked, a full HCI decode of all HCI traffic to and from the IUT will be written to the log window.	`RF_logHCIdecode( bool )`
Single line HCI decode	When checked, the HCI decode is written to the log file on a single line. When unchecked, multiple lines are used for the HCI decode.	
Raw HCI octets	When checked, the raw octets sent and received over the H4 link will be written to the log window together with an associated timestamp.	`RF_logRawOctets( bool )`

Checking “BTsnoop” will result in all HCI traffic being written to a BTsnoop file. When this item is checked, a file dialogue will pop-up so the destination for the BTsnoop data can be specified.

The corresponding API call is `RF_setBTsnoop( char *filename )`. If the filename specified in the call is empty, then any open BTsnoop file will be closed.

Prior to each HCI LE Test command issued by the moreph it is possible to run a user supplied executable. The moreph will close the corresponding comport prior to executing the user supplied code and then re-open the comport once the code has completed. This provides a facility for the user to reset the IUT prior to a new HCI LE Test command being issued. To enable this functionality check “Run .exe before test command”. A file dialogue will pop-up so that the location of the user supplied executable can be entered.

The corresponding API call is `RF_setUserExe( char* filename )`. If the filename specified is an empty string, then the corresponding functionality will be disabled.

8.4 Establishing a connection

To establish a connection press the “Connect” button on the toolbar.

Any errors incurred during the connection process will be appear in the status bar and also appear in the log window.

The number of IUTs successfully connected will appear in the status bar. The drop-down menu at the top right of the toolbar will list the comports which have been connected.

In the “IUT control” menu, devices which are selected and connected will appear in green with a tick. Devices which are selected and have not been connected will be in red with a cross.

The corresponding API call is `RF_connectIUT( char* comport )`. In the API there is a restriction on a single IUT being connected at any one time.

It is possible to determine whether a IUT is connected by calling `RF_isIUTconnected()`. This can be called ay anytime.

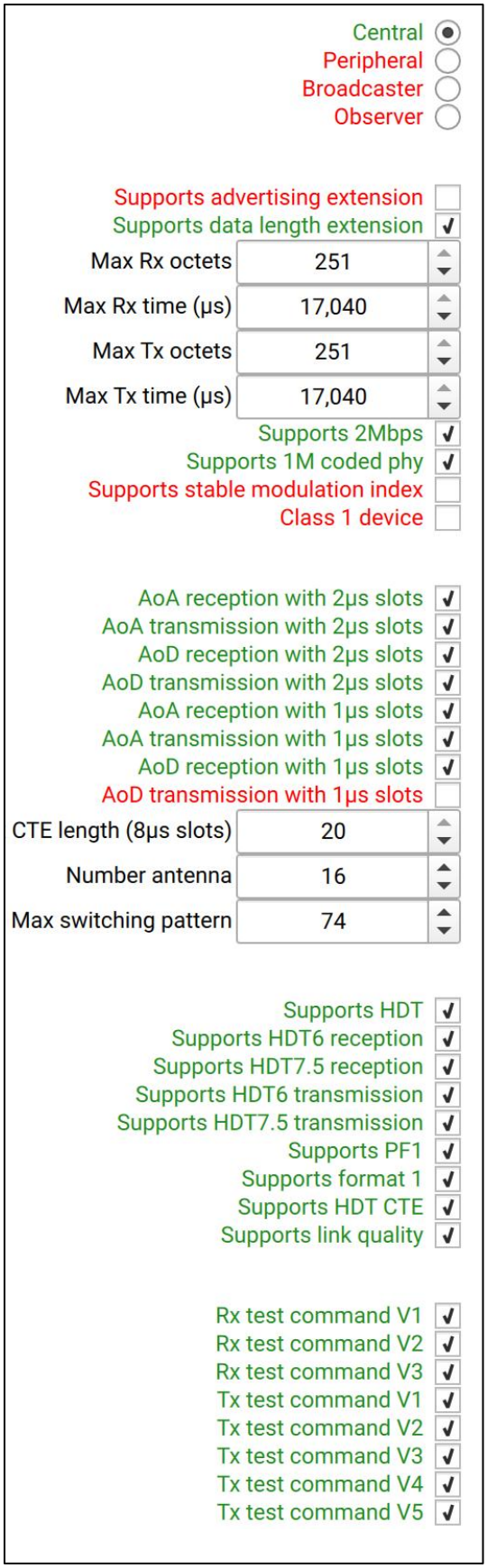
8.5 Setting supported features

Prior to any channel sounding procedure being run, it is essential that the HDT features supported by the IUT are known to the application. This can be done in two ways:

1. Querying the supported features over HCI
2. Entering the supported features manually

To query the supported features over HCI, press the query button the toolbar. Once the query is complete, the “IUT features” menu will automatically be expanded so the results of the query can be viewed. If no IUT is connected when the query button is pressed, then an attempt will automatically be made to connect to any comports selected in the “IUT control” menu prior to issuing the HCI command.

The IUT supported features can be entered or altered manually at any time. The `RF_support…` commands are used to set specific features. If a query IUT supported features command has been executed, then the resulting values can be read back using the `RF_isSupported…` commands.


Central
Peripheral
Broadcaster
Observer
Supports advertising extension
Supports data length extension
Max Rx octets
251
Max Rx time (µs)
17,040
Max Tx octets
251
Max Tx time (µs)
17,040
Supports 2Mbps
Supports 1M coded phy
Supports stable modulation index
Class 1 device
AoA reception with 2µs slots
AoA transmission with 2µs slots
AoD reception with 2µs slots
AoD transmission with 2µs slots
AoA reception with 1µs slots
AoA transmission with 1µs slots
AoD reception with 1µs slots
AoD transmission with 1µs slots
CTE length (8µs slots)
20
Number antenna
16
Max switching pattern
74
Supports HDT
Supports HDT6 reception
Supports HDT7.5 reception
Supports HDT6 transmission
Supports HDT7.5 transmission
Supports PF1
Supports format 1
Supports HDT CTE
Supports link quality
Rx test command V1
Rx test command V2
Rx test command V3
Tx test command V1
Tx test command V2
Tx test command V3
Tx test command V4
Tx test command V5

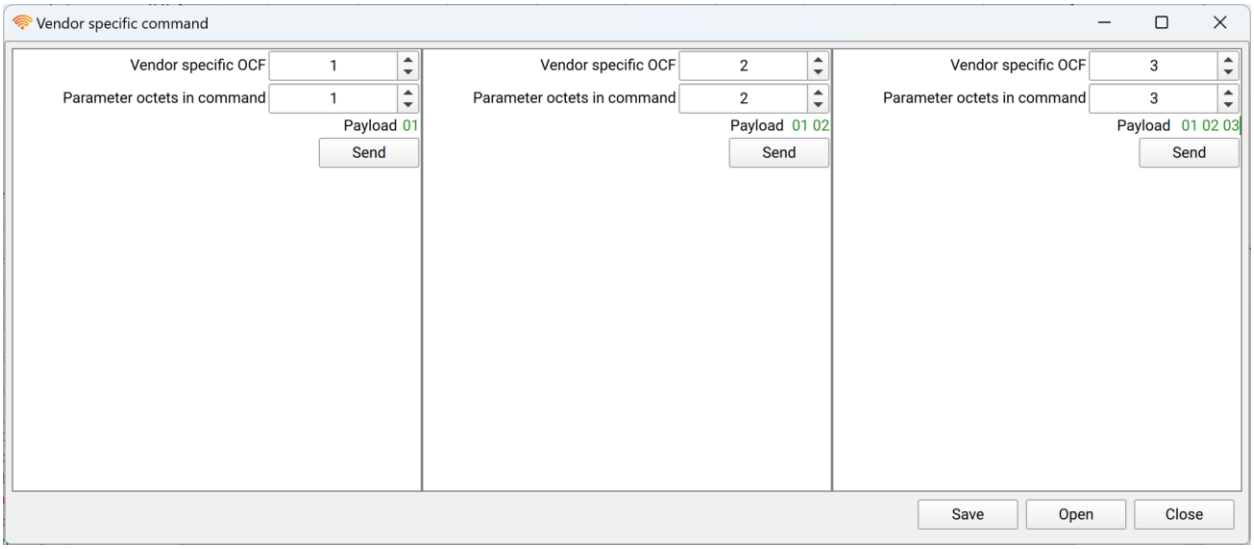
Control	Comments	API
Central/Peripheral/ Broadcaster/Observer	IUT role.	
Supports advertising extension	Not used by HDT.	
Supports data length extension	Not used by HDT.	
Max adv octets	Not used by HDT.	
Max Tx time	Not used by HDT.	
Max Tx octets	Not used by HDT.	
Max Rx time	Not used by HDT.	
Max Rx octets	Not used by HDT.	
Supports 2Mbps	Not used by HDT.	
Supports 1M coded phy	Not used by HDT.	
Supports stable modulation index	Not used by HDT.	
Class 1 device	Alters limits for P_{avg} measurements.	`RF_supportClass1(bool)` `RF_isSuportedCClass1(bool*)`
AoA reception with 2µs slots	Not used by HDT.	
AoA transmission with 2µs slots	Not used by HDT.	
AoD reception with 2µs slots	Not used by HDT.	
AoD transmission with 2µs slots	Not used by HDT.	
AoA reception with 1µs slots	Not used by HDT.	
AoA transmission with 1µs slots	Not used by HDT.	
AoD reception with 1µs slots	Not used by HDT.	
AoD transmission with 1µs slots	Not used by HDT.	
CTE length (8µs slots)		`RF_supportCTElength(uint32_t)` `RF_isSupportedCTElength(uint32_t*)`
Number of antenna		`RF_supportNumAntenna(uint32_t)` `RF_isSupportedNumAntenna(uint32_t*)`
Max switch pattern		`RF_supportSPL(uint32_t)` `RF_isSupportedSPL(uint32_t*)`
Supports HDT	Must be true.	`RF_supportHDT(bool)` `RF_isSupportedHDT(bool*)`
Supports HDT6 reception	Restricts which scripted tests can be run.	`RF_supportRxHDT6(bool)` `RF_isSupportedRxHDT6(bool*)`
Supports HDT7.5 reception	Restricts which scripted tests can be run.	`RF_supportRxHDT7_5(bool)` `RF_isSupportedRxHDT7_5(bool*)`
Supports HDT6 transmission	Restricts which scripted tests can be run.	`RF_supportTxHDT6(bool)` `RF_isSupportedTxHDT6(bool*)`
Supports HDT7.5 transmission	Restricts which scripted tests can be run.	`RF_supportTxHDT7_5(bool)` `RF_isSupportedTxHDT7_5(bool*)`
Supports multiple payloads	Not used.	`RF_supportMultiplePayloads(bool)` `RF_isSupportedMultiplePayloads(bool*)`
Supports packet format1	Restricts which scripted tests can be run.	`RF_supportFormat1(bool)` `RF_isSupportedFormat1(bool*)`
Supports HDT CTE	Restricts which scripted tests can be run.	`RF_supportHDT_CTE(bool)` `RF_isSupportedHDT_CTE(bool*)`
Supports link quality	Restricts which scripted tests can be run.	`RF_supportLinkQuality(bool)` `RF_isSupportedLinkQuality(bool*)`
Rx test command V1	Not used by HDT	`RF_supportRxTestCommandV1(bool)` `RF_isSupportedRxTestCommandV1(bool*)`
Rx test command V2	Not used by HDT	`RF_supportRxTestCommandV2(bool)` `RF_isSupportedRxTestCommandV2(bool*)`
Rx test command V3	Must be true if control of IUT is required	`RF_supportRxTestCommandV3(bool)` `RF_isSupportedRxTestCommandV3(bool*)`
Tx test command V1	Not used by HDT	`RF_supportTxTestCommandV1(bool)` `RF_isSupportedTxTestCommandV1(bool*)`
Tx test command V2	Not used by HDT	`RF_supportTxTestCommandV2(bool)` `RF_isSupportedTxTestCommandV2(bool*)`

Tx test command V3	Not used by HDT	`RF_supportTxTestCommandV3(bool)` `RF_isSupportedTxTestCommandV3(bool*)`
Tx test command V4	Not used by HDT	`RF_supportTxTestCommandV4(bool)` `RF_isSupportedTxTestCommandV4(bool*)`
Tx test command V5	Must be true if control of IUT is required	`RF_supportTxTestCommandV5(bool)` `RF_isSupportedTxTestCommandV5(bool*)`

8.6 Sending additional HCI commands

Once a IUT is connected, it is possible to send additional HCI commands from the UI:

1. HCI reset
   An HCI reset command is sent to the IUT. The UART is closed after the HCI reset command is sent and then reopened. A “Read local version information” command is then issued to ensure the H4 link is still operational. The corresponding API call is `RF_HCIreset()`.
2. RF Test End
   A LE Test End command is sent. This can be useful during initial debugging to ensure a IUT has completed the last HCI test command. The corresponding API call is `RF_HCItestEnd()`.
3. Vendor specific
   A vendor specific HCI command can be sent. A pop-up dialog allows the user to enter up to 3 vendor specific HCI commands.



For each vendor specific HCI command, it is possible to enter:

a. The vendor specific OCF
b. The number of octets in the payload
c. The payload data

The individual vendor specific HCI commands can be sent by clicking the associated “Send” button. All 3 vendor specific HCI commands can be saved/recalled from a single file by clicking the “Save” or “Open” buttons.

Individual vendor specific HCI commands can be sent via the API by calling `RF_HCI_vendor( unint32_t ocf , char * tx)`. The parameter `ocf` is the OCF which will be used in the HCI vendor specific command. The payload of the command is contained within the character string `tx`. This need only contain the payload data and not the payload count. The character string should contain the hex values of the octets to be sent, each octet separated by a space. For example, `“01 02 03”`.

8.7 Disconnecting

To disconnect the IUT(s) press the “Connect” button on the toolbar. The “Connect” button will return to its outline state. In the “IUT control” menu, all selected IUTs will now be shown in red with an associated cross. The IUT selection drop-down menu in the toolbar will now show the selected comport in red to indicate it is no longer available.

The corresponding API call is `RF_disconnectIUT( void )`. Only one IUT can be connected at a time when the API is used.

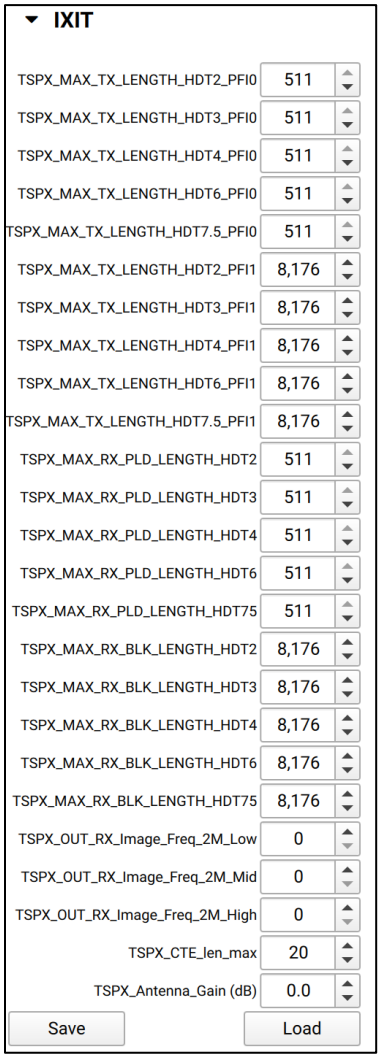
9 IXIT

9.1 Overview

In order to run the Bluetooth RF phy tests it is necessary to know what parameters the tests should be run with. Some of these parameters can be obtained by querying the IUT supported features (see Setting supported features). However, other parameters must be set by the user. These additional parameters are set under the IXIT menu.

9.2 HDT IXIT parameters

The IXIT parameters can be manually set in the UI under the “IXIT” menu. This menu also provides facilities for saving and loading IXIT parameters.



The IXIT parameters required for HDT are tabulated below along with the associated API call for setting them.

IXIT	API
TSPX_MAX_TX_LENGTH_HDT2_PFI0	RF_TSPX_MAX_TX_LENGTH_HDT2_PFI0
TSPX_MAX_TX_LENGTH_HDT3_PFI0	RF_TSPX_MAX_TX_LENGTH_HDT3_PFI0
TSPX_MAX_TX_LENGTH_HDT4_PFI0	RF_TSPX_MAX_TX_LENGTH_HDT4_PFI0
TSPX_MAX_TX_LENGTH_HDT6_PFI0	RF_TSPX_MAX_TX_LENGTH_HDT6_PFI0
TSPX_MAX_TX_LENGTH_HDT7.5_PFI0	RF_TSPX_MAX_TX_LENGTH_HDT75_PFI0
TSPX_MAX_TX_LENGTH_HDT2_PFI1	RF_TSPX_MAX_TX_LENGTH_HDT2_PFI1
TSPX_MAX_TX_LENGTH_HDT3_PFI1	RF_TSPX_MAX_TX_LENGTH_HDT3_PFI1
TSPX_MAX_TX_LENGTH_HDT4_PFI1	RF_TSPX_MAX_TX_LENGTH_HDT4_PFI1
TSPX_MAX_TX_LENGTH_HDT6_PFI1	RF_TSPX_MAX_TX_LENGTH_HDT6_PFI1
TSPX_MAX_TX_LENGTH_HDT7.5_PFI1	RF_TSPX_MAX_TX_LENGTH_HDT75_PFI1
TSPX_MAX_RX_PLD_LENGTH_HDT2	RF_TSPX_MAX_RX_PLD_LENGTH_HDT2
TSPX_MAX_RX_PLD_LENGTH_HDT3	RF_TSPX_MAX_RX_PLD_LENGTH_HDT3
TSPX_MAX_RX_PLD_LENGTH_HDT4	RF_TSPX_MAX_RX_PLD_LENGTH_HDT4
TSPX_MAX_RX_PLD_LENGTH_HDT6	RF_TSPX_MAX_RX_PLD_LENGTH_HDT6
TSPX_MAX_RX_PLD_LENGTH_HDT7.5	RF_TSPX_MAX_RX_PLD_LENGTH_HDT75
TSPX_MAX_RX_BLK_LENGTH_HDT2	RF_TSPX_MAX_RX_BLK_LENGTH_HDT2
TSPX_MAX_RX_BLK_LENGTH_HDT3	RF_TSPX_MAX_RX_BLK_LENGTH_HDT3
TSPX_MAX_RX_BLK_LENGTH_HDT4	RF_TSPX_MAX_RX_BLK_LENGTH_HDT4
TSPX_MAX_RX_BLK_LENGTH_HDT6	RF_TSPX_MAX_RX_BLK_LENGTH_HDT6
TSPX_MAX_RX_BLK_LENGTH_HDT7.5	RF_TSPX_MAX_RX_BLK_LENGTH_HDT75
TSPX_OUT_RX_Image_Freq_2M_Low	RF_TSPX_OUT_RX_Image_Freq_2M_Low
TSPX_OUT_RX_Image_Freq_2M_Mid	RF_TSPX_OUT_RX_Image_Freq_2M_Mid
TSPX_OUT_RX_Image_Freq_2M_High	RF_TSPX_OUT_RX_Image_Freq_2M_High
TSPX_CTE_len_max	RF_TSPX_CTE_len_max
TSPX_Antenna_Gain	RF_TSPX_Antenna_Gain

Clicking the “Save” button will result in the current IXIT values being exported to a csv file. The format of this file is:

IXIT_name_0,IXIT_value_0
IXIT_name_1,IXIT_value_1
IXIT_name_2,IXIT_value_2
…

Clicking the “Load” button will attempt to read a csv file of IXIT values into the application. IXIT names in the csv file which are not recognised will be silently ignored.

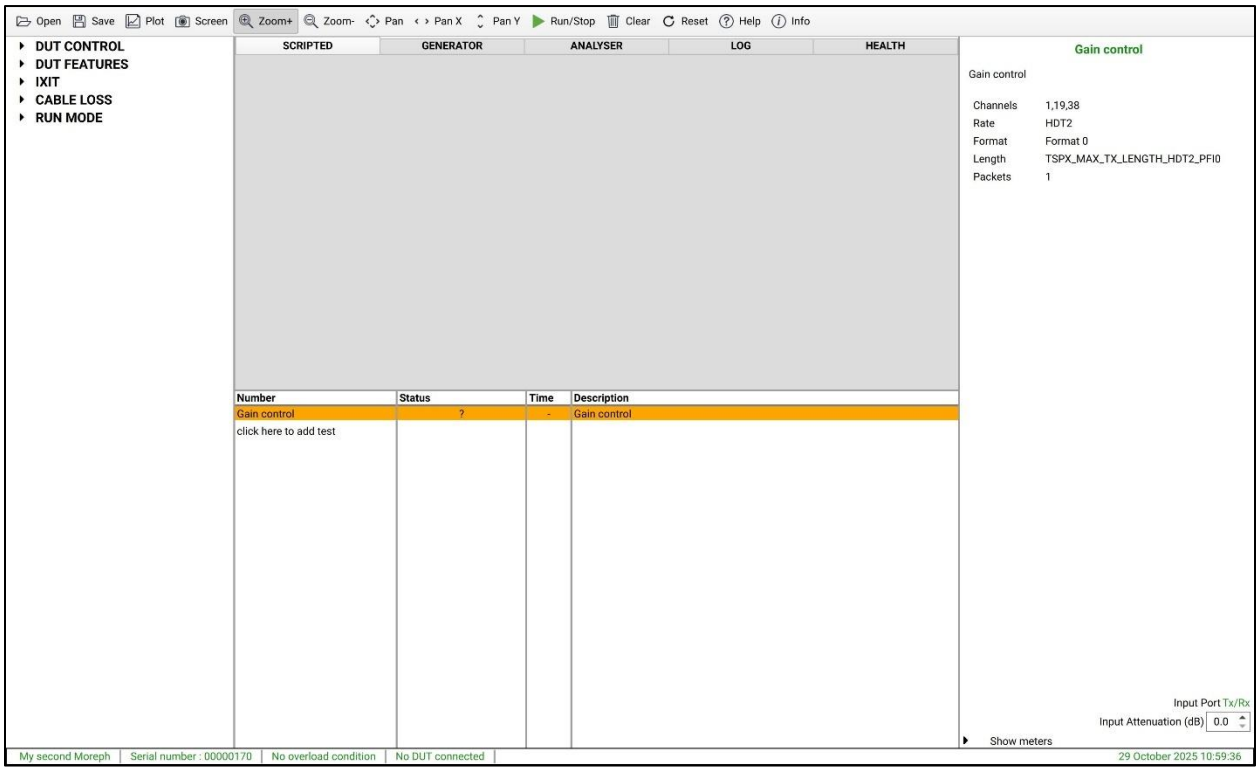
10 Scripted mode

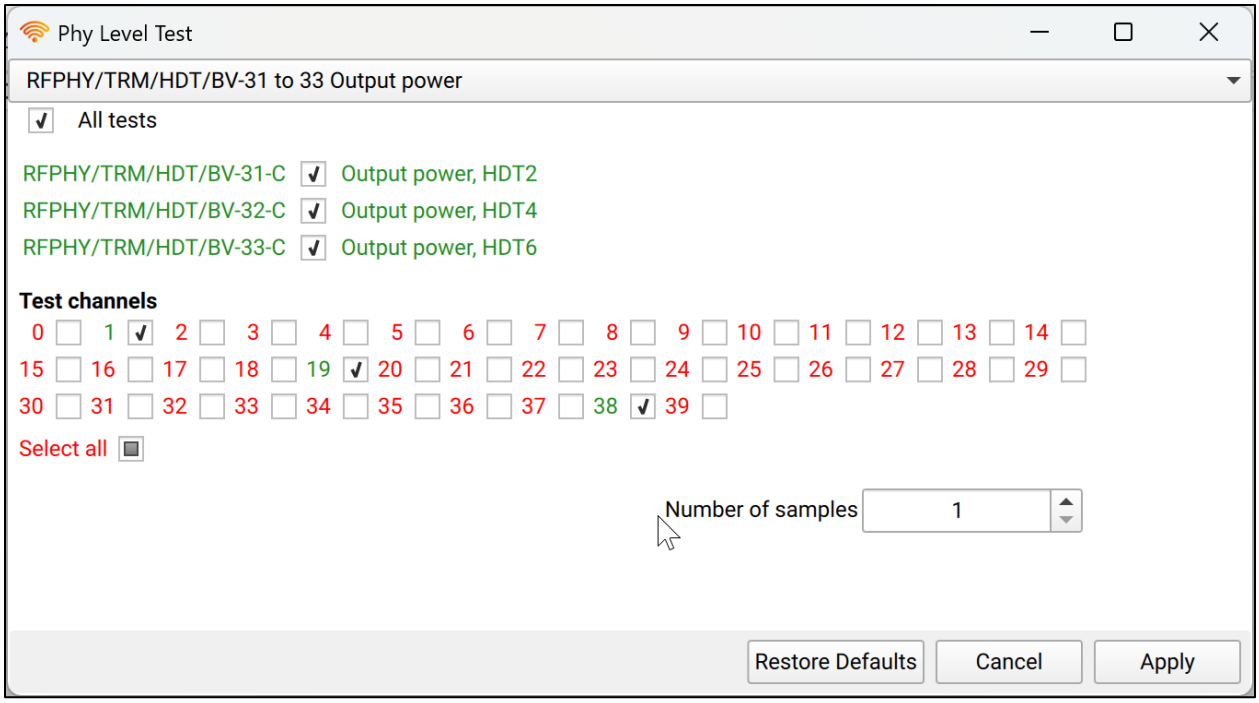
10.1 Overview

The scripted mode of operation permits the user to construct a list of tests to be executed, run the tests and then examine the results. All the tests are based on the Bluetooth test specifications.

10.2 Creating a script

To start creating a script, select the “Scripted” tab and the click on “click here to add test”. A pop-up window will then appear listing the tests which can be added.





The tests are grouped into sets of similar tests, for example the same test methodology but for different packet types or lengths. A list of the test groups can be found in the drop down menu at the top of the pop-up window.

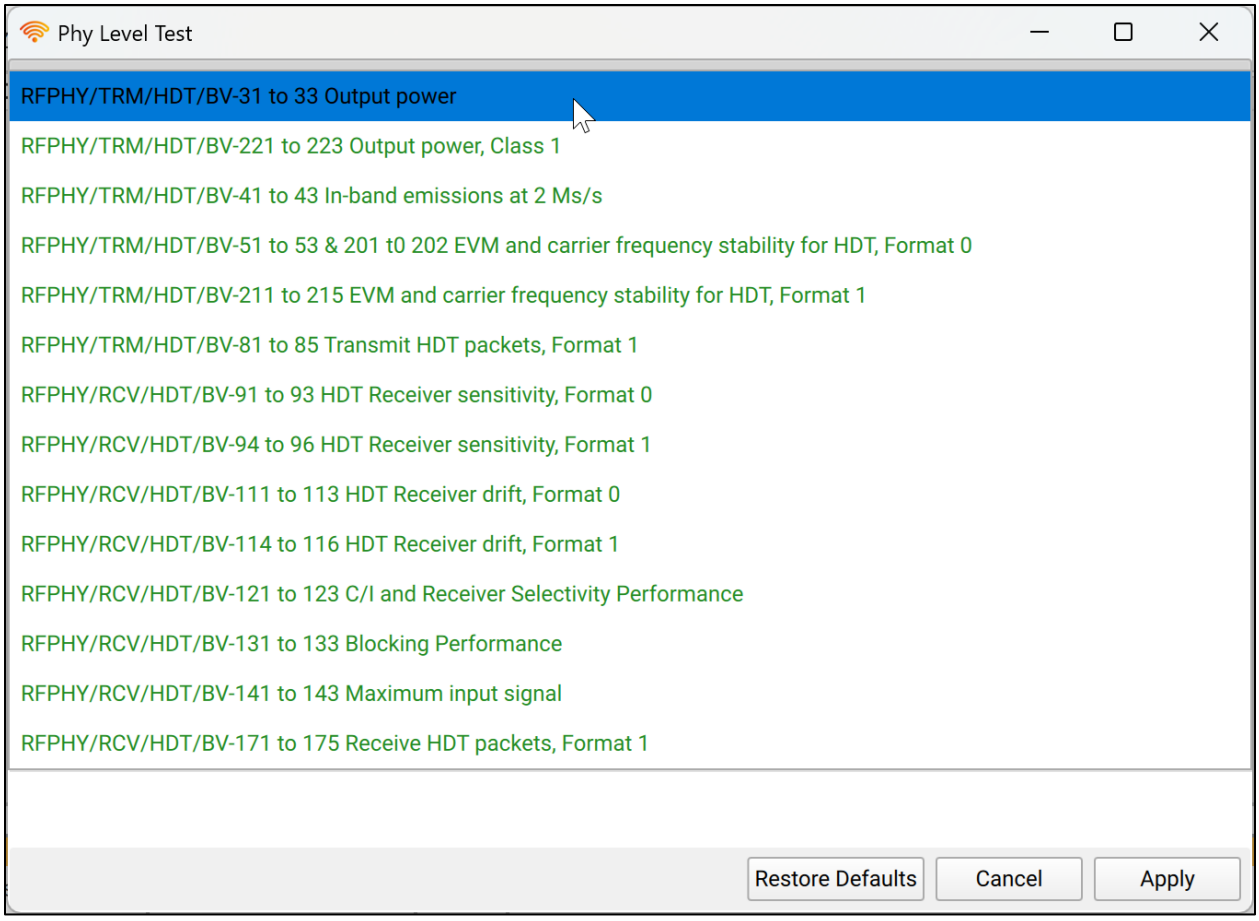
Select the group of tests you are interested in performing. The window will then show a list of all tests in that group together with a brief description of the test.

Individual tests can be selected or deselected using the checkbox next to the test name. Tests displayed in red cannot be selected because they require support for features which have not been selected in the “IUT features” menu. LE Rx Test Command version 3 is required for all receiver tests and LE Tx Test Command version 5 for all transmitter tests.

The menu make contains a quick selection checkbox at the top of the window. This allows all tests to be selected or deselected.

A series of checkboxes are provided to select the channels used by the tests. A checkbox to select or deselect all channels is also provided.

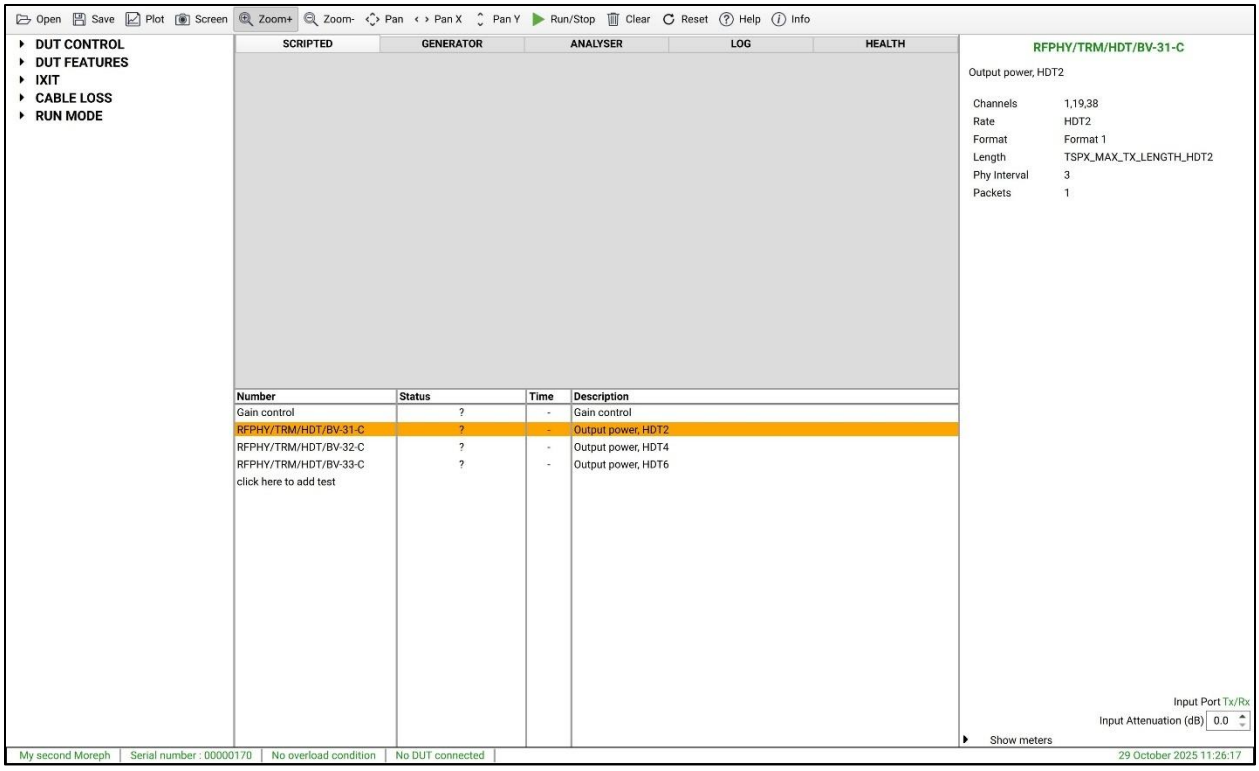
Either a “Packets” or “Blocks” spin box will be visible. This provides a means to alter how long the test will run for. Some tests, such as transmit output power, are based on the number of packets tested. For these tests the “Packets” spin box will be visible and the number of packets used in the test can be adjusted. Other tests, such receiver sensitivity tests, are based on the number of payload blocks received. For these tests the “Blocks” spin box will be visible and the number of payload blocks used in the test can be adjusted.



At the bottom of the window are various buttons:

1. Restore defaults
   Will select all available tests, restore the channel selection to the channels defined in the test specification and restore the “Packets” or “Blocks” to the value defined in the test specification.
2. Apply
   Closes the pop-up window and enters the selected tests into the test script.
3. Cancel
   Closes the pop-up window without entering the selected tests into the test script.
4. Discard
   Only visible when editing existing tests in the test script. Will result in the group of tests being edited being deleted from the test script.

On closing the pop-up window by clicking “Apply” the selected tests are displayed in the test script. The details of the highlighted test are displayed in the summary window.



Further groups of tests can be added by clicking on “click here to add test” which is always displayed at the bottom of the test script.

A test can be edited by double clicking on it. When the test is edited, the entire group to which the test belongs is opened up for editing.

To delete a group of tests, double click on any test within the group and then click “discard” in the pop-up menu.

To delete an entire test script, click the “Reset” button on the toolbar.

A test script can be saved by clicking the “Save” icon on the toolbar. An existing test script can be loaded by clicking the “Open” icon on the toolbar.

10.3 Gain control test

At the top of the test script there is a “Gain control” test which cannot be edited or deleted. When this test is run, a simple test is executed with the IUT transmitting at maximum output power. The purpose of this test is to allow the moreph to measure the maximum output power of the IUT. This is then used to set the moreph input attenuation so as to maximise the dynamic range of the unit.

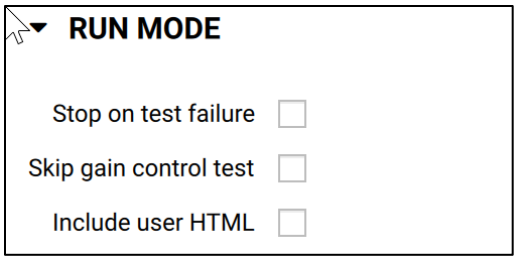
This test can be skipped by selecting “Skip gain control test” under the “Run mode” menu. However, if the test is skipped it is essential the moreph input attenuation is set prior to run the

test script. The input attenuation should be adjusted such that the input stages to the moreph do not overload when the IUT is transmitting at maximum output power. Overload conditions are shown in the status bar and by the label over the “Input power” meter on the right hand side of the window turning red.

If only receiver tests have been selected in the test script, then the gain control test will automatically be skipped.

10.4 Setting the run mode

The “Run mode” menu contains parameter which alter the execution of the scripted tests.



By default the test script will be run to completion unless communications with the IUT are lost or the user aborts the script. However, by checking “Stop on test failure” the test script will terminate as soon as a test failure is detected. This can be useful for diagnosing test failures.

The corresponding API call is `RF_stopOnFail( bool )`.

Prior to each test script run a gain control test is executed. The purpose of the gain control test is to estimate the output power from the IUT. Once this has been established the gain controls within the moreph receiver chain are automatically adjusted to maximise the dynamic range. If “Skip gain control test” is checked, then this procedure will be skipped.

The corresponding API call is `RF_skipGainControl( bool )`.

The results of the tests can be exported in html format by clicking the “Report” button on the toolbar. If under the “Run mode” menu the “Include user HTML” has been selected, then the selected file containing bespoke HTML will be included in the exported data. This provides a facility for adding additional details about the IUT or the company logo to the results file. When the “Include user HTML” button is checked, a file dialog will open up so the user can select the file containing the bespoke HTML.

The corresponding API call is is `RF_includeUserHTML( char *filename )`. If the file name is an empty string, then the inclusion of user HTML in the results file will be terminated.

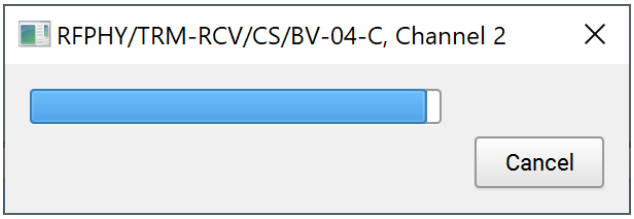
10.5 Running a script

Prior to running a script it will be necessary to:

1. Connect to a IUT
2. Ensure that the “IUT features” menu accurately represents the features the IUT supports, either by using the “Query” button on the toolbar or entering the values manually.
3. Ensure that the appropriate IXIT values have been entered in the “IXIT” menu.
4. Generate a script to run.

The script is run by clicking the green “Run” icon on the toolbar which will then turn into a red “Stop” icon.

Whilst the test script is running a pop-up window will appear showing the progress of each test. The title shows which test is being executed, the channel being tested and the fraction of the test executed. This window also contains a “Cancel” button which can be used to abort the execution of the test script.



As each test is executed a green tick or red cross is added to the status column in the test script to indicate whether the test passed or failed. For tests which are performed on more than one channel a tick or cross is added for each channel tested. The time column is also updated to show how long the test took to complete.

To run a script from the API, switch to scripted mode and then call:

```
RF_run();
```

To abort a running test script call:

```
RF_abort();
```

and to wait for a test script to complete with a timeout in milli-seconds:

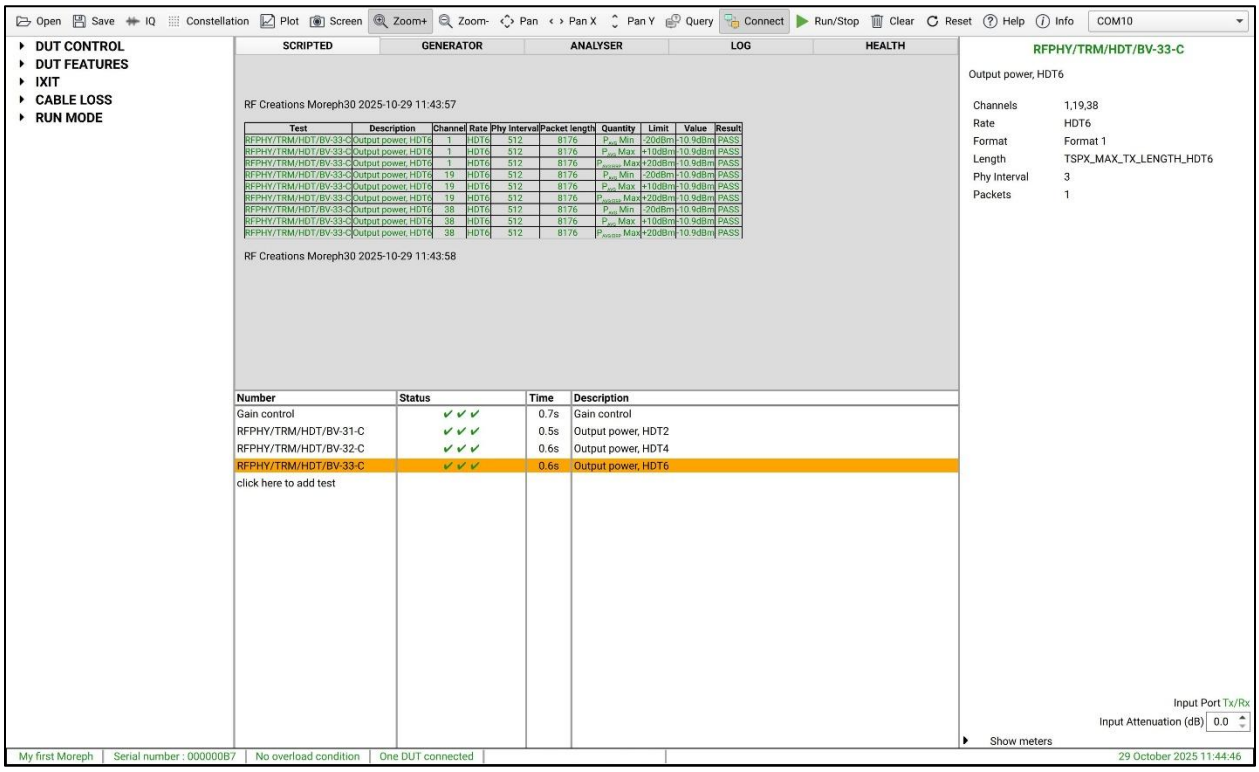
```
RF_wait( uint32_t timeout_ms );
```

10.6 Examining results

After a test script has been run the status column in the test script will indicate whether the test passed or failed. There will be one tick or cross per channel the test was executed on. An

orange warning triangle will be displayed if it was not possible to run the test. The time column will indicate the total test time for each test.

The numerical or graphical results for a particular test can be displayed by highlighting the test in the test script. The results will then be displayed in the upper portion of the window.



The table in the upper half of the window shows:

Table item	Meaning	API
Test	Unique test number	RF_getTestName
Description	Brief description of the test	RF_getTestDescription
Channel	RF channel number used in the test	RF_getTestChannel
Rate	HDT rate indicator	RF_getTestRate
Phy Interval	HDT phy interval	RF_getTestPhyInt
Packet Length	HDT packet length in octets	RF_getTestPktLen
Wanted Power	Power of the wanted signal in dBm, receive tests only	RF_getTestWantedPwr
Interferer Frequency	Frequency of the interfering signal in MHz, receive tests only	RF_getTestInterfererFreq
Interferer Power	Power of the interfering signal in dBm, receive tests only	RF_getTestInterfererPwr
Quantity	The quantity being measured by the test	RF_getTestQuantity
Limit	The limit against which the quantity being measured is tested	RF_getTestLimit
Value	The value of the measured quantity	RF_getTestValue
Result	See below	RF_getTestResult

The result item may take the following values:

1. Pass
   The test passed.
2. Fail
   The test failed.
3. Info
   The quantity being measured is not tested against a limit but it provided purely for diagnostic purposes.
4. Exception
   The measured quantity does not meet the test limit but may be allowed as an exception.
5. Exception – 1
   The measured quantity does not meet the test limit and will be retried with a relaxed interferer power.
6. Exception – 2
   The measurement has been repeats with a relaxed interferer power. The measured quantity still does not meet the test limit but may be allowed as an exception.

The results of the tests can be exported in html format by clicking the “Report” button on the toolbar. The exported data will include both the tabular results and the graphical results. If under the “Run mode” menu the “Include user HTML” has been selected, then the selected file containing bespoke HTML will be included in the exported data.

All the results can be cleared by clicking the “Discard” button on the toolbar.

10.7 Debugging test scripts

If a test script fails, select the “Stop on test failure” option in the “Run mode” menu. Run the test script which will abort once the failing test it reached. Switch to signal generator mode if it is a receiver test or switch to signal analyser mode if it is a transmitter test. The signal generator/signal analyser will now be set to precisely the same parameters which were used in the failing test. The signal analyser will also display the last packet which was collected.

For transmitter tests the last collected packet can be examined in detail to find the cause of the failure. The summary window will contain details of the last measurements made. Since the signal analyser is now configured to mirror the failing test, further packets can be analysed by clicking the “Run” button on the toolbar.

For receiver tests, clicking the “Run” button on the toolbar will result in the signal generator sending the sequence of packets and interferers used in the scripted test and reading back the BLER from the IUT. The BLER will be displayed in the top right of the graphics window.

10.8 Running test scripts from the API

After connecting to the moreph and launching the channel sounding application, the application should be placed into scripted mode:

```
RF_setOperatingMode( RF_scripted );
```

A test script should then be loaded from a file:

```
RF_loadTestScript( filename );
```

and the test script executed:

```
RF_run();
```

Wait until the test script completes with a timeout in milli-seconds:

```
RF_waitProcedure( timeout_ms );
```

The results from running the test script are contained in a single table. The first index into the table is 0. The row of a given test within the table can be obtained by calling:

```
RF_findTest( const char *testName, uint32_t  occurrence, int32_t &row);
```

where the `testName` is a character string containing either a portion or the entire test name. `occurrence` is an unsigned integer specify which occurrence of `testName` should be searched for. A value of 0 will return the first occurrence. The function returns the corresponding row in the table or -1 if the specified test name and occurrence are not found or the test is still running.

For a given row in the results table it is possible to read back the following quantities:

Table item	Meaning	API
Test	Unique test number	`RF_getTestName`
Description	Brief description of the test	`RF_getTestDescription`
Channel	RF channel number used in the test	`RF_getTestChannel`
Rate	HDT rate indicator	`RF_getTestRate`
Phy Interval	HDT phy interval	`RF_getTestPhyInt`
Packet Length	HDT packet length in octets	`RF_getTestPktLen`
Wanted Power	Power of the wanted signal in dBm, receive tests only	`RF_getTestWantedPwr`
Interferer Frequency	Frequency of the interfering signal in MHz, receive tests only	`RF_getTestInterfererFreq`
Interferer Power	Power of the interfering signal in dBm, receive tests only	`RF_getTestInterfererPwr`
Quantity	The quantity being measured by the test	`RF_getTestQuantity`
Limit	The limit against which the quantity being measured is tested	`RF_getTestLimit`
Value	The value of the measured quantity	`RF_getTestValue`
Result	See below	`RF_getTestResult`

Notes:

These functions all take the same arguments:

```
RF_error_t RF_getXXX( uint32_t row, char *result, uint32_t n )
```

`row` indicates which row in the results table is to be read.

`result` is a character string containing the requested table cell.

`n` is the maximum number of characters that can be placed in `result`.

These functions return a `RF_error_t` which should be checked.

The test description, the quantity being tested and the units in which it is measured are all returned as strings which may contain html embedded characters.

Where there is insufficient data to determine a test outcome, the measured value will be returned as NaN.

11 Signal generator mode

11.1 Overview

The signal generator mode permits up to 5 different signals to be generated simultaneously. These signals can be a mix of wanted signals and interferers. If a IUT is connected and the wanted signal contains a finite number of repetitions, then the BLER will also be calculated and displayed. This allows the user to either interactively change the wanted/interferer signal generation whilst monitoring the BLER, or interactively change the IUT configuration whilst monitoring the BLER.

In the upper half of the signal generator window is a spectrum showing the signals which are currently being generated in the 2.4GHz band. The spectrum has a resolution bandwidth of 100kHz.

If a IUT is connected and BLER is being calculated, then the BLER is shown in the top right hand corner of the spectrum.

In the lower half of the signal generator window are 5 tabs, each one assigned to a different signal. A drop down menu provides a means of selecting the type of signal which the tab is generating.

When the signal generator is active, the output power meter at the bottom right of the window (by default this is hidden and will need to be expanded) will show an approximation to the instantaneous peak power being generated by the moreph, excluding the out-of-band blocker. Messages will appear in the status bar and the above the output power meter if the moreph output is being overloaded.

The signal generator will refuse to start and will display an error message in the status bar if the sum of the signals being generated exceeds the maximum peak output power of the moreph, which is 0dBm.

11.2 Signal selection

Each of the 5 tabs controls one of the signals to be generated. Under each tab there is a drop down menu which determines the type of signal to be generated. Options for the signal type are:

1. None (signal generation is off)
2. HDT
3. CW (in-band)
4. AWGN
5. Out-of-band CW blocker

The number of each type of signal which can be selected is limited:

Signal type	Maximum number instantiated
HDT	4
CW (in-band)	2
AWGN	1
Out-of-band CW blocker	1

On selecting a particular tab, the parameters governing the generation of the associated signal are displayed in the bottom half of the signal generator window. If the tab corresponds to an active signal, then an appropriate menu will be opened on the left hand side to permit the generation parameters to be modified.

When programming the signals from the API, first call:

```
RF_setCurrentSignal( uint32_t signal_number )
```

where `signal_number` is in the range 0 to 4. All subsequent API calls will reference `signal_number`.

To set the signal type from the API:

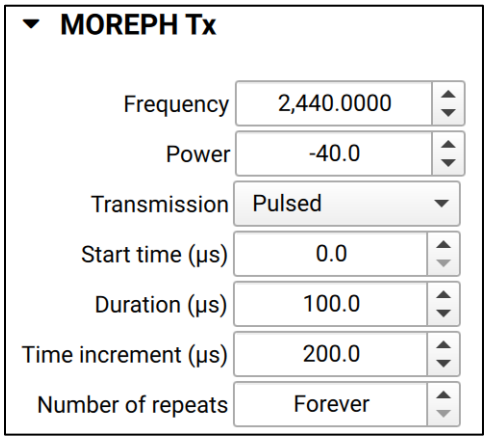
```
RF_setSignal(uint32_t sig)
```

where `sig` is an enum defining the signal type. Possible values are:

Signal type	Enum signal_t	Value
OFF	OFF	0
HDT	HDT	4
CW (in-band)	CW	6
AWGN	AWGN	7
Out-of-band CW blocker	OOB_CW	8

11.3 Common signal parameters

The parameters for configuring the selected signal are displayed under the “Moreph Tx” menu. Some parameters are common to the majority of signal types.



The meaning of these parameters and the associated API calls for each signal type are:

UI Item	Description	API calls

Frequency (MHz)	Frequency of transmission in MHz. Not relevant for AWGN.	`RF_setHDT_Frequency(double freq_MHz)`
		`RF_setCW_Frequency(double freq_MHz)`
		`RF_setOOB_Frequency(uint32_t freq_MHz)`

Power (dBm)	Power of transmission in dBm or dBm/MHz for AWGN	`RF_setHDT_Power(double dBm)`
		`RF_setCW_Power(double dBm)`
		`RF_setAWGN_Power(double dBm)`
		`RF_setOOB_Power(double dBm)`

Transmission	Whether the transmission is a continuous or pulsed/packetized. Not relevant for out-of-band blocker which is always continuous.	`RF_setHDT_Continuous(bool continuous)`
		`RF_setCW_Continuous(bool continuous)`
		`RF_setAWGN_Continuous(bool continuous)`

Start time (µs)	Start time of the first transmission. Not relevant for out-of-band blocker which starts immediately.	`RF_setHDT_StartTime(uint32_t micro_seconds)`
		`RF_setCW_StartTime(uint32_t micro_seconds)`
		`RF_setAWGN_StartTime(uint32_t micro_seconds)`

Duration (µs)	The duration of the first transmission, excluding ramp up/down times. Not relevant for out-of-band blocker which is always continuous.	`RF_setHDT_Duration(uint32_t micro_seconds)`
		`RF_setCW_Duration(uint32_t micro_seconds)`
		`RF_setAWGN_Duration(uint32_t micro_seconds)`

Time increment (µs)	The increment between the start time of successive transmissions. Not relevant for out-of-band blocker which is always continuous.	`RF_setHDT_Interval(uint32_t micro_seconds)`
		`RF_setCW_Interval(uint32_t micro_seconds)`
		`RF_setAWGN_Interval(uint32_t micro_seconds)`

Number of repeats	The number of times a pulsed/packetized transmission should be repeated. A value of zero results in indefinite repeats.	`RF_setHDT_Repeats(uint32_t nRepeat)`
		`RF_setCW_Repeats(uint32_t nRepeat)`
		`RF_setAWGN_Repeats(uint32_t nRepeat)`

UI Item	Description	API calls
	Not relevant for out-of-band blocker which is always continuous.	

The parameters have the following limitations:

Parameter	HDT	CW	AWGN	Blocker
Frequency resolution	1kHz	1kHz	-	1MHz
Maximum frequency	2485MHz	2485MHz		6000MHz
Minimum frequency	2395MHz	2395MHz	-	24MHz
Power resolution	0.1dBm	0.1dBm	0.1dBm/MHz	0.1dBm
Maximum power	-3dBm	0dBm	-41.2dBm/MHz	-28dBm
Minimum power	-120dBm	-120dBm	-120dBm/MHz	-50dBm
Maximum start time	1,800,000,000µs	1,800,000,000µs	1,800,000,000µs	1,800,000,000µs
Minimum start time	0µs	0µs	0µs	0µs
Maximum duration	Packet length	1,800,000,000µs	1,800,000,000µs	1,800,000,000µs
Minimum duration	Packet length	0µs	0µs	0µs
Maximum increment	1,800,000,000µs	1,800,000,000µs	1,800,000,000µs	1,800,000,000µs
Minimum increment	Duration + 40µs	Duration + 40µs	Duration + 40µs	Duration + 40µs
Maximum repeats	10000000	10000000	10000000	10000000
Minimum repeats	0	0	0	0

For each signal, apart from the out-of-band CW blocker, a start time can be specified. This is the time after the signal generator is started that the first transmission for that signal will be made.

Apart from the out-of-band blocker, all signals can either be continuous or pulsed/packetised. If a pulsed/packetized signal has been selected then further parameters determine the precise transmission times of subsequent pulses/packets.

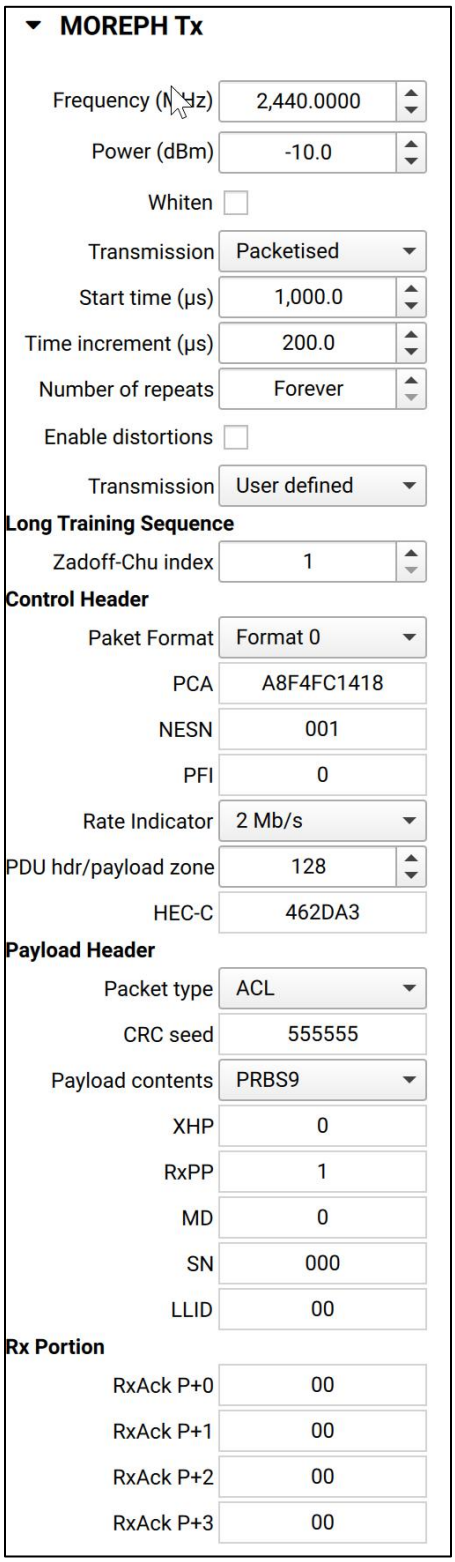
The “Duration” parameter determines the length of the transmission, excluding power ramp up/down times. For packetised signals such as HDT the duration is calculated automatically from the packet contents. For pulsed signals such as CW and AWGN, the duration must be set explicitly.

The “Time increment” parameter determines the time interval from the start of one packet/pulse to the start of the subsequent packet/pulse. If the “Time increment” is shorter than the “Duration” plus the power ramp up/down time, then the lowest possible integer multiple of the “Time increment” will be used.

Packetised/pulsed signals will be repeated “Number of repeats” times. If “Number of repeats” is set to zero, then packets/pulses will be transmitted indefinitely.

11.4 HDT parameters

11.4.1General



Where a parameter is displayed as a hexadecimal value, it can be edited by typing over the displayed characters.

Where a parameter is displayed as a binary value, it can be edited either by typing over the displayed characters or by clicking on a binary digit to invert it.

There are 3 parameters which control how the packet will be generated:

UI Item	Description	API calls
Whiten	Determines whether the packet should be whitened.	`RF_setHDT_Whiten(bool)`
Enable distortions	Determines whether distortions should be applied to the transmitted packet.	`RF_setHDT_Distort(bool)`

Transmission	Determines whether the packet should be a standard vector, a test packet or an arbitrary user defined packet.	`RF_setHDT_Type(char *)`
		`Possible values are:`
		`"Test"`
		`"User"`
		`"N" where N is the number of a standard vector in the range 1 to 33`

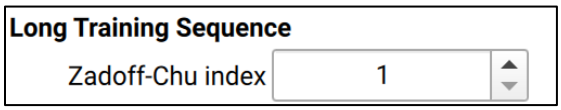
The 3 possible transmission types are:

1. **Standard vectors**. These are the 33 standard vectors defined for testing by the Bluetooth SIG. All parameters used for determining the packet contents are fixed for each standard vector.
2. **User defined**. Any of the parameters defining the packet contents can be set by the user. Arbitrary HDT packets can be generated in this mode.
3. **Test packet**. The packet format is the same as would be generated by issuing an HCI test command to the IUT. In this mode the following parameters are fixed:
    a. No whitening
    b. Zadoff-Chu index = 7
    c. PCA = 0x9F15555555
    d. NESN = 1
    e. ACL packet format
    f. Any fields not specified set to zero

If whitening is enabled, then the whitening seed is derived from:

1. For standard vectors the whitening seed is 0x7A7F
2. For user defined packets the physical channel index is used to derive the whitening seed

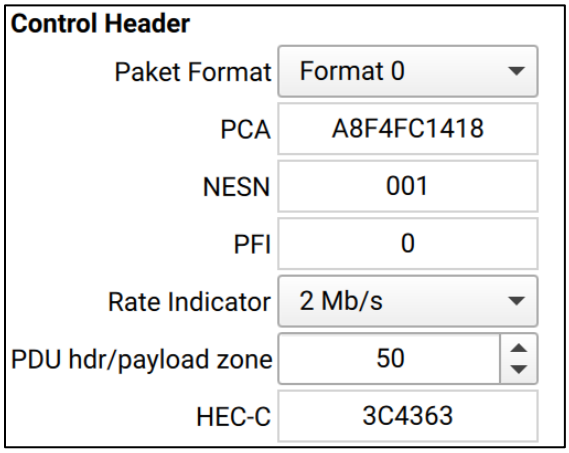
11.4.2 Long training sequence (LTS)



The LTS is defined by a single parameter which is the Zadoff-Chu index.

UI Item	Description	API calls
Zadoff-Chu index	Sets the Zadoff-Chu index for the LTS. Must be in the range 1 to 16 inclusive.	`RF_setHDT_ZadoffChu(uint32_t)`

11.4.3 Control header



The contents of the control header are defined by the following parameters:

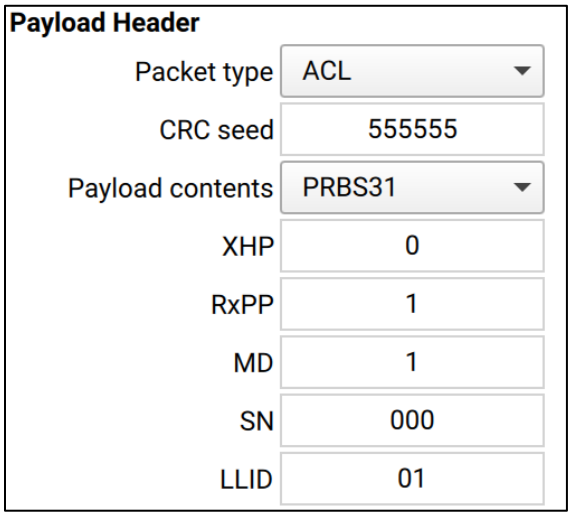
UI Item	Description	API calls

Packet format	Sets the packet format.
	Options are:
	Short format
	Format 0
	Format 1
		`Set using calls to:`
		`RF_RF_setHDT_PFI(uint32_t)`
		`RF_setHDT_RI(uint32_t)`

PCA	Physical channel address. Bits [39:24] are placed into the control header. Bits [23:0] are used as the CRC seed for HEC-C and HEC-P.	`RF_setHDT_PCA(char *)` `The argument is a 10 character string containing a hexadecimal representation of the PCA.`
NESN	The value of the NESN	`RF_setHDT_NESN(uint32_t)`
PFI	Selects between short format/format 0 packets and format 1 packets.	`RF_RF_setHDT_PFI(uint32_t)`
Rate indicator	Selects the data rate for the packet. Permissible values are 0 to 5. A value of 0 will select short format packets.	`RF_RF_setHDT_RI(uint32_t)`

UI Item	Description	API calls
PDU hdr/payload zone	The length of the header/payload zone in octets for format 0 packets.	`RF_setHDT_HdrPayload_Zone_Len(uint32_t)`
PDU hdr zone	The length of the header zone in octets for format 1 packets.	`RF_setHDT_Hdr_Zone_Len(uint32_t)`
Phy interval	The phy interval. Permissible values are b'00 to b'11.	`RF_setHDT_Phy_Interval(uint32_t)`
HEC-C	The value of the HEC-C. This is automatically calculated and cannot be set by the user.	`Calculated automatically, cannot be modified.`

11.4.4 Payload header



Contents of the payload header are determined by the following parameters:

UI Item	Description	API calls

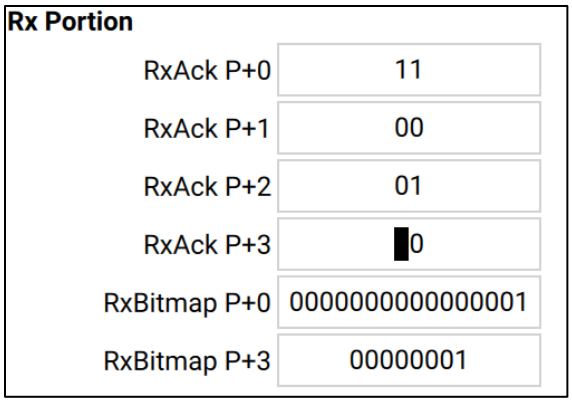
Packet type	Sets the packet type.
	Options are:
	ACL
	CIS
	BIS
		`RF_setHDT_Pkt_Type(const char*)`
		`Argument can be one of:`
		`"ACL"`
		`"CIS"`
		`"BIS"`

CRC seed	Value of 32bit CRC seed to be used for ACL and CIS packets.	`RF_setHDT_Seed(uint32_t)`
BIS number	Value of 8bit BIS number to be used to generate the CRC seed for BIS packets.	`RF_setHDT_BIS_Number(uint32_t)`
BaseCRCinit	Value of the 24bit base CRC initialisation used to generate the CRC seed for BIS packets.	`RF_setHDT_BIS_CRCinit(uint32_t)`

Payload contents	The contents of the payload.
	Possible values are:
	11110000
	10101010
	11111111
	00000000
	00001111
		`RF_setHDT_Payload(const char*)`
		`Character string argument can be one of the following:`
		`"RF_prbs9",`
		`"RF_11110000",`
		`"RF_10101010",`
		`"RF_prbs15",`
		`"RF_11111111",`

UI Item	Description	API calls
	01010101 PRBS9 PRBS15 PRBS11 PRBS20 PRBS23 PRBS29 PRBS31 Where the bit sequences are in transmission order.	"RF_00000000", "RF_00001111", "RF_01010101", "RF_PRBS11", "RF_PRBS20", "RF_PRBS23", "RF_PRBS29", "RF_PRBS31",
XHP	Determines whether extended header portion is present.	RF_setHDT_Hdr_XHP(bool)
RxPP	Determines whether RxPP portion is present.	RF_setHDT_Hdr_RXPP(bool)
CSTF	CSTF value for BIS packets.	RF_setHDT_Hdr_CSTF(bool)
MD	MD value for ACL packets.	RF_setHDT_Hdr_MD(bool)
CIE	CIE value for CIS packets.	RF_setHDT_Hdr_CIE(bool)
SN	3 bit sequence number.	RF_setHDT_Hdr_SN(uint32_t)
TxPP	Indicates whether TxPP portion is present.	RF_setHDT_Hdr_TXPP(bool)
LLID	Value of the LLID for format 0 packets.	RF_setHDT_Hdr_LLID(uint32_t)

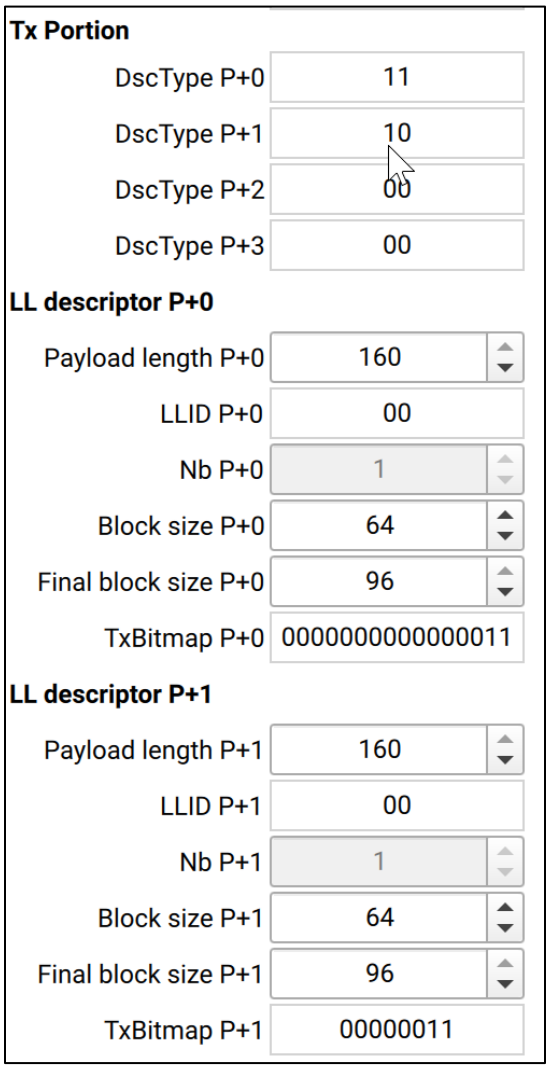
11.4.5 Rx portion



If an Rx portion is present in the payload header then the following parameters can also be defined:

UI Item	Description	API calls
RxACK P+0 RxACK P+1 RxACK P+2 RxACK P+3	2bit value of Rx acknowledgement	RF_setHDT_RxAck(uint32_t) The argument is an 8bit number containing all 4 RxACK values.
RxBitmap P+0 RxBitmap P+1 RxBitmap P+2 RxBitmap P+3	Bit maps for Rx acknowledgements for when RxACK. The bitmap is 8bits when the corresponding RxACK is b'10 and 16bits when the corresponding RxACK is b'11.	RF_setHDT_RxBitMap_0(uint32_t) RF_setHDT_RxBitMap_1(uint32_t) RF_setHDT_RxBitMap_2(uint32_t) RF_setHDT_RxBitMap_3(uint32_t)

11.4.6Tx portion

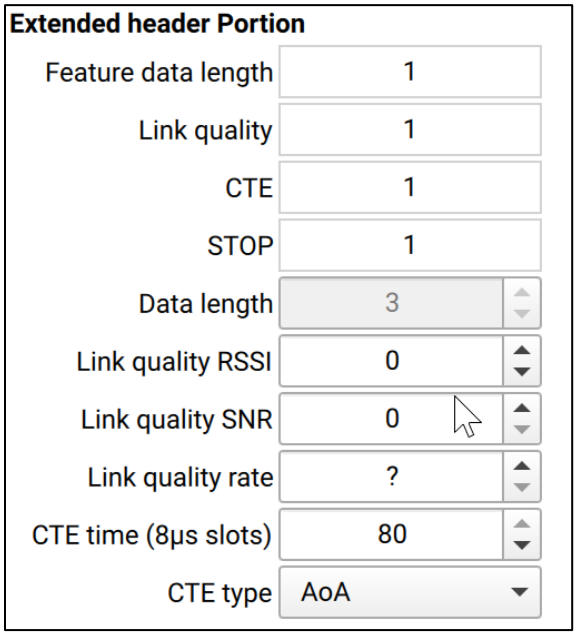


If an Tx portion is present in the payload header then the following parameters can also be defined:

UI Item	Description	API calls
DscType P+0 DscType P+1 DscType P+2 DscType P+3	2bit value of DscType.	RF_setHDT_TxDscType (uint32_t) The argument is an 8bit number containing all 4 DscType values.
Payload length P+0 Payload length P+1 Payload length P+2 Payload length P+3	The total length of the payload in octets for P+X.	RF_setHDT_TxPayload_Len_0(uint32_t) RF_setHDT_TxPayload_Len_1(uint32_t) RF_setHDT_TxPayload_Len_2(uint32_t) RF_setHDT_TxPayload_Len_3(uint32_t)
LLID P+0 LLID P+1 LLID P+2 LLID P+3	The LLID for P+X.	RF_setHDT_TxLLID_0(uint32_t) RF_setHDT_TxLLID_1(uint32_t) RF_setHDT_TxLLID_2(uint32_t) RF_setHDT_TxLLID_3(uint32_t)

UI Item	Description	API calls
Nb P+0 Nb P+1 Nb P+2 Nb P+3	The number of blocks (Nb) parameter for P+X. This value does not include the last block.	RF_setHDT_TxNb_0(uint32_t) RF_setHDT_TxNb_1(uint32_t) RF_setHDT_TxNb_2(uint32_t) RF_setHDT_TxNb_3(uint32_t)
Block size P+0 Block size P+1 Block size P+2 Block size P+3	The block size in octets for P+X. This applies to all blocks apart from the last block.	RF_setHDT_TxBlock_Size_0(uint32_t) RF_setHDT_TxBlock_Size_1(uint32_t) RF_setHDT_TxBlock_Size_2(uint32_t) RF_setHDT_TxBlock_Size_3(uint32_t)
Final block size P+0 Final block size P+1 Final block size P+2 Final block size P+3	The size of the final block in octets for P+X.	RF_setHDT_TxLast_0(uint32_t) RF_setHDT_TxLast_1(uint32_t) RF_setHDT_TxLast_2(uint32_t) RF_setHDT_TxLast_3(uint32_t)
TxBitmap P+0 TxBitmap P+1 TxBitmap P+2 TxBitmap P+3	The bitmap of blocks being transmitted. This is an 8bit map if the corresponding DscType is b'10 and a 16bit map if the corresponding DscType is b'11.	RF_setHDT_TxBitMap_0 RF_setHDT_TxBitMap_1 RF_setHDT_TxBitMap_2 RF_setHDT_TxBitMap_3

11.4.7Extended header portion



If the extended header portion is present, then it may contain the following fields:

UI Item	Description	API calls
Feature data length	Determines whether the feature data length field is included in the extended header.	RF_setHDT_XHP_Len(bool)
Link quality	Determines whether the link quality field is included in the extended header.	RF_setHDT_XHP_LinkQuality(bool)

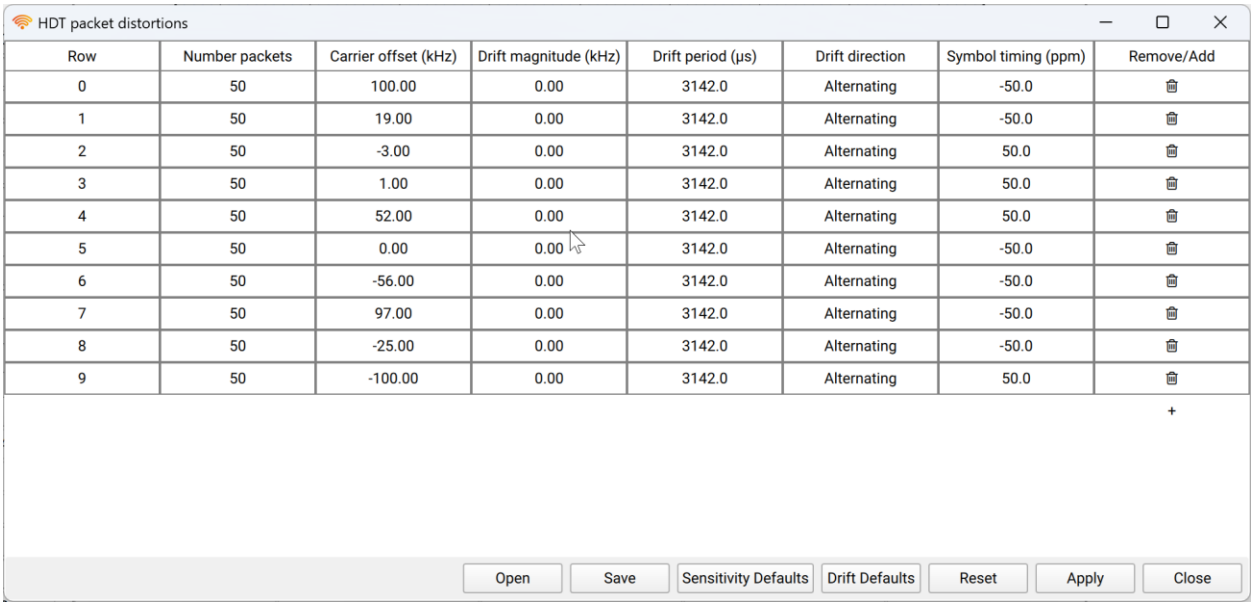
UI Item	Description	API calls
CTE	Determines whether the CTE field is included in the extended header.	`RF_setHDT_XHP_CTE(bool)`
STOP	The STOP field of the extended header.	`RF_setHDT_XHP_Stop(bool)`
Data length	The length of the extended header data in octets.	`Automatically calculated.`
Link quality RSSI	The link quality RSSI field. -127 to +20dBm, or 127 if unknown.	`RF_setHDT_XHP_LinkQuality_RSSI(int32_t)`
Link quality SNR	The link quality SNR estimate. 0 to 30dB or 21 if not available.	`RF_setHDT_XHP_LinkQuality_SNR(uint32_t)`
Link quality rate	The preferred link quality rate, with a value of 0 indicating that the preferred rate is not available.	`RF_setHDT_XHP_LinkQuality_Rate(uint32_t)`
CTE time (8µs slots)	Length of the CTE in 8µs slots. Must be between 2 and 80.	`RF_setHDT_XHP_CTE_Time(uint32_t)`

CTE type	The type of the CTE.
	0 = AoA
	1 = AoD with 1µs slots
	2 = AoD with 2µs slots
		`RF_setHDT_XHP_CTE_Type(uint32_t)`
		`0 = AoA`
		`1 = AoD with 1µs slots`
		`2 = AoD with 2µs slots`

11.5 Signal distortions

Signal distortions are selected by checking the “Enabled distortions” checkbox or calling the API function `RF_setHDT_Distort(bool)`.

To program the distortions which will be applied, click the “Distortions” button. A pop-up window will appear in which the distortions can be set.



In the HDT packet distortions window is a table of distortions. Each row in the table corresponds to a specific set of distortions. These distortions will be applied to the next *N*

packets, where *N* is defined in the “Number of packets” column. Once this set of *N* packet has been set, the distortions will be changed to those specified in the next row of the table. Once all the rows in the table have been exhausted, execution will revert to the first row in the table.

The table contains the following distortions:

Quantity	Units	Minimum	Maximum	Resolution
Number of packets	-	1	65535	1
Carrier frequency offset	kHz	-500	500	0.01
Drift magnitude	kHz	0	500	0.01
Drift period	µs	5	30000	0.1
Drift direction	-	Up / Down / Alternating		
Symbol timing error	Ppm	-200	200	0.1

The drift is applied as a sinusoidal variation on top of the carrier frequency offset. The magnitude of the sinusoid is determined by the “Drift magnitude” parameter and the period of the sinusoid by the “Drift period” parameter. The contribution of the drift to the carrier offset is always zero at the start of each packet. The drift may then proceed in either an upward or downward direction depending on the setting of the “Drift direction” parameter. If this is set to “Alternating” then the direction of the drift will be inverted on each packet.

Any value in the table can be edited by clicking on it. Once a value has been edited, the borders of the table will change from gray to red. This indicates that the displayed table is not the same as that currently being used by the HDT packet generator. The displayed table will only take affect once the “Apply” button is pressed. Once applied, the borders of the table will revert to gray to indicate that the displayed values are now the same as those being used by the HDT packet generator.

Rows can be deleted by clicking the trash can in the right most column.

Rows can be added to the end of the table by clicking the “+” symbol just below the last row under the “Remove/Add” column.

Distortion tables can be saved and reloaded using the “Open” and “Save” buttons at the bottom of the window.

The Bluetooth SIG define two sets of distortions tables. The first of these is used for receiver sensitivity tests and the second for test the performance of the receiver against carrier drift. These predefined tables can be loaded using the “Sensitivity Defaults” and “Drift Defaults” buttons at the bottom of the window.

The “Reset” button will reset the table to the either the table which was present when the window opened, or the last table programmed using the “Apply” button.

The “Close” button will close the HDT packet distortions window but will NOT apply the table. The displayed table will only take effect when the “Apply” button is pressed.

The distortion table can be programmed via the API using the function:

```
RF_setHDT_Distortions( double *Fcarrier, double *DriftMag, double
*DriftPeriod, uint32_t *DriftDirn, double *PPM, double *Squish, double
*Rotate, uint32_t *Npkts, uint32_t n)
```

The arguments to this function are:

`Fcarrier` an array of doubles of length `n` containing the carrier offset frequencies in kHz for each row in the table.

`DriftMag` an array of doubles of length `n` containing the drift magnitudes in kHz for each row in the table.

`DriftPeriod` an array of doubles of length `n` containing the drift periods in µs for each row in the table.

`DriftDirn` an array of uint32_t of length `n` containing the drift directions for each row in the table. Permissible values are 0 = alternating, 1 = upwards, 2 = downwards.

`PPM` an array of doubles of length `n` containing the symbol timing errors in ppm for each row in the table.

`Squish` an array of doubles of length `n`. The contains of this array are not currently used and should be set to 0.0.

`Rotate` an array of doubles of length `n`. The contains of this array are not currently used and should be set to 0.0.

`Npkts` an array of uint32_t of length `n` containing the number of packets to which the distortions will be applied for each row in the table.

`n` a uint32_t containing the number of row in the table, and hence the length of the arrays in the function call.

After programming the distortion table it will be necessary to stop and restart the HDT signal generator before it will take effect. It is also necessary to ensure the API function `RF_setHDT_Distort(bool)` has been called to enable the distortion table.

11.6 Starting and stopping the signal generator

The signal generator can be started by clicking the green “Run/Stop” icon on the toolbar. The corresponding API call is `RF_run()`.Prior to making this call, it is essential that the application is

in signal generator mode, which can be accomplished by calling `RF_setOperatingMode(RF_siggen)`.

If one of the signals which the signal generator has been programmed to generate contains a finite number of repetitions, then the signal generator will automatically stop once all repetitions of that signal have been transmitted. If more than one signal has been programmed with a finite number of transmissions, the signal generator will automatically stop when the repetitions of all such signals have been transmitted.

If the signal generator is set to stop automatically, then the API can wait for completion using the call `RF_wait( uint32_t timeout_ms)`. The call takes a single argument which is timeout in units of ms.

The signal generator can be stopped at any time by clicking the red “Run/Stop” icon on the toolbar. The corresponding API call is `RF_abort()`.

11.7 Measurement of BLER with signal generator

If the signal generator has been programmed to transmit a finite number of HDT packets and a IUT is connected to the application, then once the packets have been transmitted then the IUT will be queried in order to determine the BLER. The result of the BLER calculation will be displayed at the top right of the graph in the upper half of the signal generator window. The API can access the calculated value of the BLER by calling `RF_BLER( double *BLER )`.

If more than one HDT signal has been programmed with a finite number of repetitions, then it is the lowest numbered signal which is used for the BLER calculations.

When a IUT is connected and the signal generator is programmed to send a finite number of HDT packets, then the “Continuous” icon is also visible on the toolbar. If this is clicked rather than the “Run/Stop” icon, then once all packets have been sent and the BLER calculated, the transmission of a new set of packets and a new BLER calculation will be initiated. This process will be repeated indefinitely. This functionality allows the user to either interactively changed interfering signals or adjust IUT settings whilst continuously monitoring the impact on BLER. The corresponding API call is `RF_continuous()`. Use the `RF_abort()` call terminate the process.

11.8 Saving and restoring settings

The signal generator settings can be saved by clicking the “Save” icon on the toolbar. The corresponding API call is `RF_saveSigGen(const char* filename)`.

The signal generator settings can be recalled by clicking the “Open” icon on the toolbar. The corresponding API call is `RF_loadSigGen(const char* filename)`.

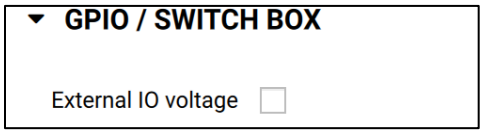
The signal generator settings can be reset to their default values by clicking the “Reset” icon on the toolbar. corresponding API call is `RF_resetRF()`. Prior to making this call, it is essential that the application is in signal generator mode, which can be accomplished by calling `RF_setOperatingMode(RF_siggen)`.

11.9 GPIO

The moreph has 5 GPIO lines which it can toggle to indicate transmission times. The 5 GPIO are assigned as follows:

GPIO	Function
3	At least one CW transmission active.
4	Lowest numbered HDT transmission active.
5	Second lowest numbered HDT transmission active.
6	Second highest numbered HDT transmission active.
7	Highest numbered HDT transmission active.

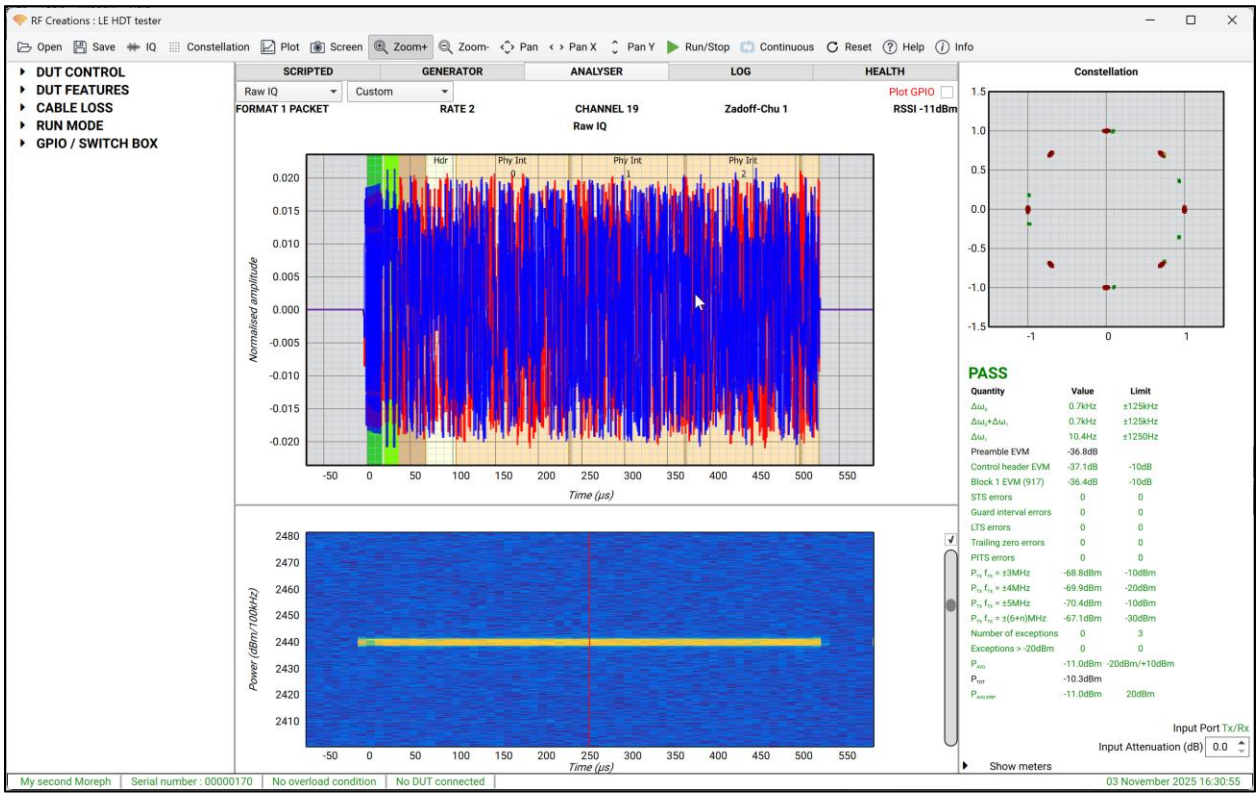
The IO voltage for the logic lines can either be applied externally or it can use the 3v3 generated internally by the moreph. The selection of the IO voltage source is made under the “GPIO / Swith box” menu.



The corresponding API call is `RF_extIOvolts(bool extIO)`.

12 Signal analyser

12.1 Overview



The signal analyser will perform detailed analysis on packets received either via a conducted path or over-the-air. All parameters which are part of the Bluetooth HDT test specification are calculated on each packet received, including in-band emissions.

To select signal analyser mode using the API, call `RF_setOperatingMode(RF_sigana)`.

The signal analyser displays a variety of data:

1. Above the upper graph, the packet format, HDT rate, channel number, Zadoff-Chu index of preamble and the approximate RSSI are displayed.
2. The upper graph can be selected to show any one of the following quantities using the drop down menu at the top left of the window:
    a. Raw IQ
    b. IQ post root-raised cosine filter
    c. Constellation points
    d. Error vector
    e. Power

f. Power measured in 3MHz Gaussian resolution bandwidth, as defined in Bluetooth HDT tests
g. Adjacent channel power (in-band emissions)
h. Packet contents

It is possible to display logic signals superimposed on the upper graph.

The lower graph shows a spectrogram encompassing the entire received packet. The frequency resolution is 100kHz (Gaussian filter) and the time resolution is 10µs. These have been chosen to correspond to the values the Bluetooth HDT tests use for in-band emission measurements.

In the right hand window, a constellation diagram is shown.

The right hand window also lists all the measurements which have been made on the received packet.

The input power meter can also be displayed in the right hand window, which can be useful to check the approximate input power to the moreph.

12.2 Demodulation

The demodulation of the HDT signal follows the procedure set out in the Bluetooth HDT test specification. This procedure has been extended to permit packets with more than 1000 symbols in the payload to be demodulated.

In the first stage of the demodulation process, a rough synchronisation with the STS of the HDT packet is sought. Once this has been achieved, the EVM of the preamble (STS + GI + LTS) is minimised over the following parameters:

1. Signal amplitude
2. Carrier phase
3. Carrier frequency
4. Symbol timing

The preamble is then compensated using these optimised parameter values and sliced.

In the second stage of the demodulation process, the control header is compensated using the parameters determined in the first stage and sliced.

In subsequent stages of the demodulation process, the payload (including payload header, terminating zeros and PITS, when present) is divided up into 1000 symbol chunks. The last chunk may be composed of fewer than 1000 symbols.

The first chunk uses the values of signal amplitude and symbol timing as calculated from the preamble. The EVM of the first 1000 symbols is then minimised over the carrier phase and carrier frequency. The first chunk is then compensated using these parameter values and sliced.

For subsequent chunks, only the signal amplitude calculated from the preamble is used. All other parameters, ie carrier phase, carrier frequency and symbol timing are estimated by minimising the EVM over the symbols within the chunk.

This demodulation process is probably inferior to that which could be achieved with well designed tracking loops for symbol timing, carrier frequency, carrier phase and signal amplitude. It has been chosen simply because it reflects what has been described in the Bluetooth Core Specification.

12.3 Time domain quantities

12.3.1 General

The contents of the upper graph are determined by the drop-down menu at the top left of the graph.

All time domain quantities are plotted from a point approximately 70µs prior to the start of the STS to a point approximately 70µs after the last symbol of the payload. The additional time before and after the packet allows the impact of synthesiser switching or power ramp up/down to be captured.

12.3.2 IQ data

Selection of “Raw IQ” shows the IQ data collected in a bandwidth which is flat over 3MHz. The “RRC filter IQ” option shows the same data but after having passed through a root-raised cosine filter.

12.3.3 Constellation data

Selection of “Constellation points” results in the display of the calculated IQ values at the symbol sampling points after compensation for carrier frequency, carrier phase and symbol timing (see Demodulation). The red points represent the I values and the blue points the Q values. On zooming in, black circles will be drawn around the sliced data, ie where the constellation points are expected to lie. For the preamble (STS + GI + LTS), terminating zeros and PITS, the sliced data will be correct. For other symbols the sliced data may be in error if the EVM of the received signal is particularly poor.

Selection of “Error vector” plots the difference between the received IQ samples and the sliced data. This may show slight increases at the start and end of each 1000 symbol chunk, since the demodulation parameters are optimised over the entire 1000 symbols.

12.3.4 Power data

Selection of “Power” shows the power profile of the packet when viewed in the 3MHz flat bandwidth. “Power 3MHz RBW” shows the power profile when viewed through the combination of a 3MHz Gaussian filter and a 3MHz wide video filter, as specified in the Bluetooth HDT test specification. This tends to show slightly lower values due the narrower bandwidth. The power variation is also less due to the impact of the video filter.

12.3.5 Background labelling

Within the packet, a coloured background is included which is labelled according to the symbols being transmitted at that point in time. The labelled time regions are:

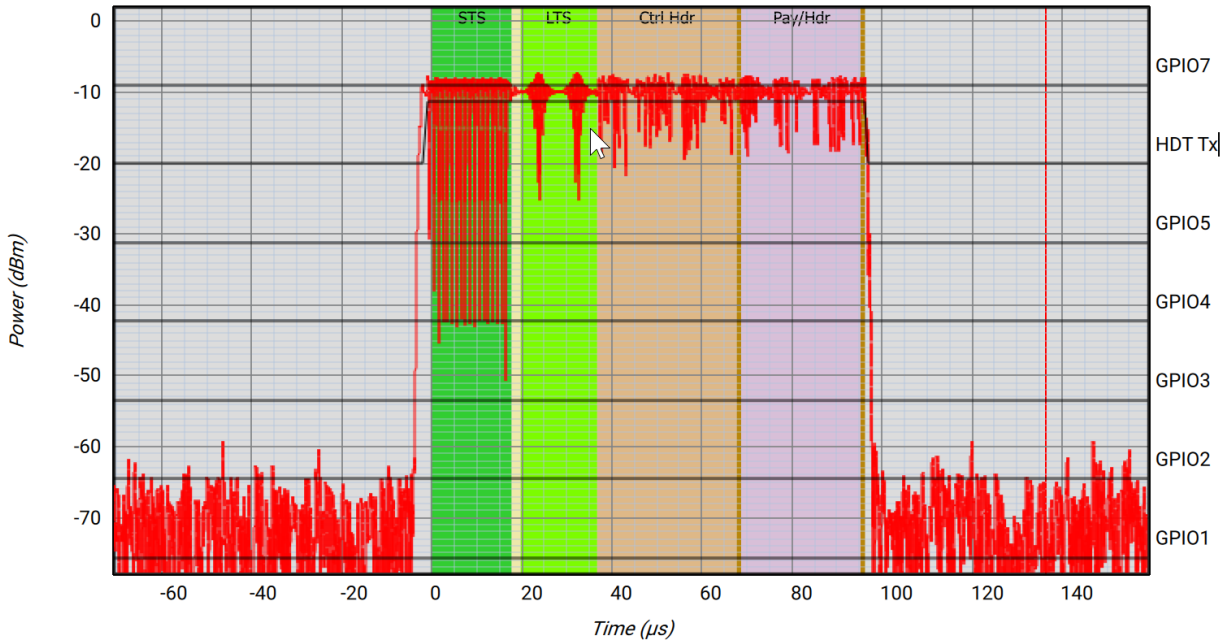
1. Short training sequence (STS)
2. Guard interval (GI)
3. Long training sequence (LTS)
4. Control header
5. Control header terminating zeros
6. Header/Payload zone (format 0 packets only)
7. Header/Payload zone terminating zeros (format 0 packets only)
8. Header zone (format 1 packets only)
9. Header zone terminating zeros (format 1 packets only)
10. Phy interval (format 1 packets only)
11. Phy interval terminating zeros (format 1 packets only)
12. Phy interval training sequence (PITS, format 1 packets only)

12.3.6 Navigation

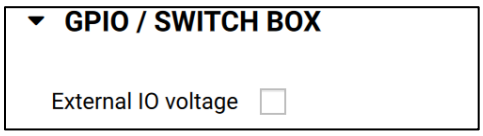
It is possible to navigate through the packet using these labels. Above the graph is a drop down menu which permits either a custom time window (determined by current zoom settings) or one of the above labelled time regions to be displayed. If multiple copies of the selected time region exist in the packet, for example phy intervals, then a spin box will appear to allow the required region to be selected.

12.3.7 GPIO

The moreph simultaneously capture logic data alongside the IQ data of the HDT packet. This permits logic signals from the IUT to be displayed alongside other time domain quantities. Logic input lines 1 to 7 are available for use.



The IO voltage for the logic pod can either be applied externally or it can use the 3v3 generated internally by the moreph. The selection of the IO voltage source is made under the “GPIO / Swith box” menu.



The corresponding API call is `RF_extIOvolts(bool extIO)`.

To display logic signals alongside the other time domain quantities, click the “Plot GPIO” checkbox at the top right of the upper graph. Each GPIO signal is shown as a black horizontal trace with a label on the right hand side of the graph. The label text can be edited.

12.3.8Delta cursors

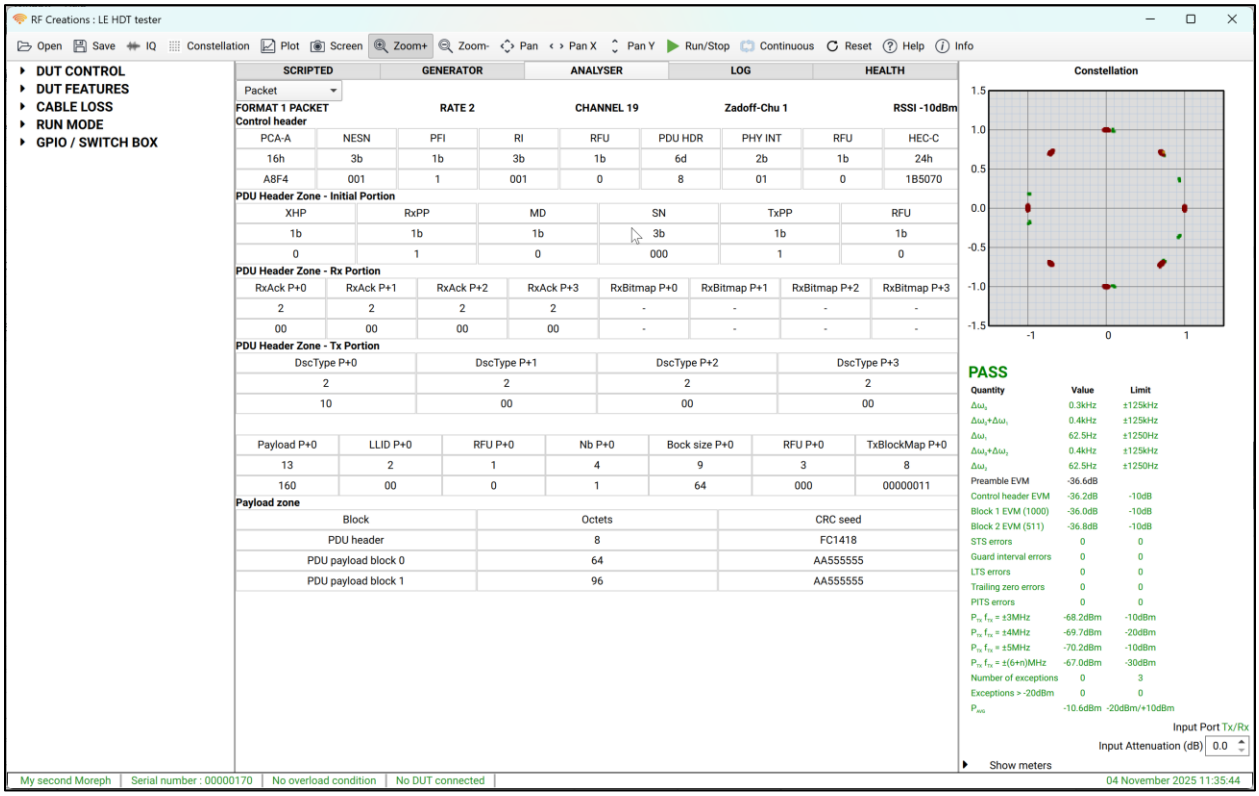
By default the vertical cursors in the upper and lower graphs are coupled whilst a time domain quantity is displayed by the upper graph. The title of the lower graph will show the corresponding cursor time.

If the user clicks on either the upper or lower graph, then a fixed cursor will be inserted into the plot. The title of the lower graph will now show the time delta between the fixed cursor and the active cursor. Clicking again will remove the fixed cursor.

12.4 Packet contents

From the drop down menu controlling the displayed quantity, it is possible to select “Packet”. Selection of this option will result in the signal analyser window displaying the contents of the last received packet. All the values displayed can be read back via the API.

The table at the bottom of the window shows the number of octets in each segment of the payload together with the associated CRC seed. The CRC seed is calculated on the basis that there are no bit errors in the received data so that the received CRC and payload contents can be used to infer the CRC seed. This provides a simple mechanism for checking that the transmission is using the expected CRC seed.



UI Item	Description	API calls
General		
Channel	The LE RF channel the packet was received on.	`RF_Rx_Channel(uint32_t *chan)`
RSSI	The approximate RSSI of the received signal.	`RF_Rx_RSSI(double *rssi)`
Preamble		
Zadoff-Chu	The Zadoff-Chu index of the received preamble.	`RF_Rx_Zadoff_Chi(uint32_t *index)`
Control Header		
NESN	The value of the NESN in the control header of the received packet.	`RF_Rx_NESN(uint32_t *NESN)`

UI Item	Description	API calls
PFI	The value of the packet format indicator in the control header of the received packet.	`RF_Rx_PFI(uint32_t *PFI)`
RI	The value of the rate indicator in the control header of the received packet.	`RF_Rx_RI(uint32_t *RI)`
PDU Header/Payload octets	Length of the format 0 PDU header/payload zone in octets.	`RF_Rx_Header_Payload(uint32_t *octets)`
PDU Header octets	Length of the format 1 PDU header zone in octets.	`RF_Rx_Header(uint32_t *octets)`
Phy Int	The value of the phy interval contained in the control header.	`RF_Rx_Phy_Interval(uint32_t *phyInt)`
HEC-C	HEC-C of control header of received packet.	`RF_Rx_HEC_C(uint32_t *HEC_C)`
-	Calculated seed for control header HEC-C.	`RF_Rx_HEC_C_SEED(uint32_t *HEC_C_seed)`
Payload header		
General		
-	HEC-P of PDU payload header of received packet.	`RF_Rx_HEC_P(uint32_t *HEC_P)`
Payload header – initial portion		
PDU header CRC seed	Calculated seed for PDU header.	`RF_Rx_HEC_P_SEED(uint32_t *HEC_P)`
XHP	Extended header flag in initial portion of payload header.	`RF_Rx_XHP(bool *XHP)`
RxPP	Rx portion flag in initial portion of payload header.	`RF_Rx_RXPP(bool *RxPP)`
MD	MD value from initial portion of payload header.	`RF_Rx_MD(bool *MD)`
SN	Sequence number from the initial portion of the payload header.	`RF_Rx_SN(uint32_t *SN)`
LLID	LLID from the initial portion of the payload header.	`RF_Rx_LLID(uint32_t *LLID)`
TxPP	Tx portion flag from the initial portion of the payload header.	`RF_Rx_TXPP(bool *TxPP)`
Payload header – Rx portion		
RxAck P+0 RxAck P+1 RxAck P+2 RxAck P+3	Combined RxACK for P+0 to P+3 from the payload header.	`RF_Rx_ACK(uint32_t *RxACK)`
RxBlockMap P=0	Rx block map for P+0 from payload header.	`RF_Rx_RxMap0(uint32_t *RxMap)`
RxBlockMap P+1	Rx block map for P+1 from payload header.	`RF_Rx_RxMap1(uint32_t *RxMap)`
RxBlockMap P+2	Rx block map for P+2 from payload header.	`RF_Rx_RxMap2(uint32_t *RxMap)`
RxBlockMap P+3	Rx block map for P+3 from payload header.	`RF_Rx_RxMap3(uint32_t *RxMap)`
Payload header – Tx portion		
DscType P+0 DscType P+1 DscType P+2 DscType P+3	Combined Tx descriptor types from payload header.	`RF_Rx_TxDSC(uint32_t *DscTyp)`
Payload P+0	Payload length in octets for Tx P+0.	`RF_Rx_TxPayload0(uint32_t *octets)`
Payload P+1	Payload length in octets for Tx P+1.	`RF_Rx_TxPayload1(uint32_t *octets)`

UI Item	Description	API calls
Payload P+2	Payload length in octets for Tx P+2.	RF_Rx_TxPayload2(uint32_t *octets)
Payload P+3	Payload length in octets for Tx P+3.	RF_Rx_TxPayload3(uint32_t *octets)
LLID P+0	LLID for Tx P+0.	RF_Rx_TxLLID0(uint32_t *LLID)
LLID P+1	LLID for Tx P+1.	RF_Rx_TxLLID1(uint32_t *LLID)
LLID P+2	LLID for Tx P+2.	RF_Rx_TxLLID2(uint32_t *LLID)
LLID P+3	LLID for Tx P+3.	RF_Rx_TxLLID3(uint32_t *LLID)
Nb P+0	The number of blocks for Tx P+0.	RF_Rx_TxNb0(uint32_t *Nb)
Nb P+1	The number of blocks for Tx P+1.	RF_Rx_TxNb1(uint32_t *Nb)
Nb P+2	The number of blocks for Tx P+2.	RF_Rx_TxNb2(uint32_t *Nb)
Nb P+3	The number of blocks for Tx P+3.	RF_Rx_TxNb3(uint32_t *Nb)
Block size P+0	The block size in octets used by P+0 for transmissions.	RF_Rx_TxBlockSize0(uint32_t *N)
Block size P+1	The block size in octets used by P+1 for transmissions.	RF_Rx_TxBlockSize1(uint32_t *N)
Block size P+2	The block size in octets used by P+2 for transmissions.	RF_Rx_TxBlockSize2(uint32_t *N)
Block size P+3	The block size in octets used by P+3 for transmissions.	RF_Rx_TxBlockSize3(uint32_t *N)
TxBlockMap P+0	Tx block map for P+0 from payload header.	RF_Rx_TxMap0(uint32_t *map)
TxBlockMap P+1	Tx block map for P+1 from payload header.	RF_Rx_TxMap1(uint32_t *map)
TxBlockMap P+2	Tx block map for P+2 from payload header.	RF_Rx_TxMap2(uint32_t *map)
TxBlockMap P+3	Tx block map for P+3 from payload header.	RF_Rx_TxMap3(uint32_t *map)
Extended header		
General		
-	Combined extended header flags.	RF_Rx_XHP_Flags(uint32_t *flags)
Len	Extended header length flag.	RF_Rx_XHP_Len_Flag(bool *len)
Link Quality	Extended header link quality flag.	RF_Rx_XHP_Link_Quality_Flag(bool *lq)
CTE	Extended header CTE flag.	RF_Rx_XHP_CTE_Flag(bool *cte)
Stop	Extended header stop flag.	RF_Rx_XHP_Stop_Flag(bool *lq)
NPI	Extended header NPI flag.	RF_Rx_XHP_NPI_Flag(bool *NPI)
XHP Length	Length value of extended header.	RF_Rx_XHP_Length(uint32_t *len)
Link quality		
RSSI	Extended header link quality RSSI.	RF_Rx_XHP_Link_Quality_RSSI(int32_t *RSSI)
SNR	Extended header link quality SNR.	RF_Rx_XHP_Link_Quality_SNR(int32_t *SNR)
Preferred rate	Extended header link quality preferred rate.	RF_Rx_XHP_Link_Quality_Rate(int32_t *rate)
CTE		
Time	Extended header CTE duration.	RF_Rx_XHP_CTE_Time(int32_t *T)
Type	Extended header CTE type.	RF_Rx_XHP_CTE_Type(int32_t *Typ)

12.5 Spectral measurements

12.5.1In-band emission plots

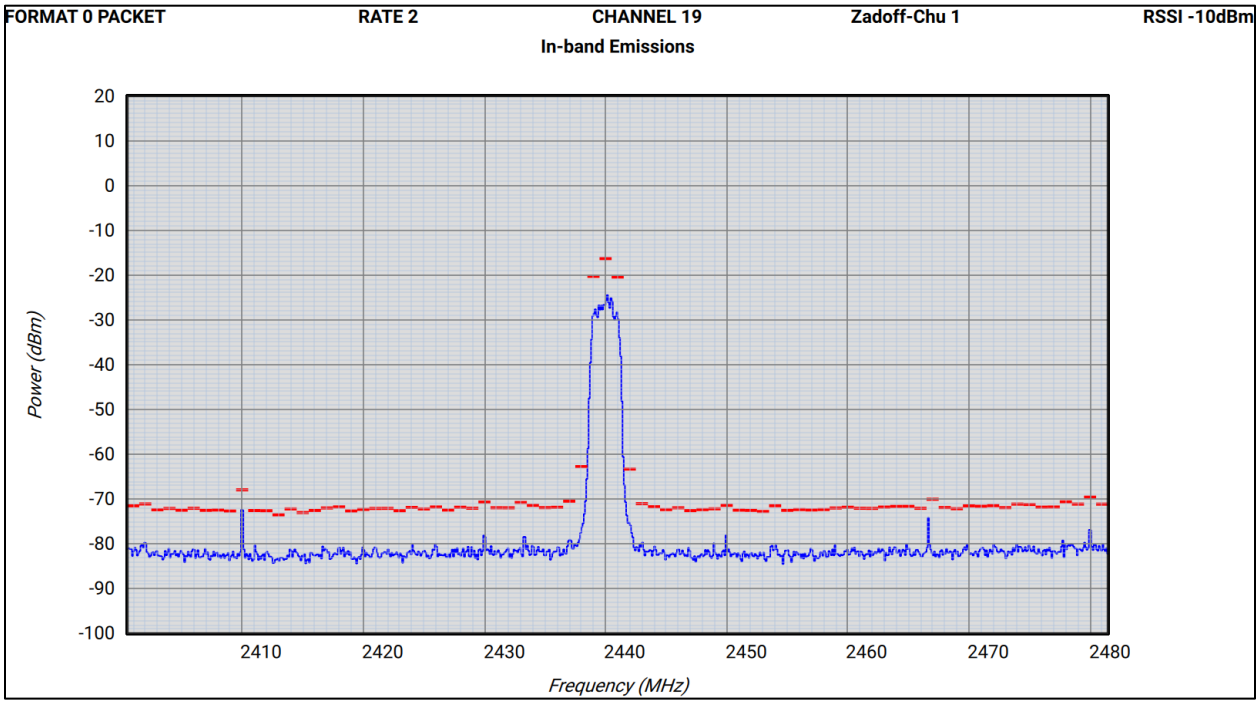
Selecting “In-band emissions” from the drop down menu of quantities to be displayed, results in an in-band emissions spectrum being displayed.

The Bluetooth HDT test specification requires the spectrum to be measured using:

1. A resolution bandwidth of 100kHz
2. A video bandwidth of 300kHz
3. An average detector
4. Trace peak hold

The results of these measurements starting at a time prior to the arrival of the packet and continuing to beyond the end of the packet are shown by the blue trace. The power in these measurements are then summed over 1MHz to produce the red trace.

Note: these measurements are performed over a single packet. To be consistent with the Bluetooth test specification this would need to be repeated for at least 10 packets. Therefore the results obtained in the scripted tests will be marginally different from those displayed and reported in the signal analyser mode.



12.5.2Spectrogram

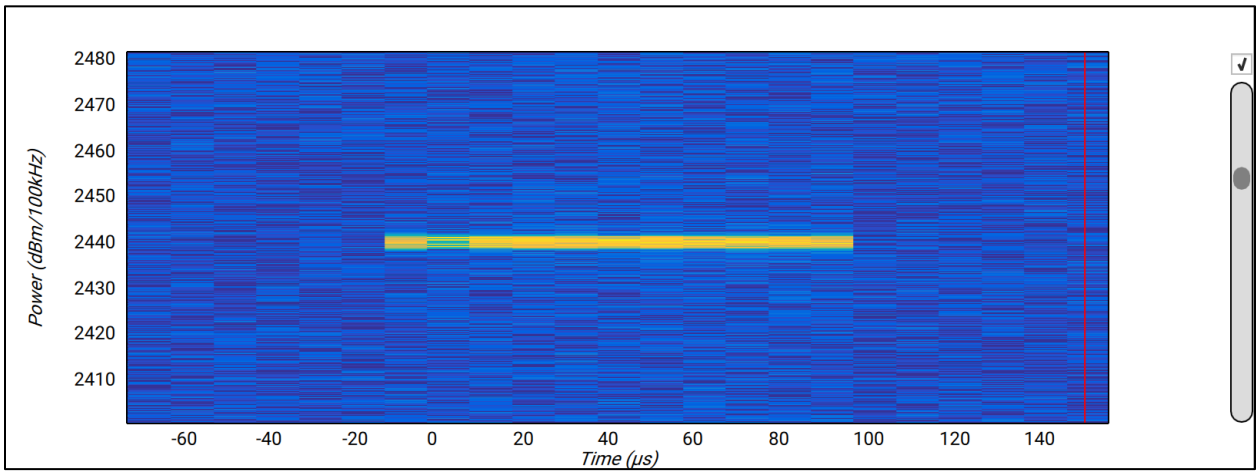
At the bottom of the signal analyser window is a spectrogram. This has a resolution of 100kHz x 10µs x 0.5dBm. It is effectively the raw data which is used in the Bluetooth in-band emission tests.

The spectrogram allows the user to determine what is causing in-band emission failures or marginal passes. It shows the distribution of energy in time and hence enables the user to identify certain events, for example synthesiser switching, power ramping, saturation, etc, which are dominating the in-band emission measurements.

The time axis of the spectrogram is coupled to that of the time domain plot displayed in the upper graph.

The checkbox to the right of the spectrogram permits the display to be either colour coded or grayscale.

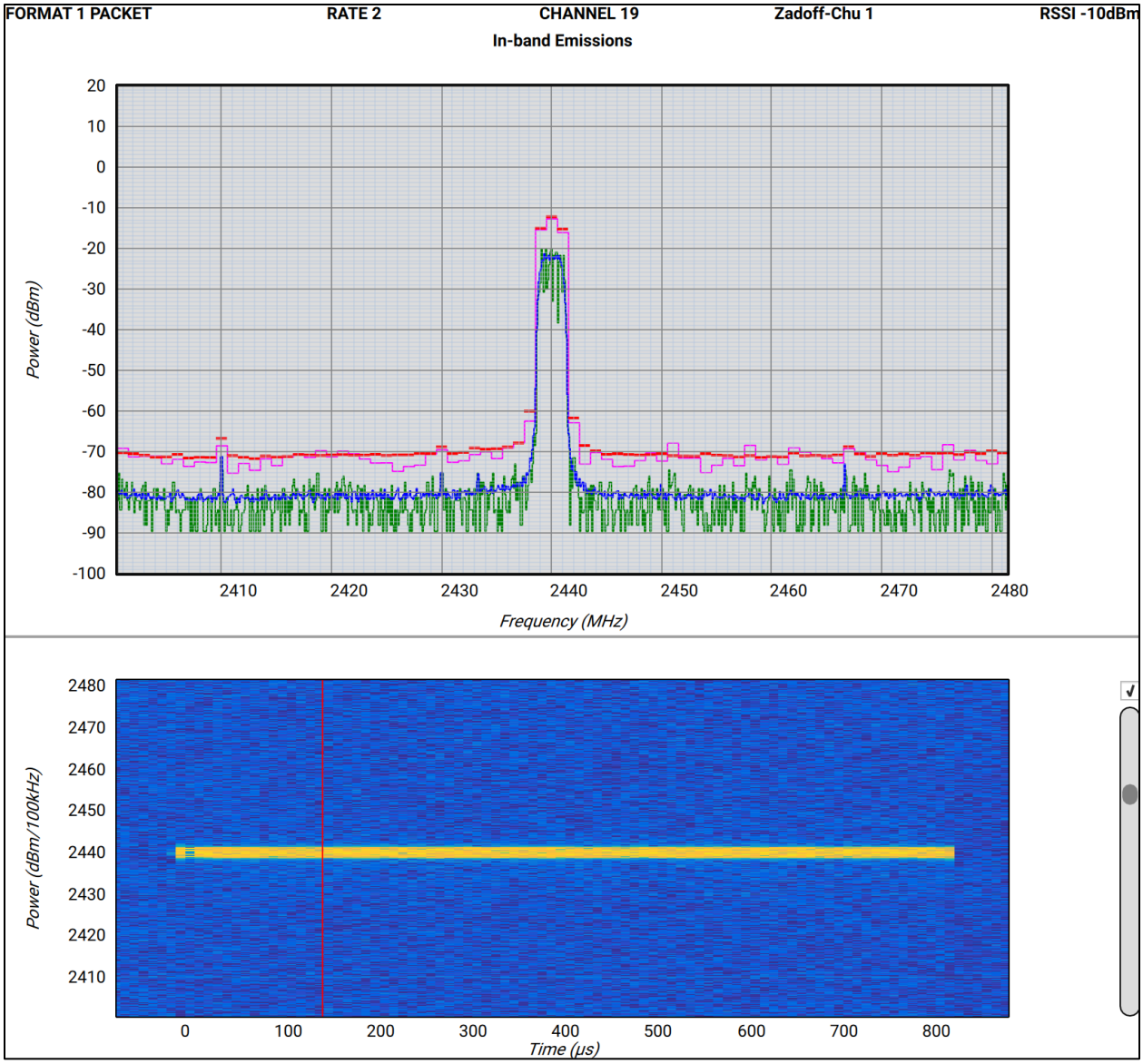
The slider to the right of the spectrogram controls the brightness.



12.5.3Coupled measurements

More detailed analysis of in-band emissions can be done by displaying the “In-band emissions” in the upper graph. When this is done, a cross section of the spectrogram is displayed on the same graph as the in-band emission spectra, whenever the mouse lies within the displayed graphs

The green line is simply the cross-section of the spectrogram at the current location of the cursor. The magenta line corresponds to 1MHz summations of the cross-section data. Note that the cross-section data has a resolution of 0.5dBm, so there are some slight inaccuracies in the resulting spectra. However, the accuracy is more than enough to investigate where in-band emission violations are occurring in both time and frequency.



12.6 Constellation diagram

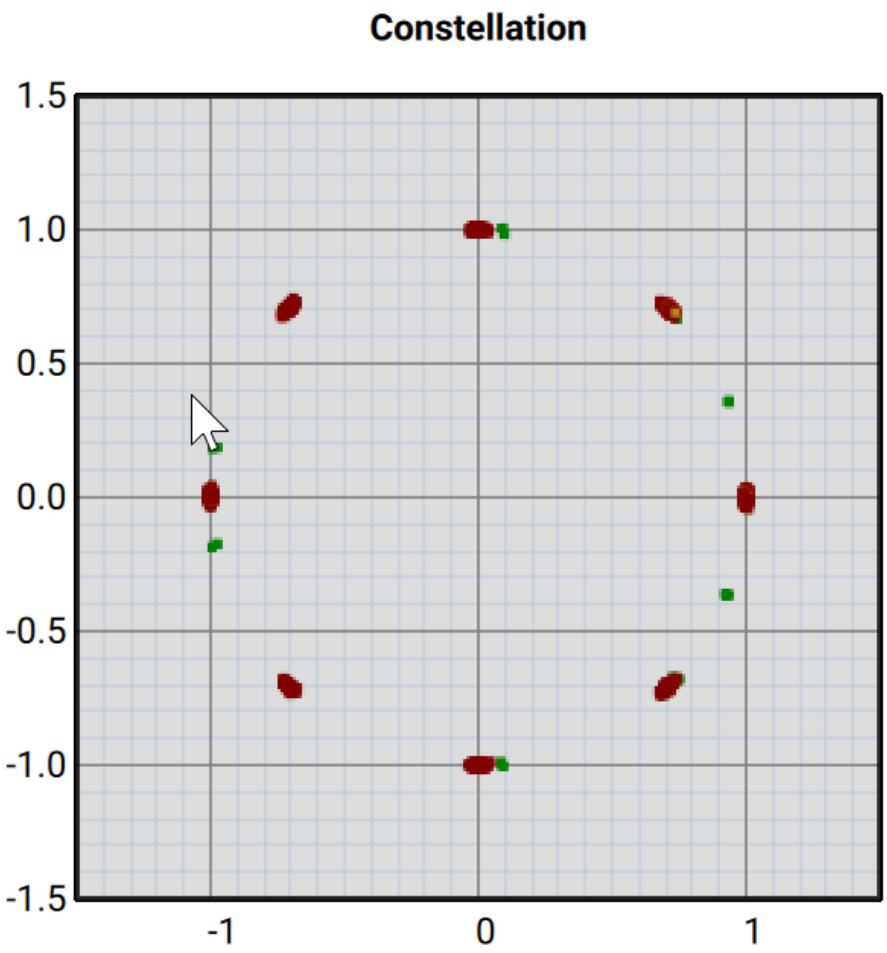
Once a packet has been captured and analysed, a constellation diagram is shown at the top of the summary panel on the right hand side.

The points in the constellation diagram are colour coded:

1. Green : Preamble symbols.
2. Blue : Control header symbols.
3. Red : Payload symbols, including PITS and trailing zeros.

The number of symbols displayed in the constellation diagram is dependent on the zooming of the time domain graph to its left. The constellation diagram will only ever display the symbols

which are encompassed by the span of the time domain graph. Hence zooming in the time domain graph will alter the symbols shown in the constellation diagram.



12.7 Measured quantities

The summary panel on the right hand side displays the quantities measured on each packet. These include EVM, drift, in-band emissions, power and symbol errors.

Note: the in-band emissions measurements are performed over a single packet, equivalent to a single sweep in the Bluetooth test specification. The Bluetooth test specification actually requires 10 sweeps to be considered, so the values reported are conservative to those which would be obtained in a scripted test.

From the API, it is possible to tell whether a valid packet has been captured and analysed by calling `RF_Rx_Have_Pkt (bool *havePkt)`.

UI Item	Description	API calls
$\Delta\omega_0$	Carrier frequency offset estimated from the preamble.	`RF_Omega0(double *omega)`
$\Delta\omega_0 + \Delta\omega_1$	Carrier frequency offset estimated from the first 1000 (if available) symbols of the payload.	`RF_Omega0_Plus_Omega1(double *omega)`
$\Delta\omega_1$	Estimated change in carrier frequency between preamble and first 1000 (if available) symbols of the payload.	`RF_Omega1(double *omega)`
$\Delta\omega_0 + \Delta\omega_n$	Carrier frequency offset estimated from the n^{th} block of 1000 (if available) symbols of the payload.	-
$\Delta\omega_n$	Estimated change in carrier frequency between preamble and n^{th} block of 1000 (if available) symbols of the payload.	-
Preamble EVM	EVM of preamble in dB.	`RF_EVM_Preamble(double *evm)`
Control header EVM	EVM of control header in dB.	`RF_EVM_Control_Header(double *evm)`
Block 1 EVM (*N*)	EVM of the first block of the payload which has been calculated over *N* symbols.	`RF_EVM_Control_Payload(double *evm)`
Block *M* EVM (*N*)	EVM of the M^{th} block of the payload which has been calculated over *N* symbols.	-
STS errors	Number of symbol errors in the short training sequence (STS).	`RF_Errors_STS(uint32_t *err)`
Guard interval errors	Number of symbol errors in the guard interval (GI).	`RF_Errors_GI(uint32_t *err)`
LTS errors	Number of symbol errors in the long training sequence (LTS).	`RF_Errors_LTS(uint32_t *err)`
Trailing zero errors	Number of symbol errors in the trailing zeros.	`RF_Errors_Zeros(uint32_t *err)`
PITS errors	Number of symbol errors in the phy interval training sequences (PITS)	`RF_Errors_PITS(uint32_t *err)`
P_{Tx} f_{TX} = ±3MHz	In-band emissions measured at ±3MHz relative to the carrier frequency.	`RF_ACP_3MHz(double *p)`
P_{Tx} f_{TX} = ±4MHz	In-band emissions measured at ±4MHz relative to the carrier frequency.	`RF_ACP_4MHz(double *p)`
P_{Tx} f_{TX} = ±5MHz	In-band emissions measured at ±5MHz relative to the carrier frequency.	`RF_ACP_5MHz(double *p)`
P_{Tx} f_{TX} = ±(6+n)MHz	In-band emissions measured at ±6MHz and greater relative to the carrier frequency.	`RF_ACP_6MHz(double *p)`
Number of exceptions	Number of in-band emission measurement exceptions.	`RF_ACP_Exceptions(uint32_t *n)`

UI Item	Description	API calls
Exceptions > -20dBm	Number of in-band emission measurement exceptions which exceeded -20dBm.	`RF_ACP_Invalid_Exceptions(uint32_t *n)`
P_{AVG}	Average power measured over the full packet in a 3MHz resolution bandwidth with a 3MHz video filter.	`RF_Pavg(double *p)`
P_{TOT}	Average power measured over the full packet in a 4MHz flat bandwidth.	-
P_{EIRP}	Effective isotropic radiated power P_{AVG} + TSPX_Antenna_Gain	-

12.8 Starting and stopping the signal analyser

Prior to starting the signal analyser it is essential to set a number of controls.

The input port should be selected using the control at the bottom right of the window. Clicking on the control will switch the input port between “Tx/Rx” and “Monitor In”. For conducted measurements the “Tx/Rx” port should be selected. For off-air measurements the “Monitor In” port should be selected and an aerial connected directly to the port. The corresponding API call is `RF_setSignal(signal_t sig)`, where `sig` is either `RF_MonitorIn` or `RF_TxRx`.

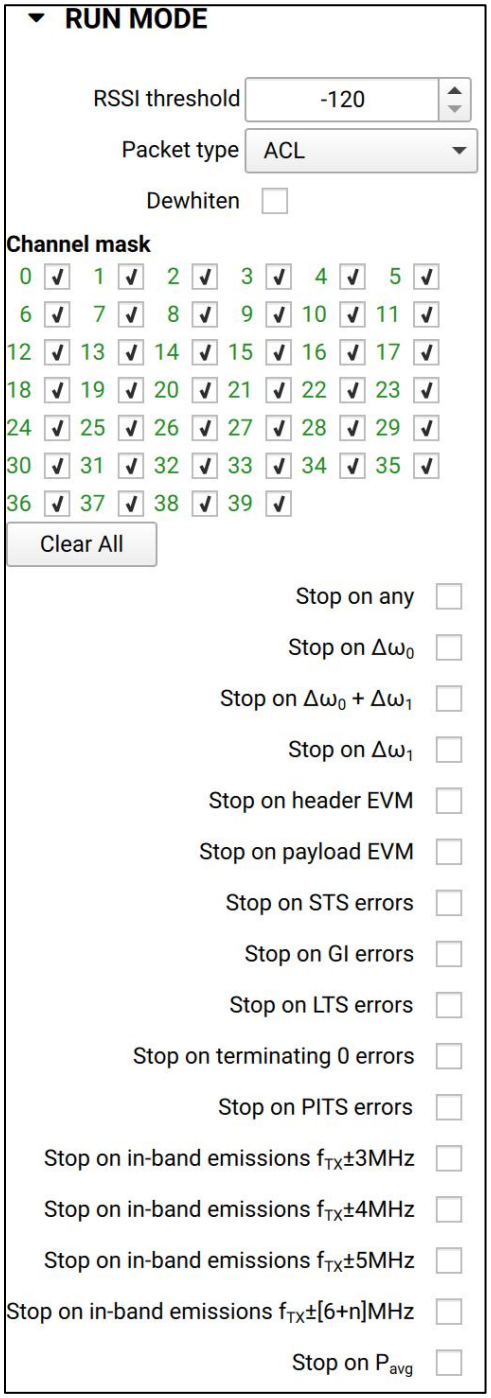
The moreph input attenuation should be set to avoid overloading the receiver front-end. The maximum input signal on the “Tx/Rx” port prior to the onset of saturation is around +7dBm. If the total power exceeds +7dBm, then 1dB of attenuation should be added for each dB the input power exceeds +7dBm. The corresponding threshold for the “Monitor In” port is around -30dBm. The input attenuation is set using the spin box at the bottom right of the window. The corresponding API call is `RF_setInpAttenuation(double atten)`.

The “Run mode” menu contains further controls which should be set prior to signal analysis.

The “RSSI threshold" sets the minimum RSSI the signal must possess if it is to be captured and analysed. This is most useful for limiting signal capture in off-air environments. However, it can also be useful if the transmitter has weak sidebands which resemble HDT signals. The corresponding API call is `RF_Rx_RSSI_threshold(int32_t rssi)`.

The “Packet type” primarily determines how the fields on the control header are displayed. However, it can also impact how the signal is dewhitened. Possible values for packet type are ACL, CIS, BIS or standard vector. For ACL, CIS and BIS it is assumed that the whitening seed is based on the physical channel index. For standard vectors the whitening seed is assumed to be 0x7A7F. The corresponding API call is `RF_Rx_Pkt_Type(const char *pkt)`, where `pkt` is one of the strings `“ACL”, “BIS”, “CIS” or "STANDARD VECTOR"`.

The ”Dewhiten” checkbox determines whether the received packet is expected to be whitened. It is essential that this control is set correctly or the payload headers of the received packet will not be interpreted correctly. The corresponding API call is `RF_Rx_dewhiten(bool dew)`.



It is possible to restrict the channels which are monitored for possible HDT packets to analyse. This can be useful in crowded environments when the channel used by the IUT is known. The

“Run mode” menu contains a checkbox for each LE channel. A button below the checkboxes permits switching between the selection or all or no channels. The channel mask can be set in the API by calling `RF_Rx_Channel_Mask(const char* mask)`. The argument `mask` is a character string containing a hexadecimal number with each representing the corresponding RF channel number. A set bit indicates that the channel is available for signal analysis.
The signal analyser can be started by clicking the “Run/Stop” button on the toolbar. Once clicked, the left hand side of the status bar will show “Trigger?” As soon as a packet is detected, this will change to “Triggered”. Alongside this label will be a progress bar indicating how much of the packet has been processed. Once the entire packet has been processed, the graphs and measurements will be updated and the “Run/Stop” button will revert to its original state ready for another capture. To automatically restart another capture, click the “Continuous” button in the toolbar rather than the “Run/Stop” button.

To run a single capture and analysis from the API, call:

```
RF_run()
```

or for repeated captures and analysis:

```
RF_continuous()
```

To abort a capture:

```
RF_abort()
```

and to wait for a capture and analysis with a timeout in milli-seconds:

```
RF_wait( uint32_t timeout_ms )
```

The presence of valid packet and measurements can be ascertained by calling:

```
RF_Rx_Have_Pkt ( bool havePkt )
```

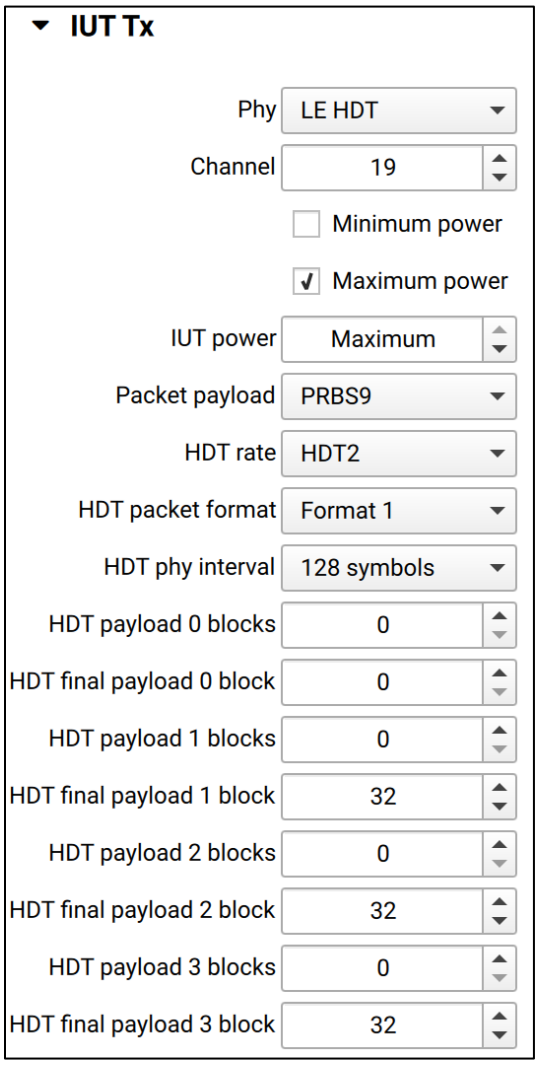
If the signal analyser is being run in continuous mode with repeated packets being analysed, then it possible to instruct the signal analyser to terminate as soon as a measured quantity exceeds the Bluetooth test limit for that quantity. This provides a convenient mechanism for capturing rogue packets. The measured quantities on which signal analyser can be made to automatically halt are:

UI Item	**Limit**	**API calls**
$\Delta\omega_0$	±125kHz	`RF_StopOn_Omega0( bool stop )`
$\Delta\omega_0 + \Delta\omega_1$	±125kHz	`RF_StopOn_Omega0_Omega1( bool stop )`
$\Delta\omega_1$	±125Hz	`RF_StopOn_Omega1( bool stop )`
Header EVM	-10dB	`RF_StopOn_EVM_Header( bool stop )`
Payload EVM	-10dB HDT2 -13dB HDT3 -16dB HDT4 -19dB HDT6 -22dB HDT7.5	`RF_StopOn_EVM_Payload( bool stop )`

UI Item	Limit	API calls
STS errors	0	RF_StopOn_STS(bool stop)
GI errors	0	RF_StopOn_GI(bool stop)
LTS errors	0	RF_StopOn_LTS(bool stop)
Terminating zero errors	0	RF_StopOn_Zeros(bool stop)
PITS errors	0	RF_StopOn_PITS(bool stop)
In-band emissions f_{TX} = ±3MHz	-10dBm	RF_StopOn_ACP_3MHz(bool stop)
In-band emissions f_{TX} = ±4MHz	-20dBm	RF_StopOn_ACP_4MHz(bool stop)
In-band emissions f_{TX} = ±5MHz	-20dBm	RF_StopOn_ACP_5MHz(bool stop)
In-band emissions ±(6+n)MHz	-30dBm	RF_StopOn_ACP_6MHz(bool stop)
P_{AVG}	-20dBm to +10dBm Class 2 -10dBm to +20dBm Class 1	RF_StopOn_Pavg(bool stop)

12.9 Controlling a IUT

When in signal analyser mode, it is possible to control the IUTs transmissions using the “IUT Tx” menu. The controls available within the “IUT Tx” menu will dependent on the features set in the “IUT Features” menu.



The IUT is made to transmit by selecting an option from the “Phy” drop down menu. Possible options are :

1. LE 1M
2. LE 2M
3. LE coded S=8
4. LE coded S=2
5. HDT

The corresponding API call is:

```
RF_IUT_Phy(const char * phy)
```

where `phy` must be one of `“OFF”`, `“1M”`, `“2M”`, `“S=8”`, `“S=2”` or `“HDT”`.

A spin box is provided for selecting the LE RF channel number on which the transmission will be made. The channel can be selected via the API by calling:

```
RF_IUT_Channel(uint32_t channel)
```

Two checkboxes can be used to set the IUT to either maximum output or minimum output power. Alternatively the output power can be set using a spin box. To set the output power via the API use:

```
RF_IUT_Power(int32_t pwr)
```

The value of `pwr` should be between -127 and +20dBm. If `pwr` is set to 0x7E then the IUT will be instructed to transmit at minimum power. A value of 0x7F will force the IUT to maximum power.

The contents of the packet payload transmitted by the IUT are determined by the “Packet payload” drop down menu. Options are:

1. PRBS9
2. 11110000
3. 10101010
4. PRBS15
5. 11111111
6. 00000000
7. 00001111
8. 01010101

The corresponding API call is:

```
RF_IUT_Payload(const char *payload)
```

The contents of the argument `payload` should one of the strings `"PRBS9"`, `"11110000"`, `"10101010"`, `"PRBS15"`, `"11111111"`, `"00000000"`, `"00001111"` or `"01010101"`.

For LE 1M, LE 2M, LE coded S=8 and LE coded S=2 transmissions, a payload length control will also be displayed. This sets the length of the packet payload in units of octets. The corresponding API call is:

```
RF_IUT_Payload_Len(uint32_t octets)
```

For HDT packets the following parameters may need to be set:

UI Item	Description	Applicable	API calls

HDT rate	Selects the HDT rate from HDT2, HDT3, HDT4, HDT6 and HDT7.5. It also enables Short Format packet to be selected.	All	`RF_IUT_RI(uint32_t RI)`
			`RF_IUT_PFI(uint32_t PFI)`
			`Set both rate indicator (RI) and packet format indicator (PFI) to zero for short format packets.`

HDT packet format	Selects between format 0 and format 1 packets.	All	`RF_IUT_PFI(uint32_t PFI)`
HDT PDU payload only	The number octets in a format 0 payload, excluding payload header. Can be in the range 0 to 510 (or 508 if a CTE is present).	Format 0	`RF_IUT_Payload_Only(uint32_t octets)`
HDT phy interval	Phy interval to be used for format 1 packets	Format 1	`RF_IUT_PhyInterval (uint32_t phyInt)`

HDT payload 0 blocks
HDT payload 1 blocks
HDT payload 2 blocks
HDT payload 3 blocks
	The number of blocks minus 1 transmitted in each payload. Must be in range 0 to 15.	Format 1	`RF_IUT_Nb_0(uint32_t Nb)`
			`RF_IUT_Nb_1(uint32_t Nb)`
			`RF_IUT_Nb_2(uint32_t Nb)`
			`RF_IUT_Nb_3(uint32_t Nb)`

HDT payload 0 block size
HDT payload 1 block size
HDT payload 2 block size
HDT payload 3 block size
	The block size to be used for each payload. Must be in range 32 to 511.	Format 1
		Nb > 0
			`RF_IUT_BlockSize_0(uint32_t octets)`
			`RF_IUT_BlockSize_1(uint32_t octets)`
			`RF_IUT_BlockSize_2(uint32_t octets)`
			`RF_IUT_BlockSize_3(uint32_t octets)`

HDT final payload 0 block
HDT final payload 1 block
HDT final payload 2 block
HDT final payload 3 block
	The size of the last block to be used for each payload. If this is the only block in the payload (Nb = 0) then it must be in the range 0 to 511, otherwise in the range 32 to 543.	Format 1	`RF_IUT_LastSize_0(uint32_t octets)`
			`RF_IUT_LastSize_1(uint32_t octets)`
			`RF_IUT_LastSize_2(uint32_t octets)`
			`RF_IUT_LastSize_3(uint32_t octets)`

If the IUT is required to send a CTE, then the CTE length must be set in the range 2µs to 80µs. A value less than 2µs will inhibit the generation of a CTE. This can be accomplished via the API by calling:

```
RF_IUT_CTE_len(uint32_t duration_micro_seconds)
```

The type of the CTE can be chosen to be AoA, AoD with 1µs slots or AoD with 2µs slots (if supported by the IUT). To set the CTE type via the API call:

```
RF_IUT_CTE_type(uint32_t typ)
```

where:

1. `typ` = 0 corresponds to AoA
2. `typ` = 1 corresponds to AoD with 1µs slots
3. `typ` = 2 corresponds to AoD with 2µs slots

If an AoD CTE is required, then it will also be necessary to set the switching pattern length and the antenna IDs. The switch pattern length is set using the "Pattern length" spin box and there are additional spin boxes available to select the antenna IDs for each slot in the switching pattern.

The switching pattern length and the associated antenna IDs are set using a single call to the API:

```
RF_IUT_CTE_Antenna_IDs(uint32_t spl, uint32_t *IDs)
```

Where `spl` is the switching pattern length and `IDs` is an array of length `spl` containing the antenna IDs.

It is possible to determine whether the IUT is believed to be transmitting by calling:

```
RF_IUT_isTx(bool *isTxing)
```

12.10 Saving and restoring settings

The current signal analyser settings can be saved by clicking the "Save" icon on the toolbar. Previously save settings can be reloaded using the "Open" icon on the toolbar.

The corresponding API calls are:

```
RF_saveSigAna(const char *filename)
```

and:

```
RF_loadSigAna(const char *filename)
```

12.11 Exporting data

IQ data can be exported to a binary file using the "IQ" icon on the toolbar. This icon is only visible in signal analyser once a packet has been successfully captured and analysed. The data is save as pairs of real*4 samples with the I sample first and then the Q sample. The data is 64x oversampled. The corresponding API call is:

```
RF_saveIQ(const char *filename)
```

The data shown in the constellation diagram (which is the same as that shown in the time-domain constellation plot) can be save by clicking on the “Constellation” icon in the toolbar. This icon is only visible in signal analyser once a packet has been successfully captured and analysed. The data is save as pairs of real*4 samples with the I sample first and then the Q sample. The corresponding API call is:

```
RF_saveConstellation (const char *filename)
```

The constellation data can also be accessed directly in the API by calling:

```
RF_Constellation(float **x, uint32_t *n)
```

where `x` points to the constellation data and `n` is the number of constellation points.

The in-band emissions data can be saved to a csv file by clicking the “Spectrum” icon on the toolbar. This icon is only visible in signal analyser once a packet has been successfully captured and analysed. The columns in the csv file contain:

Column 1: Frequency of 100kHz measurements in MHz.
Column 2: Power in 100kHz measurements in dBm.
Column 3: Frequency of 1MHz measurements in MHz.
Column 4: Power of 1MHz measurements in in dBm.

The corresponding API call is:

```
RF_saveInBandEmissions(const char *filename)
```

The in-band emissions data can also be accessed directly in the API by:

```
RF_ACP_100kHz(double **x)
```

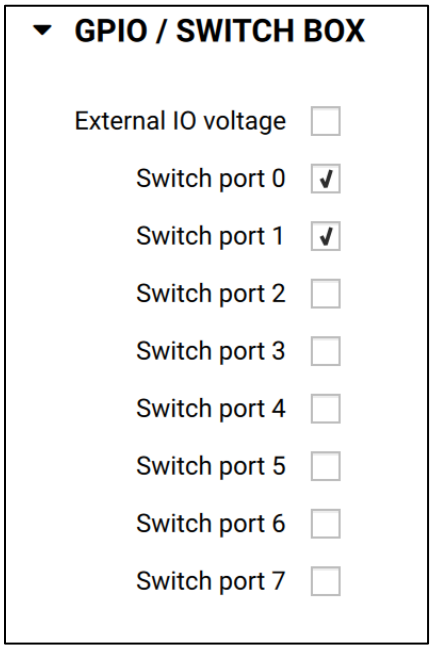
and:

```
RF_ACP_1MHz(double **x)
```

where `x` points to the in-band emissions data. There will be 810 values of 100kHz resolution data, the first value being at 2400.55MHz. There will be 81 values of 1MHz resolution data, the first value being at 2401MHz.

13 GPIO / Switch box

The “GPIO/Switch“ menu controls the operation of the external 8-way RF switch box and the external logic pod, if attached.



The external 8-way RF switch box controls which combination of IUT antennas are combined and connected to the moreph Tx/Rx port. Unused IUT antennas are terminated in 50Ω.

If the IUT attached to the external 8-way RF switch box has more than 1 antenna, then “Switch port N” refers to the Nth antenna port on the IUT.

The logic pod can be used to sense digital signals on the IUT. These can then be displayed on the upper graph in the signal analyser. Up to 7 logic lines can be sensed. These are located on GPIO 1 to 7.

If the logic pod is attached, then the moreph will also output digital timing signals. GPIO 3 is used to indicate when a CW transmission is active. GPIOs 4 to 7 are used to indicate when HDT signals are active, GPIO 4 corresponding to the lowest numbered HDT signal and GPIO 7 to the highest. The labels on the right hand side of the time-domain plot can be edited by overtyping the existing text.

The IO voltage for the logic signals can either be 3v3 supplied by the moreph or an externally applied IO voltage. If the external 8-way switch box is attached then the IO voltage must be close to 3v3 since this is also used to power the circuitry within the switch box.

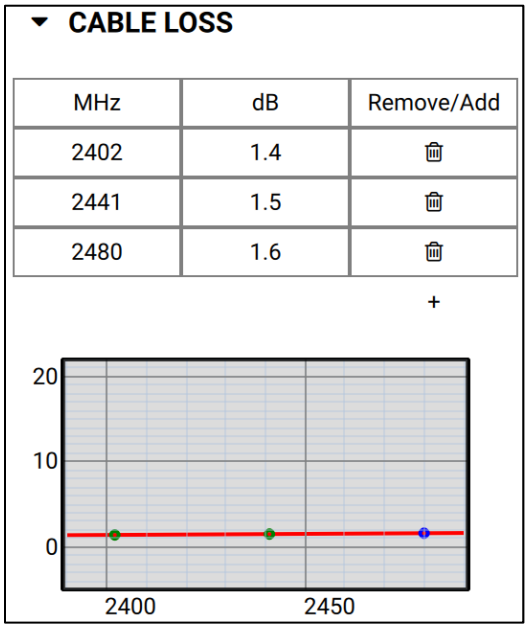
The moreph will automatically attempt to detect the presence of the switch box when the application is started. The status bar indicates whether a switch box has or has not been

discovered. Toggling the source of the IO voltage will also force the moreph to search for an attached switch box.

UI Item	Description	API call
External IO voltage	Determines whether an external IO voltage is used by the GPIO connector or the 3v3 generated by the moreph is used.	`RF_extIOvolts(bool extIO`
Switch port 0 Switch port 1 Switch port 2 Switch port 3 Switch port 4 Switch port 5 Switch port 6 Switch port 7	Determines the state of switch port. Unused switch ports are terminated in 50Ω.	`RF_setSwitchPort(uint32_t port, bool on)`

14 Cable loss

It is essential that the information in the “Cable” menu is entered accurately prior to making measurements. This applies to all modes of operation.



The cable loss is used by the signal generator to adjust its output power level so that the power specified in the UI is that received at the IUT antenna port, not the power at the moreph SMA connector.

The cable loss is used by the signal analyser to adjust its measurements so that they reflect the power at the IUT antenna port, not the power at the moreph SMA connector. This impacts both power and spectral measurements.

The out-of-band CW blocker uses the value of the cable loss at the centre of the 2.4GHz ISM band. There is no frequency dependent cable loss compensation for the out-of-band blocker power.

The cable loss can be specified at a number of different frequencies with the 2.4GHz ISM band. A table and graph show the current cable loss configuration. Additional rows can be added to the table by clicking the “+” symbol at the bottom of the “Remove/Add” column. Rows can be deleted by clicking on the trash can icons. The table values can be made editable by clicking on them.

The API can be used to set the cable loss by calling:

```
RF_CableLoss(uint32_t *frequency, double *loss_dB, uint32_t nEntries)
```

where:

`frequency` is an array of uint32_t of length `nEntries` which contains the frequencies at which the losses will are specified in units of MHz,

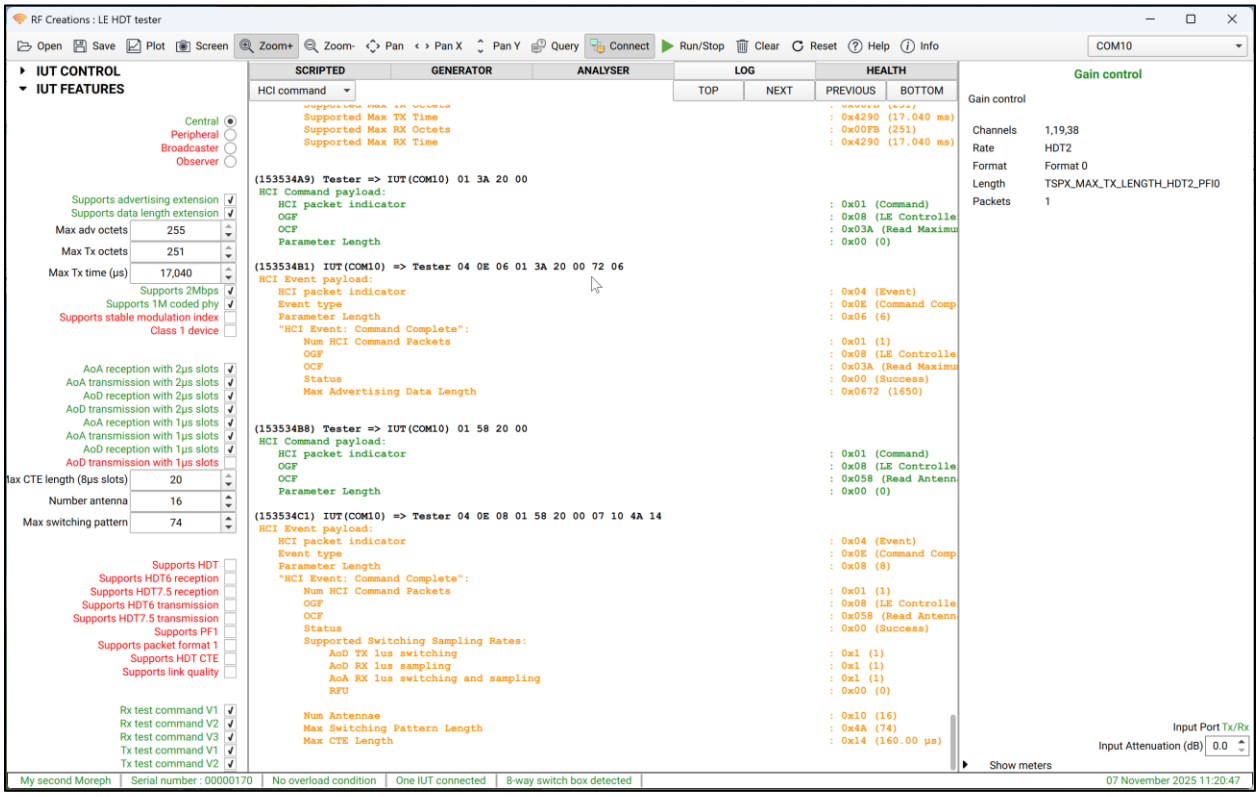
`loss` is an array of double of length `nEntries` which contains the cable losses at the specified frequencies in units of dB.

15 Log window

15.1 Overview

The log window is used to display:

1. Raw octets transferred over the HCI interface with the IUT.
2. Decoded HCI messages transferred between the moreph and the IUT.
3. Informational messages from the moreph.
4. Errors detected by the moreph.



15.2 Raw octets

The raw octets transferred over the HCI interface will only be include in the log window if “Log raw HCI octets” has been checked under the “IUT control” menu.

Raw octets are displayed in the log window in black.

At the start of each line in parentheses is a timestamp in units of milliseconds.

If the octets are being transferred from the moreph to the IUT, the line will commence:

```
Tester => IUT(comport)
```

If the octets are being transferred from the IUT to the moreph, the line will commence:

```
IUT(comport) => Tester
```

The remainder of the line contains the octets transferred in hex.

15.3 HCI decode

An HCI decode will always be entered into the log window. This will normally be displayed on multiple lines. To display on a single line check “Single line HCI decode” in the IUT control menu.

HCI messages from the moreph to the IUT are displayed in green. Those from the IUT to the moreph are displayed in orange.

15.4 Informational messages

Informational messages are entered into the log window in black. Examples of informational messages are those generated when a IUT is connected or disconnected from the moreph.

15.5 Error messages

Error message are entered into the log window in red.

15.6 Test script messages

Test script messages are entered into the log window in blue. These messages are entered every time a new test or phase of a test. They may contain:

a. The test name
b. The channel on which the test will be performed
c. Phy interval in used for transmitter tests
d. Interferer frequency for receiver tests

The messages appear immediately before the first corresponding HCI test command. They therefore provide a convenient method of navigating through the log file to find the HCI traffic associated with a particular test.

15.7 Navigating within the log window

It is possible to navigate within the log window using the menu and push buttons at the top of the window.

The “Top” button will navigate to the start of the log. The “Bottom” button will navigate to the last entry in the log.

The “Previous” and “Next” buttons will navigate to the previous or next entity described by the drop down menu at the top left of the window. Possible entities to navigate to are:

1. HCI command
2. HCI test command
3. HCI event
4. HCI sub-event report
5. Scripted test
6. Error
7. Information

15.8 Exporting log window contents

The contents of the log window can be exported in html format by clicking the “Save” icon on the toolbar whilst the log window is visible. From the API call `RF_saveLog( filename)`.

The log window can be cleared by clicking the “Discard” button on the toolbar. From the API call `RF_clearLog()`.

16 Calibration

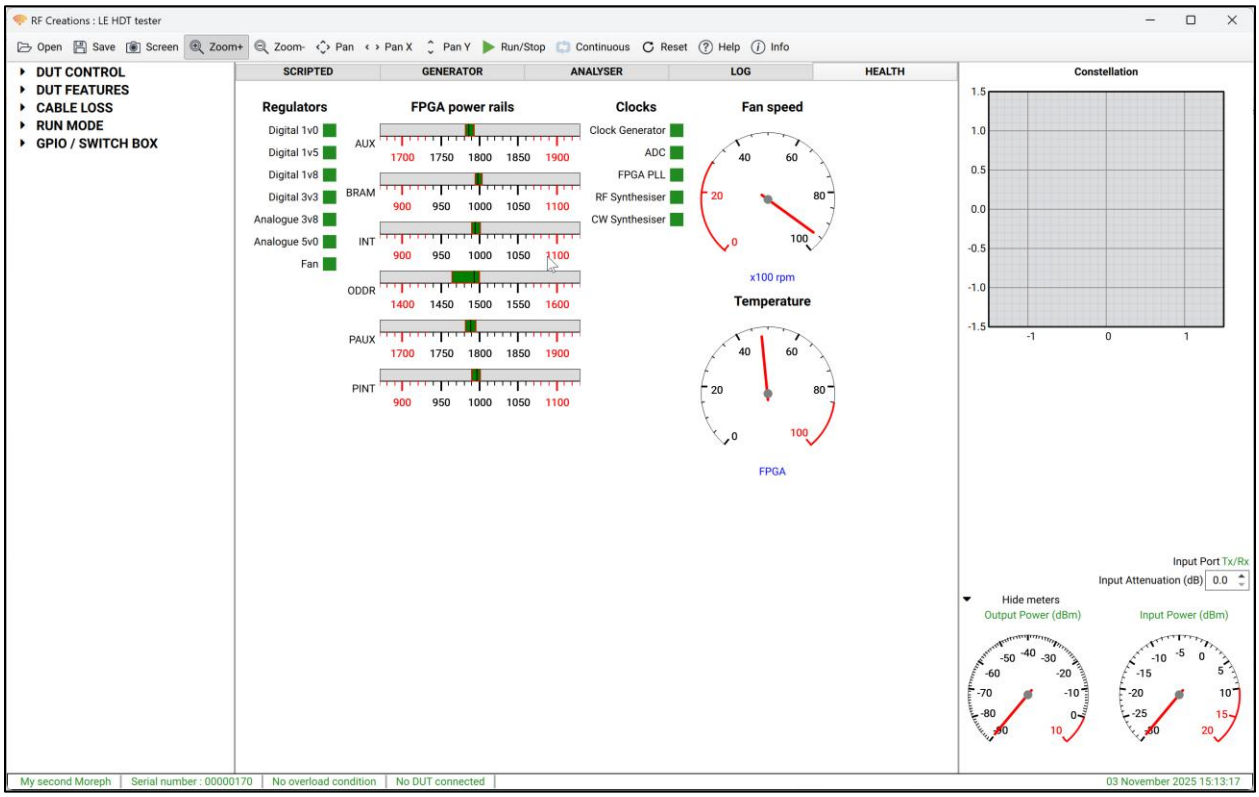
The moreph is calibrated at factory. There are no user defined routines for calibrating the moreph within the HDT application. The moreph can be returned to the manufacturer at any time for re-calibration. RFcreations recommend a re-calibration interval of 1 year.

17 Health

The health tab displays information on:

1. Power rails
2. Clocks
3. FPGA die temperature
4. Fan speed

The moreph will automatically shut down if any of these parameters remained outside their specified operating regions for a prolonged period.



18 C API

18.1 Overview

The C API provides access to most features of the UI. It currently does not support the generation of test scripts, but it can load and run test scripts created in the UI.

The API is reliant on an API server. It is the user's responsibility to start and terminate the API server appropriately. This differs from earlier versions of the API where the API server was launched and terminated automatically.

The C API permits the UI to be either visible or hidden.

The header file RFapi.h must be included in any user code. This file also includes RFresults.h. The executable must have access to the dynamic libraries contained in the install directory. Note that Cinnabar.rfcplugin is a dynamic library.

An example program named RF_example.cpp is available in the install directory.

18.2 API server

The API is reliant on the API server running. The API server must be started prior to any API calls.

The API server executable is named RFapi_server.exe and resides in the install directory, which by default is C:/Program Files/RFcreations/Phy.

The C example code does not include statements for launching the API server since these may be OS dependent. It is therefore essential that the user either adds code to launch the API server or starts it manually before running the example code.

The API server will be terminated by any call to `RF_disconnect()`. If the user wishes to make further calls to the API after calling `RF_disconnect()`, then it is essential that they restart the API server first.

18.3 Error handling

Most API calls return a `RF_error_t` which are listed in api.h.

If the call returns `RF_ERROR_UNKNOWN` then further information on the error may be contained in the character string returned by `RF_error_msg()`.

18.4 Connecting to a moreph

The moreph devices connected to the host computer can be found using `RF_search()`. These devices can then be scanned to find the one of interest, for example, by looking for a specific serial number. The selected device can then be connected to using `RF_connect()`.

```
// Connect to Moreph
// -----------------

int serial_number = 0x150;
bool connected = false;
RF_search_result_t* results;
std::cout <<  "Searching for Moreph..." << std::endl;
while (!connected) {
    int N;
    RF_search(&results, &N);
    for (int i = 0; i < N; i++) {
        std::cout << "Found Moreph " << std::hex << std::uppercase <<
                        results[i].serial_number << std::dec << std::endl;
        if(results[i].serial_number == serial_number) {
            std::cout << "Connecting to Moreph " << std::hex << std::uppercase <<
                        serial_number << std::dec << std::endl;
            RF_connect(results[i].handle);
            connected = true;
            break;
        }
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
}
```

Alternatively, it is possible to connect directly to a moreph with a given serial number:

```
RF_connect_serial(uint32_t serial_number);
```

18.5 Making the UI visible

It is possible to make the UI visible whilst using the API:

```
// Make UI visible
// ---------------

RF_show_gui();
std::cout << "UI launched" << std::endl;
```

18.6 Connecting to a IUT

Prior to connecting to the IUT it is necessary to configure the UART. Once this has been done a `RF_connectIUT()` call can be made to connect to the IUT.

```
// Connect to IUT
// --------------

RF_setBaud( 115200 );
RF_setHWflow( RF_HW_RTS_CTS );
RF_setSWflow( RF_SW_none );
RF_setStopBits( 1 );
RF_setParity(RF_PARITY_none);
std::cout << "UART configured" << std::endl;

RF_error_t err;
err = RF_connectIUT("com10");
if (err != RF_NO_ERROR) {
    std::cout << "Couldn't connect to IUT" << std::endl;
    return 1;
}
std::cout << "Connected to IUT" << std::endl;
```

Having successfully connected to the IUT it is necessary to inform the application of the features which the IUT supports. This can either be done by individual commands (see section Setting supported features) or by enquiring from the IUT over HCI which features it supports:

```
// Find IUT supported features
// ---------------------------

err = RF_queryIUT();
if (err != RF_NO_ERROR) {
    std::cout << "Couldn't query IUT supported features " << uint32_t(err) <<
                 RF_error_msg() << std::endl;
    return 1;
}
std::cout << "Queried IUT supported features" << std::endl;
```

18.7 Set cable loss

The HDT application needs to be able to compensate for the cable loss between the mopreh and the IUT. This is done by specifying the cable loss at a number of points within the 2.4GHz ISM band.

```
// Set cable parameters
// --------------------

uint32_t Freqs[] = {2402, 2440, 2480};
double Losses[] = {1.1, 1.2, 1.4};
RF_CableLoss(Freqs, Losses);

std::cout << "Cable loss configured" << std::endl;
```

18.8 Setting operating mode

The operating is set by:

```
// Set operating mode and reset
// ----------------------------

RF_setOperatingMode(RF_scripted);
std::cout << "Switched to scripted mode" << std::endl;
```

Prior to performing any further operations the log window can be cleared and all parameters set to their default values:

```
RF_clearLog();
RF_resetRF();
```

18.9 Loading a test script

To load the test script it is necessary to switch to the scripted mode of operating. Test scripts can be generated using the UI. These can then be loaded into the channel sounding application using the API.

```
// Loading a test script
// ---------------------

RF_setOperatingMode(RF_scripted);
std::cout << "Switched to scripted mode" << std::endl;

if( RF_loadTestScript( "My_test_script.tsf") != RF_NO_ERROR ) {
    std::cout << "Failed to load test script" << std::endl;
    return 1;
}
std::cout << "Test script loaded" << std::endl;
```

18.10 Creating a test script

The current test script can be cleared by calling `RF_clearTestScript()`.

New tests can be appended to the current test script by calling `RF_addTest()`. This function takes the following arguments:

const char *	testName	Name of the test to add to the test script. Must be match one of the test names available in the UI. For example, “RFPHY/TRM/HDT/BV-31”.
const char *	channels	The channels on which the test must be performed. Only applicable to those tests which are run on a single channel, ie primarily the RF-PHY tests. The channels are represented as common separated channel numbers with no spaces, eg "2,38,76”. Specifying an empty string, ie “”, will lead to the default channels for the test being used.
uint32_t	nSamples	The number of samples used in the test. Specifying a value of 0 will result in the default number of samples being used for the test.

```
// Clear the test script and add a new test
// ----------------------------------------

RF_setOperatingMode(RF_scripted);
std::cout << "Switched to scripted mode" << std::endl;

RF_clearTestScript();
RF_addTest("RFPHY/TRM/HDT/BV-31", "2,19,38", 10);
```

18.11 Running a test script

Running the test script is straightforward:

```
// Run the test script
// -------------------

RF_run();
err = RF_wait( 2 * 60 * 1000 );
if( err == RF_ERROR_TIMEOUT ) {
    std::cout << "Test script timed out" << std::endl;
    return 1;
} else if( err != RF_NO_ERROR ) {
    std::cout << "Error waiting for test script" << std::endl;
    return 1;
}
std::cout << "Ran test script" << std::endl;
```

18.12 Reading the results of a test script

The results of running a test script are contained in a single table. This table is the concatenation of all the tables shown in the UI.

It is possible to use the test name to locate the row in the table at which that test commenced. The example code then loops through each of the subsequent rows until and start of a new test is encountered. For each row the code reads back the strings in each column of the table.

```
// Get the results for the first test
// ----------------------------------

int32_t row = RF_findTest("RFPHY/TRM/HDT/BV-31", 0);
if (row < 0) {
    std::cout << "Test results not found" << std::endl;
    return 1;
}

while (1) {
    char name[64];
    if (RF_getTestName(row, name, 64) != RF_NO_ERROR)
        break;
    if (strcmp(name, "RFPHY/TRM/HDT/BV-31"))
        break;

    char quantity[64];
    RF_getTestQuantity(row, quantity, 64);

    char val[64];
    RF_getTestValue(row, val, 64);

    char res[64];
    RF_getTestResult(row, res, 64);

    std::cout << "Quantity " << quantity << " Value" << val << " Result " << res;
    row++;
}
```

The results of the test script can also be saved to an html file:

```
// Print the test script results
// -----------------------------

err = RF_saveResults( "TestResults.html" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write test results to file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "Test results written to file" << std::endl;
}
```

18.13 Saving a test script file

The current test script can be saved to a file:

```
// Save the test script
// --------------------

err = RF_saveTestScript( "TestScript.tsf" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write test script file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "Test script written to file" << std::endl;
}
```

18.14 Building an HDT signal

To add a signal to the signal generator it is first necessary to specify which signal out of the possible 5 is to be programmed. A further call is then required to label this signal as an HDT signal. Once this has been done, all the parameters for the HDT signal can be programmed. Depending on the packet format, not all parameters need to be specified. This example is for a format 1 packet.

Setting the current signal and its type:

```
// Build an HDT signal
// -------------------

RF_setCurrentSignal(0);
RF_setSignal(RF_HDT);
```

Common parameters for the signal:

```
RF_setHDT_Frequency(2432);
RF_setHDT_Power(-40);
RF_setHDT_Continuous(False);
RF_setHDT_StartTime(100);
RF_setHDT_Interval(1000);
RF_setHDT_Repeats(5000);
RF_setHDT_Whiten(True);
```

Parameters to define the HDT packet:

```
RF_setHDT_Type(0);
RF_setHDT_ZadoffChu(3);
RF_setHDT_PCA("9876543210");
RF_setHDT_NESN(0);
RF_setHDT_PFI(1);
RF_setHDT_RI(4);
RF_setHDT_Phy_Interval(1);
```

```
    RF_setHDT_Pkt_Type("ACL");
    RF_setHDT_Seed(0x555555);
    RF_setHDT_Hdr_XHP(False);
    RF_setHDT_Hdr_RXPP(True);
    RF_setHDT_Hdr_MD(True);
    RF_setHDT_Hdr_SN(0);
    RF_setHDT_TXPP(True);
    RF_setHDT_RxAck(0x03);
    RF_setHDT_RxBitMap_0(0x01FF);
    RF_setHDT_TxDscType(0x0C);
    RF_setHDT_TxPayload_Len_1(4096);
    RF_setHDT_TxLLID_1(2);
    RF_setHDT_TxNb_1(3);
    RF_setHDT_TxBlock_Size_1(511);
    RF_setHDT_TxBitMap_1(0x01E0);
    RF_setHDT_TxLast_1(8);
    RF_setHDT_Payload("PRBS9");
```

If RF distortions are required, then distortion table must be programmed and distortions enabled:

```
    // Build distortion table
    // ----------------------

    double Fcarrier[] = {-10, 10, 100};
    double DriftMag[] = {0, 200, 10};
    double DriftPeriod[] = {1000, 1000, 3140};
    uint32_t DriftDirn[] = {0, 0, 1};
    double PPM[] = {50, 50, -50};
    double Squish[] = {1, 1, 1};
    double Rotate[] = {0, 0, 0};
    uint32_t Npkts[] = {50, 20, 30};
    RF_setHDT_Distortions(Fcarrier, DriftMag, DriftPeriod, DriftDirn, PPM, Squish,
Rotate, Npkts);
    RF_setHDT_Distort(True);
```

18.15 Building a CW signal

To add a signal to the signal generator it is first necessary to specify which signal out of the possible 5 is to be programmed. A further call is then required to label this signal as an CW signal. Once this has been done, all the parameters for the CW signal can be programmed. This example is for a pulsed CW signal which repeats indefinitely.

```
    // Build in-band CW signal
    // -----------------------

    RF_setCurrentSignal(1);
    RF_setSignal(RF_CW);

    RF_setCW_Frequency(2467);
    RF_setCW_Power(-30);
    RF_setCW_Continuous(False);
    RF_setCW_StartTime(100);
    RF_setCW_Duration(100);
```

```
RF_setCW_Interval(200);

RF_setCW_Repeats(0);
```

18.16 Adding AWGN

To add a signal to the signal generator it is first necessary to specify which signal out of the possible 5 is to be programmed. A further call is then required to label this signal as an AWGN signal. Once this has been done, all the parameters for the AWGN signal can be programmed. This example is for a constant AWGN signal to be transmitted continuously.

```
// Build AWGN signal
// -----------------

RF_setCurrentSignal(2);
RF_setSignal("RF_AWGN");

RF_setAWGN_Power(-80);

RF_setCW_Continuous(True);
```

18.17 Running the signal generator

If the signal generator contain a packetised or pulsed signals with a finite number of repeats, then it will stop automatically when all the repeats have been transmitted and the call to `RF_wait()` will return. If this is not the case, the signal generator will need to be terminated by a call to `RF_abort()`.

```
// Start signal generator
// ----------------------

RF_run();
std::cout << "Signal generator started" << std::endl;

// Wait for HDT packets to be transmitted
// --------------------------------------

RF_wait(5000);
std::cout << "Signal generator finished" << std::endl;
```

18.18 Configuring the signal analyser

Prior to running the signal analyser, it is necessary to specify which input port to use, how much front-end attenuation should be applied, which packet type is expected and whether it should be dewhitened. An RSSI threshold and channel mask may also optionally be set.

```
// Configure signal analyser
// -------------------------

RF_setInpAttenuation(0);
RF_setRxPort(RF_TxRx);
RF_Rx_Pkt_Type("ACL");
RF_Rx_RSSI_threshold(-50);
RF_Rx_dewhiten(True);
RF_Rx_Channel_Mask("FFFFFFFFFF");
```

If the signal analyser is to be run in continuous mode, then it may be appropriate to also specify criteria which would cause the signal analyser to automatically terminate. For example:

```
// Configure signal analyser stop criteria
// ---------------------------------------

RF_stopOn_Omega0(true);
```

18.19 Running the signal analyser

To analyse just a single packet:

```
// Start signal analyser
// ---------------------

RF_run();
std::cout << "Signal analyser started" << std::endl;

// Wait for HDT packet to be received
// ----------------------------------

if (RF_wait(10000) == RF_ERROR_TIMEOUT) {
    std::cout << "No HDT packet received" << std::endl;
    return 1;
}
std::cout << "Signal analysis completed " << std::endl;
```

To analyse continuously call `RF_continuous()`. If the signal analyser is not set to stop on a limit failure then it will be necessary to stop is by calling `RF_abort()`.

18.20 Reading back measurements

Once the signal analysis has completed, the results of the measurements can be read back. For example a subset of the possible results being read back:

```
bool havePkt;
RF_Rx_Have_Pkt(&havePkt);
if (havePkt) {
    uint32_t chan;
    RF_Rx_Channel(&chan);
    std::cout << "Channel : " << chan << std::endl;
    int32_t rssi;
    RF_Rx_RSSI(&rssi);
    std::cout << "RSSI    : " << rssi << std::endl;
    uint32_t ri;
    RF_Rx_RI(&ri);
    std::cout << "Rate    : " << ri << std::endl;
    bool pfi;
    RF_Rx_PFI(&pfi);
    std::cout << "PFI     : " << pfi << std::endl;
    double omega0;
    RF_Omega0(&omega0);
    std::cout << "Omega 0 : " << omega0 << std::endl;
    double ebm;
    RF_EVM_Control_Header(&evm);
    std::cout << "EVM Hdr : " << evm << std::endl;
}
```

The entire list of measurements can be saved to an html file:

```
// Save the test script
// --------------------

err = RF_saveSummary( "Summary.html" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write summary file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "Summary written to file" << std::endl;
}
```

18.21 Accessing in-band emissions data

A pointer to the in-band emissions data can be obtained as follows:

```
// Get pointer to in-band emissions data
// -------------------------------------

double* acp1MHz;
RF_ACP_1MHz(&acp1MHz);

double* acp100kHz;
RF_ACP_1MHz(&acp100kHz);
```

The first call returns the data for 1MHz bins. There are 81 bins, the first bin being centred on 2401MHz.

The second call returns the for 100kHz bins, from which the 1MHz bins are generated by summation of powers. There are 810 bins, the first bin being centred on 2400.55MHz.

The in-band emissions data can also be saved to a csv file:

```
// Save in-band emissions data
// ---------------------------

err = RF_RF_saveInBandEmissions ( "Emissions.csv" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write in-band emissions file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "In-band emissions written to file" << std::endl;
}
```

18.22 Accessing constellation data

The constellation data can be accessed via:

```
// Get pointer to constellation data
// ---------------------------------

double* constellation;
uint32_t nPoints;

RF_Constellation(&constellation, &nPoints);
```

The constellation data consists of IQ pairs representing each constellation point. The value returned in `nPoints` is the total number of elements in the array constellation. The actually number of IQ pairs will be half this value.

The constellation data can also be saved to a file. The file is a binary file contained real*4 elements.

```
// Save constellation data
// -----------------------

err = RF_RF_saveConstellation ( "Constellation.dat" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write constellation file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "Constellation data written to file" << std::endl;
}
```

18.23 Saving IQ data

The IQ data can be saved to a binary file. The data is written to the file as real*4. The data is written at IQ pairs with an oversampling ratio of 64.

```
// Save IQ data
// ------------

err = RF_RF_saveIQ ( "IQ.dat" );
if( err != RF_NO_ERROR ) {
    std::cout << "Failed to write IQ file" << std::endl;
    if (err == RF_ERROR_UNKNOWN)
        std::cout << RF_error_msg() << std::endl;
} else {
    std::cout << "IQ data written to file" << std::endl;
}
```

18.24 Disconnecting

To disconnect the IUT:

```
// Disconnect IUT
// --------------

RF_disconnectIUT();
std::cout << "IUT disconnected" << std::endl;
```

To close the UI:

```
// Close the GUI
// -------------

RF_hide_gui();
std::cout << "GUI hidden" << std::endl;
```

To disconnect the moreph and terminate the API server:

```
// Disconnecting the moreph and terminating the API server
// ------------------------------------------------------

RF_disconnect();
std::cout << "Disconnecting" << std::endl;
```

If the program terminates without hiding the UI and not disconnecting from the API server, then the UI will remain visible and active. This provides a mechanism of running a problematic test via a script and then switching to an interactive mode to examine the data.

18.25 Resetting moreph hardware

The moreph hardware can be reset by first performing a `RF_search()`, selecting which device to reset from the search results and then executing a `RF_hardwareReset(idx)`, using the index of the selected device. An example is provided in RF_example_reset.py contained in the installation folder.

19 Python API

19.1 Overview

Function calls in the Python API are identical to those for the C API with the following exception:

Enums are replaced by character strings. For example, the C enum `RF_scripted` simply becomes the string `"RF_scripted"`.

The python library is held in RFapi.py in the install directory.

An example Python program named RF_example.py is provided in the installation directory.

19.2 API server

The API server can be started from a Python script as follows:

```
import subprocess

# Start the API server
# --------------------

subprocess.Popen(["C:/Program Files/RFcreations/Phy/RFapi_server"])
```

The API server will be closed by any call to `RF_disconnect()`, and must be restarted before any further API calls can be made.

19.3 Reseting moreph hardware

An example Python program for resetting the moreph hardware can be found in RF_example_reset.py.

20 Standalone mode

The HDT application can be run without a moreph being attached. In this standalone mode the UI can be used to build and save test scripts.

Standalone mode can be entered using the following Python script:

```
import subprocess

import sys

sys.path.append("C:/Program Files/RFcreations/Phy")

from RFapi import *

subprocess.Popen(["C:/Program Files/RFcreations/Phy/RFapi_server"])

RF_standalone()

RF_show_gui()
```

This script is contained in the installation folder with the name RF_standalone.py.

21 Supported graphics formats

The following graphics formats are supported for saving plots or screen captures from the UI:

1. Windows bitmap files (*.bmp)
2. Joint photographic expert group files (*.jpg)
3. Portable network graphics files (*.png)
4. Portable bitmap files (*.pbm)
5. Portable graymap files (*.pgm)
6. Portable pixmap files (*.ppm)
7. X11 bitmap files (*.xbm)
8. X11 pixmap files (*.xpm)