Sapphire Programmer's Manual V4.00

A *Bluetooth®* 5 *LE* standard phy level and production tester application for the TLF3000 software-defined receiver, signal analyzer and signal generator.

5 January 2021

1 Contents

2 Overview.

TLF3000 is a wideband, ultra-high dynamic range 2.4 GHz software-defined radio, signal analyser and signal generator. It captures and analyses the entire 2402-2480 MHz band simultaneously. It can also generate arbitrary waveforms occupying the band 2395-2485 MHz with a maximum peak level of 0 dBm. In addition, it includes a CW signal generator covering 25 MHz-6GHz with an output level of -50 dBm to -28 dBm

Sapphire is a *Bluetooth*® 5 LE application for the *TLF3000* software-defined receiver, signal analyser and signal generator. The Sapphire application can:

1. Perform all phy level tests as specified in Bluetooth Low Energy RF PHY Test Specification (with minor restrictions). Testing beyond the limits of the specification is also supported.
2. Act as a signal generator, creating all necessary signals for receiver testing, including signals outside the specification as well as continuous tone extenstions for AoA/AoD testing. It can either simulate an AoD antenna array or drive an external array. Signals as weak as -120 dBm can be generated for testing long range modes.
3. Act as a signal analyser, performing transmitter tests on conducted or off-air signals without knowledge of the payload format or hopping sequence. Test coverage includes AoA/AoD continuous tone extension tests.
4. Generate advertising or scan request packets to provoke activity from items on a production line and analyse the captured packets.

The application has been honed for speed. Particularly impressive is the ability to perform in-band emission tests over the entire 2.4GHz band in just a few milliseconds. This is possible due to *TLF3000*‘s unique parallel architecture.

A key feature of the unit is its ability to perform C/I, receiver selectivity and intermodulation tests without the need for additional test equipment. This is possible due to *TLF3000*’s ultra-linear wideband signal generator, permitting both wanted and interfering signals to be generated through the same signal path. The high linearity and low noise floor ensure that there is ample dynamic range to encompass both the wanted and interfering signals. Furthermore, high fidelity filtering of the interfering signals ensures that they are correctly bandlimited and that unwanted sidebands are not responsible for test failures, which is frequently overlooked when external test equipment is used to provide these signals. The single signal path also removes the need for time consuming and laborious calibration of signal combiners as well as eliminating the need to ensure that the injected interfering and wanted signals do not generate intermodulation products before arriving at the DUT.

Unique to *TLF3000* is a 25 MHz to 6 GHz signal generator. This enables the majority of the receiver blocking performance to be explored prior to committing the DUT for formal inspection at a test house with its associated costs.

The Sapphire application is highly parameterised, permitting it to be configured for different scenarios. For example:

1. The unit will function with arbitrary access addresses, allowing multiple devices to be tested simultaneously without cross-talk.
2. The unit may be controlled directly from a host machine via USB or Ethernet, or operated stand-alone with digital IO used to start, stop and report test results.
3. The DUT may be controlled directly from the unit or via a common host platform.

3 Modes of Operation.

3.1 Overview

The Sapphire application has 4 modes of operation:

1. Phy level tester
2. Signal generator
3. Signal analyser
4. Advertise/scan with phy level testing

4 Phy Level Tester Mode

4.1 Overview

The phy level tester mode aims to perform all the tests set out in the Bluetooth Low Energy RF PHY Test Specification including all AoA/AoD tests.

In this mode, a parameterised test script is downloaded to the *TLF3000*. A DUT is then connected to the *TLF3000* and the test script executed. Results can be reported back to a host machine or signalled via digital IO lines, permitting the unit to be embedded into a wide range of test systems.

4.1.1 Supported tests

Sapphire supports the following phy level tests:

Test number	Test description	Switching	Phy	Notes
RF-PHY/TRM/BV-01-C	Output power		Uncoded, 1Mbps	
RF-PHY/TRM/BV-03-C	In-band emissions		Uncoded, 1Mbps	
RF-PHY/TRM/BV-05-C	Modulation characteristics		Uncoded, 1Mbps	
RF-PHY/TRM/BV-06-C	Carrier frequency offset & drift		Uncoded, 1Mbps	
RF-PHY/TRM/BV-08-C	In-band emissions		2Mbps	
RF-PHY/TRM/BV-09-C	Modulation characteristics		Stable, uncoded, 1Mbps	
RF-PHY/TRM/BV-10-C	Modulation characteristics		2Mbps	
RF-PHY/TRM/BV-11-C	Modulation characteristics		Stable, 2Mbps	
RF-PHY/TRM/BV-12-C	Carrier frequency offset & drift		2Mbps	
RF-PHY/TRM/BV-13-C	Modulation characteristics		LE Coded, S=8	
RF-PHY/TRM/BV-14-C	Carrier frequency offset & drift		LE Coded, S=8	
RF-PHY/TRM/BV-15-C	Output power, with CTE		Uncoded, 1Mbps	
RF-PHY/TRM/BV-16-C	Carrier frequency offset & drift, CTE		Uncoded, 1Mbps	
RF-PHY/TRM/BV-17-C	Carrier frequency offset & drift, CTE		Uncoded, 2Mbps	
RF-PHY/TRM/PS/BV-01-C	Tx power stability, AoD	2µs	Uncoded, 1Mbps	
RF-PHY/TRM/PS/BV-02-C	Tx power stability, AoD	1µs	Uncoded, 1Mbps	
RF-PHY/TRM/PS/BV-03-C	Tx power stability, AoD	2µs	Uncoded, 2Mbps	
RF-PHY/TRM/PS/BV-04-C	Tx power stability, AoD	1µs	Uncoded, 2Mbps	
RF-PHY/TRM/ASI/BV-01-C	Antenna switching integrity AoD	2µs	Uncoded, 1Mbps	
RF-PHY/TRM/ASI/BV-02-C	Antenna switching integrity AoD	1µs	Uncoded, 1Mbps	
RF-PHY/TRM/ASI/BV-03-C	Antenna switching integrity AoD	2µs	Uncoded, 2Mbps	
RF-PHY/TRM/ASI/BV-04-C	Antenna switching integrity AoD	1µs	Uncoded, 2Mbps	
RF-PHY/RCV/BV-01-C	Receiver sensitivity		Uncoded, 1Mbps	
RF-PHY/RCV /BV-03-C	C/I & receiver selectivity		Uncoded, 1Mbps	
RF-PHY/RCV /BV-04-C	Blocking		Uncoded, 1Mbps	See (a)
RF-PHY/RCV /BV-05-C	Intermodulation		Uncoded, 1Mbps	See (b)
RF-PHY/RCV /BV-06-C	Maximum input signal		Uncoded, 1Mbps	
RF-PHY/RCV /BV-07-C	PER report integrity		Uncoded, 1Mbps	
RF-PHY/RCV /BV-08-C	Receiver sensitivity		2Mbps	
RF-PHY/RCV /BV-09-C	C/I & receiver selectivity		2Mbps	

RF-PHY/RCV /BV-10-C	Blocking		2Mbps	See (a)
RF-PHY/RCV /BV-11-C	Intermodulation		2Mbps	See (b)
RF-PHY/RCV /BV-12-C	Maximum input signal		2Mbps	
RF-PHY/RCV /BV-13-C	PER report integrity		2Mbps	
RF-PHY/RCV /BV-14-C	Receiver sensitivity		Stable, uncoded, 1Mbps	
RF-PHY/RCV /BV-15-C	C/I & receiver selectivity		Stable, uncoded, 1Mbps	
RF-PHY/RCV /BV-16-C	Blocking		Stable, uncoded, 1Mbps	See (a)
RF-PHY/RCV /BV-17-C	Intermodulation		Stable, uncoded, 1Mbps	See (b)
RF-PHY/RCV /BV-18-C	Maximum input signal		Stable, uncoded, 1Mbps	
RF-PHY/RCV /BV-19-C	PER report integrity		Stable, uncoded, 1Mbps	
RF-PHY/RCV /BV-20-C	Receiver sensitivity		Stable, 2Mbps	
RF-PHY/RCV /BV-21-C	C/I & receiver selectivity		Stable, 2Mbps	
RF-PHY/RCV /BV-22-C	Blocking		Stable, 2Mbps	See (a)
RF-PHY/RCV /BV-23-C	Intermodulation		Stable, 2Mbps	See (b)
RF-PHY/RCV /BV-24-C	Maximum input signal		Stable, 2Mbps	
RF-PHY/RCV /BV-25-C	PER report integrity		Stable, 2Mbps	
RF-PHY/RCV /BV-26-C	Receiver sensitivity		LE coded, S=2	
RF-PHY/RCV /BV-27-C	Receiver sensitivity		LE coded, S=8	
RF-PHY/RCV /BV-28-C	C/I & receiver selectivity		LE coded, S=2	
RF-PHY/RCV /BV-29-C	C/I & receiver selectivity		LE coded, S=8	
RF-PHY/RCV /BV-30-C	PER report integrity		LE coded, S=2	
RF-PHY/RCV /BV-31-C	PER report integrity		LE coded, S=8	
RF-PHY/RCV /BV-32-C	Receiver sensitivity		Stable, LE coded, S=2	
RF-PHY/RCV /BV-33-C	Receiver sensitivity		Stable, LE coded, S=8	
RF-PHY/RCV /BV-34-C	C/I & receiver selectivity		Stable, LE coded, S=2	
RF-PHY/RCV /BV-35-C	C/I & receiver selectivity		Stable, LE coded, S=8	
RF-PHY/RCV /BV-36-C	PER report integrity		Stable, LE coded, S=2	
RF-PHY/RCV /BV-37-C	PER report integrity		Stable, LE coded, S=8	
RF-PHY/RCV /IQC/BV-01-C	IQ sample coherency, AoD receiver	2µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQC/BV-02-C	IQ sample coherency, AoD receiver	1µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQC/BV-03-C	IQ sample coherency, AoD receiver	2µs	Uncoded, 2Mbps	
RF-PHY/RCV /IQC/BV-04-C	IQ sample coherency, AoD receiver	1µs	Uncoded, 2Mbps	
RF-PHY/RCV /IQC/BV-05-C	IQ sample coherency, AoA receiver	2µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQC/BV-06-C	IQ sample coherency, AoA receiver	2µs	Uncoded, 2Mbps	
RF-PHY/RCV /IQDR/BV-01-C	IQ samples dynamic range, AoD rx	2µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQDR/BV-02-C	IQ samples dynamic range, AoD rx	1µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQDR/BV-03-C	IQ samples dynamic range, AoD rx	2µs	Uncoded, 2Mbps	
RF-PHY/RCV /IQDR/BV-04-C	IQ samples dynamic range, AoD rx	1µs	Uncoded, 2Mbps	
RF-PHY/RCV /IQDR/BV-05-C	IQ samples dynamic range, AoA rx	2µs	Uncoded, 1Mbps	
RF-PHY/RCV /IQDR/BV-06-C	IQ samples dynamic range, AoA rx	2µs	Uncoded, 2Mbps	
RF-PHY/TRM/BV-90-C	Output power		Uncoded, 2Mbps	see (c)
RF-PHY/TRM/BV-91-C	Output power		LE coded, S=2	see (c)
RF-PHY/TRM/BV-92-C	Output power		LE coded, S=8	see (c)
RF-PHY/TRM/BV-93-C	In-band emissions		LE coded, S=2	see (c)
RF-PHY/TRM/BV-94-C	In-band emissions		LE coded, S=8	see (c)

Limitations:

a) Blocking tests are limited to the range 25MHz to 6GHz. The Bluetooth 5.2 test specification has an upper limit of 12.75GHz. The supported range does include the 2nd harmonic of the 2.4GHz band. If the blocking test passes at the 2nd harmonic, then it is highly likely to pass for higher test frequencies. The blocking source used in the *TLF3000* has significant harmonic content. Some internal filtering of the source is applied to attenuate harmonics falling within the 2.4GHz band. If blocking failures do occur, then further testing may be required to ascertain whether these are due to the fundamental of the blocker or one of its harmonics.

b) The specification states that the intermodulation tests should be performed with the interfering signals both on the low side and the high side of the wanted signal. Under some circumstances, when the wanted signal is close to the band edge, this can lead to the interfering signals being placed outside the *TLF3000* signal generator bandwidth. When these circumstances arise, the test will only be performed with the interfering signals positioned closer to the band centre. The requirement to perform intermodulation tests with the signals both above and below the wanted signal is somewhat perverse. If the wanted signal is at the band edge and the interfering signals are positioned outside the band, then the interfering signals will be attenuated by frontend filtering and the intermodulation performance of the device improved. Hence the inability of *TLF3000* to test this scenario is considered somewhat academic.
   The table below sets out which tests can be performed if the *TLF3000* unit has undergone extended calibration:

	Interferers below			**Interferers above**		
Channel	**N = 3**	**N = 4**	**N = 5**	**N = 3**	**N = 4**	**N = 5**
2402	Y	Y	Y	Y	Y	Y
2403	Y	Y	Y	Y	Y	Y
…	…	…	…	…	…	…
2478	Y	Y	Y	Y	Y	Y
2479	Y	Y	Y	Y	Y	N
2480	Y	Y	Y	Y	Y	N

c) These tests have not been specified by the Bluetooth SIG but have been added due to customer requests.

4.1.2 AoA/AoD

Sapphire is fully complinat with the Bluetooth 5.2 AoA/AoD specification. The signal generator is capable of producing packets with an arbitrary constant tone extension as well as monitoring the IQ samples received from the DUT. The signal analyser is also capable of capturing and processing the constant tone extension of received packets.

In order to perform the antenna switching integrity tests (RF-PHY/TRM/ASI/BV-0X-C), many reconnections of the antenna array to the *TLF3000* and terminators are required. This process can be automated by connecting the 8-way switch unit between the *TLF3000* and the antenna array. The *TLF3000* can automatically control the settings of the 8-way switch unit as the test progresses.

An external antenna array and switch can be connected to the either of the *TLF3000* RF ports. The switch matrix can be controlled via *TLF3000's* digital IO connector.

4.1.3 DUT control

Sapphire supports UART communication with a DUT using the following protocols:

1. Direct test mode
2. H4
3. H5
4. BCSP
5. Qualcomm debug SPI (requires access to Qualcomm's test engine dll)

Supported baud rates for a DUT connected directly to the *TLF3000* unit are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. If the DUT is connected via a common host, then the supported baud rates are 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800 and 921600. An automatic baud rate detection features is also supported for H5 and BCSP. Additional baud rates can be added on request.

Hardware handshaking, software flow control, stop bits and parity are all configurable.

The IO voltage for the UART can be either:

1. An internal 3.3 V source from the *TLF3000* unit
2. An external voltage in the range 0.8 V to 3.6 V supplied to the *TLF3000* unit

At the present time Sapphire does not support USB control of the DUT. This is a software limitation which may be removed in the future.

It is possible for the DUT to be controlled by the host machine which is controlling Sapphire. In this scenario, Sapphire passes the UART data which it wishes to be sent to the DUT to the host machine. The host machine issues these commands to the DUT, using whatever transport is appropriate. The host machine then returns UART data from the DUT to the Sapphire application.

4.2 Multi-DUT support

TLF3000 can perform phy level tests on up to 32 DUTs simultaneosuly. The control of the DUTs must be via the host machine. The RF ports of the DUTs must be combined using an N-way splitter to connect to the single Tx/Rx Port of the *TLF3000*. The Teledyne 8-way switch unit can be used to provide this combination.

The time consuming receiver tests are performed in parallel whilst the much faster transmitter tests are performed sequentially.

4.2.1 DUT supported features

In order to correctly configure the tests, Sapphire must be aware of the DUT supported features, such as:

1. Support for 2 Mbps
3. Support for coded phy
4. Support for stable modulation index
5. Maximum transmit and receive payload sizes
6. Whether the device is a central, peripheral, broadcaster or observer
7. Support of AoA/AoD features

This information can be downloaded directly to Sapphire. Alternatively, Sapphire can be requested to interrogate the DUT to obtain this information. If various interrogation commands are not supported by the DUT, then Sapphire will assume that the relevant features are also not supported.

The ability to query the device for supported features is useful in scenarios such as RMA processing where a range of different devices may be encountered. However, the interrogation process adds to test time and may be inappropriate on a production line. Hence the ability to download the supported features information directly.

4.2.2 Scripting

RF Phy level tests are performed by downloading a test script to the Sapphire application and then executing the script. When multiple units are to be tested, the test script need only be downloaded once.

The test script is an ASCII formatted file. Each line in the test script corresponds to one of the Bluetooth test cases. If only the test name is specified, then the default parameters for that test are assumed; i.e. the test it carried out in accordance with the Bluetooth 5 LE RF PHY Test Specification (RF-PHY.TS.p15). However, each test also permits a number of quantities to be parameterized, such as:

1. RF channels to be tested
2. Packet lengths to be used
3. Number of packets to be used in the test
4. Wanted signal levels
5. Interferer signal levels
6. Interferer signal frequencies
7. CTE lengths

This parameterisation of the tests permits DUTs to be tested outside the Bluetooth specification and hence the margin against the Bluetooth specification to be determined. It also allows devices to be tested against a manufacturers' datasheet rather than the Bluetooth specification.

4.2.3 Parameter searches

Each parameter which can be specified can be entered either as a single value or a range of values. For example, the channels to be tested could be specified as:

1. 39 – a single channel
8. 1,12,39 – a list of channels
9. 1,12,39,3:10 – a list of channels plus a range of channels

By enabling complex ranges of parameters to be specified, automated testing of specific scenarios can be achieved.

Furthermore, by specifying a range of parameters for quantities such as wanted signal level, the sensitivity of the DUT can automatically be searched for.

4.2.4 Test duplication

Running any of the modulation characteristics tests will automatically generate results for the corresponding carrier offset and drift tests. This occurs since the packets the DUT must transmit for the carrier offset and drift tests are a subset of those required for the modulation characteristic tests. Furthermore, the test channels used for modulation characteristic measurements are the same as those required for carrier offset and drift measurements. It is therefore unnecessary to explicitly include the carrier offset and drift tests within the test script if the corresponding modulation characteristics test are present.

Running any of the in-band emission tests will automatically generate results for the corresponding output power tests. This occurs since the packet types used in the two tests are identical. However, the test channels for in-band emission tests are not always the same as those for output power tests. Where there is overlap in the test channels, the overlapping channels can be excluded from the output power tests if the corresponding in-band emissions tests are included in the test script.

4.2.5 Run modes

When the test script is executed, a number of different run modes can be specified:

1. Only test against limits.
   If this flag is set, each test will be terminated as soon as the test limit is passed. For example, when testing receiver sensitivity for 30.8% PER over 1500 packets, if the first 1038 packets all pass, then the test must pass irrespective of whether the remaining 462 packets pass or fail. This allows test time to be minimized when accurate estimates of the test parameters are not required. Hence it may be appropriate to set this flag on a production line but possibly not when characterising silicon over temperature and voltage.
2. Run to completion
   If this flag is set, then all the tests specified in the test script will be executed. If this flag is not set, then the test script will be terminated as soon as any test fails. This is useful on a production line where is prevents wasting test time on devices which have already failed. However, it limits the test coverage which may be useful in ascertaining the root cause of repeated failures.

A further parameter can be specified to indicate the number of times the test script should be run. If required, the test script can be made to loop indefinitely. This facility can be useful for tracking down occasional anomalies in the DUT performance.

4.2.6 External Connectivity

In phy test mode the following connectivity is required:

1. RF connection between the DUT and the *TLF3000*. This would normally be a calibrated cabled connection between the DUT and the *TLF3000* Tx/Rx port. However, it is also possible to perform radiated testing. In radiated test mode, the unit will transmit on the Tx/Rx port and can receive on either the Tx/Rx port or the more sensitive Monitor In port. If multiple DUTs are to be tested simultaneously, then the DUT RF ports must be combined using an N-way splitter or the Teledyne 8-way switch units. The combined signals are then connected to the Tx/Rx port of the *TLF3000*.
2. Control connection between *TLF3000* and the DUT(s). This would normally be a UART connection. However, *TLF3000* also has the ability to communicate with the DUT via a common host. If multiple DUTs are being tested simultaneously then the control of the DUTs must be via the common host.
3. Control connection to the *TLF3000*. This would normally be a USB or Ethernet connection to a host machine. However, the starting and stopping of a test script can also be controlled via digital input lines and the results displayed using digital output lines. This facility allows *TLF3000* to be incorporated into a range of test facilities.

4.3 Signal Generator Mode

4.3.1 Overview

In signal generator mode, the Sapphire application can simultaneously produce all the wanted and interfering signals required to perform all the tests within the Bluetooth 5 LE RF PHY Test Specification, with the following limitations:

1. CW blocking signals can only be generated to 6 GHz; the Bluetooth specification has an upper limit of 12.75 GHz. The upper frequency of 6 GHz does include the second harmonic of the 2.4 GHz band. If the blocking test passes at the second harmonic of 2.4 GHz, then it is highly likely that the DUT will not be blocked by higher interfering frequencies.
2. For the intermodulation tests, interfering signals can only be generated within the *TLF3000* transmit bandwidth. This limitation prevents testing of either the high or low side interference case when the wanted signal is close to a band edge. In practice this is of little significance since under these circumstances, the interfering signals would be attenuated by the front end filtering of the DUT and the intermodulation performance would be improved. The table below sets out which tests can be performed if the *TLF3000* unit has undergone extended calibration:

	Interferers below			**Interferers above**		
Channel	**N = 3**	**N = 4**	**N = 5**	**N = 3**	**N = 4**	**N = 5**
2402	Y	Y	Y	Y	Y	Y
2403	Y	Y	Y	Y	Y	Y
…	…	…	…	…	…	…
2478	Y	Y	Y	Y	Y	Y
2479	Y	Y	Y	Y	Y	N
2480	Y	Y	Y	Y	Y	N

The signal generator can simultaneously produce six independent signals:

1. A packetized wanted signal
2. A continuously modulated interferer signal
3. A primary in-band CW signal
4. A secondary in-band CW signal
5. Broadband AWGN
6. An out-of-band CW signal

The signal generator can also communicate with a DUT. This permits real-time PER measurements to be made. It is also possible to read back the IQ data generated by the DUT on receipt of a CTE.

4.3.2 Wanted signal

A packetized wanted signal to be received by the DUT can be generated. This is parameterized by:

1. Frequency: 2392 to 2488MHz in 1MHz steps
2. Amplitude: -120 dBm to 0 dBm with a resolution of 0.1 dBm
3. Modulation:
    a. 2Mbps, uncoded
    b. 1Mbps, uncoded
    c. 1Mbps, coded, s=2
    d. 1Mbps, coded, s=8
4. Arbitrary access address
5. Arbitrary CRC seed
6. Packet payload:
    a. PRBS9 sequence
    b. PRBS15 sequence
    c. PRBS11 sequence
    d. PRBS20 sequence
    e. PRBS23 sequence
    f. PRBS29 sequence
    g. PRBS31 sequence
    h. `11110000` in transmission order
    i. `10101010` in transmission order
    j. `11111111` in transmission order
    k. `00000000` in transmission order
    l. `00001111` in transmission order
    m. `01010101` in transmission order
7. Payload length: 0 to 255 octets
8. Packet interval: packet duration to 100ms
9. Number of packets to be transmitted: 1 to 65535, or continuous
10. Dirty or standard transmitter
10. Dirty transmitter parameters:
    a. Carrier frequency offset: -500 kHz to +500 kHz for 2 Mbps uncoded and -250 kHz to +250 kHz for all other phys.
    b. Modulation index: 0.4 to 0.6
    c. Drift magnitude: 0 to 156 kHz for 2 Mbps uncoded and 0 to 78 kHz for all other phys.
    d. Drift rate: 0 to 2440 Hz
    e. Symbol timing error: -100 ppm to +100 ppm
11. Presence of a CTE for use in AoA/Aod

12. CTE parameters:
    f. Whether the CTE is constant antenna phase (AoA) or switched antenna phase (AoD)
    g. The number of 8µs slots in the supplemental
    h. Wether the transmitted CTEinfo field specifies AoA or AoD
    i. If AoD, whether the switching slots or 1 µs or 2 µs
    j. If AoD, whether the antenna switching pattern is 1,2,3,4 … 1,2,3,4 or 1,2,3,4 … 4,3,2,1,2,3,4 …
    k. If AoD, whether a real antenna array is connected or whether the antenna array is to be simulated
    l. If AoD and the antenna array is to be simulated, the phases and amplitudes to be transmitted in each slot
    m. If AoD and a real antenna array is connected, the setting of the DIO lines for each slot
    n. If AoD and a real antenna array is connected, the timing delay between the *TLF3000* setting the DIO lines and the antenna array switching
11. The time taken for the signal amplitude to ramp up and down
12. The length of unmodulated carrier after the power ramp up and prior to the first bit of the preamble
13. The length of unmodulated carrier after the last bit of the CRC and the power ramp down
13. Digital output lines to be toggled when a wanted packet is transmitted

The drift magnitude and drift rate follow the definitions in the Bluetooth 5 LE RF PHY Test Specification.

The drift follows a sinusoidal waveform whose period is given by the drift rate and whose amplitude is determined by the drift magnitude. The drift is always zero at the start of a packet. The sign of the drift amplitude is inverted on alternate packets.

The dirty transmitter is defined by an arbitrary length list, each element in the list specifying:

1. Carrier frequency offset
2. Modulation index
3. Drift magnitude
4. Drift rate
5. Symbol timing error

The first element in the list is used to define the first 50 transmitted packets, with alternate packets having the sign of the drift reversed. The next 50 packets to be transmitted are defined by the second element in the list, and so on. If more packets are to be transmitted than there are definitions in the list, then the contents of the list are repeated until all packets have been transmitted.

An important feature of the wanted signal generator is its ability to create signals with amplitudes as low as -120 dBm. This is essential for testing 1 Mbps, coded, s=8 receivers whose sensitivity can easily approach -110 dBm.

If any of the wanted signal parameters are changed whilst the wanted signal is being transmitted, then the current transmission is aborted and the wanted signal plus any interfering signals restarted.

4.3.3 Interfering signal

An interfering signal with PRBS payload to be used in C/I and intermodulation tests. This signal is parameterised by:

1. Frequency: 2392 MHz to 2488 MHz in steps of 1 MHz
2. Amplitude: -120 dBm to 0 dBm with a resolution of 0.1 dBm
3. Modulation:
    a. 2 Mbps
    b. 1 Mbps, uncoded
    c. 1 Mbps, coded, s=2
    d. 1 Mbps, coded, s=8
4. Contents of payload:
    a. PRBS9
    b. PRBS11
    c. PRBS15
    d. PRBS20
    e. PRBS23
    f. PRBS29
    g. PRBS31
5. Whether the interferer is a continuous signal or packetised
6. The payload length, if the interferer signal is packetised
7. The packet interval, if the interferer signal is packetised
8. Digital output lines to be toggled when this signal is active

Although this is can be a continuous signal, on commencement of a transmission a valid preamble and access code are sent prior to the PRBS payload. Hence there are a few µs at the start of each test period before the PRBS sequence commences. The wanted signal is always delayed until this period has passed.

If any of the interfering signal parameters are changed whilst the interferer signal is active, then the current transmission is aborted, and all wanted and interfering signals restarted.

4.3.4 Primary and Seconadry In-band CW signals

Two in-band CW signals which can be used in intermodulation testing. These signals could also be used for custom receiver selectivity testing. The signal is parameterized by:

1. Frequency: 2392 MHz to 2488 MHz in steps of 1 MHz
2. Amplitude: -120 dBm to 0 dBm with a resolution of 0.1 dBm

If any of the in-band CW signal parameters are changed whilst the in-band CW signal is active, then the current transmission is aborted, and all wanted and interfering signals restarted.

4.3.5 AWGN signal

A AWGN signal covering 2395 to 2485MHz can be generated. The amplitude of this signal can be set between -162dBm/MHz and -42dBm/MHz. This signal permits the SNR of the wanted signal to be accurately controlled, which can be useful for testing the implementation of digital demodulators.

4.3.6 Out-of-band CW signal

An out-of-band CW signal to be used in receiver blocking tests. This signal can typically be generated at a level up to -25 dBm. This permits up to 5 dB cable loss if the required Bluetooth test level of -30 dBm is to be available at the DUT input. The frequency of the signal can be in the range 25 MHz to 6 GHz with a resolution of 1 MHz.

The CW generator within the *TLF3000* unit possesses significant harmonic content. When a harmonic lies within the 2.4 GHz ISM band, it is possible for the DUT to be blocked by the harmonic and not the fundamental. This issue is not unique to the *TLF3000*; harmonic content is also an issue when using external signal generators to provide the blocking signal. For example, an R&S SMW200A specifies harmonic content as < -30 dBc. If this unit were to generate a blocking signal at 1201 MHz with a level of -30 dBm, then it may generate a second harmonic at 2402 MHz with a level of -60 dBm. Given that the DUT is being tested at a sensitivity level of at least -67 dBm, it is clear that the DUT will be blocked by the harmonic and not the fundamental.

To help alleviate this problem, the *TLF3000* contains a 2.4 GHz notch filter in the out-of-band CW signal path. This notch filter is switched in whenever the blocking signal is below 1500 MHz. Hence the harmonic content of the blocker within the 2.4 GHz ISM band is greatly attenuated. However, when testing sensitive receivers, it should always be borne in mind that any blocking failures could be due to harmonics of the blocking signal and not the fundamental.

The out-of-band CW signal can be modified at any time without affecting the wanted or in-band interfering signals.

4.4 Signal Analyser Mode

4.4.1 Overview

The Sapphire application can be run in a signal analyser mode in which it monitors the integrity of the waveforms which it receives. Each received packet is analysed in accordance with the transmitter tests of the Bluetooth LE RF PHY Test Specification, including the in-band emissions test. The analysed packets may be received over either a conducted link or over-the-air. The packets can be analysed irrespective of whether they are Bluetooth test packets or whether they have been whitened.

The Sapphire application contains extensive data analysis features which permit results to be interrogated and displayed in a number of different parameterised formats.

The signal analyser can also communicate with a DUT. This permits the signal being transmitted by the DUT, and hence analysed by the signal analyser, to be interactively controlled.

4.4.2 Capture port

The Sapphire application can receive packets on one of two ports:

1. Tx/Rx port. This port has a noise figure of +46 dB and can handle input signals as high as +27 dBm. Use this port for conducted measurements.
2. Monitor In port. This port has a noise figure of +6 dB and a maximum usable input signal level of -10 dBm. Use this port for off-air measurements.

4.4.3 Capture criteria

All 40 LE channels are monitored simultaneously by the *TLF3000*'s wideband receiver. Packets are captured for further analysis when they satisfy a range of criteria:

1. The packet must possess a specified access address. A single, arbitrary access address can be specified.
2. The packet must be modulated using one of the specified phys. One or more of the following phys can be specified:
    a. 2 Mbps, uncoded
    b. 1 Mbps, uncoded
    c. 1 Mbps, coded, S=2
    d. 1 Mbps, coded, S=8
3. The packet must arrive on one of a number of specified RF channels. Arbitrary combinations of LE RF channels may be specified.
4. The packet length must fall within one of a number of selected packet length bins. Up to *16* packet length bins can be specified.

The ability to monitor all 40 LE channels simultaneously permits the Sapphire application to follow hopping links without the need for knowledge of the hopping sequence or link encryption.

If it is desired to test multiple units simultaneously on a production line, then confusion can arise if all devices use the same access address. The Sapphire application allows each tester to look for a distinct access address and hence avoid nuisance packets from other devices on the production line.

4.4.4 Captured data and analysis

The captured data can be subject to a number of different tests:

1. Average and peak power measurements
2. Modulation characteristics
3. Carrier offset and drift measurements
4. In-band emissions
5. CTE analysis

One or more of these tests may be selected.

The data used for the modulation characteristics, carrier offset, drift and CTE analysis is 32x oversampled in full compliance with the Bluetooth LE RF PHY Test Specification. Many LE testers do not fulfil this requirement or simply fail to specify the oversampling used. This complete oversampling of the waveform is accomplished without any degradation in the overall test time.

It is also possible to capture the raw data of individual packets. This can be returned as:

1. IQ complex samples
2. Amplitude profile in dBm
3. Deviation profile
4. CTE phases

The oversampling ratio for this data can be 32x, 16x, 8x or 4x. Because of the large amounts of data to be transferred, turning on collection of raw waveform data increases the interval between the end of one packet and the start of the next packet which can be captured and analysed.

4.4.5 Capture termination

Capture termination can be caused by:

1. A stop capture command
2. A limit failure, provided the stop capture on limit failure flag has been set

For each test being conducted, appropriate limits can be downloaded to the Sapphire application. As each packet is analysed, the results are compared against the limits. If a limit failure is detected, then data capture will be terminated if the stop capture on limit failure flag has been set.

If raw waveform data collection has been enabled, and data collection is terminated by a limit failure, then the last waveform stored will be that corresponding to the limit failure. This facility permits the rogue waveform to be examined in detailed so that the exact cause of the limit failure can be understood. This can be particularly useful when observing real-world waveforms captured off-air.

4.4.6 Capture modes

The Sapphire application supports two capture modes:

1. Standard LE test packet mode
2. Off-air mode

In the standard LE test packet mode, the application is expecting to receive LE test packets as defined in the core specification. These are unwhitened packets whose header determines both the payload contents and the payload length. Different payload contents are required to perform different phy level tests. The signal analyzer automatically determines the payload contents from the packet header and then performs the relevant phy level tests. Only three of the available 8 payload formats are used in the

Bluetooth specification, however, the Sapphire application can utilise all possible payloads to extract the maximum possible information.

Packet payload	Phy level tests	Bluetooth Specified
PRBS9	Output power In-band emissions	✓
11110000	Δf_{1max}, Δf_{1avg}	✓
10101010	Δf_{2max}, Δf_{2avg}, carrier offset & drift	✓
11111111	Carrier offset & drift	
00001111	Δf_{1max}, Δf_{1avg}	
01010101	Δf_{2max}, Δf_{2avg}, carrier offset & drift	
00000000	Carrier offset & drift	
PRBS15	Output power In-band emissions	

Table 1: Available Payloads for the Sapphire Application

Although the Sapphire application will accept test packets with the standard test access address (0x71764129), it can also be configured to accept test packets with an arbitrary access address.

In the off-air mode, the Sapphire application will apply an approximation to the Bluetooth 5 LE RF PHY Test Specification to packets with arbitrary payloads. The approximations used are relatively accurate and produce results in exactly the same format as test packets. This mode will function even if the DUT is hopping or the packets are whitened. In this mode the Sapphire application can apply all the phy level tests to packets collected from a real-world device or link.

When operating in standard test packet mode, all calculations are performed in hardware. However, in the off-air mode, some of the calculations are performed in software (but still within the *TLF3000* unit). If long packets are being analysed, then the off-air mode may require a larger interval between the end of one packet and the start of the next packet which can be collected and analysed.

4.4.7 Collected statistics

When the appropriate tests are enabled, the Sapphire application collects statistics on the following quantities defined within the Bluetooth 5 LE RF PHY Test Specification:

1. P_{avg}
2. Pk - P_{avg}
3. $\Delta F1_{max}$
4. $\Delta F1_{avg}$
5. $\Delta F2_{max}$
6. $\Delta F2_{avg}$
7. $\Delta F2_{avg} / \Delta F1_{avg}$
8. $\Delta F2_{max}$ 99% percentile
9. F_0
10. F_n
11. F_0 - F_n
12. $F_1 – F_0$
13. $F_{n+5} – F_n$
14. F_{tx} @ ±2 MHz offset
15. F_{tx} @ ≥±3 MHz offset
16. Number of in-band emission exceptions
17. P_{avg} – CTE
18. Pk – P_{avg} – CTE
19. Fs_i
20. $Fs_1 – F_p$
21. Fs_i - F_0
22. $Fs_i – Fs_{i-3}$
23. $P_{ref,dev} / P_{ref,ave}$
24. $P_{n,dev} / P_{n,ave}$
25. Phase within the reference period and each sampling slot of the supplemental

For each of these quantities the following are stored:

1. Maximum value
2. Minimum value
3. Average value
4. Last value observed
5. Histogram of observed values
6. Number of samples analysed
7. Number of packets analysed

In addition, the following statistics are also held for in-band emissions:

1. Maximum value for each of the 81 1 MHz bands
2. Minimum value for each of the 81 1 MHz bands
3. Average value for each of the 81 1 MHz bands
4. Last observed value for each of the 81 1 MHz bands
5. Maximum value for each of the 810 100 kHz bands used in the computation
6. Minimum value for each of the 810 100 kHz bands used in the computation
7. Average value for each of the 810 100 kHz bands used in the computation
8. Last observed value for each of the 810 100 kHz bands used in the computation
9. Number of packets analysed

For tests excluding the CTE measurements, separate sets of statistics are maintained for each of:

1. RF channel
2. Packet length group
3. Modulation scheme
4. Standard test packet mode and off-air mode (excluding in-band emissions)

For CTE measurements, separate sets of statistics are maintained for each of:

1. RF channel
2. CTE length
3. Modulation scheme
4. AoA, AoD with 1 µs switching slots, AoD with 2µs switching slots

By saving statistics on the 100 kHz resolution intermediate stage in the in-band emissions test, much greater insight is available into why there are test failures or where there is low margin against the test specification.

In areas of light Wi-Fi pollution, the statistics on average in-band emissions can be useful since they tend to attenuate the Wi-Fi pollution. In some cases this can make the difference between being able to perform the test in a lab environment or resorting to the use of a screened chamber.

4.4.8 Statistical analysis

Two types of statistical data can be requested from the Sapphire application:

1. Tabular statistics
2. Graphical statistics

Tabular statistics consist of a table showing the minimum, maximum, average and current value of various quantities for the selected measurement. Measurements which can be selected and the returned quantities are:

Measurement set	Returned quantities
Output power	P_{avg} $Pk - P_{avg}$
Modulation characteristics	$\Delta F1_{max}$ $\Delta F1_{avg}$ $\Delta F2_{max}$ $\Delta F2_{avg}$ $\Delta F2_{avg} / \Delta F1_{avg}$ $\Delta F2_{max}$ 99% percentile
Drift and carrier offset	F_0 F_n $F_0 - F_n$ $F_1 – F_0$ $F_{n+5} – F_n$
In-band emissions	F_{tx} @ ±2 MHz offset F_{tx} @ ≥±3 MHz offset Number of in-band emission exceptions
CTE	P_{avg} $Pk – P_{avg}$ Fs_i $Fs_1 – F_p$ $Fs_i - F_0$ $Fs_i – Fs_{i-3}$ $P_{ref,dev} / P_{ref,ave}$ $P_{n,dev} / P_{n,ave}$ 8 phases within the reference period A phase for each of the sampling slots

Table 2: Measurements which can be selected and their returned quantities for signal analyzer mode

The CTE phases are calculated by:

1. Estimating the average frequency and initial phase of the first 7 1 µs slots within the reference period
2. Removing the average frequency from the supplemental
3. Subtracting the initial phase from the frequency compensated supplemental

Graphical statistics consist of one of the following quantities:

1. P_{avg}
2. Pk - P_{avg}
3. $\Delta F1_{max}$
4. $\Delta F1_{avg}$
5. $\Delta F2_{max}$
6. $\Delta F2_{avg}$
7. $\Delta F2_{avg} / \Delta F1_{avg}$
8. $\Delta F2_{max}$ 99% percentile
9. F_0
10. F_n
11. F_0 - F_n
12. $F_1 – F_0$
13. $F_{n+5} – F_n$
14. F_{tx} @ ±2MHz offset
15. F_{tx} @ ≥±3MHz offset
16. Number of in-band emission exceptions
17. P_{avg} – CTE
18. Pk – P_{avg} – CTE
19. Fs_i
20. $Fs_1 – F_p$
21. Fs_i - F_0
22. $Fs_i – Fs_{i-3}$
23. $P_{ref,dev} / P_{ref,ave}$
24. $P_{n,dev} / P_{n,ave}$
25. The 8 1µs slots in the reference period of the supplemental and all subsequent sampling slots

as a function of one of:

1. RF channel
2. Modulation
3. Packet length group (non-CTE measurements)

or as a histogram showing the relative occurrence of different values.

When requesting statistical data from the Sapphire application, it is possible to qualify the request with the following criteria:

1. The modulation(s) pertaining to the statistics, which can be one or more of:
    a. 2 Mbps uncoded
    b. 1 Mbps, uncoded
    c. 1 Mbps, coded, S=2
    d. 1 Mbps, coded, S=8
2. Which set of RF channel numbers, which can be any combination of channels 0 to 39
3. Which packet length bin (non-CTE measurements) or CTE length and type (CTE measurements).

This exceptional flexibility as to how the statistical data is presented allows for rapid correlation of packet parameters with potential test failures.

4.5 Advertise/Scan Mode

4.5.1 Overview

The advertise/scan mode permits restricted testing of devices for which no test interface is available, for example, devices assembled into final products on a production line. In the advertise/scan mode, normal LE packets are used to force a response from the DUT so that the receiver performance can be checked and the transmission quality verified.

4.5.2 Modes of operation

There are two modes of operation:

1. Advertise. In this mode of operation, the Sapphire application sends out pre-defined advertising packets at a specified signal level and waits for scan or connection requests from the device under test. Sapphire can advertise on a single channel or multiple channels simultaneously.
2. Scan. In this mode the DUT advertises and the Sapphire application sends out scan or connection requests at a specified level to trigger a response from the DUT.

4.5.3 Capture port

The Sapphire application always transmits on the Tx/Rx port. The packets from the DUT can be received on either:

1. Tx/Rx port. This port has a noise figure of +46 dB and can handle input signals as high as +27 dBm. Use this port for conducted measurements.
2. Monitor In port. This port has a noise figure of +6 dB and a maximum usable input signal level of -10 dBm. Use this port for off-air measurements.

4.5.4 Selecting the correct DUT

In most environments multiple DUTs may be advertising or scanning. By default, the *TLF3000* will talk to any device it sees. To ensure that only the wanted device is tested, the *TLF3000* can be instructed to look for:

1. Devices with specific AdvA, InitA or ScanA addresses
2. Only packets which exceed an RSSI threshold
3. Only certain packet types

The *TLF3000* can generate a list of advertising devices in the vicinity and their associated RSSI. This can enable the current device to be found and the appropriate address and RSSI threshold set.

4.5.5 Capture criteria

All 3 LE primary advertising channels are monitored simultaneously by the *TLF3000's* receiver. Packets are captured for further analysis when they satisfy a range of criteria:

1. The packet must possess the advertising address
2. The packet must be modulated using 1 Mbps, uncoded
3. Scan responses, scan requests or connection requests must occur around the inter frame spacing
4. Scan responses, scan requests or connection requests must arrive on the associated advertising channel

4.5.6 Captured data and analysis

The captured data can be subject to a number of different tests:

1. Average and peak power measurements
2. Modulation characteristics
3. Carrier offset and drift measurements
4. In-band emissions

One or more of these tests may be selected.

The data used for the modulation characteristics, carrier offset, drift and supplemental analysis is 32x oversampled in full compliance with the Bluetooth LE RF PHY Test Specification. Many LE testers do not fulfil this requirement or simply fail to specify the oversampling used. This complete oversampling of the waveform is accomplished without any degradation in the overall test time.

It is also possible to capture the raw data of individual packets. This can be returned as:

1. IQ complex samples
2. Amplitude profile in dBm
3. Deviation profile

The oversampling ratio for this data can be 32x, 16x, 8x or 4x. Because of the large amounts of data to be transferred, turning on collection of raw waveform data increases the interval between the end of one packet and the start of the next packet which can be captured and analysed.

4.5.7 Capture termination

Capture termination can be caused by:

1. A stop capture command
2. A limit failure, provided the stop capture on limit failure flag has been set

For each test being conducted, appropriate limits can be downloaded to the Sapphire application. As each packet is analysed, the results are compared against the limits. If a limit failure is detected, then data capture will be terminated if the stop capture on limit failure flag has been set.

If raw waveform data collection has been enabled, and data collection is terminated by a limit failure, then the last waveform stored will be that corresponding to the limit failure. This facility permits the rogue waveform to be examined in detailed so that the exact cause of the limit failure can be understood. This can be particularly useful when observing real-world waveforms captured off-air.

4.5.8 Transmit power

The power at which the *TLF3000* unit transmits advertising packets of scan requests can be specified in the range -120 dBm to 0 dBm. It is also possible to specify a range of powers for the *TLF3000* transmissions. To specify a range of powers, a maximum power, minimum power and step size must be specified. The unit will then transmit *N* packets at the maximum power level, reduce the power level by step size and then transmit a further *N* packets, etc. A total of 32 power levels can be specified in this way. This facility can be used to estimate the sensitivity of the DUT.

4.5.9 Collected quantities

When the appropriate tests are enabled, the Sapphire application collects statistics on the following quantities defined within the Bluetooth 5 LE RF PHY Test Specification:

1. P_{avg}
2. Pk - P_{avg}
3. $\Delta F1_{max}$
4. $\Delta F1_{avg}$
5. $\Delta F2_{max}$
6. $\Delta F2_{avg}$
7. $\Delta F2_{avg} / \Delta F1_{avg}$
8. $\Delta F2_{max}$ 99% percentile
9. F_0
10. F_n
11. F_0 - F_n
12. $F_1 – F_0$
13. $F_{n+5} – F_n$
14. F_{tx} @ ±2 MHz (or ±4 MHz for 2 Mbps) offset
15. F_{tx} @ ≥±3 MHz (or ≥±5 MHz for 2 Mbps) offset
16. Number of in-band emission exceptions

For each of these quantities the following are stored:

1. Maximum value
2. Minimum value
3. Average value
4. Last value observed
5. Histogram of observed values
6. Number of samples analysed
7. Number of packets analysed

The following statistics are also held for in-band emissions:

1. Maximum value for each of the 81 1 MHz bands
2. Minimum value for each of the 81 1 MHz bands
3. Average value for each of the 81 1 MHz bands
4. Last observed value for each of the 81 1 MHz bands
5. Maximum value for each of the 810 100 kHz bands used in the computation
6. Minimum value for each of the 810 100 kHz bands used in the computation
7. Average value for each of the 810 100 kHz bands used in the computation
8. Last observed value for each of the 810 100 kHz bands used in the computation
9. Number of packets analysed

In addition, the number of Number of scan responses (scan mode) or scan and connection requests (advertise mode) received are also recorded.

Separate sets of statistics are maintained for each of:

1. Primary advertising channel number
2. *TLF3000* transmit power

4.5.10 Statistical analysis

Two types of statistical data can be requested from the Sapphire application:

1. Tabular statistics
2. Graphical statistics

Tabular statistics consist of a table showing the minimum, maximum, average and current value of various quantities for the selected measurement. Measurements which can be selected, and the returned quantities are:

Measurement set	**Returned quantities**
Output power	P_{avg} $Pk - P_{avg}$
Modulation characteristics	$\Delta F1_{max}$ $\Delta F1_{avg}$ $\Delta F2_{max}$ $\Delta F2_{avg}$ $\Delta F2_{avg} / \Delta F1_{avg}$ $\Delta F2_{max}$ 99% percentile
Drift and carrier offset	F_0 F_n $F_0 - F_n$ $F_1 – F_0$ $F_{n+5} – F_n$
In-band emissions	F_{tx} @ ±2 MHz (or ±4 MHz for 2 Mbps) offset F_{tx} @ ≥±3 MHz (or ≥±5 MHz for 2 Mbps) offset Number of in-band emission exceptions
Received packets	Number of received scan responses (scan mode) or number of received scan and connection requests (advertise mode)

Table 3: Measurements which can be selected and their returned quantities for advertise/scan mode

Graphical statistics consist of one of the following quantities:

1. P_{avg}
2. Pk - P_{avg}
3. $\Delta F1_{max}$
4. $\Delta F1_{avg}$
5. $\Delta F2_{max}$
6. $\Delta F2_{avg}$
7. $\Delta F2_{avg}/\Delta F1_{avg}$
8. $\Delta F2_{max}$ 99% percentile
9. F_0
10. F_n
11. F_0 - F_n
12. $F_1 – F_0$
13. $F_{n+5} – F_n$
14. F_{tx} @ ±2 MHz (or ±4 MHz for 2 Mbps) offset
15. F_{tx} @ ≥±3 MHz (or ≥ ±5 MHz for 2 Mbps) offset
16. Number of in-band emission exceptions
17. Number of received scan responses (scan mode) or scan and connection requests (advertise mode)

as a function of one of:

1. Primary advertising channel
2. *TLF3000* transmit power

or as a histogram showing the relative occurrence of different values, except for number of received responses.

When requesting statistical data from the Sapphire application, it is possible to qualify the request with the following criteria:

1. A combination of the 3 primary advertising channels
2. A selection of the *TLF3000* transmit powers

5 Python Interface

5.1 Overview

Support is provided for driving the *TLF3000* directly from Python on a Windows platform. This support is available for Pyhton 2.7 onwards. Both 32 bit and 64 bit versions are available.

5.1.1 Prerequistes

To import the sapphire module, include the location of the appropriate Sapphire library on the current path. Then import the following:

```
sys.path.append('D:/Users/timbo/build-PyFrontline_MSVC2017_32bit_2v7-Release')

from Frontline import MorephDevice, MorephSearch, MorephInterface, SapphireInterface,
vector_uchar, vector_float, vector_int16, vector_char
```

5.1.2 Connecting to the *TLF3000*

To connect to the *TLF3000* it is first necessary to create a search engine by calling MorephSearch.get(). Having obtained a search engine, the following operations should be performed:

1. Attach a callback to the search engine by invoking the callback() method. The callback will process the *TLF3000s* that are discovered by the search engine.
2. Start the search engine by invoking the start() method.

Whenever a *TLF3000* is discovered, the callback will be entered with a handle to the device and a flag to indicate whether it was discovered on USB or Ethernet. In the example code below, the callback appends the device to a list if it is a USB device or an Ethernet device.

In the example program, the main thread monitors the contents of the list. Whenever a new entry is placed on the list, the main thread obtains an interface to the device by invoking the MorephInterface() method. Having obtained an interface to the device, the friendly name, serial number and current IP address are interrogated. If these satisfy certain search criteria, then the required device has been found and opened, otherwise the entry is discarded from the list.

Once the required device has been found, the search engine is stopped by invoking the stop() method.

```
from __future__ import print_function
import sys, struct
from collections import namedtuple
from time import sleep

sys.path.append('D:/Users/timbo/build-PyFrontline_MSVC2017_32bit_2v7-Release')

from Frontline import MorephDevice, MorephSearch, MorephInterface, SapphireInterface,
vector_uchar, vector_float, vector_int16, vector_char

TLF3000Name = "My TLF3000"
TLF3000SerialNumber = 172
```

```
"""
Connect to first device found which matches either TLF3000Name or TLF3000SerialNumber
"""

def connect():
    tlf3000 = []

    ms = MorephSearch.get()

    def callback(m):
        if ( (m.type == MorephDevice.USB) | (m.type == MorephDevice.Eth) ):
            tlf3000.append(m)

    ms.set_callback(callback)
    ms.start()

    found = False
    while True:
        while len(tlf3000):
            try:
                transport = tlf3000[0].getTransport();
                mi = MorephInterface(transport)
                name = mi.getFriendlyName()
                sn = mi.getSerialNumber()
                net = mi.getNetwork()
                print( "Found" , name , sn , hex(net.cur_ip) )
                if( (name == TLF3000Name) | (sn == TLF3000SerialNumber) ):
                    found = True
                    break
                del tlf3000[0]
                break
            except:
                del tlf3000[0]
        if found:
            break
        sleep(0.1)

    print( "Stop" )
    ms.stop()

    print("Opening TLF3000" )

    return transport
```

5.1.3 Launching the Sapphire application

The Sapphire application is launched by calling SapphireInterface(). This returns an interface to the Sapphire application.

A callback should be attached to handle asynchronous data from the *TLF3000*. This is done uiing the setDataCallback method.

```
    transport = connect()                           # Connect to the TLF3000 unit
    sapphire = SapphireInterface(transport)         # Launch the Sapphire application
    sapphire.setDataCallback(dataCallback,False)    # Register callback for asynch data
```

5.1.4 Handling asynchronouos data

Whenever an asynchronous message is received from the *TLF3000*, the asynchronous data callback will be entered. In the example below, the data is unpacked to determine its length and type. Having determined the message type, it is dispatched to an appropriate routine.

```
"""
Main callback for handling messages on the asynch data channel
These are then dispatched to the next level callback handler
"""

pktType = {
    0: startScript,
    1: txTest,
    2: rxTest,
    3: endScript,
    4: startLine,
    5: endLine,
    6: advScan,
    7: advScan,
    10: uartSearch,
    16: siganaLimFail,
    40: dutProperties,
    50: extraTestPower,
    51: extraTestInitialFreq,
    64: siggenPER,
    65: phytstEnded,
    66: advscanSniff,
    67: siggenEnded,
    96: antInfo,
    128: lostDUT,
    129: haveDUT,
    130: dutTxed,
    131: dutAsync,
    193: envData,
    194: pwrData}

pkthdr = namedtuple('hdr', 'n nMsb typ')
pkthdrbin = '<HBB'

def dataCallback(d):
    s = pkthdr(*struct.unpack(pkthdrbin, d[0:4]))
    pktType.get( s.typ )(d)
```

5.1.5 Handling errors

Many of the calls to the Sapphire library return a flag to indicate wtheter the call was successful or not. When an error occurs the cause of the error can be read back by using the Sapphire interface DispErrors() method. Each invocation of this method will remove the oldest error from the Sapphire error queue and return it to the caller. When no more messages are available, an empty string is returned.

```
"""
Routine for retrieving errors messages from TLF3000 and then exiting
"""

def getError(s):
    msg = s.DispErrors()
    while( msg != "" ):
        print( msg )
        msg = s.DispErrors()
```

5.1.6 Closing down

The Sapphire application can be shutdown by invoking the Sapphire interface exitApp() method..

```
sapphire.exitApp()          # Close the sapphire applications
sapphire = 0
transport = 0
```

5.2 Examples

5.2.1 Running a phy test script

The example demonsrates the sequence of instruction necessary to download and run a phy test script. It also demonstrates how the messages on the asynchronous data channel can be handled. This is a useful starting point for code running on a production line or characterisation rig.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. The Sapphire application is launched using the code described in Launching the Sapphire application
3. A callback for asynchronous messages is registered using the code described inHandling asynchronouos data
4. The operating mode is set phy test mode using setMode
5. The receiver frontend attenuation is set using setRxAtten
6. The receiver port to be used is set using setRxPort
7. The source of the IO voltage which will be used for the UART communication with the DUT is set using setDIOVolts
8. The meaning of the digital output lines is established using setPhyTestDIO. For many applications this call can be omitted.
9. The cable loss between the DUT and the *TLF3000* unit is set using setCableLoss
10. Details of the UART communications between the *TLF3000* unit and the DUT are set using setDUTPort and setDUTComms
11. The supported features of the DUT are programmed using setDUTProperties
12. A test script is then downloaded to the *TLF3000* unit. In this case the test script is read in from a .sta file which has been exported from the Sapphire GUI. This is the most convenient means of generating test scripts. However, test scripts can be generated within the Python code, the format of the test script being set out in Test Script Format. The test script is downloade using a startScript command, followed by a series of contScript script commands and a final endScript
13. Execution of the test script is triggered by issuing a runScript command
14. The code then waits for the test script to terminate. During this time a sequence of messages appear on the asynchronous data channel. The callback handles each of these messages and dispactches them to the appropriate routine which then prints out the test results.

```
"""
Main script for phy test example
"""

if __name__ == '__main__':
    global done
    done = False

    transport = connect()                                   # Connect to the TLF3000 unit
    sapphire = SapphireInterface(transport)                 # Launch the Sapphire application
    sapphire.setDataCallback(dataCallback,False)            # Register callback for asynch data channel

    if( not sapphire.setMode( 0 ) ):                        # Enter phy tester mode
        getError(sapphire)

    if( not sapphire.setRxAtten( 0 ) ):                     # Set rx frontend attenuation in units of 0.5dB
        getError(sapphire)

    if( not sapphire.setRxPort( 1 ) ):                      # Set the rx port to be tx/rx
        getError(sapphire)

    if( not sapphire.setDioVolts( True ) ):                 # Turn on 3.3V IO supply from TLF3000
        getError(sapphire)

    if( not sapphire.setPhyTestDIO( 0, 0, 0, 0 ) ):         # Program DIOs to indicate run/stop pass/fail
        getError(sapphire)

    if( not sapphire.setCableLoss( 0 ) ):                   # Set cable loss in units of 0.1dB
        getError(sapphire)

    if( not sapphire.setDUTPort( 5 ) ):                     # Set which comport to use
        getError(sapphire)                                  # negative values imply direct connection of DUT to TLF3000

                                                            # Serial communications to DUT
                                                            # Arg 1 : Transport
                                                            #         0 = direct test mode
                                                            #         1 - H4
                                                            #         2 = H5
                                                            #         3 = BCSP
                                                            # Arg 2 : Baud rate
                                                            # Arg 3 : h/w flow control
                                                            # Arg 4 : s/w flow control
                                                            # Arg 5 : stop bits
                                                            # Arg 6 : parity : 0 = none, 1 = even, 2 = odd
                                                            # Arg 7 : crc present : 0 = no
```

```
    if( not sapphire.setDUTComms( 0 , 19200, 0, 0, 1, 0, 0 ) ):
        getError(sapphire)
                                                        # DUT supported features
                                                        # Arg 1 : Role
                                                        #         0 = central
                                                        #         1 = peripheral
                                                        #         2 = broadcaster
                                                        #         3 = observer
                                                        # Arg 2 : Support for advertising extensions
                                                        # Arg 3 : Support for data length extensions
                                                        # Arg 4 : Max supported advertising octets
                                                        # Arg 5 : Max supported rx octets
                                                        # Arg 6 : Max supported rx time
                                                        # Arg 7 : Max supported tx octets
                                                        # Arg 8 : Max supported tx time
                                                        # Arg 9 : Support for 2Mbps
                                                        # Arg 10: Support for coded phys
                                                        # Arg 11: Support for stable modulation index
                                                        # Arg 12: Support for Rx AoA 2us slots
                                                        # Arg 13: Support for Tx AoA 2us slots
                                                        # Arg 14: Support for Rx AoD 2us slots
                                                        # Arg 15: Support for Tx AoD 2us slots
                                                        # Arg 16: Support for Rx AoA 1us slots
                                                        # Arg 17: Support for Tx AoA 1us slots
                                                        # Arg 18: Support for Rx AoD 1us slots
                                                        # Arg 19: Support for Tx AoD 1us slots
                                                        # Arg 20: Maximum CTE length in 8us slots
                                                        # Arg 21: Number of antenna
                                                        # Arg 22: Maximum switching pattern length
    if( not sapphire.setDUTProperties(0, True, True, 255, 251, 17040, 251, 17040, False, False, False,
                                      False, False, False, False, False, False, False, False, 0, 0, 0 ) ):
        getError(sapphire)

                                                        # Start download of test script
                                                        # Arg 1 : Overwrite any file with existing name
                                                        # Arg 2 : Do not place in FLASH (currently not supported)
                                                        # Arg 3 : Test script file name
    if( not sapphire.startScript( True , False , "TestScript" ) ):
        getError(sapphire)

    n = 0
    with open( TestScriptFile , "r" ) as f:
        for line in f:
            n = n + 1
            if( n > 5 ):                                # Skip header lines in .sta file
```

```
            if( not sapphire.contScript(line) ):        # Add lines to test script file
                getError(sapphire)
                quit()

if( not sapphire.endScript() ):                         # Signal that the test script is now complete
    getError(sapphire)

f.close()
                                                        # Run the test script
                                                        # Arg 1 : Number of times script should be run
                                                        # Arg 2 : Run to completion
                                                        # Arg 3 : Reduce test time by aborting rx tests early
                                                        # Arg 4 : Reorder tx/rx tests to reduce test time
                                                        # Arg 5 : Test script not in FLASH
                                                        # Arg 6 : Test script file nane - same as in startScript
if( not sapphire.runScript( 1, True, True, True, False, "TestScript" ) ):
    getError(sapphire)

while( not done ):                                      # Loop until test script terminates
    msg = sapphire.DispErrors()
    if( msg != "" ):
        print( msg )
    else:
        sleep(0.1)

getError(sapphire)

sapphire.exitApp()                                      # Close the sapphire applications

sapphire = 0
transport = 0
```

5.2.1.1 Test progress

The progress of the testing is reported via asynchronous callbacks. Callbacks are generated whenever:

1. A new script is started
2. A new line in a script is started
3. A line in a script is completed, with an indication of pass or fail
4. A script is completed, with an indication of pass or fail

```
"""
callbacks for handling script/line start/end messages
"""

script = namedtuple('script', 'hdr unused id ok')
scriptbin = '<LHBB'

def startScript( d ):
    s = script(*struct.unpack(scriptbin, d[0:8]))
    print( "Test script execution starting : id" , s.id )

def endScript( d ):
    global done
    s = script(*struct.unpack(scriptbin, d[0:8]))
    print( "Test script execution terminating : id" , s.id )
    if( s.ok == 0 ):
        print( "Test script result : PASS" )
    else:
        print( "Test script result : FAIL" )
    done = True

startline = namedtuple('startline', 'hdr line id num')
startlinebin = '<LHBB'

def startLine( d ):
    s = startline(*struct.unpack(startlinebin, d[0:8]))
    print( "Starting test script line %d" % s.line )

doneline = namedtuple('doneline', 'hdr line id num result')
donelinebin = '<LHBBL'

def endLine( d ):
    s = doneline(*struct.unpack(donelinebin, d[0:12]))
    print( "End test script line %d" % s.line , end = "" )
    if( s.result == 0 ):
        print( " : FAIL" )
    else:
        print( " : PASS" )
```

5.2.1.2 Tables for decoding results

The example code utilises the following tables to decode the test results returned.

```
"""
Table to indicate which tests are using stable modulation index
"""
txStable = {
        1: 0,
        3: 0,
        5: 0,
        6: 0,
        8: 0,
        9: 1,
        10: 0,
        11: 1,
        12: 0,
        13: 0,
        14: 0,
        90: 0,
        91: 0,
        92: 0,
        93: 0,
        94: 0
        }
rxStable = {
        1: 0,
        3: 0,
        4: 0,
        5: 0,
        6: 0,
        7: 0,
        8: 0,
        9: 0,
        10: 0,
        11: 0,
        12: 0,
        13: 0,
        14: 1,
        15: 1,
        16: 1,
        17: 1,
        18: 1,
        19: 1,
        20: 1,
        21: 1,
        22: 1,
        23: 1,
        24: 1,
        25: 1,
        26: 0,
        27: 0,
        28: 0,
        29: 0,
        30: 0,
        31: 0,
        32: 1,
        33: 1,
        34: 1,
        35: 1,
        36: 1,
        37: 1
        }
```

```
"""
Text strings for output
"""

phys = [ "2Mbps uncoded" , "1Mbps uncoded" , "500kbps coded s=2" , "125kbps coded s=8"
]

stables = [ "standard modulation index" , "stable modulation index" ]
```

5.2.1.3 Transtmitter test results

Transmitter test results are directed through a single routine called by the main asynchronous callback handler. The nature of the transmitter test is determined and an appropriate routine for handling the asynchronous data is called.

```
"""
Dictionary to map test numbers to callbacks
"""

txType = {
        1: outputPower,
        3: inBandEmissions,
        5: modulationCharacteristics,
        6: carrierDrift,
        8: inBandEmissions,
        9: modulationCharacteristics,
        10: modulationCharacteristics,
        11: modulationCharacteristics,
        12: carrierDrift,
        13: modulationCharacteristics,
        14: carrierDrift,
        90: outputPower,
        91: outputPower,
        92: outputPower,
        93: inBandEmissions,
        94: inBandEmissions
        }

"""
Callbacks for handling transmit messages
These are passed to the appropraite message handler
"""

txrx = namedtuple('txrx', 'hdr line id num')
txrxbin = '<LHBB'

def txTest( d ):
    s = txrx(*struct.unpack(txrxbin, d[0:8]))
    txType.get(s.num)(d)
```

5.2.1.3.1 Transmitter output power results

Measurements related to output power are decoded by the following routine:

```
"""
Callback for tx power messages
"""

power = namedtuple('power', 'hdr line id num chan phy pktlen Pmin Pmax Pkmax result
frame')
powerbin = '<LHBBLLLfffLL'

def outputPower(d):
    s = power(*struct.unpack(powerbin, d))
    print( "    Output Power" , end = "" )
    print( " : " , phys[s.phy] , stables[txStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Min power  : %.1f dBm" % s.Pmin )
    print( "        Max power  : %.1f dBm" % s.Pmax )
    print( "        Peak power : %.1f dB" % s.Pkmax )
    if( s.result ):
        print( "        Result     : FAIL" )
    else:
        print( "        Result     : PASS" )
```

5.2.1.3.2 Transmitter in-band emission results

Measurements related to in-band emissions are decoded by the following routine. The object "spectrum" holds the results of the in-band emission measurements on 81 channels in units of dBm. The first channel is 2400MHz and the last channel is 2480MHz.

```
"""
Callback for tx in-band emission messages
"""

inband = namedtuple('inband', 'hdr line id num chan phy pktlen P1max P2max
maxXcep numXcep result')
inbandbin = '<LHBBLLLfffLL'

def inBandEmissions(d):
    print( "In-band emissions" )
    s = inband(*struct.unpack(inbandbin, d[0:40]))
    print( "    In-band emissions" , end = "" )
    print( " : " , phys[s.phy] , stables[txStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    if( s.phy == 0 ):
        print( "        Ptx +/- 3MHz         : %.1f dBm" % s.P1max )
        print( "        Ptx >= +/- 4MHz      : %.1f dBm" % s.P2max )
    else:
        print( "        Ptx +/- 2MHz         : %.1f dBm" % s.P1max )
        print( "        Ptx >= +/- 3MHz      : %.1f dBm" % s.P2max )
    print( "        Maximum exception    : %.1f dBm" % s.maxXcep )
    print( "        Number of exceptions : %d" % s.numXcep )
    if( s.result & 1 ):
        print( "        Result               : FAIL" )
    else:
        print( "        Result               : PASS" )
    spectrum = struct.unpack('<81f', d[40:364])
```

5.2.1.3.3 Transmitter modulation characteristics results

Measurements related to modulation characteristics are decoded by the following routine:

```
"""
Callback for tx modulation characteristics messages
"""

modchar = namedtuple('modchar', 'hdr line id num chan phy pktlen DF1min DF1max
minRatio frac result frame')
modcharbin = '<LHBBLLLffffLL'

def modulationCharacteristics(d):
    s = modchar(*struct.unpack(modcharbin, d))
    print( "    Modulation characteristics" , end = "" )
    print( " : " , phys[s.phy] , stables[txStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        minimum Delta F1max  : %.1f kHz" % s.DF1min )
    print( "        maximum Delta F1max  : %.1f kHz" % s.DF1max )
    print( "        min DeltaF2/DeltaF1  : %.3f" % s.minRatio )
    if( s.phy == 0 ):
        print( "        DeltaF2 > 370kHz     : %.3f %%" % (100*s.frac) )
    else:
        print( "        DeltaF2 > 185kHz     : %.3f %%" % (100*s.frac) )
    if( s.result ):
        print( "        Result               : FAIL" )
    else:
        print( "        Result               : PASS" )
```

5.2.1.3.4 Transmitter carrier frequency offset and drift results

Measurements related to carrier frequency offset and drift are decoded by the following routine:

```
"""
Callback for carrier offset & drift messages
"""

carDrift = namedtuple('carDrift', 'hdr line id num chan phy pktlen minFo maxFo minFn
maxFn minFoFn maxFoFn minF1Fo maxF1Fo minFnFm maxFnFm result frame')
carDriftbin = '<LHBBLLLfffffffffLL'

def carrierDrift(d):
    s = carDrift(*struct.unpack(carDriftbin, d))
    print( "    Carrier offset & drift" , end = "" )
    print( " : " , phys[s.phy] , stables[txStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        minimum Fo         : %.1f kHz" % s.minFo )
    print( "        maximum Fo         : %.1f kHz" % s.maxFo )
    print( "        minimum Fn         : %.1f kHz" % s.minFn )
    print( "        maximum Fn         : %.1f kHz" % s.maxFn )
    print( "        minimum Fo - Fn    : %.1f kHz" % s.minFoFn )
    print( "        maximum Fo - Fn    : %.1f kHz" % s.maxFoFn )
    print( "        minimum F1 - Fo    : %.1f kHz" % s.minF1Fo )
    print( "        maximum F1 - Fo    : %.1f kHz" % s.maxF1Fo )
    print( "        minimum Fn - Fn-5  : %.1f kHz" % s.minFnFm )
    print( "        maximum Fn - Fn-5  : %.1f kHz" % s.maxFnFm )
    if( s.result ):
        print( "        Result             : FAIL" )
    else:
        print( "        Result             : PASS" )
```

5.2.1.4 Receiver test results

Receiver test results are directed through a single routine called by the main asynchronous callback handler. The nature of the receiver test is determined and an appropriate routine for handling the asynchronous data is called.

```
"""
Dictionary to map test numbers to callbacks
"""

rxType = {
        1: sensitivity,
        3: ci,
        0x83: ciXcep,
        4: blocking,
        0x84: blockingXcep,
        5: intermodulation,
        6: maxinput,
        7: integrity,
        8: sensitivity,
        9: ci,
        0x89: ciXcep,
        10: blocking,
        0x8A: blockingXcep,
        11: intermodulation,
        12: maxinput,
        13: integrity,
        14: sensitivity,
        15: ci,
        0x8F: ciXcep,
        16: blocking,
        0x90: blockingXcep,
        17: intermodulation,
        18: maxinput,
        19: integrity,
        20: sensitivity,
        21: ci,
        0x95: ciXcep,
        22: blocking,
        0x96: blockingXcep,
        23: intermodulation,
        24: maxinput,
        25: integrity,
        26: sensitivity,
        27: sensitivity,
        28: ci,
        0x9C: ciXcep,
        29: ci,
        0x9D: ciXcep,
        30: integrity,
        31: integrity,
        32: sensitivity,
        33: sensitivity,
        34: ci,
        0xA2: ciXcep,
        35: ci,
        0xA3: ciXcep,
        36: integrity,
        37: integrity
        }
```

```
"""
Callbacks for handling receive messages
These are passed to the appropraite message handler
"""

def rxTest( d ):
    s = txrx(*struct.unpack(txrxbin, d[0:8]))
    rxType.get(s.num)(d)
```

5.2.1.4.1 Receiver sensitivity results

Measurements related to receiver sensitivity are decoded by the following routine:

```
"""
Callback for rx sensitivity messages
"""

sens = namedtuple('sens', 'hdr line id num chan phy pktlen wPwr numPkt rxPkt PER
result frame')
sensbin = '<LHBBLLLiLLfLL'

def sensitivity(d):
    s = sens(*struct.unpack(sensbin, d))
    print( "    Rx sensitivity" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level : %.1f dBm" % (0.1*s.wPwr))
    print( "        Transmitted packets : %d" % s.numPkt )
    print( "        Received packets    : %d" % s.rxPkt )
    print( "        PER                 : %.1f %%" % (100*s.PER) )
    if( s.result ):
        print( "        Result              : FAIL" )
    else:
        print( "        Result              : PASS" )
```

5.2.1.4.2 Receiver C/I results

Measurements related to receiver C/I measurements are decoded by the following two routines. One routine handles C/I measurements and the other handles the number of C/I exceptions for the entire C/I sweep across channels.

```
"""
Callback for rx C/I messages
"""

ciblock = namedtuple('sens', 'hdr line id num chan phy pktlen wPwr iFreq iPwr numPkt
rxPkt PER result frame')
ciblockbin = '<LHBBLLLiLiLLfLL'

def ci(d):
    s = ciblock(*struct.unpack(ciblockbin, d))
    print( "    C/I" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level     : %.1f dBm" % (0.1*s.wPwr))
    print( "        Interferer frequency    : %d MHz" % s.iFreq )
    print( "        Interferer signal level : %.1f dBm" % (0.1*s.iPwr))
    print( "        Transmitted packets     : %d" % s.numPkt )
    print( "        Received packets        : %d" % s.rxPkt )
    print( "        PER                     : %.1f %%" % (100*s.PER) )
    if( s.result & 1 ):
        print( "        Result                  : FAIL" )
    elif( s.result == 0 ):
        print( "        Result                  : PASS" )
    else:
        print( "        Result                  : EXCEPTION" )


"""
Callback for rx C/I exception messages
"""

cixcep = namedtuple('sens', 'hdr line id num chan phy pktlen wPwr nXcep result frame')
cixcepbin = '<LHBBLLLiLLL'

def ciXcep(d):
    s = cixcep(*struct.unpack(cixcepbin, d))
    print( "    C/I exceptions" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num & 0x7F)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Number of exceptions    : %d" % s.nXcep )
    if( s.result ):
        print( "        Result                  : FAIL" )
    else:
        print( "        Result                  : PASS" )
```

5.2.1.4.3 Receiver blocking results

Measurements related to receiver blocking performance are decoded by the following two routines. The first routine handles individual blocking measurements, whilst the second handles the overall number of blocking exceptions.

```
"""
Callback for rx blocking messages
"""

def blocking(d):
    s = ciblock(*struct.unpack(ciblockbin, d))
    print( "    Blocking" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level  : %.1f dBm" % (0.1*s.wPwr))
    print( "        Blocker frequency    : %d MHz" % s.iFreq )
    print( "        Blocker signal level : %.1f dBm" % (0.1*s.iPwr))
    print( "        Transmitted packets  : %d" % s.numPkt )
    print( "        Received packets     : %d" % s.rxPkt )
    print( "        PER                  : %.1f %%" % (100*s.PER) )
    if( s.result & 1 ):
        print( "        Result               : FAIL" )
    elif( s.result & 2 ):
        print( "        Result               : EXCEPTOPM - level 1" )
    elif( s.result & 4 ):
        print( "        Result               : EXCEPTOPM - level 2" )
    else:
        print( "        Result               : PASS" )

"""
Callback for rx blocking exception messages
"""

blockxcep = namedtuple('sens', 'hdr line id num chan phy pktlen wPwr nXcep mXcep
result frame')
blockxcepbin = '<LHBBLLLiLLLL'

def blockingXcep(d):
    s = blockxcep(*struct.unpack(blockxcepbin, d))
    print( "    Blocking exceptions" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num & 0x7F)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Level 1 exceptions   : %d" % s.nXcep )
    print( "        Level 2 exceptions   : %d" % s.mXcep )
    if( s.result ):
        print( "        Result               : FAIL" )
    else:
        print( "        Result               : PASS" )
```

5.2.1.4.4 Receiver intermodulation results

Measurements related to the receiver intermodulation performance are handled by the following routine:

```
"""
Callback for rx intermodulation messages
"""

intermod = namedtuple('intermod', 'hdr line id num chan phy pktlen wPwr n side iPwr
numPkt rxPkt PER result frame')
intermodbin = '<LHBBLLLiLLiLLfLL'

def intermodulation(d):
    s = intermod(*struct.unpack(intermodbin, d))
    print( "    Intermodulation" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level     : %.1f dBm" % (0.1*s.wPwr))
    print( "        Intermodulation spacing : %d" % s.n )
    if( s.side == 0 ):
        print( "        Interferers are below wanted signal" )
    else:
        print( "        Interferers are above wanted signal" )
    print( "        Interferer powers       : %.1f dBm" % (0.1*s.iPwr) )
    print( "        Transmitted packets     : %d" % s.numPkt )
    print( "        Received packets        : %d" % s.rxPkt )
    print( "        PER                     : %.1f %%" % (100*s.PER) )
    if( s.result ):
        print( "        Result                  : FAIL" )
    else:
        print( "        Result                  : PASS" )
```

5.2.1.4.5 Receiver maximum input results

Measurements related to the receiver maximum input signal level are handled by the following routine:

```
"""
Callback for rx max input messages
"""

def maxinput(d):
    s = sens(*struct.unpack(sensbin, d))
    print( "    Rx max input" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level : %.1f dBm" % (0.1*s.wPwr))
    print( "        Transmitted packets : %d" % s.numPkt )
    print( "        Received packets    : %d" % s.rxPkt )
    print( "        PER                 : %.1f %%" % (100*s.PER) )
    if( s.result ):
        print( "        Result              : FAIL" )
    else:
        print( "        Result              : PASS" )
```

5.2.1.4.6 Receiver PER integrity results

Measurements related to the receiver PER integrity reporting are handled by the following routine:

```
"""
Callback for rx PER report integrity messages
"""

integ = namedtuple('integ', 'hdr line id num chan phy pktlen wPwr rep numPkt rxPkt PER
result frame')
integbin = '<LHBBLLLiLLLfLL'

def integrity(d):
    s = integ(*struct.unpack(integbin, d))
    print( "    Integrity report" , end = "" )
    print( " : " , phys[s.phy] , stables[rxStable.get(s.num)] , end = "" )
    print( " : RF Channel " , s.chan , end = "" )
    print( " : Packet length " , s.pktlen )
    print( "        Wanted signal level : %.1f dBm" % (0.1*s.wPwr))
    print( "        Repeat number       : %d" % s.rep )
    print( "        Transmitted packets : %d" % s.numPkt )
    print( "        Received packets    : %d" % s.rxPkt )
    print( "        PER                 : %.1f %%" % (100*s.PER) )
    if( s.result ):
        print( "        Result              : FAIL" )
    else:
        print( "        Result              : PASS" )
```

5.2.2 Programming the signal generator

This example code demonstrates how to program the various signal generator sources and then turn on the signal generator output.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. The Sapphire application is launched using the code described in Launching the Sapphire application
3. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
4. The operating mode is set to signal generator mode using setMode
5. Set the cable loss using setCableLoss
6. Vectors are created to hold the dirty transmitter table. In this example, the dirty transmitter table contains two rows.
7. The wanted signal is programmed using setWanted. At this stage there is not output from the signal generator since it has not been started.
8. An interferer signal is programmed using setInterferer
9. An in-band CW blocker is programmed using setInBandCW
10. An AWGN source is enabled using setAWGN
11. An out–of-band CW blocker is programmed using setOutOfBandCW
12. The signal generator is started by calling startStopSigGen. It is only at this point that the signal generator starts outputting power.
13. After 5 seconds the signal generator is stopped by calling stopSigGen. This does not affect the programming which has previously been performed.

```
"""
Main script for controlling signal generator
"""

if __name__ == '__main__':
    global done
    done = False
    transport = connect()                                   # Connect to the TLF3000 unit
    sapphire = SapphireInterface(transport)                 # Launch the Sapphire application
    transport.setDataCallback(dataCallback,False)           # Register callback for asynch data channel

    if( not sapphire.setMode( 1 ) ):                        # Enter sig gen tester mode
        getError(sapphire)

    if( not sapphire.setCableLoss( 0 ) ):                   # Set cable loss in units of 0.1dB
        getError(sapphire)


                                                            # Distortion table for wanted signal
                                                            # carrier offset in kHZ
                                                            # modulation x 10000
                                                            # drift magnitude in kHz
                                                            # drift rate in Hz
                                                            # symbol timing error in ppm
    distortions = vector_int16( array.array( "h" ,
        ( 30, 4000, 20, 100, 30,
         -30, 5000, 20, 100, -30 ) ) )


                                                            # Program wanted signal
                                                            # Arg1 : Set wanted signal on/off
                                                            # Arg2 : RF channel
                                                            # Arg3 : output power
                                                            # Arg4 : phy 0 = 2Mbps, 1 = 1Mbps, 2 = S=2, 3 = S=8
                                                            # Arg5 : access address
                                                            # Arg6 : crc seed
                                                            # Arg7 : payload type
                                                            #    0  = PRBS9
                                                            #    1  = PRBS11
                                                            #    2  = PRBS15
                                                            #    3  = PRBS20
                                                            #    4  = PRBS23
                                                            #    5  = PRBS29
                                                            #    6  = PRBS31
                                                            #    7  = 0F0F0F0F
                                                            #    8  = 55555555
                                                            #    9  = FFFFFFFF
```

```
                                                    #   10 = 00000000
                                                    #   11 = F0F0F0F0
                                                    #   12 = AAAAAAAA
                                                    # Arg 8 : payload length in octets
                                                    # Arg 9 : interval between start of packets in us
                                                    # Arg 10: number of packets, 0 = continuous
                                                    # Arg 11: DIO mask for packet transmission
                                                    # Arg 12: distortion table

if( not sapphire.setWanted( True, 19, -30.0, 1, 0x71764129, 0x555555, 8, 37, 625, 1500, 0, distortions) ):
    getError(sapphire)

                                                    # Program interferer
                                                    # Arg 1 : Set interferer on/off
                                                    # Arg 2 : Frequency in MHz
                                                    # Arg 3 : Output power in dBm
                                                    # Arg 4 : phy, 0 = 2MBps, 1 = 1Mbps, 2 = S=2, 3 = S=8
                                                    # Arg 5 : payload type
                                                    #   0  = PRBS9
                                                    #   1  = PRBS11
                                                    #   2  = PRBS15
                                                    #   3  = PRBS20
                                                    #   4  = PRBS23
                                                    #   5  = PRBS29
                                                    #   6  = PRBS31
                                                    # Arg 6 : DIO mask for interferer active
                                                    # Arg 7 : continuous = 1, packetised transmission = 0
                                                    # Arg 8 : number of payload octets
                                                    # Arg 9 : interval between start of packets in us
if( not sapphire.setInterferer(True, 2405, -40.0, 1, 6, 0, 0, 255, 2500) ):
    getError(sapphire)

                                                    # Program in band CW source
                                                    # Arg 1 : Set CW source on/off
                                                    # Arg 2 : number of CW source (0 or 1)
                                                    # Arg 3 : frequency in Hz
                                                    # Arg 4 : power in dBm
if( not sapphire.setInBandCW( True, 0, 2479000000, -15.0 ) ):
    getError(sapphire)

                                                    # Program the AWGN source
                                                    # Arg 1 : Set AWGN source on/off
                                                    # Arg 2 : power in dBm/MHz
if( not sapphire.setAWGN( True , -80.0 ):
    getError(sapphire)
```

```
                                                    # Program out of band CW source
                                                    # Arg 1 : Set CW source on/off
                                                    # Arg 2 : frequency in MHz
                                                    # Arg 3 : power in dBm
if( not sapphire.setOutOfBandCW( True, 3000, -35.0 ) ):
    getError(sapphire)

                                                    # Start signal generator
                                                    # 0 - stop, 1 = start
if( not sapphire.startStopSigGen( 1 ) ):
    getError(sapphire)

sleep(30)

if( not sapphire.startStopSigGen( 0 ) ):
    getError(sapphire)

getError(sapphire)

sapphire.exitApp()                                  # Close the sapphire applications

sapphire = 0
transport = 0
```

5.2.2.1 Completion of transmission of wanted signal

When the transmission of the wanted signal is complete, an asynchronous callback is made. This is handled by the following routine:

```
def siggenDone( d ):
    print( "Signal generator has sent packets" )
```

5.2.3 Analysing waveforms and obtaining IQ data using the signal analyser

This example code demonstrates how to program the signal analyser and instruct it to run. Various results are then retrieved from the signal analyser. The power profile, FM deviation and IQ data of a waveform are then plotted.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. The Sapphire application is launched using the code described in Launching the Sapphire application
3. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
4. The operating mode is set to signal analyser mode using setMode
5. The receiver port to be used is set using setRxPort
6. The receiver frontend attenuation is set using setRxAtten
7. The cable loss is set using setCableLoss
8. The signal analayser is set running using startStopSigAna. In this instance it is programmed to pickup off-air advertising packets.
9. After 1 second the signal analyser is halted using stopSigAna. It is not actually neceassry to stop the signal analyser running before reading back results.
10. pollResultsTable is used to recover the modulation characteristics results table and then further calls are used to receover the carrier frequency and drift results table and the in-band emissions results table. All these tables are printed out.
11. A pollResultsPlot call is made to capture waveform data. The meta data associated with the captured data is then read back and printed out using getResultsPlotMeta
12. Two calls are made to getResultsPlotInt16 to recover the waveform data, firstly as an amplitude series and then as a deviation series. These data are plotted.
13. A third call to recover the stored data is then made using getResultsPlotFloat which returns the IQ complex data. This is also plotted.

```
phys = (
        "2Mbps uncoded",
        "1Mbps uncoded",
        "Coded S=2",
        "Coded S=8"
        )

PowerHeadings = (
        "Pavg",
        "Pk-Pavg"
        )

ModulationHeadings = (
        "DF1max",
        "DF1avg",
        "DF2max",
        "DF2avg",
        "DF2avg/DF1avg",
        "DF2max 99.9%"
        )

DriftHeadings = (
        "Fo",
        "Fn",
        "|F1-Fo|",
        "|Fo-Fn|",
        "|Fn-Fn-5|"
        )

InBandHeadings = (
        "Ftx+/-2MHz",
        "Ftx+/-(3+n)MHz",
        "# exceptions",
        "Max exception"
        )

"""
routine for printing results tables
"""

def printTable( headings , x ):
    print( "" )
    print( "%20s %10s %10s %10s %10s %10s" % ( "Quantity" , "# Pkts" , "Min" , "Max" , "Avg" , "Current" ))
    n = 0;
    for h in headings:
```

```
        print( "%20s " % h , end = "" )
        print( "%10d %10.1f %10.1f %10.1f %10.1f" % ( x[n], x[n+1], x[n+2], x[n+3], x[n+4] )  )
        n = n + 5
    print( "" )


"""
Main script for driving signal analyser
"""

if __name__ == '__main__':
    transport = connect()                                   # Connect to the TLF3000 unit
    sapphire = SapphireInterface(transport)                 # Launch the Sapphire application
    transport.setDataCallback(dataCallback,False)           # Register callback for asynch data channel

    if( not sapphire.setMode( 2 ) ):                        # Enter sig ana tester mode
        getError(sapphire)


                                                            # Select input port
                                                            # 0 - MONITOR IN
                                                            # 1 - Tx/Rx
    if( not sapphire.setRxPort( 1 ) ):
        getError(sapphire)

    if( not sapphire.setCableLoss( 0 ) ):                   # Set cable loss in units of 0.1dB
        getError(sapphire)


                                                            # Start the signal analyser
                                                            # Arg1: run (True), stop (False)
                                                            # Arg2: access address to analyse
                                                            # Arg3: bit mask of channels to analyse
                                                            #   bitn = RF channel number n
                                                            # Arg4: bit mask of tests to perform
                                                            #
                                                            # Arg5: packet header mask
                                                            # Arg6: bit mask of phys to process
                                                            #   bit0 = 2Mbps uncoded
                                                            #   bit1 = 1Mbps uncoded
                                                            #   bit2 = s=2 coded
                                                            #   bit3 = s=8 coded
                                                            # Arg7: dehitten packet header (True)
                                                            #       don't dewhitten packet header (False)
                                                            # Arg8: off-air processing (True)
                                                            #       processing as per specification (False)
                                                            # Arg9: oversampling rate for waveform readback
```

```
                                                        # Arg10: stop on limit failure (True)
                                                        #        don't stop on limit failure (False)
                                                        # Arg11: packet length mask
                                                        #   bit0 = packet lengths
                                                        # Arg12: RSSI threshold for packets in dBm
    if( not sapphire.startStopSigAna( True , 0x71764129 , 0xFFFFFFFFFF , 0x11F , 0xFFFF , 0xF , False , True , 7 ,
False , 0xFFFFFFFF , -120 ) ):
        getError(sapphire)


    """
    Wait for some data to be colelected
    """

    sleep(2)

    """
    Stop the signal analyser (not essential)
    """

    if( not sapphire.stopSigAna() ):
        getError(sapphire)

    """
    Output power results
    """
                                                        # Poll results table
                                                        # Arg1: measurement to be retrieved
                                                        # Arg2: bit mask for channel filter
                                                        #   bitn = RF channel number n
                                                        # Arg3: bit mask for phy filter
                                                        #   bit0 = 2Mbps uncoded
                                                        #   bit1 = 1Mbps uncoded
                                                        #   bit2 = s=2 coded
                                                        #   bit3 = s=8 coded
                                                        # Arg4: bit mask for packet length group filter
    x = sapphire.pollResultsTable( 0, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
    printTable( PowerHeadings , x )

    """
    Output modulation results in off-air mode
    """
                                                        # Poll results table
                                                        # Arg1: measurement to be retrieved
                                                        # Arg2: bit mask for channel filter
```

```
                                                    #   bitn = RF channel number n
                                                    # Arg3: bit mask for phy filter
                                                    #   bit0 = 2Mbps uncoded
                                                    #   bit1 = 1Mbps uncoded
                                                    #   bit2 = s=2 coded
                                                    #   bit3 = s=8 coded
                                                    # Arg4: bit mask for packet length group filter
x = sapphire.pollResultsTable( 1+0x80000000, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
printTable( ModulationHeadings , x )

"""
Output drift results in off-air mode
"""
                                                    # Poll results table
                                                    # Arg1: measurement to be retrieved
                                                    # Arg2: bit mask for channel filter
                                                    #   bitn = RF channel number n
                                                    # Arg3: bit mask for phy filter
                                                    #   bit0 = 2Mbps uncoded
                                                    #   bit1 = 1Mbps uncoded
                                                    #   bit2 = s=2 coded
                                                    #   bit3 = s=8 coded
                                                    # Arg4: bit mask for packet length group filter
x = sapphire.pollResultsTable( 2+0x80000000, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
printTable( DriftHeadings , x )

"""
Output inband mession results in off-air mode
"""
                                                    # Poll results table
                                                    # Arg1: measurement to be retrieved
                                                    # Arg2: bit mask for channel filter
                                                    #   bitn = RF channel number n
                                                    # Arg3: bit mask for phy filter
                                                    #   bit0 = 2Mbps uncoded
                                                    #   bit1 = 1Mbps uncoded
                                                    #   bit2 = s=2 coded
                                                    #   bit3 = s=8 coded
                                                    # Arg4: bit mask for packet length group filter
x = sapphire.pollResultsTable( 3, 0x1000, 0x2 , 0xFFFFFFFF )
printTable( InBandHeadings , x )

"""
Use poll plot command with method = 4 to stash a waveform
This only needs to be done if the subsequent reads are to refer to the same packet
```

```
    """

    n = sapphire.pollResultsPlot( 0, 4, 0xFFFFFFFFFF, 0xF, 0xFFFFFFFF)
    if( n ):

        """
        Read back meta data for stashed packet
        """

        x = sapphire.getResultsPlotMeta()
        print ( "Time stamp : " , x.timestamp )
        print ( "RF channel : " , x.rfchan )
        print ( "Phy        : " , phys[x.phy] )
        print ( "Octets     : " , x.pktlen )
        print ( "Power      : %.2f dBm" % x.pwr )

        """
        Read back waveform amplitude
        """

        m = sapphire.pollResultsPlot( 0x3E9, 0, 0xFFFFFFFFFF, 0xF, 0xFFFFFFFF )
        if( m ):
            m = m - 32
            amp = []
            for offset in range( 32 , m , 32768 ):
                count = min( 32768 , m )
                m = m - count;
                x = sapphire.getResultsPlotInt16( int(offset/2) , int(count/2) );
                for y in x:
                    amp.append(y)
            plt.figure(1)
            n = len(amp)
            x = [ 0.03125*float(x)-100.0 for x in range(0,n) ]
            plt.plot( x  , amp , 'r' )
            plt.xlabel( 'Time (us)' )
            plt.ylabel( 'Power (dBm)' )
            plt.title( 'Output power profile' )
            plt.grid( True )
            plt.show()

        """
        Read back waveform deviation
        """

        m = sapphire.pollResultsPlot( 0x3EA, 0, 0xFFFFFFFFFF, 0xF, 0xFFFFFFFF )
```

```
if( m ):
    m = m - 32
    fm = []
    for offset in range( 32 , m , 32768 ):
        count = min( 32768 , m )
        m = m - count;
        x = sapphire.getResultsPlotInt16( int(offset/2) , int(count/2) );
        for y in x:
            fm.append(y)
    plt.figure(1)
    n = len(fm)
    x = [ 0.03125*float(x)-100.0 for x in range(0,n) ]
    plt.plot( x  , fm , 'r' )
    plt.xlabel( 'Time (us)' )
    plt.ylabel( 'Frequency deviation (kHz)' )
    plt.title( 'FM demodulated data' )
    plt.grid( True )
    plt.show()

"""
Read back waveform IQ data
"""

m = sapphire.pollResultsPlot( 0x3EC, 0, 0xFFFFFFFFFF, 0xF, 0xFFFFFFFF )
if( m ):
    m = m - 32
    re = [];
    im = [];
    for offset in range( 32 , m , 32768 ):
        count = min( 32768 , m )
        m = m - count;
        x = sapphire.getResultsPlotFloat( int(offset/4) , int(count/4) );
        k = 0
        for y in x:
            if k:
                im.append(y)
            else:
                re.append(y)
            k = 1 - k
    n = len(re)
    x = [ 0.03125*float(x)-100.0 for x in range(0,n) ]
    plt.figure(1)
    plt.plot( x , re , 'r' )
    plt.hold(True)
    plt.plot( x , im , 'b' );
```

```
            plt.xlabel( 'Time (us)' )
            plt.ylabel( 'Amplitude' )
            plt.title( 'IQ data' )
            plt.grid( True )
            plt.show()

    getError(sapphire)

    sapphire.exitApp()                                  # Close the sapphire applications

    sapphire = 0

    transport = 0
```

5.2.4 Testing a device which is advertising

This example code demonstrates how a device which is advertising can be tested. The advertising device must send either ADV_IND or ADV_SCAN_IND packets. The *TLF3000* will issue SCAN_REQ packets to the advertising device and listen for the SCAN_RSP packets. The *TLF3000* will measure:

1. Output power
2. Modulation characteristics
3. Carrier frequency offset and drift
4. In-band emissions
5. Sensitivity

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. The Sapphire application is launched using the code described in Launching the Sapphire application
3. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
4. The operating mode is set to advertise/scan mode using setMode
5. The receiver port to be used is set using setRxPort
6. The *TLF3000* is instructed to scan using startScan. The *TLF3000* can be instructed to respond to only certain advertising addresses, but in this example it is permitted to respond to any advertising addresses which it sees.
7. When the requested data has been collected, the scanning is halted using stopAdvScan.
8. pollResultsTable is used to recover the output power results table and then further calls are used to recover the modulation characteristics results table, carrier frequency and drift results table and the in-band emissions results table. All these tables are printed out.
9. A second scan is made by caling startScan. However, this time the output power of the *TLF3000* is gradually reduced. At each power level the number of SCAN_RESP packets is recorded. In this way the sensitivity can be determined.
10. When the requested data has been collected, the scanning is halted using stopAdvScan.
11. Calls topollResultsPlot and getResultsPlotFloat are used to recover the packet error rate at each power level and these results are printed out.

```
"""
callbacks for handling end messages
"""

def scanEnd( d ):
    global doneScan
    doneScan = True
    print( "Scan complete" )

"""
callbacks for handling aux messages which should not occur
"""

def envData( d ):
    print( "Environmental data" )

def pwrData( d ):
    print( "Input/output power data" )

"""
Main callback for handling messages on the asynch data channel
These are then dispatched to the next level callback handler
"""

pktType = {
    7: scanEnd,
    193: envData,
    194: pwrData}

pkthdr = namedtuple('hdr', 'n nMsb typ')
pkthdrbin = '<HBB'

def dataCallback(d):
    s = pkthdr(*struct.unpack(pkthdrbin, d[0:4]))
    pktType.get( s.typ )(d)

"""
Routine for retrieving errors messages from TFL3000 and then exiting
"""

def getError(s):
    msg = s.DispErrors()
    while( msg != "" ):
        print( msg )
        msg = s.DispErrors()
```

```
PowerHeadings = (
        "Pavg",
        "Pk-Pavg"
        )

ModulationHeadings = (
        "DF1mzx",
        "DF1avg",
        "DF2max",
        "DF2avg",
        "DF2avg/DF1avg",
        "DF2max 99.9%"
        )

DriftHeadings = (
        "Fo",
        "Fn",
        "|F1-Fo|",
        "|Fo-Fn|",
        "|Fn-Fn-5|"
        )

InBandHeadings = (
        "Ftx+/-2MHz",
        "Ftx+/-(3+n)MHz",
        "# exceptions",
        "Max exception"
        )


"""
routine for printing results tables
"""

def printTable( headings , x ):
    print( "" )
    print( "%20s %10s %10s %10s %10s %10s" % ( "Quantity" , "# Pkts" , "Min" , "Max" , "Avg" , "Current" ))
    n = 0;
    for h in headings:
        print( "%20s " % h , end = "" )
        print( "%10d %10.1f %10.1f %10.1f %10.1f" % ( x[n], x[n+1], x[n+2], x[n+3], x[n+4] )  )
        n = n + 5
    print( "" )
```

```
"""
Main script
"""

if __name__ == '__main__':
    global doneScan
    doneScan = False
    transport = connect()                                   # Connect to the TLF3000 unit
    sapphire = SapphireInterface(transport)                 # Launch the Sapphire application
    transport.setDataCallback(dataCallback,False)           # Register callback for asynch data channel

    if( not sapphire.setMode( 3 ) ):                        # Enter adv/scan mode
        getError(sapphire)


                                                            # Select input port
                                                            # 0 - MONITOR IN
                                                            # 1 - Tx/Rx
    if( not sapphire.setRxPort( 1 ) ):
        getError(sapphire)


                                                            # Start scanning
                                                            # Arg0: Initial power in power sweep
                                                            # Arg1: Final power in power sweep
                                                            # Arg2: Step size for power sweep
                                                            # Arg3: Mask of channels to advertise on
                                                            #       1 = channel 0
                                                            #       2 = channel 12
                                                            #       4 = channel 39
                                                            #       To help ensure DUT sees advertising packet
                                                            #       recommend value of 7, ie all channels
                                                            # Arg4: 0 = cycle round the channels in the channel mask
                                                            #       sequentially
                                                            #       1 = transmit on all the channels in the channel
                                                            #       mask concurrently
                                                            # Arg5: TxAdd bit for header
                                                            #       False = AdvA is public
                                                            #       True = AdvA is random
                                                            # Arg6: ScanA as an ascii string representing octets in hex
                                                            # Arg7: Number of advertising packets to respond to
                                                            #       0 = respond forever
                                                            # Arg8: 0 = run to completion
                                                            #       1 = stop on test failure
```

```
                                                    # Arg9:  Timeout in ms to prevent hanging if no advertising
                                                    #        packets seen
                                                    # Arg10: Mask of data to collect
                                                    #        1 = Power
                                                    #        2 = Modulation charateristics
                                                    #        4 = Drift and carrier offset
                                                    #        8 = In-band emissions
                                                    #        256 = Waveform
                                                    # Arg11: RSSI threshold for packets to analyse in dBm
                                                    #        Only packets greater than this threshold will be
                                                    #        analysed
                                                    # Arg12: Mask of PDU types to analyse
                                                    #        Packets with PDUs not in the mask will not be
                                                    #        analysed
                                                    #        1 = ADV_IND
                                                    #        64 = ADV_SCAN_IND
                                                    # Arg13: False = ignore address in received packets
                                                    #        True = check address in received packets
                                                    # Arg14: 0 = address to check against is public
                                                    #        1 = address to check against is random
                                                    #        Ignored if Arg13 is False
                                                    # Arg15: Address to check against as an ascii string
                                                    #        representing octets in hex
                                                    #        Ignored if Arg16 is False

ScanA = vector_uchar( array.array( "B" , ( 0x0A , 0x1B , 0x2C , 0x3D , 0x4E , 0x5E  ) ) )
CmpAddr = vector_uchar( array.array( "B" , ( 0x11 , 0x22 , 0x33 , 0x44 , 0x55 , 0x66 ) ) )

# Do each channel sequentially so that we get statistics on each channel
# This also checks that the DUT can respond on each channel

if( not sapphire.startScan(  -10, -10, -0.5,
                             7, 0,
                             False, ScanA,
                             10,
                             0, 500,
                             0xF,
                             -30, 1 + 64, False, False, CmpAddr ) ):
    getError(sapphire)

n = 0
while 1:
    if( doneScan or (n >= 100) ):
        break
    n = n + 1
```

```
        sleep( 0.1 )

    if( not sapphire.stopAdvScan() ):
        getError(sapphire)

    """
    Output power results
    """
                                                        # Poll results table
                                                        # measurement
                                                        # channel filter
                                                        # phy filter
                                                        # power filter
    x = sapphire.pollResultsTable( 0, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
    printTable( PowerHeadings , x )

    """
    Output modulation results
    """
                                                        # Poll results table
                                                        # measurement
                                                        # channel filter
                                                        # phy filter
                                                        # power filter
    x = sapphire.pollResultsTable( 1+0x80000000, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
    printTable( ModulationHeadings , x )

    """
    Output drift results
    """
                                                        # Poll results table
                                                        # measurement
                                                        # channel filter
                                                        # phy filter
                                                        # power filter
    x = sapphire.pollResultsTable( 2+0x80000000, 0xFFFFFFFFFF, 0x2 , 0xFFFFFFFF )
    printTable( DriftHeadings , x )

    """
    Output inband mession results in off-air mode
    """
                                                        # Poll results table
                                                        # measurement
                                                        # channel filter
                                                        # phy filter
```

```
                                                                # pkt len group filter
x = sapphire.pollResultsTable( 3, 0x1000, 0x2 , 0xFFFFFFFF )
printTable( InBandHeadings , x )

"""
Do a sensitivity search
"""

# There can be no more than 32 power steps

startPwr = -85
stepPwr = -1
stopPwr = startPwr + 15*stepPwr;

# Do channels sequentially so we get sensitivity for each channel

doneScan = False

if( not sapphire.startScan(  startPwr, stopPwr, stepPwr,
                           7, 1,
                           False, ScanA,
                           50,
                           0, 5000,
                           0xF,
                           -30, 1 + 64, False, False, CmpAddr ) ):
    getError(sapphire)

while 1:
    if( doneScan ):
        break
    sleep( 0.1 )

if( not sapphire.stopAdvScan() ):
    getError(sapphire)

"""
Read back number of packets vs power
"""
n = int( sapphire.pollResultsPlot( 0x80000011, 2, 0x0000000001, 0x2, 0xFFFFFFFF) / 12 )
c1 = sapphire.getResultsPlotFloat( 0 , n );
n = int( sapphire.pollResultsPlot( 0x80000011, 2, 0x0000001000, 0x2, 0xFFFFFFFF) / 12 )
c12 = sapphire.getResultsPlotFloat( 0 , n );
n = int( sapphire.pollResultsPlot( 0x80000011, 2, 0x8000000000, 0x2, 0xFFFFFFFF) / 12 )
c39 = sapphire.getResultsPlotFloat( 0 , n );
print( "                         Packet error rate" )
```

```
    print( "Power (dBm)   Channel 0    Channel 12   Channel 39" )
    for k in range(0,n):
        print( "%6.1f         %6.1f       %6.1f       %6.1f" % ( startPwr , c1[k] , c12[k] , c39[k] ) )
        startPwr = startPwr + stepPwr;

    if( not sapphire.stopAdvScan() ):
        getError(sapphire)

    """

    """

    getError(sapphire)

    sapphire.exitApp()                                    # Close the sapphire applications

    sapphire = 0
    transport = 0
```

5.3 Library reference

5.3.1 Overview

This section lists the Python library commands which are available for the Sapphire application.

5.3.2 setMode

setMode() sets the operating mode of the Sapphire application:

res = sapphire.setMode(mode)

mode defines the operating mode:

Value	Operating mode
0	Phy level tester
1	Signal generator
2	Signal analyzer
3	Advertise/scan
> 3	Illegal

res will be set to False if the the command fails.

5.3.3 getMode

getMode() returns the current operarting mode of the Sapphire application:

mode = sapphire.getMode()

mode is the current operating mode. Possible returned values:

Value	Operating mode
0	Phy level tester
1	Signal generator
2	Signal analyzer
3	Advertise/scan

5.3.4 startScript

startScript() instructs the Sapphire application to prepare for the download of a new test script. A test script can only be started when the phy level tester is not running. This command can be issued irrespective of which mode the Sapphire application is operating in. Any existing but incomplete test script download will be aborted.

res = sapphire.startScript(overwrite , flash , testScriptName)

overwrite is a Boolean. If True, then if a test script file of the same name already exists it will be overwritten. If False, then if a test script file of the same name already exits the command will fail.

flash is a Boolean. If True, then the test script file will be placed in non-volatile FLASH memory. If False, then the test script file will be placed in volatile RAM. Currenty only RAM is supported.

testScriptName is a string containing the name of the test script file. Valid test script filenames must start with an alphabetic character. The remainder of the file name must be composed from the characters a-z, A-Z, 0-9 and _.

res will be set to False if the the command fails.

5.3.5 contScript

contScript() appends a new line to a test script previously opened using startScript but not yet closed with an *endScript*. A test script can only be added to when the phy level tester is not running. This command can be issued irrespective of which mode the Sapphire application is operating in.

Each line in the test script is a sequence of ASCII characters defining a test to be performed and its associated parameters. The test script is case insensitive. Full details of the test script format can be found in section TBD.

res = sapphire.addLine(line)

line is a character string containing the next line to be appended to the test script file. It is not necessary to include an end of line character.

res will be set to False if the the command fails.

5.3.6 endScript

endScript() signifies that the entire content of a test script has been download. This can only be called after startScript() has successfully be executed. This command cannot be executed if the phy level tester is running. This command can be issued irrespective of which mode the Sapphire application is operating in.

res = sapphire.endScript()

res will be set to False if the the command fails.

5.3.7 deleteScript

deleteScript() will delete a test script from the *TLF3000* memory.

res = sapphire.deleteScript(fals, testScriptName)

value but will dramatically reduce test time.

flash is a Boolean. If True, then the test script file to be deleted resides in non-volatile FLASH memory. If False, then the test script file to be deleted resides in volatile RAM. Currenty only RAM is supported.

testScriptName is a string containing the name of the test script file to be deleted. Valid test script filenames must start with an alphabetic character. The remainder of the file name must be composed from the characters a-z, A-Z, 0-9 and _.

res will be set to False if the the command fails.

5.3.8 runScript

runScript() instructs the phy level tester to start executing a test script. This command can only be executed when the Sapphire application is in phy level test mode. It cannot be executed whilst a test script is being downloaded, i.e. after a *startScript()* command and prior to an *endScript()* command.

res = sapphire.startScript(repeat , runToCompletion , reorder , onlyLimts , flash , testScriptName)

repeat specifies the number of times the testScript will be executed.

runToCompletion is a Boolean. If set, then all the tests specified in the test script will be performed. If not set, then the test script will terminate as soon as a test failure is detected.

reorder is a Boolean. If this flag is set, then the test script will be reordered so that the transmitter tests are performed first followed by the receiver tests. If this flag is not set, then the tests will be performed in the order specified in the test script.

onlyLimits is a Boolean. If set, the receiver tests will terminate as soon as sufficient packets have been sent to determine whether the test will pass or fail the PER limit. This will reduce the accuracy of the returned PER value but will dramatically reduce test time.

flash is a Boolean. If True, then the test script file to be run resides in non-volatile FLASH memory. If False, then the test script file to be run resides in volatile RAM. Currenty only RAM is supported.

testScriptName is a string containing the name of the test script file to be run. Valid test script filenames must start with an alphabetic character. The remainder of the file name must be composed from the characters a-z, A-Z, 0-9 and _.

res will be set to False if the the command fails.

5.3.9 abortScript

abortScript() instructs the phy level tester to abort an executing a test script. This command can only be executed when the Sapphire application is in phy level test mode and a test script is running.

res = sapphire.abortScript()

res will be set to False if the the command fails.

5.3.10 setCableLoss

setCableLoss() informs the Sapphire application of the cable loss between the Tx/Rx port of the *TLF3000* unit and the DUT. The loss is specified at 2.4 GHz. The Sapphire application requires knowledge of this loss to compensate its transmit power and to calibrate its received signal strength measurements. The same loss is also used to compensate the out-of-band CW blocker. Since the cable will vary between 24 MHz and 6 GHz, this compensation can only be approximate.

res = sapphire.setCableLoss(dB)

dB is the cable loss in dB. The cable loss must be between 0 and 25dB.

res will be set to False if the the command fails.

5.3.11 getCableLoss

getCableLoss() retrieves from the Sapphire application the programmed cable loss between the Tx/Rx port of the *TLF3000* unit and the DUT. The loss is specified at 2.4 GHz. The Sapphire application requires knowledge of this loss to compensate its transmit power and to calibrate its received signal strength measurements. The same loss is also used to compensate the out-of-band CW blocker. Since the cable will vary between 24 MHz and 6 GHz, this compensation can only be approximate.

loss = sapphire.getCableLoss()

loss is the programmed cable loss in dB. It will be in the range 0 to 25dB.

5.3.12 setDUTProperties

setDUTProperties() informs the Sapphire application of the DUT's supported features. This information is only required for the phy test mode. This command can be executed at any time.

res = sapphire.setDUTProperties(role , extAdv , extData , maxAdvOct , maxRxOct , maxRxTime , maxTxOct , maxTxTime , ext2Mbps, extLR , extStable , RxAoA2, TxAoA2, RxAoD2 , TxAoD2, RxAoA1, TxAoA1, RxAoD1, RxAoD1, CTEslots , NumAnt , maxPatt)

role indicates which role the DUT supports. This determines whether both transmit and receive tests can be done, and if so, on which channels. Possible values are:

Value	Role
0	Central
1	Peripheral
2	Broadcaster
3	Observer

extAdv is a Boolean indicating whether extended advertising features are supported.

extData is a Boolean indiciating whether extended data lengths are supported.

maxAdvOct is the maximum number of advertising octets.

maxRxOct is the maximum number of octets in a packet payload which the DUT can receive.

maxRxTime is the maximum packet length that can be received in units of µs.

maxTxOct. is the maximum number of octets in a packet payload which the DUT can transmit.

maxTxTime is the maximum packet length that can be transmitted in units of µs.

ext2Mbps is a Boolean indicating whether 2Mbps is supported.

extLR is a Boolean indicating whether coded phys are supported.

extStable is a Boolean indicating whether stable modulation index is supported.

RxAoA2 is a Boolean indicating whether the receiver can receive an AoA CTE with 2µs slots.

TxAoA2 is a Boolean indicating whether the transmitter can transmit an AoA CTE with 2µs slots.

RxAoD2 is a Boolean indicating whether the receiver can receive an AoD CTE with 2µs slots.

TxAoD2 is a Boolean indicating whether the transmitter can transmit an AoD CTE with 2µs slots.

RxAoA1 is a Boolean indicating whether the receiver can receive an AoA CTE with 1µs slots.

TxAoA1 is a Boolean indicating whether the transmitter can transmit an AoA CTE with 1µs slots.

RxAoD1 is a Boolean indicating whether the receiver can receive an AoD CTE with 1µs slots.

TxAoD1 is a Boolean indicating whether the transmitter can transmit an AoD CTE with 1µs slots.

CTEslots is the maximum supported CTE length in 8µs slots

NumAnt is the number of antenna attached to the DUT.

maxPatt is the maximum length switch pattern the DUT supports.

res will be set to False if the the command fails.

5.3.13 setDUTPort

setDUTPort informs the Sapphire application which comport the DUT is attached to. This information is required by the Sapphire application when operating in phy level test mode. This command can be executed at anytime.

res = sapphire.setDUTPort(port)

port is the number of the comport to which the DUT is connected. If this number is positive, then it is interpreted as a comport on the host machine. If this number if negative, then it implies that the DUT is connected to the DIO connector on the rear of the *TLF3000* unit.

res will be set to False if the the command fails.

5.3.14 setDUTComms

set*DUTComms* informs the Sapphire application how it should communicate with DUT. This information is required by the Sapphire application when operating in phy level test mode. This command can be executed at anytime.

res = sapphire.setDUTComms(comms , baud , hwFlow , swFlow , stopBits , parity , crc)

comms specifies method to be used to communicate with the DUT. Possible values are:

0	Direct test mode
1	H4 UART
2	H5 UART
3	BCSP UART

baud contains the baud rate used to communicate with the DUT. Supported baud rates for a DUT connected directly to the *TLF3000* unit are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. Automatic baud detection will occur if this field is set to zero. Automatic baud rate detection is only available for H5 and BCSP transports.

hwFlowIndicates the type of hardware flow control which will be used in communicating with the DUT. Possible values are:

0	No hardware flow control
1	RTS/CTS

swFlow *i*ndicates the type of software flow control which will be used in communicating with the DUT. Possible values are:

0	No software flow control
1	XON/XOFF

stopBits is the number of stop bits to be used when communicating over UART. Possible values are:

1	1 stop bit
2	2 stop bits

parity determines the parity bits to be used when communicating over UART. Possible values are:

0	None
1	Odd
2	Even

crc specifies whether a CRC should be appended to the packet when using H5 or BCSP. Possible values are:

0	No CRC
1	16 bit CCITT CRC

res will be set to False if the the command fails.

5.3.15 setPhyTestDIO

setPhyTestDIO() defines how the digital output lines should be programmed when the Sapphire application is in the phy tester mode. The digital output lines can be used to show when the tester is running and whether tests have passed or failed. This command can be executed at any time. If a digital output line is programmed to respond to both the phy tester running and the pass/fail status, then the pass/fail status takes priority.

res = sapphire.setPhyTestDIO(runBits , runMask , passBits , passMask)

run bits is an 8 bit mask determining the polarity of the 8 digital output lines when the phy tester is running. A bit set to '1' indicates that the corresponding digital output line should be set high whilst the phy tester is running. A bit set to '0' indicates that the corresponding digital output line should be set low whilst the phy tester is running.

runMask is an 8 bit mask indicating which bits in the *RunMask* are relevant and which should be ignored. A bit set to '1' indicates that the corresponding bit in *RunBits* should be acted on. A bit set to '0' indicates that the corresponding bit in *RunBits* should be ignored.

passBits is an 8 bit mask determining how the 8 digital output lines should be set to indicate pass or fail. A bit set to '1' indicates that the corresponding digital output line should be set high if all tests have passed. A bit set to '0' indicates that the corresponding digital output line should be set low if all tests have passed.

passMask is an 8 bit mask indicating which bits in the *PassMask* are relevant and which should be ignored. A bit set to '1' indicates that the corresponding bit in *PassBits* should be acted on. A bit set to '0' indicates that the corresponding bit in *PassBits* should be ignored.

res will be set to False if the the command fails.

5.3.16 setWanted

setWanted() programs the wanted signal when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setWanted(on , chan , pwr , phy , aa , seed , payload , pktLen , pktInt , pktNum , dio , distortions)

on is a Boolean to indicate whether the wanted signal generator should be started. If this flag set, then the wanted signal generator is started, otherwise the wanted signal generator is halted if it is already running.

chan is the RF channel number for the wanted signal. Valid channels are in the range 0 to 39.

pwr is the power of the wanted signal in units of dBm. This should be in range -170 dBm to 0 dBm.

phy is the phy to be used by the wanted signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

aa is the access address to be used in the transmitted packets.

seed is the CRC seed to be used in the transmitted packets.

payload defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31
7	11110000 in transmission order
8	10101010 in transmission order
9	11111111
10	00000000
11	00001111 in transmission order
12	01010101 in transmission order

pktLen is the length of the wanted signal packet payload in octets. This must be in the range 0 to 255.

pktInt is the interval between the start of one packet and the start of the subsequent packet in units of µs.

pktNum is the number of packets to be transmitted. If this value is specified as 0, then the wanted signal transmitter will continue to generate packets until it is terminated. The maximum value which can be specified is 635535.

dio is a mask indicating which digital IO lines should be toggled high when the wanted signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when a packet is being transmitted. Only lines 2 to 7 can be specified.

distortions is a vector__int16 which must be imported from the sapphire module. See example signal generator code. Its contents are groups of 5 int16 which described the distortions to be applied:

1. *int16_t : CarrierOff*. The carrier offset in units of kHz. Valid range is -500 kHz to +500kHz for 2Mbps uncoded and -250 kHz to +250 kHz for all other phy.
2. *uint16_t : ModIndex*. The modulation index in units of 0.0001. Valid range is 0.4 to 0.6.
3. *uint16_t : DriftMag*. The drift magnitude in units of kHz. Valid range is -156 kHz to +156 kHz for 2 Mbps uncoded and -78 kHz to +78 kHz for all other phy.
4. *uint16_t : DriftRate*. The drift rate in units of Hz. Valid range is 0 to 2400 Hz.
5. *int16_t : SymbolTime*. The symbol timing error in units of ppm. Valid range is -100 to +100.

Each group of distortions is applied to 50 transmitted packets. Once all the groups of distortions have been exhausted, the list is rewound, and the first group reused. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *DriftMag* and whose frequency is determined by *DriftRate*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets.

res will be set to False if the the command fails.

5.3.17 setWantedAoA

setWantedAoA() programs a wanted signal with an AoA CTE when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setWantedAoA(on , chan , pwr , phy , aa , seed , payload , pktLen , pktInt , pktNum , dio , distortions , cteSlots , cteInfo)

on is a Boolean to indicate whether the wanted signal generator should be started. If this flag set, then the wanted signal generator is started, otherwise the wanted signal generator is halted if it is already running.

chan is the RF channel number for the wanted signal. Valid channels are in the range 0 to 39.

pwr is the power of the wanted signal in units of dBm. This should be in range -170 dBm to 0 dBm.

phy is the phy to be used by the wanted signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

aa is the access address to be used in the transmitted packets.

seed is the CRC seed to be used in the transmitted packets.

payload defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31
7	11110000 in transmission order
8	10101010 in transmission order
9	11111111
10	00000000
11	00001111 in transmission order
12	01010101 in transmission order

pktLen is the length of the wanted signal packet payload in octets. This must be in the range 0 to 255.

pktInt is the interval between the start of one packet and the start of the subsequent packet in units of µs.

pktNum is the number of packets to be transmitted. If this value is specified as 0, then the wanted signal transmitter will continue to generate packets until it is terminated. The maximum value which can be specified is 635535.

dio is a mask indicating which digital IO lines should be toggled high when the wanted signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the

corresponding digital output line will be set high when a packet is being transmitted. Only lines 2 to 7 can be specified.

distrotions is a vector__int16 which must be imported from the sapphire module. See example signal generator code. Its contents are groups of 5 int16 which described the distortions to be applied:

6. *int16_t : CarrierOff*. The carrier offset in units of kHz. Valid range is -500 kHz to +500kHz for 2Mbps uncoded and -250 kHz to +250 kHz for all other phy.
7. *uint16_t : ModIndex*. The modulation index in units of 0.0001. Valid range is 0.4 to 0.6.
8. *uint16_t : DriftMag*. The drift magnitude in units of kHz. Valid range is -156 kHz to +156 kHz for 2 Mbps uncoded and -78 kHz to +78 kHz for all other phy.
9. *uint16_t : DriftRate*. The drift rate in units of Hz. Valid range is 0 to 2400 Hz.
10. *int16_t : SymbolTime*. The symbol timing error in units of ppm. Valid range is -100 to +100.

Each group of distortions is applied to 50 transmitted packets. Once all the groups of distortions have been exhausted, the list is rewound, and the first group reused. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *DriftMag* and whose frequency is determined by *DriftRate*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets.

cteSlots is the length of the AoA CTE in 8µs slots.

cteInfo determines the contents to be placed in the packets CTEInfo field. Options are:

Value	Meaning
0	AoA
1	AoD with 1 µs switching slots
2	AoD with 2 µs switching slots

res will be set to False if the the command fails.

5.3.18 setWantedAoD

setWantedAoD() programs a wanted signal with an AoD CTE when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setWantedAoD(on , chan , pwr , phy , aa , seed , payload , pktLen , pktInt , pktNum , dio , distortions , cteSlots , cteInfo ,
numAnt , antPatt , phis , amps)

on is a Boolean to indicate whether the wanted signal generator should be started. If this flag set, then the wanted signal generator is started, otherwise the wanted signal generator is halted if it is already running.

chan is the RF channel number for the wanted signal. Valid channels are in the range 0 to 39.

pwr is the power of the wanted signal in units of dBm. This should be in range -170 dBm to 0 dBm.

phy is the phy to be used by the wanted signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

aa is the access address to be used in the transmitted packets.

seed is the CRC seed to be used in the transmitted packets.

payload defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31
7	`11110000` in transmission order
8	`10101010` in transmission order
9	`11111111`
10	`00000000`
11	`00001111` in transmission order
12	`01010101` in transmission order

pktLen is the length of the wanted signal packet payload in octets. This must be in the range 0 to 255.

pktInt is the interval between the start of one packet and the start of the subsequent packet in units of µs.

pktNum is the number of packets to be transmitted. If this value is specified as 0, then the wanted signal transmitter will continue to generate packets until it is terminated. The maximum value which can be specified is 635535.

dio is a mask indicating which digital IO lines should be toggled high when the wanted signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when a packet is being transmitted. Only lines 2 to 7 can be specified.

distrotions is a vector__int16 which must be imported from the sapphire module. See example signal generator code. Its contents are groups of 5 int16 which described the distortions to be applied:

11. *int16_t : CarrierOff*. The carrier offset in units of kHz. Valid range is -500 kHz to +500kHz for 2Mbps uncoded and -250 kHz to +250 kHz for all other phy.
12. *uint16_t : ModIndex*. The modulation index in units of 0.0001. Valid range is 0.4 to 0.6.
13. *uint16_t : DriftMag*. The drift magnitude in units of kHz. Valid range is -156 kHz to +156 kHz for 2 Mbps uncoded and -78 kHz to +78 kHz for all other phy.
14. *uint16_t : DriftRate*. The drift rate in units of Hz. Valid range is 0 to 2400 Hz.
15. *int16_t : SymbolTime*. The symbol timing error in units of ppm. Valid range is -100 to +100.

Each group of distortions is applied to 50 transmitted packets. Once all the groups of distortions have been exhausted, the list is rewound, and the first group reused. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *DriftMag* and whose frequency is determined by *DriftRate*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets.

cteSlots is the length of the AoD CTE in 8µs slots.

cteInfo determines the contents to be placed in the packets CTEInfo field. Options are:

Value	Meaning
0	AoA
1	AoD with 1 µs switching slots
2	AoD with 2 µs switching slots

numAnt is the number of antenna to be simulated by the signal generator. For AoD the number of transmit antennas should be in the range 2 to 63.

antPatt defines the antenna switching pattern to be simulated. Options are:

Value	Meaning
0	1,2,3,4 … 1,2,3,4 … 1,2,3,4 …
1	1,2,3,4 … 4,3,2,1,2,3,4 … 4,3,2,1,2,3,4 …

phis is a vector__float which must be imported from the sapphire module. See example signal generator code. Each entry represents the phase of the *N*th transmit antenna in degrees. Valid phases are in the range -360 to +360. There must be numAnt entries.

amps is a vector__float which must be imported from the sapphire module. See example signal generator code. Each entry represents the amplitude of the *N*th transmit antenna relative to the main body of the packet in dB. Valid amplitudes are in the range -10 to +6dB. There must be numAnt entries.

res will be set to False if the the command fails.

5.3.19 setInterferer

setInterferer() programs the modulated interferer signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setInterferer(on , freq , pwr , phy , payload , dio , cont , pktLen , pktInt)

on is a Boolean to indicate whether the modulated interferer signal generator should be started. If this flag is set, then the modulated interferer signal generator is started, otherwise the modulated interferer signal generator is halted if it is already running.

Freq is the carrier frequency for the interferer signal in MHz. Valid range is 2392 MHz to 2488 MHz inclusive.

pwr is the power of the modulated interferer signal in dBm. Valid range is -170dBm to 0dBm.

phy is the phy to be used by the modulated interferer signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

payload defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31

dio is a mask indicating which digital IO lines should be toggled high when the interferer signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when the interferer is being transmitted. Only lines 2 to 7 can be specified.

cont is a Boolean which indicates whether the interferer transmission is continuous (True) or packetised (False).

pktLen is the length of the inteferer signal packet payload in octets. This must be in the range 0 to 255. This parameter is ignored if **cont** is True.

pktInt is the interval between the start of one packet and the start of the subsequent packet in units of µs. This parameter is ignored if **cont** is True.

res will be set to False if the the command fails.

5.3.20 setInBandCW

setInBandCW() programs the in-band CW signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setInBandCW(on , num , freq , pwr)

on is a Boolean to indicate whether the in-band CW generator should be turned on or off. If set, then the in-band CW generator is turned on, otherwise the in-band CW generator is turned off.

num indicates which of the two in-band CW sources is being programmed. Valid values are 0 and 1.

freq is the frequency of the in-band CW signal in Hz. Valid range is 2,395,000,000 Hz to 2,485,000,000 Hz inclusive.

pwr is the power of the in-band CW signal in units of dBm. Valid range is -170 dBm to 0 dBm.

res will be set to False if the the command fails.

5.3.21 setAWGN

setAWGN programs the AWGN generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = sapphire.setAWGN(on , pwr)

on is a Boolean to indicate whether the AWGN generator should be turned on or off. If set, then the AWGN generator is turned on, otherwise the AWGN generator is turned off.

pwr is the power of the AWGN signal in units of dBm/MHz. Valid range is -162 dBm/MHz to -42 dBm/MHz.

res will be set to False if the the command fails.

5.3.22 setOutOfBandCW

setOutOfBandCW programs the out-of-band CW signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

res = setOutOfBandCW(on , freq , pwr)

on is a Boolean to indicate whether the out-of-band CW generator should be turned on or off. If set, then the out-of-band CW generator is turned on, otherwise the out-of-band CW generator is turned off.

freq is the frequency of the out-of-band CW signal in MHz. Valid range is 24 MHz to 6,000 MHz.

pwr is the power of the out-of-band CW signal in units of dBm. Valid range is -50dBm to -28 dBm.

res will be set to False if the the command fails.

5.3.23 startStopSigGen

startStopSigGen determines when the signal generator is active. This is a global enable/disable for the signal generator output; it overrides the individual on fields for the wanted signal, modulated interferer, in-band CW, AWGN and out-of-band CW. This command can only be executed when the Sapphire application is in signal generator mode.

res = sapphire.startStopSigGen(run)

run determines whether the signal generator is running (1) or stopped (0).

res will be set to False if the the command fails.

5.3.24 stopSigGen

stopSigGen will stop all output from the signal generator. It will not affect the programming of the various signal sources, ie the wanted signal, modulated interferer, in-band CW, AWGN and out-of-band CW. This command can only be executed when the Sapphire application is in signal generator mode.

res = sapphire.stopSigGen( )

res will be set to False if the the command fails.

5.3.25 startStopSigAna

startStopSigAna() starts or stops the signal analyser. It also determines various run parameters for the signal analyser. This command can only be executed when the Sapphire application is in signal analyser mode.

res = sapphire.startStopSigAna(run , aa , chanMask , testMask , pktMask , phyMask , deWhiten , offAir , oversample , stopOnFail , lenMask , RSSIthresh)

run is a Boolean which indicates whether the signal analyser is to be started. If set., the signal analyser will be started, otherwise it will be stopped.

aa contains the 32 bit access address for the packets to be analysed.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers.

testMask is a bit mask indicating which types of analysis should be performed.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

pktMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *PktMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

phyMask is a mask containing the phys which will be analysed. Only packets which match the selected phys will be analysed. The meaning of the bits in this mask are:

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

deWhiten is a Boolean to indicate whether the packets to be analysed have been whitened. If deWhiten is set then the packets are whitened, otherwise the packets are unwhitened. The signal analyser needs to know whether the packet has been whitened in order to extract the correct length from the packet header.

offAir if True then the off-air mode will be used. This enables packets whose payloads do not conform to those in the Bluetooth LE phy test specification to be analysed.

overSample is the oversampling ratio to be applied to raw captured waveform data. This only applies to the raw waveform data; all transmitter tests are performed with data that has been 32x oversampled. Valid values are:

0	4x oversampling
1	8x oversampling
3	16x oversampling
7	32x oversampling

stopOnFail is a Boolean to indicate how the signal analyser should respond when a limit failure is detected. If clear, then the signal analyser ignores the limit failure and continues to run. If set, then the signal analyser will complete the analysis of the current failing packet and then halt.

lenMask this mask contains the packet length filter. Only packets which pass the packet length filter will analysed. There are 32 packet length groups. The first length group corresponds to packets with payloads in the range 0 to 7 octets, the second group to packets with payloads of 8 to 15 octets, etc. If a bit is '1', then packets whose payloads fall within the the corresponding packet length group will be analysed. If a bit is '0', then packets whose payloads fall within the the corresponding packet length group are ignored.

RSSIthresh Only packet with an RSSI greater than **RSSIthresh** will be analysed. **RSSIthresh** is in units of dBm.

res will be set to False if the the command fails.

5.3.26 stopSigAna

stopSigAn() will stop the signal analyser running. This command can only be executed when the Sapphire application is in signal analyser mode.

res = sapphire.stopSigAna()

res will be set to False if the the command fails.

5.3.27 getSigAnaState

getSigAnaState() returns whether the signal analyser was running. This is useful if the stop on test fail flag had been set when the signal analyser was started. This command can only be executed when the Sapphire application is in signal analyser mode.

run = sapphire.getSigAnaState()

run will be zero if the signal analyser is halted and 1 if the signal analyser is running.

5.3.28 startScan

startScan() instructs the *TLF3000* to look for advertising events and issue scan requests in reply. The *TLF3000* then looks to see if a scan response was received, indicating that the DUT received the scan request. The power, modulation, carrier frequency offset, drift and in-band emissions are simultaneously calculated. The *TLF3000* can be instructed to gradually reduce its transmit power so that the sensitivity of the DUT can be established.

res = sapphire.startScan(startdBm, stopdBm, stepdBm, chanMask, concurrent, TxAdd, scanAddr, numPkt, stopCond, timeoutms, capture, rssiThreshdBm, pduMask, checkAddrIn, randAddrIn, cmpAddr)

startdBm the power at which the *TLF3000* should send its first **numPkt** packets in units of dBm.

stopdBm the power at which the *TLF3000* should send its last **numPkt** packets in units of dBm. Note that the More *TLF3000* ph30 will step down from **startdBm** in steps of **stepdBm** until **stopdBm** is reached. Hence the final packets may be sent at a power level slightly higher than **stopdBm**.

stepdBm is the power reduction between each set of **numPkt** in units of dBm. Not more than 32 steps are permitted.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers. Only advertising channels should be used.

concurrent. If this is False, then the *TLF3000* will test each of the 3 advertising channels in turn. If this is True, then the *TLF3000* will test which ever advertising channel the next advertising packet is received on.

TxAdd if this is True then the ScanAddr used by the *TLF3000* is random address, otherwise it is a public address.

scanAddr is the ScanAddr used by the *TLF3000*. This argument is a 6 element vector_uchar, with each element representing one octet of the ScanAddr. vector_uchar must be imported from Frontline.

numPkt If **concurrent** is False, then **numPkt** is the number of packets tested on each advertising channel. If **concurrent** is True, then **numPkt** is the total number of packets tested.

stopCond If True, the scan will terminate as soon as a test limit fails. If False, the tests will run to completion, even if a test fails.

timeoutms If the time taken to issue **numPkt** scan requests exceeds **timeoutms** then the current test will be aborted and the test test started. The units of the timeout are ms.

capture is a bit mask indicating which types of analysis should be performed. It is recommended that raw waveform data is not collected unless necessary since this slows down the processing of advertising packets.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

rssiThreshdBm Only received packets which have a signal strength greater than or equal to **rssiThreshdBm** will be processed. The units of the parameter are dBm.

pduMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *pduMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

checkAddrIn If True, then the *TLF3000* will only issue scan requests to the advertising device whose address is **cmpAddr**. If False, the *TLF3000* will issue scan requests to any advertising device.

randAddrIn If True, then **cmpAddr** is a random address, otherwise **cmpAddr** is a public address. Ignored if **checkAddr** is False.

cmpAddr is the advertising address of the device which *TLF3000* should issue scan requests to if **checkAddrIn** is True. This argument is ignored otherwise. This argument is a 6 element vector_uchar, with each element representing one octet of the advertising address. vector_uchar must be imported from Frontline.

res will be set to False if the the command fails.

5.3.29 startAdv

startAdv() instructs the *TLF3000* to start sending advertising packets and listen for scan requests. The power, modulation, carrier frequency offset, drift and in-band emissions of the scan requests are calculated. The *TLF3000* can be instructed to gradually reduce its transmit power so that the sensitivity of the DUT can be established.

res = sapphire.startAdv(startdBm, stopdBm, stepdBm, chanMask, concurrent, advPDU, ChSel, TxAdd, RxAdd, AdvA, TargetA, PayLoad, numPkt, period, stopCond, capture, rssiThreshdBm, pduMask, checkAddrIn, randAddrIn, cmpAddr)

startdBm the power at which the *TLF3000* should send its first **numPkt** packets in units of dBm.

stopdBm the power at which the *TLF3000* should send its last **numPkt** packets in units of dBm. Note that the *TLF3000* will step down from **startdBm** in steps of **stepdBm** until **stopdBm** is reached. Hence the final packets may be sent at a power level slightly higher than **stopdBm**.

stepdBm is the power reduction between each set of **numPkt** in units of dBm. Not more than 32 steps are permitted.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers. Only advertising channels should be used.

concurrent. If this is False, then the *TLF3000* will transmit on each of the 3 advertising channels in turn. If this is True, then the *TLF3000* will transmit on all 3 advertising channels simultaneously.

advPDU is the value of the advertising PDU which the *TLF3000* should transmit.

chSel is the value of the chSel field in the packet header which the *TLF3000* transmits.

TxAdd is the value of the TxAdd field in the packet header which the *TLF3000* transmits.

RxAdd is the value of the RxAdd field in the packet header which the *TLF3000* transmits.

AdvA is the advertising address of the *TLF3000*. This argument is a 6 element vector_uchar, with each element representing one octet of the advertising address. vector_uchar must be imported from Frontline.

TargetA is the TargetA address used in the *TLF3000* transmitted packets. This argument is ignored if advPDU does not require a TargetA address. This argument is a 6 element vector_uchar, with each element representing one octet of the advertising address. vector_uchar must be imported from Frontline.

Payload is the advertising data to be included in the *TLF3000* transmitted packets. This argument is a vector_uchar, with each element representing one octet of the advertising data. vector_uchar must be imported from Frontline.

numPkt If **concurrent** is False, then **numPkt** is the number of advertising packets transmitted on each channel. If **concurrent** is True, then **numPkt** is the total number of advertising packets transmitted.

period is the interval between transmitted advertising packets in units of µs.

stopCond If True, advertising will terminate as soon as a test limit fails. If False, the tests will run to completion, even if a test fails.

capture is a bit mask indicating which types of analysis should be performed. It is recommended that raw waveform data is not collected unless necessary since this slows down the processing of advertising packets.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

rssiThreshdBm Only received packets which have a signal strength greater than or equal to **rssiThreshdBm** will be processed. The units of the parameter are dBm.

pduMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *pduMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

checkAddrIn If True, then the *TLF3000* will only analyse scan requests to the scanning device whose address is **cmpAddr**. If False, the *TLF3000* will analyse scan requests from any device.

randAddrIn If True, then **cmpAddr** is a random address, otherwise **cmpAddr** is a public address. Ignored if **checkAddr** is False.

cmpAddr is the ScanA address of the device which *TLF3000* should analyse scan requests from if **checkAddrIn** is True. This argument is ignored otherwise. This argument is a 6 element vector_uchar, with each element representing one octet of the ScanA address. vector_uchar must be imported from Frontline.

res will be set to False if the the command fails.

5.3.30 stopAdvScan

stopAdvScan() terminates a pervious advertising or scanning session started on the *TLF3000*.

Sapphire.stopAdvScan()

res will be set to False if the the command fails.

5.3.31 pollResultsTable

pollResultsTable() returns results in a tabular form when operating in signal analyser mode or advertise/scan mode.

For non-CTE tests, the results contributing to the returned table are filtered by:

1. An RF channel filter
2. A phy filter
3. A packet length group filter

Any packets which satisfy all the filters will contribute to the statistics in the returned table.

For CTE measurements, the filtering operating has to be modified to prevent packets with different CTE configurations being combined. A search is performed to find the last received packet which satisfies:

1. An RF channel filter
1. A phy filter
2. A CTE type filter
3. A CTE length filter

The most recent packet which satisfies these filters is used to determine the CTE type and length which will be used to populate the table. All packets with the same CTE type and length, and which also satisfy the RF channel and phy filters, will contribute to the statistics in the returned table.

table = sapphire.pollResultsTable(meas , chanMask , phyMask , pktMask)

meas is the measurement set for which tabular results are requested. Possible values are:

Meas	Measurement set	Collection mode for signal analyser
0x00000000	Output power	Standard test packets
0x00000001	Modulation characteristics	Standard test packets
0x00000002	Carrier offset & drift	Standard test packets
0x00000003	In-band emissions	Standard test packet & Off-air
0x00000004	#rxed packets in advertise/scan	N/A
0x00000005	CTE – absolute phases	Standard test packet & Off-air
0x40000005	CTE – differential phases	Standard test packet & Off-air
0x80000000	Output power	Off-air mode
0x80000001	Modulation characteristics	Off-air mode
0x80000002	Carrier offset & drift	Off-air mode
0x80000003	In-band emissions	Standard test packet & Off-air
0x80000004	#rxed packets in advertise/scan	N/A
0x80000005	CTE – absolute phases	Standard test packet & Off-air
0xC0000005	CTE – differential phases	Standard test packet & Off-air

The *Meas* types of 0x00000004 and 0x80000004 are equivalent and are only used in advertise/scan mode. When advertising, this *Meas* type returns the number of received scan requests or connection requests. When scanning, this *Meas* type returns the number of received scan responses.

chanMask is a 40 bit mask indicating containing the RF channel filter. Only packets which pass the RF channel filter results will contribute to the statistics in the returned table. A '1' in the mask indicates that the corresponding LE RF channel results should be included in the table, a '0' in the mask indicates that the corresponding LE RF channel results should be ignored.

phyMask is a 4 bit mask containing the phy filter. Only packets which pass the phy filter will contribute to the statistics in the returned table. If a bit is '1', then the results for the corresponding phy scheme will contribute to the statistics in the returned table. If a bit is '0', then the results for the corresponding phy scheme are ignored.

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

pktMask. For non-CTE measurements, this mask contained the packet length filter. Only packets which pass the packet length filter will contribute to the statistics in the returned table. There are 32 packet length groups. The first length group corresponds to packets with payloads in the range 0 to 7 octets, the second group to packets with payloads of 8 to 15 octets, etc. If a bit is '1', then the results for the corresponding packet length group will contribute to the statistics in the returned table. If a bit is '0', then the results for the corresponding packet length group are ignored.

For CTE measurements, the top 8 bits contain the supplemental type filter. During the initial search for the most recent packet, only packets which are consistent with the CTE type filter will be considered. If a bit is set to '1', then packets with the corresponding phy are included in the search. If a bit is set to '0', then packets with the corresponding phy are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Type
24	0x01000000	AoA
25	0x02000000	AoD, 1 µs slots
26	0x04000000	AoD, 2 µs slots

The lower 19 bits contain a filter for the CTE length. During the initial search for the most recent packet, only packets which are consistent with the CTE length filter will be considered. If a bit is set to '1', then packets with the corresponding CTE length are included in the search. If a bit is set to '0', then packets with the corresponding CTE length are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Length
0	0x00000001	16 µs
1	0x00000002	24 µs
2	0x00000004	32 µs
…	…	…
18	0x00040000	160 µs

table is a vector_float type which must be imported from the Sapphire module. See example signal analyser code. It contains the requested table of results. The first 5 values in table correspond to the first row in the table, the second 5 values to the second row, etc. The tables returned for the various measurement sets are:

1. Output power

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					

2. Modulation characteristics

Quantity	Packet Count	Minimum	Maximum	Average	Current
$\Delta F1_{max}$					NaN
$\Delta F1_{avg}$					
$\Delta F2_{max}$					NaN
$\Delta F2_{avg}$					
$\Delta F2_{avg} / \Delta F1_{avg}$					
$\Delta F2_{max\ 99.9\%}$					

3. Carrier offset and drift

Quantity	Packet Count	Minimum	Maximum	Average	Current
F_0					
F_n					NaN
$\lvert F_1 - F_0 \rvert$					
$\lvert F_0 - F_n \rvert$					NaN
$\lvert F_n - F_{n-5} \rvert$					NaN

4. In-band emissions

Quantity	Packet Count	Minimum	Maximum	Average	Current
F_{tx} ± 2MHz					
F_{tx} ± (3+n)MHz					
Exceptions					

5. # Received packets

Quantity	Packet Count	Minimum	Maximum	Average	Current
#rxed pkts					

This table is only available in advertising/scan mode. This table refers to the number of received scan or connection requests when advertising or the number of received scan responses when scanning. All the entries in the table are identical.

6. CTE- absolute phases

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					
Fs_i					
Fs_1-F_p					
Fs_i-F_0					
Fs_i-Fs_j					
$P_{ref,dev} / P_{ref,ave}$					
$P_{n,dev} / P_{n,ave}$					
Φ_0					
Φ_1					
….					
Φ_N					

Φ_o to Φ_7 correspond to the eight measurements made at 1µs intervals throughout the reference period. Φ_8 to Φ_N refer to phases measured in the subsequent sampling slots. The supplemental is processed by using the first 7µs of the reference period to estimate a starting phase and a frequency offset. The frequency offset is then removed from the supplemental. The starting phase is then subtracted from the frequency compensated supplemental prior to the phases being sampled.

7. CTE- differential phases

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					
Fs_i					
Fs_1-F_p					
Fs_i-F_0					
Fs_i-Fs_j					
$P_{ref,dev} / P_{ref,ave}$					
$P_{n,dev} / P_{n,ave}$					
$\Phi_1 - \Phi_o$					
$\Phi_2 - \Phi_1$					
….					
$\Phi_N - \Phi_{N-1}$					

Φ_o to Φ_7 correspond to the eight measurements made at 1µs intervals throughout the reference period. Φ_8 to Φ_N refer to phases measured in the subsequent sampling slots. The supplemental is processed by using the first 7µs of the reference period to estimate a starting phase and a frequency offset. The frequency offset is then removed from the supplemental. The starting phase is then subtracted from the frequency compensated supplemental prior to the phases being sampled.

Some current values are vector quantities rather than scalar quantities, for example, F_n, the average frequency over 10 bits in the drift calculation. The current values for these vector quantities are returned as NaN, as shown in the tables above.

5.3.32 pollResultsPlot

polltResultsPlot() instructs Sapphire to capture data ready for reading back in a subsequent command,

n = sapphire.pollResultsPlot(quantity , method , chanMask , phyMask , pktMask)

To access plot data, a sequence of two commands is required:

1. *pollPlotResults*. This command determines whether data is available, and if so, stashes it within the Sapphire application.
2. *getPlotResults*. This command reads back sections of the stashed data.

Multiple *getPlotResults* commands can be issued to retrieve different sections of the stashed data.

The *pollResultsPlot* command has 5 arguments:
Quantity. The measured quantity for which the plot is requested. Possible values are:

Quantity	Measurement	Notes
0x00000000	P_{avg}	Standard test packets
0x00000001	$Pk - P_{avg}$	Standard test packets
0x00000002	$\Delta F1_{max}$	Standard test packets
0x00000003	$\Delta F1_{avg}$	Standard test packets
0x00000004	$\Delta F2_{max}$	Standard test packets
0x00000005	$\Delta F2_{avg}$	Standard test packets
0x00000006	$\Delta F2_{avg}/\Delta F1_{avg}$	Standard test packets
0x00000007	$\Delta F2_{max}$ *99% percentile*	Standard test packets
0x00000008	F_0	Standard test packets
0x00000009	F_n	Standard test packets
0x0000000A	$F_0 - F_n$	Standard test packets
0x0000000B	$F_1 - F_0$	Standard test packets
0x0000000C	$F_{n+5} - F_n$	Standard test packets
0x0000000D	P_{tx} *@ ±2 MHz offset*	-
0x0000000E	P_{tx} *@ ≥±3 MHz offset*	-
0x0000000F	*# in-band emission exceptions*	-
0x00000010	*Max in-band exception*	-
0x00000011	*# rxed packets*	-
0x00000012	P_{avg} *CTE*	Standard test packets
0x00000013	$P_k - P_{avg}$ *CTE*	Standard test packets
0x00000014	Fs_i	Standard test packets
0x00000015	$Fs_1 - Fs_p$	Standard test packets
0x00000016	$Fs_i - F_0$	Standard test packets
0x00000017	$Fs_i - Fs_j$	Standard test packets
0x00000018	$P_{ref,dev} / P_{ref,ave}$	Standard test packets
0x00000019	$P_{n,dev} / P_{n,ave}$	Standard test packets
0x0000001A	*CTE* Φ_0	Standard test packets
0x0000001B	*CTE* Φ_1	Standard test packets
…	…	Standard test packets
0x0000006B	*CTE* Φ_{81}	Standard test packets
0x4000001A	*CTE* $\Phi_1 - \Phi_0$	Standard test packets
0x4000001B	*CTE* $\Phi_2 - \Phi_1$	Standard test packets
…	…	Standard test packets
0x4000006A	*CTE* $\Phi_{81} - \Phi_{80}$	Standard test packets
0x80000000	P_{avg}	Off-air mode
0x80000001	$Pk - P_{avg}$	Off-air mode
0x80000002	$\Delta F1_{max}$	Off-air mode
0x80000003	$\Delta F1_{avg}$	Off-air mode
0x80000004	$\Delta F2_{max}$	Off-air mode

0x80000005	$\Delta F2_{avg}$	Off-air mode
0x80000006	$\Delta F2_{avg}/\Delta F1_{avg}$	Off-air mode
0x80000007	$\Delta F2_{max}$ *99% percentile*	Off-air mode
0x80000008	F_0	Off-air mode
0x80000009	F_n	Off-air mode
0x8000000A	$F_0 - F_n$	Off-air mode
0x8000000B	$F_1 – F_0$	Off-air mode
0x8000000C	$F_{n+5} – F_n$	Off-air mode
0x80000012	P_{avg} *CTE*	Off-air mode
0x80000013	P_k - P_{avg} *CTE*	Off-air mode
0x80000014	Fs_i	Off-air mode
0x80000015	$Fs_1 – Fs_p$	Off-air mode
0x80000016	$Fs_i – F_0$	Off-air mode
0x80000017	$Fs_i – Fs_j$	Off-air mode
0x80000018	$P_{ref,dev} / P_{ref,ave}$	Off-air mode
0x80000019	$P_{n,dev} / P_{n,ave}$	Off-air mode
0x8000001A	*CTE* Φ_0	Off-air mode
0x8000001B	*CTE* Φ_1	Off-air mode
…	…	Off-air mode
0x8000006B	*CTE* Φ_{81}	Off-air mode
0xC000001A	*CTE* Φ_1- Φ_0	Off-air mode
0xC000001B	*CTE* $\Phi_{2\_}\Phi_1$	Off-air mode
…	…	Off-air mode
0xC000006A	*CTE* Φ_{81}- Φ_{80}	Off-air mode
0x000003E8	ACP spectra	-
0x000003E9	Amplitude – stashed packet	-
0x000003EA	FM deviation – stashed packet	-
0x000003EB	CTE phase – stashed packet	-
0x000003EC	IQ data – stashed packet	-
0x000003ED	FM deviation – CTE – stashed packet	-
0x000003EF	Amplitude – CTE – last packet received	-
0x000007D1	Amplitude – last packet received	-
0x000007D2	FM deviation – last packet received	-
0x000007D3	CTE phase last packet received	-
0x000007D4	IQ data – last packet received	-
0x000007D5	FM deviation – CTE – last packet received	-
0x000007D6	Amplitude – CTE - last packet received	-

Some quantities can be estimated when the packet payload does not contain a standard test packet sequence. For these quantities it is necessary to specify whether the 'Standard test packet' or 'off-air' measurement should be returned.

Method. Describes the type of plot which is requested. Possible values are:

0x00000000	Quantity vs RF channel
0x00000001	Quantity vs phy
0x00000002	Quantity vs packet length group
0x00000003	Quantity as a histogram
0x00000004	Quantity vs time or frequency (in-band emissions)

ChanMask. A 40 bit mask indicating which RF channels' results should be include in the plot. A '1' in the mask indicates that the corresponding LE RF channel results should be included in the plot, a '0' in the mask indicates that the corresponding LE RF channel results should be ignored. This mask is ignored when *Method* equals 0.

PhyMask. This is a mask indicating which phy schemes should be included in the plot. If a bit is '1', then the results for the corresponding phy are included in the plot. If a bit is '0', then the results for the corresponding phy are ignored. This mask is ignored when *Method* equals 1.

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

PktMask. For non-CTE meausrements this mask indicates which packet length groups should be included in the results plot. There are 32 packet length groups. The first packet length group includes packets with payload lengths in the range 0 to 7 octets, the second group contains packets with payload lengths of 8 to 15 octets, etc. If a bit is '1', then the results for the corresponding packet length group are included in the plot. If a bit is '0', then the results for the corresponding packet length group are ignored. This mask is ignored when *Method* equals 2.

For CTE measurements, the top 8 bits contain the supplemental type filter. During the initial search for the most recent packet, only packets which are consistent with the CTE type filter will be considered. If a bit is set to '1', then packets with the corresponding phy are included in the search. If a bit is set to '0', then packets with the corresponding phy are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Type
24	0x01000000	AoA
25	0x02000000	AoD, 1 µs slots
26	0x04000000	AoD, 2 µs slots

The lower 19 bits contain a filter for the CTE length. During the initial search for the most recent packet, only packets which are consistent with the CTE length filter will be considered. If a bit is set to '1', then packets with the corresponding CTE length are included in the search. If a bit is set to '0', then packets with the corresponding CTE length are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Length
0	0x00000001	16 µs
1	0x00000002	24 µs
2	0x00000004	32 µs
…	…	…
18	0x00040000	160 µs

n is the number of bytes of data which are available for reading back.

If the parameter **method** has been set to read back data vs RF channel, phy or packet length group, then three sequences of values are returned. The first sequence corresponds to the minimum value of the quantity, the second sequence to the average value of the quantity and the third sequence to the maximum value of the quantity.

The number of samples available to be read back is:

Method	Number of returned samples	Bytes available to read
0	3*40 = 120	480
1	3*4 = 12	48
2	3*32	128

For *Method* = 0, the first 40 samples correspond to the minimum value observed over RF channels 0 to 39. The next 40 samples correspond to the average value observed over RF channels 0 to 39. The final 40 samples correspond to the maximum value observed over RF channels 0 to 39.

For *Method* = 1, the first 4 samples correspond to the minimum value observed for each modulation scheme. The order of the modulation schemes is:

1. 2 Mbps, uncoded
2. 1 Mbps, uncoded
3. 1 Mbps, coded, S = 2
4. 1 Mbps, coded, S = 8

The next 4 samples correspond the average value observed for each modulation scheme. The final 4 samples correspond to the maximum value observed for each modulation scheme.

For *Method* = 2, the first N samples correspond the minimum value observed for each of the N packet length groups which have been defined. The next N samples correspond the average value observed for each of the N packet length groups. The final N samples correspond to the maximum value observed for each of the N packet length groups.

The units of the returned quantities are:

Measurement	**Units**
P_{avg}	dBm
Pk - P_{avg}	dB
$\Delta F1_{max}$	kHz
$\Delta F1_{avg}$	kHz
$\Delta F2_{max}$	kHz
$\Delta F2_{avg}$	kHz
$\Delta F2_{avg} / \Delta F1_{avg}$	Dimensionless
$\Delta F2_{max}$ 99% percentile	kHz
F_0	kHz
F_n	kHz
F_0 - F_n	kHz
$F_1 – F_0$	kHz
$F_{n+5} – F_n$	kHz
F_{tx} @ ±2MHz offset	dBm
F_{tx} @ ≥±3MHz offset	dBm
# in-band emission exceptions	Dimensionless
Max in-band exception	dBm
P_{avg} *CTE*	dBm
P_k - P_{avg} *CTE*	dBm
Fs_i	kHz
$Fs_1 – Fs_p$	kHz
$Fs_i – F_0$	kHz
$Fs_i – Fs_j$	kHz
$P_{ref,dev} / P_{ref,ave}$	dB
$P_{n,dev} / P_{n,ave}$	dB
Φ_n	°
$\Phi_n - \Phi_{n-1}$	°

If **method** is set to 3 then a histogram of the specified quantity will be collected. Each histogram is composed of 128 floating point samples. These form a histogram with equally spaced bins. The lower and upper edges of the first and last bins are:

Quantity	**Lower edge of 1st bin**	**Upper edge of last bin**
P_{avg}	-100 dBm	+28 dBm
$Pk - P_{avg}$	0 db	6.4 dB
$\Delta F1_{max}$	-512 kHz	+512 kHz
$\Delta F1_{avg}$	0 kHz	+512 kHz
$\Delta F2_{max}$	-512 kHz	+512 kHz
$\Delta F2_{avg}$	0 kHz	+512 kHz
$\Delta F2_{avg}/\Delta F1_{avg}$	0	1.28
$\Delta F2_{max}$ *99% percentile*	0	512 kHz
F_0	-256 kHz	+256 kHz
F_n	-256 kHz	+256 kHz
$F_0 - F_n$	-128 kHz	+128 kHz
$F_1 - F_0$	-64 kHz	+64 kHz
$F_{n+5} - F_n$	-64 kHz	+64 kHz
F_{tx} *@ ±2MHz offset*	-100 dBm	+28 dBm
F_{tx} *@ ≥±3MHz offset*	-100 dBm	+28 dBm
# in-band emission exceptions	0	128
Max in-band exception	-100 dBm	+28 dBm
P_{avg} *CTE*	-100 dBm	+28 dBm
$P_k - P_{avg}$ *CTE*	0 db	6.4 dB
Fs_i	-256 kHz	+256 kHz
$Fs_1 - Fs_p$	-32 kHz	+32 kHz
$Fs_i - F_0$	-32 kHz	+32 kHz
$Fs_1 - Fs_p$	-32 kHz	+32 kHz
$P_{ref,dev} / P_{ref,ave}$	0	1
$P_{n,dev} / P_{n,ave}$	0	1
Φ_n	-180°	+180°
$\Phi_n - \Phi_{n-1}$	-180°	+180°

If **method** is set to 4, then the specified quantity is collected as a function of time or frequency (in-band emission spectra).

For those quantities which are collected as a function of time, the saved data consists of 24 bytes of meta data followed by the requested quantity as a series of floating point numbers. The meta data is composed of 6 uint32_t:

1. *uint32_t : MSB*. MSB of timestamp of packet P_0 location in units of 250 ns since the Unix epoch of 1970-01-01 00:00:00 + 0000 (UTC).
2. *uint32_t : LSB*. LSB of timestamp of packet P_0 location in units of 250 ns since the Unix epoch of 1970-01-01 00:00:00 + 0000 (UTC).
3. *uint32_t : N.* The number of samples available.
4. *uint32_t : RF Chan*. The LE RF channel number on which the packet was collected.
5. *uint32_t : Phy*. phy of the packet for which the data was collected. Possible values are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

6. *uint32_t : Len*. For non-CTE quantities this istThe number of octets in the payload of the packet for which the data was collected. For CTE quantities the lower 24 bits contain the number of slots in the CTE whilst the upper 8 bits contain the CTE type.

Amplitude, FM deviation and IQ data commence 100µs prior to the starts of the packet and continue for 100µs past the end of the packet.

In standard test packet collection mode:

1. The first samples of $\Delta F1_{max}$ and $\Delta F2_{max}$ correspond to bit 4 of the payload. The interval between samples is 1 bit. Samples which are not valid contain NaN.
1. The first samples of *Fn* correspond to the average starting at bit 1 of the payload. The interval between samples is 10 bits. These samples are used to derive F_0 - F_n and $F_{n+5} – F_n$.

In off-air mode:

1. The first samples of $\Delta F1_{max}$ and $\Delta F2_{max}$ correspond to bit 1 of the packet. The interval between samples is 1 bit. Samples which are not valid contain NaN.
2. The first samples of *Fn* correspond to the average starting at bit 1 of the payload. The interval between samples is 16 bits. Samples which are not valid contain NaN. These samples are used to derive F_0 - F_n and $F_{n+5} – F_n$.
3. F_1 is taken as the value of F_n that is closest to the averaging period used to calculate F_1 in the standard test packet mode.

The units of the returned quantities are:

Measurement	**Units**
P_{avg}	dBm
Pk - P_{avg}	dB
$\Delta F1_{max}$	kHz
$\Delta F1_{avg}$	kHz
$\Delta F2_{max}$	kHz
$\Delta F2_{avg}$	kHz
$\Delta F2_{avg} / \Delta F1_{avg}$	Dimensionless
$\Delta F2_{max}$ 99% percentile	kHz
F_0	kHz
F_n	kHz
F_0 - F_n	kHz
$F_1 - F_0$	kHz
$F_{n+5} - F_n$	kHz
F_{tx} @ ±2 MHz offset	dBm
F_{tx} @ ≥± 3MHz offset	dBm
# in-band emission exceptions	Dimensionless
Max in-band exception	dBm
P_{avg} *CTE*	dBm
P_k - P_{avg} *CTE*	dBm
Fs_i	kHz
$Fs_1 - Fs_p$	kHz
$Fs_i - F_0$	kHz
$Fs_i - Fs_j$	kHz
$P_{ref,dev} / P_{ref,ave}$	dB
$P_{n,dev} / P_{n,ave}$	dB
Φ_n	°
$\Phi_n - \Phi_{n-1}$	°
Amplitude	dBm
FM deviation	kHz

If **method** is set to 4 and the quantity is an in-band emission quantity then both 100 kHz resolution and 1MHz resolution spectra are collected.

The data which can be read back represents in-band emissions as a function of frequency. Results are available for both the final 1MHz resolution spectrum and the 100kHz resolution spectrum which was summed to generate the 1MHz spectrum. This provides greater visibility of what is dominating the in-band emissions.

If the **ChanMask** contains a single channel, then the available data includes the average as well as the minimum, maximum and current values of the spectra. If the **ChanMask** contains more than one channel, then the average spectra are not available.

The available data can contain the following fields:

1. *81*float : Min_1MHz*. The lowest recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
2. *81*float : Max_1MHz*. The highest recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
3. *81*float : Curr_1MHz*. The last recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
4. *81*float : Avg_1MHz*. The average recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz. This field is only present if the *ChanMask* contained a single channel.
5. *810*float : Min_100kHz*. The lowest recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
6. *810*float : Max_100kHz*. The highest recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
7. *810*float : Curr_100kHz*. The last recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
8. *810*float : Avg_100kHz*. The average recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz. This field is only present if the *ChanMask* contained a single channel.

All spectra are in units of dBm.

n is the number of bytes of data which are available to be read back, or zero if no data is available.

5.3.33 getResultsPlotInt16

getResultsPlotInt16() reads back the data which was captured using a previous polltResultsPlot() command. This command should be used when the captured data consisted of an array of int16.

x = sapphire.getResultsPlotInt16(offset , count)

offset is the offset from the start of the buffer of the data to be read back (see pollResultsPlot). The offset is in units of 2 bytes.

count is the number of int16 to be read back from the buffer.

x is the data read back from the buffer. x is a vector_int16 which must be imported from the Sapphire module.

5.3.34 getResultsPlotFloat

getResultsPlotFloat() reads back the data which was captured using a previous polltResultsPlot() command. This command should be used when the captured data consisted of an array of float.

x = sapphire.getResultsPlotFloat(offset , count)

offset is the offset from the start of the buffer of the data to be read back (see pollResultsPlot). The offset is in units of 4 bytes.

count is the number of float to be read back from the buffer.

x is the data read back from the buffer. x is a vector_float which must be imported from the Sapphire module.

5.3.35 getResultsPlotMeta

getResultsPlotMeta() returns the meta data associated with the last pollResultsPlot() capture.

meta = sapphire.getResultsPlotMeta()

meta is a structure with the following fields:

1. timestamp indicating the time of the P_0 location in units of 250 ns since the Unix epoch of 1970-01-01 00:00:00 + 0000 (UTC).
2. nsamples indicating the number of samples available to be read back.
3. rfchan indicating the LE RF channel number on which the packet was collected.
4. phy indicating the phy of the packet for which the data was collected. Possible values are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	Mbps, coded, S = 8

5. pkttlen. For non-CTE quantities this indicates the number of octets in the payload of the packet for which the data was collected. For CTE quantities the lower 24bits indicate the length of the CTE in slots, whilst the top 8 bits indicate the CTE type.

5.3.36 clearResults

clearResults() erases all test results accumulated in the Sapphire application by the signal analyser mode or the advertise/scan mode. This command can be executed at any time.

res = sapphire.clearResults()

res will be set to False if the the command fails.

5.3.37 setLimits

setLimits() informs the Sapphire application of the test limits to be applied. The test limits are used in phy tester mode, signal analyser mode and advertise/scan mode. The signal analyser mode and advertise/scan mode share the same set of limits, whilst the phy tester mode has its own set of limits. This command can be executed at any time

res = sapphire.setLimits(x)

x is a vector_int16 containing the test limits. vector_int16 must be imported from the Sapphire module (see example signal analyser code). The length of this vector determines which limits are being set.

If the vector is of length 400, then phy tester limits are being downloaded. These are comprised of 8 groups of 50 limits. Not all the limits in each group are used. The test limits in each group are:

Limit	Test	Quantity	Limit type	Units
1	Ouput power	P_{avg}	Lower	0.1 dBm
2	Ouput power	P_{avg}	Upper	0.1 dBm
3	Ouput power	Pk - P_{avg}	Upper	0.1 dB
4	Modulation characteristics	$\Delta F1_{avg}$	Lower	100 Hz
5	Modulation characteristics	$\Delta F1_{avg}$	Upper	100 Hz
6	Modulation characteristics	$\Delta F2_{avg} / \Delta F1_{avg}$	Lower	0.001
7	Modulation characteristics	ΔF2 99.9% percentile	-	100 Hz
8	Carrier frequency & drift	$\lvert F_0 \rvert$	Upper	100 Hz
9	Carrier frequency & drift	$\lvert F_n \rvert$	Upper	100 Hz
10	Carrier frequency & drift	$\lvert F_1 - F_0 \rvert$	Upper	100 Hz
11	Carrier frequency & drift	$\lvert F_0 - F_n \rvert$	Upper	100 Hz
12	Carrier frequency & drift	$\lvert F_n - F_{n-5} \rvert$	Upper	100 Hz
13	In-band emissions	P_{tx} @ ±2 or ±4 MHz	Upper	0.1 dBm
14	In-band emissions	P_{tx} @ ≥3 or ≥5 MHz	Upper	0.1dBm
15	In-band emissions	Number of exceptions	Upper	Integer
16	In-band emissions	P_{tx} maximum exception	Upper	0.1 dBm
17	-	Not used	-	-
18	-	Not used	-	-
19	-	Not used	-	-
20	PER for scan mode	PER	Upper	0.1%
21	C/I	Number of exceptions	Upper	Integer
22	C/I	C/I exception level	Upper	0.1 dBm

23	Blocking	Number of exceptions at higher level	Upper	Integer
24	Blocking	Number of failures at lower level	Upper	Integer
25	Blocking	Lower level at which to allow exceptions	-	0.1 dBm
26	All receiver tests	BER for payload ≤ 37	Upper	0.0001%
27	All receiver tests	BER for 37 < payload ≤ 63	Upper	0.0001%
28	All receiver tests	BER for 63 < payload ≤ 127	Upper	0.0001%
29	All receiver tests	BER for 127 < payload ≤ 255	Upper	0.0001%
30	Ouput power – CTE	P_{avg}	Lower	0.1 dBm
31	Ouput power – CTE	P_{avg}	Upper	0.1 dBm
32	Ouput power- CTE	Pk - P_{avg}	Upper	0.1 dB
33	CTE receive tests	Max MRP	Upper	0.001
34	CTE receive tests	Max MRP percent	Upper	0.1 %
35	CTE receive tests	MRPD	Upper	0.001
36	CTE drift tests	Fs_i	Upper	100 Hz
37	CTE drift tests	Fs_1-Fp	Upper	100 Hz
38	CTE drift tests	Fs_i-F_0	Upper	100 Hz
39	CTE drift tests	Fs_i-Fs_j	Upper	100 Hz
40	CTE power stability	$P_{ref,dev}$ / $P_{ref,ave}$ 1µs slots	Upper	0.01
41	CTE power stability	$P_{n,dev}$ / $P_{n,ave}$ 1µs slots	Upper	0.01
42	CTE power stability	$P_{ref,dev}$ / $P_{ref,ave}$ 2µs slots	Upper	0.01
43	CTE power stability	$P_{n,dev}$ / $P_{n,ave}$ 2µs slots	Upper	0.01
44	CTE switching integrity	$P_{m,X,AVE,ON}$-$P_{m,X,AVE,OFF}$	Lower	0.1 dB
45	-	Not used	-	-
46	-	Not used	-	-
47	-	Not used	-	-
48	-	Not used	-	-
49	-	Not used	-	-
50	-	Not used	-	-

The 8 groups of test limits are applied to different phys:

Group	Bit rate	Coding	Modulation index
1	2 Mbps	Uncoded	Standard
2	1 Mbps	Uncoded	Standard
3	500 kbps	Coded, S=2	Standard
4	125 kbps	Coded, S=8	Standard
5	2 Mbps	Uncoded	Stable
6	1 Mbps	Uncoded	Stable
7	500 kbps	Coded, S=2	Stable
8	125 kbps	Coded, S=8	Stable

If the vector is of length 80 then of 4 groups of 20 test limits are downloaded. These test limits are used by the signal analyser and the advertise/scan mode. The test limits in each group are:

Limit	Test	Quantity	Limit type	Units
1	Ouput power	P_{avg}	Lower	0.1 dBm
2	Ouput power	P_{avg}	Upper	0.1 dBm
3	Ouput power	Pk - P_{avg}	Upper	0.1 dB
4	Modulation characteristics	$\Delta F1_{avg}$	Lower	100 Hz
5	Modulation characteristics	$\Delta F1_{avg}$	Upper	100 Hz
6	Modulation characteristics	$\Delta F2_{avg} / \Delta F1_{avg}$	Lower	0.001
7	Modulation characteristics	ΔF2 99.9% percentile	-	100 Hz
8	Carrier frequency & drift	$\lvert F_0 \rvert$	Upper	100 Hz
9	Carrier frequency & drift	$\lvert F_n \rvert$	Upper	100 Hz
10	Carrier frequency & drift	$\lvert F_1 - F_0 \rvert$	Upper	100 Hz
11	Carrier frequency & drift	$\lvert F_0 - F_n \rvert$	Upper	100 Hz
12	Carrier frequency & drift	$\lvert F_n - F_{n-5} \rvert$	Upper	100 Hz
13	In-band emissions	P_{tx} @ ±2 or ±4 MHz	Upper	0.1 dBm
14	In-band emissions	P_{tx} @ ≥3 or ≥5 MHz	Upper	0.1dBm
15	In-band emissions	Number of exceptions	Upper	Integer
16	In-band emissions	P_{tx} maximum exception	Upper	0.1 dBm
17	-	Not used	-	-
18	-	Not used	-	-
19	-	Not used	-	-
20	PER for scan mode	PER	Upper	0.1%

The 4 groups of test limits are applied to different phys:

Group	Bit rate	Coding
1	2 Mbps	Uncoded
2	1 Mbps	Uncoded
3	500 kbps	Coded, S=2
4	125 kbps	Coded, S=8

If the vector length is 24, then 2 groups of 12 test limits are downloaded. These limits are used by the signal analyser. These limits correspond to the transmitter CTE tests:

Limit	Test	Quantity	Limit type	Units
1	Ouput power – CTE	P_{avg}	Lower	0.1 dBm
2	Ouput power – CTE	P_{avg}	Upper	0.1 dBm
3	Ouput power- CTE	Pk - P_{avg}	Upper	0.1 dB
4	CTE drift tests	Fs_i	Upper	100 Hz
5	CTE drift tests	Fs_1-Fp	Upper	100 Hz
6	CTE drift tests	Fs_i-F_0	Upper	100 Hz
7	CTE drift tests	Fs_i-Fs_j	Upper	100 Hz
8	CTE power stability	$P_{ref,dev}$ / $P_{ref,ave}$ 1µs slots	Upper	0.01
9	CTE power stability	$P_{n,dev}$ / $P_{n,ave}$ 1µs slots	Upper	0.01
10	CTE power stability	$P_{ref,dev}$ / $P_{ref,ave}$ 2µs slots	Upper	0.01
11	CTE power stability	$P_{n,dev}$ / $P_{n,ave}$ 2µs slots	Upper	0.01
12	CTE switching integrity	$P_{m,X,AVE,ON}$-$P_{m,X,AVE,OFF}$	Lower	0.1 dB

The 2 groups of test limits are applied to different phys:

Group	Bit rate	Coding
1	2 Mbps	Uncoded
2	1 Mbps	Uncoded

res will be set to False if the the command fails.

5.3.38 getLimits

getLimits(p) returns from the Sapphire application of the test limits which are being applied. The test limits are used in phy tester mode, signal analyser mode and advertise/scan mode. The signal analyser mode and advertise/scan mode share the same set of limits, whilst the phy tester mode has its own set of limits. The limits returned are determined by the parameter p. This command can be executed at any time. This command is useful if it is desired to alter a few limits from their default values. It makes it possible to read back the default limits, modify them and the write back using setLimits().

x = sapphire.getLimits(p)

p determines which set of limits are read back. Possible values are:

p	Limits returned
0	8 groups fo 50 phy tester limits
1	4 groups of 20 signal analyser and advertise/scan limits
2	2 groups of 12 signal analyser CTE limits

x is a vector_int16 containing the test limits. vector_int16 must be imported from the Sapphire module (see example signal analyser code). The length of this vector varies depending on the parameter **p**. See setLimits for a description of the limits format.

5.3.39 setRxAtten

setRxAtten() sets the receiver frontend attenuation. This command can be executed at any time.

res = sapphire.setRxAtten(atten)

atten contains the receiver frontend attenuation in units of 0.5 dB. The permissible attenuator range is 0 to 31.5 dB.

res will be set to False if the the command fails.

5.3.40 getRxAtten

getRxAtten returns the current setting of the receiver front end attenuator.

atten = sapphire.getRxAtten()

atten contains the receiver frontend attenuation in units of 0.5 dB. The permissible attenuator range is 0 to 31.5 dB.

5.3.41 setRxPort

setRxPort() determines whether the Monitor In port or the Tx/Rx port should be used for reception. The Monitor In port is has a noise figure of +6 dB and so is ideally suited for performing off-air measurements. The Tx/Rx port has a noise figure of +46 dB but can handle signals as large as +27 dBm. It is therefore ideally suited for conducted measurements. This command can be executed at any time.

res = sapphire.setRxPort(port)

port is the receiver port to use for reception and can be one of the following values:

0	Monitor In
1	Tx/Rx port

res will be set to False if the the command fails.

5.3.42 getRxPort

getRxPort() returns the port currently in use for reception. The Monitor In port is has a noise figure of +6 dB and so is ideally suited for performing off-air measurements. The Tx/Rx port has a noise figure of +46 dB but can handle signals as large as +27 dBm. It is therefore ideally suited for conducted measurements. This command can be executed at any time.

port = sapphire.getPort()

port is the receiver port used for reception and can be one of the following values:

0	Monitor In
1	Tx/Rx port

5.3.43 setDIOVolts

setDIOVolts() determines whether the digital IO voltage is supplied by *TLF3000* or is supplied by an external source. If *TLF3000* supplies the digital IO voltage then it is fixed at 3.3 V. The *TLF3000* is able to provide up to 500 mA on the 3v3 supply. If an external voltage is used for the digital IO then it must be in the range 0.8 V to 3.6 V. This command can be executed at any time.

res = sapphire.setDIOVolts(ext)

ext determines whether the IO voltage is supplied internally or externally:

0	3.3 V IO supplied by *TLF3000*
1	0.8 V to 3.6 V IO supplied externally

res will be set to False if the the command fails.

5.3.44 getDIOVolts

getDIOVolts() returns whether the digital IO voltage is supplied by *Mor TLF3000 eph30* or is supplied by an external source. If *TLF3000* supplies the digital IO voltage then it is fixed at 3.3 V. The *TLF3000* is able to provide up to 500 mA on the 3v3 supply. If an external voltage is used for the digital IO then it must be in the range 0.8 V to 3.6 V. This command can be executed at any time.

ext = sapphire.getDIOVolts( )

ext indicates whether the IO voltage is supplied internally or externally:

0	3.3 V IO supplied by *TLF3000*
1	0.8 V to 3.6 V IO supplied externally

5.3.45 getError

getError() returns the oldest error message from the Sapphire error queue. Once read, this eror message is removed from the error queue.

msg = sapphire.getError()

msg is a string containing the oldest message of the Sapphire error queue. If the queue is empty, then an empty string is returned.

5.3.46 exitApp

exitApp() will cause the Sapphire application to exit and control return to the *TLF3000* supervistor program,

res = sapphireexitApp()

res will be set to False if the the command fails.

5.3.47 hardwareReset

hardwareReset() will cause the *TLF3000* to reboot.

sapphire.hardwareReset()

5.3.48 powerDown

powerDown() will power down the *TLF3000*.

sapphire.powerDown()

5.3.49 getFriendlyName

getFriendlyName() will return the friendly name of the *TLF3000* unit.

name = getFriendlyName()

name is a string containging the friendly name of the *TLF3000* unit.

5.3.50 getSerialNumber

geSerialNumber() will return the serial number of the *TLF3000* unit.

sn = getSerialNumber()

sn is an integer containing the serial number of the *TLF3000* unit.

6 C dll Interface

6.1 Overview

Support is provided for driving the *TLF3000* using a C dll. This support is available for both 32 bit and 64 bits.

6.1.1 Connecting to the *TLF3000*

To connect to the *TLF3000* it is first necessary to find a list of accessible *TLF3000* by calling sapphire_search(). The required *TLF3000* can be identified and connected to using sapphire_connect(). This will start the Sapphire application running on the *TLF3000*.

```
sapphire_search_result_t* results;

while (true) {
	int N;
	sapphire_search(&results, &N);
	for (int i = 0; i < N; i++) {
		printf("Found: %d\n", results[i].serial_number);
	}
	if (N) {
		// This simply connects to the first TLF3000 which is found
		printf("Connecting\n");
		sapphire_connect(results[0].handle);
		break;
	}
	std::this_thread::sleep_for(500ms);
}
```

6.1.2 Handling asynchronouos data

A callback should be attached to handle asynchronous data from the *TLF3000*. This is done by calling sapphire_set_data_callback().

```
// Register callback for asynch data channel
sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);
```

Whenever an asynchronous message is received from the *TLF3000*, the asynchronous data callback will be entered. In the example below, the data is unpacked to determine its length and type. Having determined the message type, it is dispatched to an appropriate routine. deserialise is a utility function provided by Frontline to unpack the data returned in the callback.

```
void dataCallback(uint8_t *data, uint32_t len) {
	std::vector<unsigned char> d;
	for (uint32_t k = 0; k < len; k++) d.push_back(data[k]);
	uint16_t len_lsb;
	uint8_t len_msb;
	uint8_t type;
	deserialise(d, len_lsb, len_msb, type);
	switch (type) {
	case 0: StartScript(d); break;
	case 1: TxTest(d); break;
	case 2: RxTest(d); break;
	case 3: EndScript(d); break;
	case 4: StartLine(d); break;
	case 5: EndLine(d); break;
	case 35: VendorSpecific(d); break;
	case 36: DUTClosed(); break;
	case 193: EnvData(d); break;
	case 194: PwrData(d); break;
	default: break;
	}
}
```

6.1.3 Handling errors

Many of the calls to the Sapphire library return a flag to indicate wtheter the call was successful or not. When an error occurs the cause of the error can be read back by calling sapphire_getError(). Each invocation of this method will remove the oldest error from the Sapphire error queue and return it to the caller. When no more messages are available, an empty string is returned.

```
void getError() {
	char err[1024];
	sapphire_getError(err,1024);
	while (err[0]) {
		printf("%s\n", err);
		sapphire_getError(err,1024);
	}
}
```

6.1.4 Closing down

The Sapphire application can be shutdown by calling sapphire_disconnect()

```
sapphire_disconnect();
```

6.1.5 Switching between applications on the *TLF3000*

There is a significant overhead in downloading a new application to the *TLF3000*. The C dll provides a mechanism for stashing an application on the *TLF3000*. This dramatically reduces the time needed to switch between applications. A common example might be the switching between Sapphire (LE) and Zircon (BR/EDR). This can be accomplished the following sequence:

```
zircon_connect(results[0].handle);      // Startup Zircon
zircon_stashExe();                      // Save Zircon in the TLF3000
zircon_suspend();                       // Close down Zircon
sapphire_connect(s_results[0].handle);  // Startup Sapphire
sapphire_stashExe();                    // Save Sapphire in the TLF3000

… run the LE tests here …

sapphire_swapExe(16);                   // Tell the TLF3000 that the next application
                                        // we want to run is Zircon
                                        // (application number 16)
sapphire_suspend();                     // Exit the Sapphire application
zircon_resume();                        // Resume the Zircon application

… run the BR/EDR tests here …
```

6.2 Examples

6.2.1 Running a phy test script

The example demonsrates the sequence of instruction necessary to download and run a phy test script. It also demonstrates how the messages on the asynchronous data channel can be handled. This is a useful starting point for code running on a production line or characterisation rig.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
3. The operating mode is set phy test mode using sapphire_setMode
4. The receiver frontend attenuation is set using sapphire_setRxAtten
5. The receiver port to be used is set using sapphire_setRxPort
6. The source of the IO voltage which will be used for the UART communication with the DUT is set using sapphire_setDIOVolts
7. The cable loss between the DUT and the *TLF3000* unit is set using sapphire_setCableLosses
8. The comport on which the DUT resides is set by calling sapphire_setDUTPort
9. Details of the UART communications between the *TLF3000* unit and the DUT are set using sapphire_setDUTComms
10. The supported features of the DUT are programmed using sapphire_setDUTProperties
11. A test script is then downloaded to the *TLF3000* unit. In this case the test script is read in from a .sta file which has been exported from the Sapphire GUI. This is the most convenient means of generating test scripts. However, test scripts can be generated within the Python code, the format of the test script being set out inTest Script Format . The test script is downloade using sapphire_startScript, followed by a series of calls tosapphire_contScript and a final call to sapphire_endScript .
12. Execution of the test script is triggered by issuing sapphire_runScript .
13. The code then waits for the test script to terminate. During this time a sequence of messages appear on the asynchronous data channel. The callback handles each of these messages and dispactches them to the appropriate routine which then prints out the test results.
14. Finally sapphire_DUTclose is used to termination communications with the DUT. At this point a new DUT could be attached and the test script re-run on the new DUT.

```
HANDLE    sem;

// Cable loss definition

#define nLoss    (3)
float frqs[nLoss] = { 2402.0f , 2440.0f , 2480.0f };
float loss[nLoss] = { 1.0f , 1.1f , 1.2f };


int main(int argc, char **argv) {

   DWORD err;

   sem = CreateSemaphoreA(NULL, 0, 1, "Done");

   // --------------------
   // CONNNECT TO TLF3000
   // --------------------

   sapphire_search_result_t* results;

   while (true) {
         int N;
         sapphire_search(&results, &N);
         for (int i = 0; i < N; i++) {
               printf("Found: %d\n", results[i].serial_number);
         }
         if (N) {
               // This simply connects to the first TLF3000 which is found
               printf("Connecting\n");
               sapphire_connect(results[0].handle);
               break;
         }
         std::this_thread::sleep_for(500ms);
   }

   // -------------
   // GENERAL SETUP
   // -------------

   // Register callback for asynch data channel
   sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);
```

```
// Enter phy tester mode
if (sapphire_setMode(0) != SAPPHIRE_NO_ERROR) getError();

// Set rx frontend attenuation in units of 0.5dB
if (sapphire_setRxAtten(0) != SAPPHIRE_NO_ERROR) getError();

// Set the rx port to be tx / rx
if (sapphire_setRxPort(1) != SAPPHIRE_NO_ERROR) getError();

// Turn on 3.3V IO supply from TLF3000
if (sapphire_setDioVolts(true) != SAPPHIRE_NO_ERROR) getError();

// Set cable loss
if (sapphire_setCableLosses( nLoss, frqs, loss ) != SAPPHIRE_NO_ERROR) getError();

// Set which comport to use
// A negative value indicates direct connection to the TLF3000
if (sapphire_setDUTPort(9) != SAPPHIRE_NO_ERROR) getError();

// Serial communications to DUT
// Transport : 0 = direct test mode, 1 - H4, 2 = H5, 3 = BCSP
// Baud rate
// h/w flow control
// s/w flow control
// stop bits
// parity : 0 = none, 1 = even, 2 = odd
// crc
if (sapphire_setDUTComms(1 | 0x80, 115200, false, false, 1, 0, false) != SAPPHIRE_NO_ERROR) getError();

// DUT supported features
// Role: 0 = central, 1 = peripheral, 2 = broadcaster, 3 = observer
// Support for advertising extensions
// Support for data length extensions
// Max supported advertising octets
// Max supported rx octets
// Max supported rx time
// Max supported tx octets
// Max supported tx time
// Support for 2Mbps
// Support for coded phys
// Support for stable modulation index
// Support for Rx AoA 2us slots
// Support for Tx AoA 2us slots
```

```
// Support for Rx AoD 2us slots
// Support for Tx AoD 2us slots
// Support for Rx AoA 1us slots
// Support for Tx AoA 1us slots
// Support for Rx AoD 1us slots
// Support for Tx AoD 1us slots
// Maximum CTE length
// Number of antenna
// Maximum switching pattern length

if (sapphire_setDUTProperties(0, false, false, 37, 27, 328, 27, 328, false, false, false,
                              false, false, false, false, false, false, false, 0, 0, 0)
                             != SAPPHIRE_NO_ERROR) getError();

// ----------------------------
// Load and run the test script
// ----------------------------

// Start download of test script
// arg1 = Overwrite file with existing name
// arg2 = Do not place in FLASH(currently not supported)
// arg3 = Test script file name
if (sapphire_startScript(true, false, "TestScript") != SAPPHIRE_NO_ERROR) getError();

FILE *fid;
fopen_s(&fid, "D:/LE_1M_production_fast_short.sta", "rt");

int n = 0;
char line[4096];
while (fgets(line, 4096, fid)) {
      n++;
      // Skip first 5 header lines in .sta file
      if (n <= 5) continue;
      // Add lines to test script file
      if (sapphire_contScript(line) != SAPPHIRE_NO_ERROR) getError();
}

fclose(fid);

// Signal that the test script is now complete
if (sapphire_endScript() != SAPPHIRE_NO_ERROR) getError();

// Run the test script
```

```
    // arg1 = Number of times script should be run
    // arg2 = Run to completion
    // arg3 = Raeduce test time by aborting rx tests when result is known
    // arg4 = Reorder tx / rx tests to reduce test time
    // arg5 = Test script not in FLASH
    // arg6 = Test script file nane - same as in startScript call
    if (sapphire_runScript(1, true, true, true, false, "TestScript") != SAPPHIRE_NO_ERROR) getError();

    err = WaitForSingleObject(sem, 120 * 1000);
    if (err != WAIT_OBJECT_0) printf("Error waiting for script to terminate\n");

    getError();

    // ------------------------------------------------------------------
    // Close the DUT so we are ready to run new tests on a new DUT
    // It is not necessary to reload test scripts of reload configuration
    // ------------------------------------------------------------------

    if (sapphire_DUTClose()) getError();

    // Wait for the callback to verify that the DUT has been closed
    err = WaitForSingleObject(sem, 1000);
    if (err != WAIT_OBJECT_0) printf("Error waiting for DUT to close\n");

    // A new DUT can be attached at this point

    // …

    // Stop the Sapphire application running on the TLF3000

    sapphire_disconnect();

    printf("All done\n");
    getchar();

    return 0;
}
```

6.2.1.1 Test progress

The progress of the testing is reported via asynchronous callbacks. Callbacks are generated whenever:

1. A new script is started
2. A new line in a script is started
3. A line in a script is completed, with an indication of pass or fail
4. A script is completed, with an indication of pass or fail
5. The serial communications to the DUT are terminated

```
void StartScript(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	unused;
	uint8_t		id;
	uint8_t		ok;
	deserialise(d, hdr, unused, id, ok);
	printf("Test script execution starting : %d\n", id);
}

void EndScript(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	unused;
	uint8_t		id;
	uint8_t		ok;
	deserialise(d, hdr, unused, id, ok);
	printf("Test script execution terminating : %d\n", id);
	if (ok == 0)
		printf("Test script result : PASS\n");
	else
		printf("Test script result : FAIL\n");
	ReleaseSemaphore(sem, 1, NULL);
}

void StartLine(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	deserialise(d, hdr, line, id, num);
	printf("Starting test script line %d\n" , line);
}

void EndLine(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	result;
	deserialise(d, hdr, line, id, num, result);
	printf("End test script line %d" , line);
	if (result == 0)
		printf(" : FAIL\n");
	else
		printf(" : PASS\n");
}

void DUTClosed() {
	printf("DUT Closed ... ready to run new test\n");
	ReleaseSemaphore(sem, 1, NULL);
}
```

6.2.1.2 Tables for decoding results

The example code utilises the following tables to decode the test results returned.

```
// Table to indicate which tests are using stable modulation index

int txStable[] = { 0, 0, 0, 0, 0, 0, 0, 0, 1,
                     0, 1, 0, 0, 0, 0, 0, 0, 0 };

int rxStable[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                     0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
                     1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
                     0, 0, 1, 1, 1, 1, 1, 1, 1, 1 };

// Text strings for output

const char *phys[] = { "2Mbps uncoded", "1Mbps uncoded", "500kbps coded s=2", "125kbps
coded s=8" };

const char *stables[] = { "standard modulation index", "stable modulation index" };
```

6.2.1.3 Transtmitter test results

Transmitter test results are directed through a single routine called by the main asynchronous callback handler. The nature of the transmitter test is determined and an appropriate routine for handling the asynchronous data is called.

```
void TxTest(std::vector<unsigned char> d) {
       uint32_t     hdr;
       uint16_t     line;
       uint8_t      id;
       uint8_t      num;
       deserialise(d, hdr, line, id, num);
       switch (num) {
       case 1: outputPower(d); break;
       case 3 : inBandEmissions(d); break;
       case 5 : modulationCharacteristics(d); break;
       case 6 : carrierDrift(d); break;
       case 8 : inBandEmissions(d); break;
       case 9 : modulationCharacteristics(d); break;
       case 10 : modulationCharacteristics(d); break;
       case 11 : modulationCharacteristics(d); break;
       case 12 : carrierDrift(d); break;
       case 13 : modulationCharacteristics(d); break;
       case 14 : carrierDrift(d); break;
       default: break;
       }
}
```

6.2.1.3.1 Transmitter output power results

Measurements related to output power are decoded by the following routine:

```
// Callback for tx power messages

void outputPower(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	float		Pmin;
	float		Pmax;
	float		Pkmax;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, Pmin, Pmax, Pkmax, result,
			frame);
	printf("    Output Power");
	printf(" : %s %s", phys[phy], stables[txStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        Min power  : %.1f dBm\n", Pmin);
	printf("        Max power  : %.1f dBm\n", Pmax);
	printf("        Peak power : %.1f dB\n", Pkmax);
	if (result)
		printf("        Result     : FAIL\n");
	else
		printf("        Result     : PASS\n");
}
```

6.2.1.3.2 Transmitter in-band emission results

Measurements related to in-band emissions are decoded by the following routine. The vector "spectrum" holds the results of the in-band emission measurements on 81 channels in units of dBm. The first channel is 2400MHz and the last channel is 2480MHz.

```
// Callback for tx in - band emission messages

void inBandEmissions(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	float		P1max;
	float		P2max;
	float		maxXcep;
	uint32_t	numXcep;
	uint32_t	result;
	uint32_t	frame;
	std::vector<float> spectrum;
	spectrum.resize(81);
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, P1max, P2max, maxXcep,
			numXcep, result, sprectrum, frame);
	printf("    In-band emissions");
	printf(" : %s %s", phys[phy], stables[txStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	if (phy == 0) {
		printf("	Ptx +/- 3MHz		: %.1f dBm\n", P1max);
		printf("	Ptx >= +/- 4MHz		: %.1f dBm\n", P2max);
	}
	else {
		printf("	Ptx +/- 2MHz		: %.1f dBm\n", P1max);
		printf("	Ptx >= +/- 3MHz		: %.1f dBm\n", P2max);
	}
	printf("	Maximum exception    : %.1f dBm\n", maxXcep);
	printf("	Number of exceptions : %d\n", numXcep);
	if (result & 1)
		printf("	Result			: FAIL\n");
	else
		printf("	Result			: PASS\n");
}
```

6.2.1.3.3 Transmitter modulation characteristics results

Measurements related to modulation characteristics are decoded by the following routine:

```
// Callback for tx modulation characteristics messages

void modulationCharacteristics(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	float		DF1min;
	float		DF1max;
	float		minRatio;
	float		frac;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, DF1min, DF1max, minRatio,
			frac, result, frame);
	printf("    Modulation characteristics");
	printf(" : %s %s", phys[phy], stables[txStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("         minimum Delta F1max  : %.1f kHz\n", DF1min);
	printf("         maximum Delta F1max  : %.1f kHz\n", DF1max);
	printf("         min DeltaF2/DeltaF1  : %.3f\n", minRatio);
	if (phy == 0)
		printf("         DeltaF2 > 370kHz     : %.3f %%\n", 100 * frac);
	else
		printf("         DeltaF2 > 185kHz     : %.3f %%\n", 100 * frac);
	if (result)
		printf("         Result               : FAIL\n");
	else
		printf("         Result               : PASS\n");
}
```

6.2.1.3.4 Transmitter carrier frequency offset and drift results

Measurements related to carrier frequency offset and drift are decoded by the following routine:

```
// Callback for carrier offset & drift messages

void carrierDrift(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	float		minFo;
	float		maxFo;
	float		minFn;
	float		maxFn;
	float		minFoFn;
	float		maxFoFn;
	float		minF1Fo;
	float		maxF1Fo;
	float		minFnFm;
	float		maxFnFm;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, minFo, maxFo, minFn,
			maxFn, minFoFn, maxFoFn, minF1Fo, maxF1Fo, minFnFm, maxFnFm,
			result, frame);
	printf("    Carrier offset & drift");
	printf(" : %s %s", phys[phy], stables[txStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        minimum Fo         : %.1f kHz\n", minFo);
	printf("        maximum Fo         : %.1f kHz\n", maxFo);
	printf("        minimum Fn         : %.1f kHz\n", minFn);
	printf("        maximum Fn         : %.1f kHz\n", maxFn);
	printf("        minimum Fo - Fn    : %.1f kHz\n", minFoFn);
	printf("        maximum Fo - Fn    : %.1f kHz\n", maxFoFn);
	printf("        minimum F1 - Fo    : %.1f kHz\n", minF1Fo);
	printf("        maximum F1 - Fo    : %.1f kHz\n", maxF1Fo);
	printf("        minimum Fn - Fn-5  : %.1f kHz\n", minFnFm);
	printf("        maximum Fn - Fn-5  : %.1f kHz\n", maxFnFm);
	if (result)
		printf("        Result             : FAIL\n");
	else
		printf("        Result             : PASS\n");

	// Calculate average frequency error
	FrequencyError = 0.5f * (minFo + maxFo);
}
```

6.2.1.4 Receiver test results

Receiver test results are directed through a single routine called by the main asynchronous callback handler. The nature of the receiver test is determined and an appropriate routine for handling the asynchronous data is called.

```
void RxTest(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	deserialise(d, hdr, line, id, num);
	switch (num) {
	case 1: sensitivity(d); break;
	case 3 : ci(d); break;
	case 0x83 : ciXcep(d); break;
	case 4 : blocking(d); break;
	case 0x84 : blockingXcep(d); break;
	case 5 : intermodulation(d); break;
	case 6 : maxinput(d); break;
	case 7 : integrity(d); break;
	case 8 : sensitivity(d); break;
	case 9 : ci(d); break;
	case 0x89 : ciXcep(d); break;
	case 10 : blocking(d); break;
	case 0x8A : blockingXcep(d); break;
	case 11 : intermodulation(d); break;
	case 12 : maxinput(d); break;
	case 13 : integrity(d); break;
	case 14 : sensitivity(d); break;
	case 15 : ci(d); break;
	case 0x8F : ciXcep(d); break;
	case 16 : blocking(d); break;
	case 0x90 : blockingXcep(d); break;
	case 17 : intermodulation(d); break;
	case 18 : maxinput(d); break;
	case 19 : integrity(d); break;
	case 20 : sensitivity(d); break;
	case 21 : ci(d); break;
	case 0x95 : ciXcep(d); break;
	case 22 : blocking(d); break;
	case 0x96 : blockingXcep(d); break;
	case 23 : intermodulation(d); break;
	case 24 : maxinput(d); break;
	case 25 : integrity(d); break;
	case 26 : sensitivity(d); break;
	case 27 : sensitivity(d); break;
	case 28 : ci(d); break;
	case 0x9C : ciXcep(d); break;
	case 29 : ci(d); break;
	case 0x9D : ciXcep(d); break;
	case 30 : integrity(d); break;
	case 31 : integrity(d); break;
	case 32 : sensitivity(d); break;
	case 33 : sensitivity(d); break;
	case 34 : ci(d); break;
	case 0xA2 : ciXcep(d); break;
	case 35 : ci(d); break;
	case 0xA3 : ciXcep(d); break;
```

```
        case 36 : integrity(d); break;
        case 37 : integrity(d); break;
        default: break;
        }
}
```

6.2.1.4.1 Receiver sensitivity results

Measurements related to receiver sensitivity are decoded by the following routine:

```
// Callback for rx sensitivity messages

void sensitivity(std::vector<unsigned char> d) {
        uint32_t      hdr;
        uint16_t      line;
        uint8_t       id;
        uint8_t       num;
        uint32_t      chan;
        uint32_t      phy;
        uint32_t      pktlen;
        int32_t       wPwr;
        uint32_t      numPkt;
        uint32_t      rxPkt;
        float         PER;
        uint32_t      result;
        uint32_t      frame;
        deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, numPkt, rxPkt, PER,
                    result, frame);
        printf("    Rx sensitivity");
        printf(" : %s %s", phys[phy], stables[rxStable[num]]);
        printf(" : RF Channel %d", chan);
        printf(" : Packet length %d\n", pktlen);
        printf("         Wanted signal level : %.1f dBm\n", 0.1*wPwr);
        printf("         Transmitted packets : %d\n", numPkt);
        printf("         Received packets    : %d\n", rxPkt);
        printf("         PER                 : %.1f %%\n", 100 * PER);
        if (result)
              printf("         Result              : FAIL\n");
        else
              printf("         Result              : PASS\n");
}
```

6.2.1.4.2 Receiver C/I results

Measurements related to receiver C/I measurements are decoded by the following two routines. One routine handles C/I measurements and the other handles the number of C/I exceptions for the entire C/I sweep across channels.

```
// Callback for rx C / I messages

void ci(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	iFreq;
	int32_t		iPwr;
	uint32_t	numPkt;
	uint32_t	rxPkt;
	float		PER;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, iFreq, iPwr, numPkt,
			rxPkt, PER, result, frame);
	printf("    C/I");
	printf(" : %s %s", phys[phy], stables[rxStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        Wanted signal level     : %.1f dBm\n", 0.1*wPwr);
	printf("        Interferer frequency    : %d MHz\n", iFreq);
	printf("        Interferer signal level : %.1f dBm\n", 0.1*iPwr);
	printf("        Transmitted packets     : %d\n", numPkt);
	printf("        Received packets        : %d\n", rxPkt);
	printf("        PER                     : %.1f %%\n", 100 * PER);
	if (result & 1)
		printf("        Result                  : FAIL\n");
	else if (result == 0)
		printf("        Result                  : PASS\n");
	else
		printf("        Result                  : EXCEPTION\n");
}

// Callback for rx C / I exception messages

void ciXcep(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	nXcep;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, nXcep, result,
			frame);
	printf("    C/I exceptions");
```

```
        printf(" : %s %s", phys[phy], stables[rxStable[num & 0x7F]]);
        printf(" : RF Channel %d", chan);
        printf(" : Packet length %d\n", pktlen);
        printf("        Number of exceptions    : %d\n", nXcep);
        if (result)
              printf("        Result                  : FAIL\n");
        else
              printf("        Result                  : PASS\n");
}
```

6.2.1.4.3 Receiver blocking results

Measurements related to receiver blocking performance are decoded by the following two routines. The first routine handles individual blocking measurements, whilst the second handles the overall number of blocking exceptions.

```
// Callback for rx blocking messages

void blocking(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	iFreq;
	int32_t		iPwr;
	uint32_t	numPkt;
	uint32_t	rxPkt;
	float		PER;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, iFreq, iPwr, numPkt,
			rxPkt, PER, result, frame);
	printf("    Blocking");
	printf(" : %s %s", phys[phy], stables[rxStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        Wanted signal level  : %.1f dBm\n", 0.1*wPwr);
	printf("        Blocker frequency    : %d MHz\n", iFreq);
	printf("        Blocker signal level : %.1f dBm\n", 0.1*iPwr);
	printf("        Transmitted packets  : %d\n", numPkt);
	printf("        Received packets     : %d\n", rxPkt);
	printf("        PER                  : %.1f %%\n", 100 * PER);
	if (result & 1)
		printf("        Result               : FAIL\n");
	else if (result & 2)
		printf("        Result               : EXCEPTOPM - level 1\n");
	else if (result & 4)
		printf("        Result               : EXCEPTOPM - level 2\n");
	else
		printf("        Result               : PASS\n");
}

// Callback for rx blocking exception messages

void blockingXcep(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	nXcep;
	uint32_t	mXcep;
	uint32_t	result;
	uint32_t	frame;
```

```
        deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, nXcep, mXcep,
                    result, frame);
        printf("    Blocking exceptions");
        printf(" : %s %s", phys[phy], stables[rxStable[num & 0x7F]]);
        printf(" : RF Channel %d", chan);
        printf(" : Packet length %d\n", pktlen);
        printf("         Level 1 exceptions   : %d\n", nXcep);
        printf("         Level 2 exceptions   : %d\n", mXcep);
        if (result)
              printf("         Result               : FAIL\n");
        else
              printf("         Result               : PASS\n");
}
```

6.2.1.4.4 Receiver intermodulation results

Measurements related to the receiver intermodulation performance are handled by the following routine:

```
// Callback for rx intermodulation messages

void intermodulation(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	n;
	uint32_t	side;
	int32_t		iPwr;
	uint32_t	numPkt;
	uint32_t	rxPkt;
	float		PER;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, n, side, iPwr,
			numPkt, rxPkt, PER, result, frame);
	printf("    Intermodulation");
	printf(" : %s %s", phys[phy], stables[rxStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d", pktlen);
	printf("\n\t\tWanted signal level     : %.1f dBm\n", 0.1*wPwr);
	printf("\t\tIntermodulation spacing : %d\n", n);
	if (side == 0)
		printf("\t\tInterferers are below wanted signal\n");
	else
		printf("\t\tInterferers are above wanted signal\n");
	printf("\t\tInterferer powers       : %.1f dBm\n", 0.1*iPwr);
	printf("\t\tTransmitted packets     : %d\n", numPkt);
	printf("\t\tReceived packets        : %d\n", rxPkt);
	printf("\t\tPER                     : %.1f %%\n", 100 * PER);
	if (result)
		printf("\t\tResult                  : FAIL\n");
	else
		printf("\t\tResult                  : PASS\n");
}
```

6.2.1.4.5 Receiver maximum input results

Measurements related to the receiver maximum input signal level are handled by the following routine:

```
// Callback for rx max input messages

void maxinput(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	numPkt;
	uint32_t	rxPkt;
	float		PER;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, numPkt, rxPkt, PER,
			result, frame);
	printf("    Rx max input");
	printf(" : %s %s", phys[phy], stables[rxStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        Wanted signal level : %.1f dBm\n", 0.1*wPwr);
	printf("        Transmitted packets : %d\n", numPkt);
	printf("        Received packets    : %d\n", rxPkt);
	printf("        PER                 : %.1f %%\n", 100 * PER);
	if (result)
		printf("        Result              : FAIL\n");
	else
		printf("        Result              : PASS\n");
}
```

6.2.1.4.6 Receiver PER integrity results

Measurements related to the receiver PER integrity reporting are handled by the following routine:

```
// Callback for rx PER report integrity messages

void integrity(std::vector<unsigned char> d) {
	uint32_t	hdr;
	uint16_t	line;
	uint8_t		id;
	uint8_t		num;
	uint32_t	chan;
	uint32_t	phy;
	uint32_t	pktlen;
	int32_t		wPwr;
	uint32_t	rep;
	uint32_t	numPkt;
	uint32_t	rxPkt;
	float		PER;
	uint32_t	result;
	uint32_t	frame;
	deserialise(d, hdr, line, id, num, chan, phy, pktlen, wPwr, rep, numPkt, rxPkt,
			PER, result, frame);
	printf("    Integrity report");
	printf(" : %s %s", phys[phy], stables[rxStable[num]]);
	printf(" : RF Channel %d", chan);
	printf(" : Packet length %d\n", pktlen);
	printf("        Wanted signal level : %.1f dBm\n", 0.1*wPwr);
	printf("        Repeat number       : %d\n", rep);
	printf("        Transmitted packets : %d\n", numPkt);
	printf("        Received packets    : %d\n", rxPkt);
	printf("        PER                 : %.1f %%\n" , 100 * PER);
	if (result)
		printf("        Result              : FAIL\n");
	else
		printf("        Result              : PASS\n");
}
```

6.2.2 Programming the signal generator

This example code demonstrates how to program the various signal generator sources and then turn on the signal generator output.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
3. The operating mode is set phy test mode using sapphire_setMode
4. Vectors are created to hold the dirty transmitter table. In this example, the dirty transmitter table contains ten rows.
5. The wanted signal is programmed using sapphire_setWanted. At this stage there is not output from the signal generator since it has not been started.
6. An interferer signal is programmed using sapphire_setInterferer.
7. An in-band CW blocker is programmed using sapphire_setInBandCW.
8. An out–of-band CW blocker is programmed using sapphire_setOutOfBandCW.
9. The signal generator is started by calling sapphire_startSigGen. It is only at this point that the signal generator starts outputting power.
10. Execution is paused until the asynchronous callback for all programmed packets has been called.
11. The signal generator is stopped using sapphire_stopSigGen. This does not affect the programming which has previously been performed.

```
HANDLE sem;

int main(int argc, char **argv) {

	sapphire_search_result_t* results;

	while (true) {
		int N;
		sapphire_search(&results, &N);
		for (int i = 0; i < N; i++) {
			printf("Found: %d\n", results[i].serial_number);
		}
		if (N) {
			// This simply connects to the first TLF3000 which is found
			printf("Connecting\n");
			sapphire_connect(results[0].handle);
			break;
		}
		std::this_thread::sleep_for(500ms);
	}

	// Register callback for asynch data channel
	sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);

	// Enter signal generator mode
	if (sapphire_setMode(1) != SAPPHIRE_NO_ERROR) getError();

	// Transmitter distortion table
	// Each row of 5 entries are:
	//     carrier offset in kHZ
	//     modulation index
	//     drift magnitude in kHz
	//     drift rate in Hz
	//     symbol timing error in ppm

	double distortions[] = {
		100, 0.45, 50, 1250, -50,
		 19, 0.48, 50, 1250, -50,
		 -3, 0.46, 50, 1250,  50,
		  1, 0.52, 50, 1250,  50,
		 52, 0.53, 50, 1250,  50,
		  0, 0.54, 50, 1250, -50,
		-56, 0.47, 50, 1250, -50,
		 97, 0.50, 50, 1250, -50,
```

```
            -25, 0.45, 50, 1250, -50,
           -100, 0,55, 50, 1250,  50 };

      // Program the wanted signal
      // Arg 0  : turn wanted transmission on or off
      // Arg 1  : RF channel number
      // Arg 2  : tranmission power in dBm
      // Arg 3  : phy
      //          0 = 2Mbps uncoded
      //          1 = 1Mbps uncoded
      //          2 = s=2 coded
      //          3 = s=8 coded
      // Arg 4  : access address
      // Arg 5  : CRC seed
      // Arg 6  : false = do not whiten, true = whitem
      // Arg 7  : Payload type
      //          0 = PRBS9
      //          1 = PRBS11
      //          2 = PRBS15
      //          3 = PRBS20
      //          4 = PRBS23
      //          5 = PRBS29
      //          6 = PRBS31
      //          7 = 11110000
      //          8 = 10101010
      //          9 = 00000000
      //         10 = 11111111
      //         11 = 00001111
      //         12 = 01010101
      // Arg 8  : payload length in octets
      // Arg 9  : Interval between start of successive packets in us
      // Arg 10 : power ramp time in us
      // Arg 11 : length of unmodulated carrier at start of packet in us
      // Arg 12 : length of unmodulated carrier at end of packet in us
      // Arg 13 : Mask of IO lines to set when wanted signal is active
      // Arg 14 : Number of entries in distortion table
      // Arg 15 : Distortion table
      if (sapphire_setWanted(true, 0, -10.0, 1, 0x71764129, 0x555555, false, 0, 27, 625, 1500, 5, 2, 2, 0, 2,
                             distortions) != SAPPHIRE_NO_ERROR) getError();

      // Program an interferer
      // Arg 0  : turn interferer transmission on or off
      // Arg 1  : transmission frequency in MHz
      // Arg 2  : tranmission power in dBm
      // Arg 3  : phy
```

```
//          0 = 2Mbps uncoded
//          1 = 1Mbps uncoded
//          2 = s=2 coded
//          3 = s=8 coded
// Arg 4  : Payload type
//          0 = PRBS9
//          1 = PRBS11
//          2 = PRBS15
//          3 = PRBS20
//          4 = PRBS23
//          5 = PRBS29
//          6 = PRBS31
// Arg 5  : Mask of IO lines to set when interferer is active
// Arg 6  : true = continuous transmission, false = packetised transmission
// Arg 7  : payload length in octets (ignored for continuous transmissions)
// Arg 8  : interval between the start of successive packets in us (ignored for continuous transmissions)
if (sapphire_setInterferer(true, 2440, -30, 2, 2, 0, true, 27, 625) != SAPPHIRE_NO_ERROR) getError();

// Program an in-band CW signal
// Arg 0  : turn CW transmission on or off
// Arg 1  : CW source to program
// Arg 2  : transmission frequency in Hz
// Arg 3  : tranmission power in dBm
if (sapphire_setInBandCW(true, 0, 2470000000, -30) != SAPPHIRE_NO_ERROR) getError();

// Program an out-of-band CW signal
// Arg 0  : turn CW transmission on or off
// Arg 1  : transmission frequency in MHz
// Arg 2  : tranmission power in dBm
if (sapphire_setOutOfBandCW(true, 2390, -40) != SAPPHIRE_NO_ERROR) getError();

sem = CreateSemaphoreA(NULL, 0, 1, "Done");

// Start the signal generator
if (sapphire_startSigGen() != SAPPHIRE_NO_ERROR) getError();

DWORD err = WaitForSingleObject(sem, 5000);
if (err == WAIT_TIMEOUT) {
      printf("Wanted signal transmission has not completed\n");
}
else if (err != WAIT_OBJECT_0) {
      printf("Error waiting for sniff\n");
}

// Stop the signal generator
```

```
        if (sapphire_stopSigGen() != SAPPHIRE_NO_ERROR) getError();

        getError();

        sapphire_disconnect();

        return 0;
}
```

6.2.2.1 Completion of transmission of wanted signal

When the transmission of the wanted signal is complete, an asynchronous callback is made. This is handled by the following routine:

```
def siggenDone( d ):
    print( "Signal generator has sent packets" )
```

6.2.3 Analysing waveforms using the signal analyser

This example code demonstrates how to program the signal analyser and instruct it to run. Various results are then retrieved from the signal analyser.

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described in Connecting to the TLF3000
2. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
3. The operating mode is set phy test mode using sapphire_setMode
4. The receiver port to be used is set using sapphire_setRxPort
5. The receiver frontend attenuation is set using sapphire_setRxAtten
6. The signal analayser is set running using sapphire_signalAnalyser . In this instance it is programmed to process packets with the test access address.
7. After 1 second the signal analyser is halted using sapphire_signalAnalyser . It is not actually neceassry to stop the signal analyser running before reading back results.
8. sapphire_pollResultsTable is used to recover the modulation characteristics results table and then further calls are used to receover the carrier frequency and drift results table and the in-band emissions results table. All these tables are printed out.

```
const char *PowerHeadings[] = {
                    "Pavg",
                    "Pk-Pavg"
};

const char *ModulationHeadings[] = {
                    "DF1max",
                    "DF1avg",
                    "DF2max",
                    "DF2avg",
                    "DF2avg/DF1avg",
                    "DF2max 99.9%"
};


const char *DriftHeadings[] = {
                    "Fo",
                    "Fn",
                    "|F1-Fo|",
                    "|Fo-Fn|",
                    "|Fn-Fn-5|"
};

const char *InBandHeadings[] = {
                    "Ftx+/-2MHz",
                    "Ftx+/-(3+n)MHz",
                    "# exceptions",
                    "Max exception"
};


// routine for printing results tables

void printTable(const char**headings, float *x, int num) {
      printf("\n");
      printf("%20s %10s %10s %10s %10s %10s\n", "Quantity", "# Pkts", "Min", "Max", "Avg", "Current");
      int n = 0;
      for (int k = 0; k < num; k++) {
            printf("%20s ", headings[k]);

            printf("%10d %10.1f %10.1f %10.1f %10.1f\n", (int)x[n], x[n + 1], x[n + 2], x[n + 3], x[n + 4]);
            n = n + 5;
      }
      printf("\n");
}
```

```
int main(int argc, char **argv) {

	sapphire_search_result_t* results;

	while (true) {
		int N;
		sapphire_search(&results, &N);
		for (int i = 0; i < N; i++) {
			printf("Found: %d\n", results[i].serial_number);
		}
		if (N) {
			// This simply connects to the first TLF3000 which is found
			printf("Connecting\n");
			sapphire_connect(results[0].handle);
			break;
		}
		std::this_thread::sleep_for(500ms);
	}

	// Register callback for asynch data channel
	sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);

	// Enter signal analyser mode
	if (sapphire_setMode(2) != SAPPHIRE_NO_ERROR) getError();

	// Select input port
	// 0 - MONITOR IN
	// 1 - Tx / Rx

	if (sapphire_setRxPort(0) != SAPPHIRE_NO_ERROR) getError();

	// Set rx frontend attenuation in units of 0.5dB
	if (sapphire_setRxAtten(0) != SAPPHIRE_NO_ERROR) getError();

	// Start signal analyser
	// Arg0  : start or stop signal analyser
	// Arg1  : advertising address of packets to be analysed
	// Arg2  : bit mask of channels to analyse
	// Arg3  : bit mask of tests to perform
	//         bit 0 = power measurements
	//         bit 1 = modulation characteristics
	//         bit 2 = carrier frequency offset and drift
	//         bit 3 = in-band emissions
	//         bit 4 = CTE
```

```
//         bit 8 = collect waveform data
// Arg4  : bit mask of packets to analyse
//         16 bit mask corresponding to the lower 4 bits of the header
// Arg5  : bit mask of phys to analyse
//         bit 0 = 2Mbps uncoded
//         bit 1 = 1Mbps uncoded
//         bit 2 = 500kbps coded
//         bit 3 = 125kbps coded
// Arg6  : bit mask of packet lengths to analyse
// Arg7  : RSSI threshold in dBm
// Arg8  : indicates whether packets are whitened
// Arg9  : perform off - air measurements
// Arg10 : overSample rate for waveform data
//         0 = 4x
//         1 = 8x
//         3 = 16x
//         7 = 32x
// Arg11 : stop on fail

if (sapphire_signalAnalyser(true, 0x71764129, 0xFFFFFFFFFF, 0xF, 0xFFFF, 0xF, 0xFFFFFFFF, -100, false, true, 0,
                                  false) != SAPPHIRE_NO_ERROR) getError();

// Wait for some data to be collected

Sleep(1000);

// Stop the signal analyser

if (sapphire_signalAnalyser(false, 0x71764129, 0xFFFFFFFFFF, 0xF, 0xFFFFFFFF, 0xF, 0xFFFFFFFF, -100, false, true,
                                  0, false) != SAPPHIRE_NO_ERROR) getError();

// Output results on low, mid and high channels for 1M modulation

float x[5 * 6 + 1];

for (int chan = 0; chan < 3; chan++) {

      uint64_t chanFilter;
      if (chan == 0) {
            printf("Results for channel 0\n");
            chanFilter = 0x0000000001;
      }
      else if (chan == 1) {
            printf("Results for channel 19\n");
            chanFilter = 0x0000080000;
```

```
            }
            else {
                  printf("Results for channel 39\n");
                  chanFilter = 0x8000000000;
            }

            // Poll results table
            // Arg1: measurement, addition of 0x80000000 recovers off-air measurements
            // (not relevant for power or inband emissions)
            // Arg2: channel mask filter
            // Arg3: phy mask filter
            // Arg4: power mask filter

            // Output power results

            sapphire_pollResultsTable(0, chanFilter, 0x2, 0xFFFFFFFF, x);
            printTable(PowerHeadings, x, 2);

            //Output modulation results

            sapphire_pollResultsTable(1 + 0x80000000, chanFilter, 0x2, 0xFFFFFFFF, x);
            printTable(ModulationHeadings, x, 6);

            // Output drift results

            sapphire_pollResultsTable(2 + 0x80000000, chanFilter, 0x2, 0xFFFFFFFF, x);
            printTable(DriftHeadings, x, 5);

            // Output inband emissions results in off - air mode
            // Results may not be useful if connection to DUT is over-the-air

            sapphire_pollResultsTable(3, chanFilter, 0x2, 0xFFFFFFFF, x);
            printTable(InBandHeadings, x, 4);

      }

      getError();

      sapphire_disconnect();

      printf("All done\n");
      getchar();

      return 0;
}
```

6.2.4 Testing a device which is advertising

This example code demonstrates how a device which is advertising can be tested. The advertising device must send either ADV_IND or ADV_SCAN_IND packets. The *TLF3000* will issue SCAN_REQ packets to the advertising device and listen for the SCAN_RSP packets. The *TLF3000* will measure:

1. Output power
2. Modulation characteristics
3. Carrier frequency offset and drift
4. In-band emissions
5. Sensitivity

The sequence of operations is as follows:

1. A connection is made to the wanted *TLF3000* unit using the connection code described inConnecting to the TLF3000
2. A callback for asynchronous messages is registered using the code described in Handling asynchronouos data
3. The operating mode is set phy test mode using sapphire_setMode
4. The receiver port to be used is set using sapphire_setRxPort
5. The receiver frontend attenuation is set using sapphire_setRxAtten
6. The *TLF3000* looks for advertising devices using sapphire_sniffAdvA . Any advertising device which is found causes the callback SniffAdvA to be executed. The address and RSSI of the device are returned. In this example the first device with an RSSI above a threshold is selected.
7. Once an advertising device has been selected, sapphire_sniffAdvA is called again to exit the snffing mode.
8. The *TLF3000* is instructed to scan using sapphire_startScan.
9. When the requested data has been collected the callback ScanEnd is executed. If a timeout happens before the data has been collected then sapphire_stopAdvScan is called to terminate the scan.
10. sapphire_pollResultsTable is used to recover the output power results table and then further calls are used to recover the modulation characteristics results table, carrier frequency and drift results table and the in-band emissions results table. All these tables are printed out.
11. A second scan is made by caling sapphire_startScan. However, this time the output power of the *TLF3000* is gradually reduced. At each power level the number of SCAN_RESP packets is recorded. In this way the sensitivity can be determined.
12. When the requested data has been collected, the scanning is halted using sapphire_stopAdvScan .
13. Calls to sapphire_pollResultsPlot and sapphire_getResultsPlotFloat are used to recover the packet error rate at each power level and these results are printed out.

```
HANDLE sem;
int32_t bestRSSI;
uint64_t advA;
uint64_t MacAddr;
bool advArandom;
const int32_t rssiThreshold = -40;

const char *PowerHeadings[] = {
                    "Pavg",
                    "Pk-Pavg"
};

const char *ModulationHeadings[] = {
                    "DF1max",
                    "DF1avg",
                    "DF2max",
                    "DF2avg",
                    "DF2avg/DF1avg",
                    "DF2max 99.9%"
};


const char *DriftHeadings[] = {
                    "Fo",
                    "Fn",
                    "|F1-Fo|",
                    "|Fo-Fn|",
                    "|Fn-Fn-5|"
};

const char *InBandHeadings[] = {
                    "Ftx+/-2MHz",
                    "Ftx+/-(3+n)MHz",
                    "# exceptions",
                    "Max exception"
};


// routine for printing results tables

void printTable(const char**headings, float *x, int num) {
      printf("\n");
      printf("%20s %10s %10s %10s %10s %10s\n", "Quantity", "# Pkts", "Min", "Max", "Avg", "Current");
      int n = 0;
```

```
	for (int k = 0; k < num; k++) {
		printf("%20s ", headings[k]);

		printf("%10d %10.1f %10.1f %10.1f %10.1f\n", (int)x[n], x[n + 1], x[n + 2], x[n + 3], x[n + 4]);
		n = n + 5;
	}
	printf("\n");
}

void ScanEnd(std::vector<unsigned char> d) {
	deserialise(d, Pad<4>(), MacAddr);
	ReleaseSemaphore(sem, 1, NULL);
}

void SniffAdvA(std::vector<unsigned char> d) {
	uint32_t	dummy;
	uint64_t	addr;
	uint32_t	chan;
	int32_t		rssi;
	deserialise(d, dummy, addr, chan, rssi);
	bool found = bestRSSI > rssiThreshold;
	if (rssi > bestRSSI) {
		advA = addr & 0xFFFFFFFFFFFF;
		advArandom = (addr & 0x1000000000000) ? true : false;
		bestRSSI = rssi;
		printf("Found device with %s address of %012llX with RSSI %d\n", advArandom ? "random" : "public",
			advA, rssi);
	}
	if( (bestRSSI > rssiThreshold) && (!found) ) ReleaseSemaphore(sem, 1, NULL);
}

void EnvData(std::vector<unsigned char> d) {}

void PwrData(std::vector<unsigned char> d) {}

void dataCallback(uint8_t *data, uint32_t len) {
	std::vector<unsigned char> d;
	for (uint32_t k = 0; k < len; k++) d.push_back(data[k]);
	uint16_t len_lsb;
	uint8_t len_msb;
	uint8_t type;
	deserialise(d, len_lsb, len_msb, type);
	switch (type) {
	case 7: ScanEnd(d); break;
	case 66: SniffAdvA(d); break;
```

```
        case 193: EnvData(d); break;
        case 194: PwrData(d); break;
        default: printf("Callback %d\n", type);  break;
        }
}

int main(int argc, char **argv) {

        sapphire_search_result_t* results;

        while (true) {
                int N;
                sapphire_search(&results, &N);
                for (int i = 0; i < N; i++) {
                        printf("Found: %d\n", results[i].serial_number);
                }
                if (N) {
                        // This simply connects to the first TLF3000 which is found
                        printf("Connecting\n");
                        sapphire_connect(results[0].handle);
                        break;
                }
                std::this_thread::sleep_for(500ms);
        }

        // Register callback for asynch data channel
        sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);

        // Enter Adv/Scan mode
        if (sapphire_setMode(3) != SAPPHIRE_NO_ERROR) getError();

        sem = CreateSemaphoreA(NULL, 0, 1, "Done");

        // Select input port
        // 0 - MONITOR IN
        // 1 - Tx / Rx

        if (sapphire_setRxPort(1) != SAPPHIRE_NO_ERROR) getError();

        // Set rx frontend attenuation in units of 0.5dB
        if (sapphire_setRxAtten(0) != SAPPHIRE_NO_ERROR) getError();

        // Sniff for advertising devices
        bestRSSI = -120;
        if (sapphire_sniffAdvA(true) != SAPPHIRE_NO_ERROR) getError();
```

```
DWORD err = WaitForSingleObject(sem, 1000);
bool haveAdvA = true;
if (err == WAIT_TIMEOUT) {
	printf("Failed to find advertising device\n");
	haveAdvA = false;
}
else if (err != WAIT_OBJECT_0) {
	printf("Error waiting for sniff\n");
}
getError();

if (haveAdvA) {

	if (sapphire_sniffAdvA(false) != SAPPHIRE_NO_ERROR) getError();

	// Start scanning
	// Arg0 : Initial power in power sweep
	// Arg1 : Final power in power sweep
	// Arg2 : Step size for power sweep
	// Arg3 : Mask of channels to advertise on
	//        1 = channel 0
	//        2 = channel 12
	//        4 = channel 39
	//        To help ensure DUT sees advertising packet recommend value of 7, ie all channels
	// Arg4: 0 = cycle round the channels in the channel mask sequentially
	//        1 = transmit on all the channels in the channel mask concurrently
	// Arg5: TxAdd bit for header
	//        False = AdvA is public
	//        True = AdvA is random
	// Arg6: ScanA as an ascii string representing octets in hex
	// Arg7 : Number of advertising packets to respond to
	//        0 = respond forever
	// Arg8: 0 = run to completion
	//        1 = stop on test failure
	// Arg9: Timeout in ms to prevent hanging if no advertising packets seen
	// Arg10 : Mask of data to collect
	//         1 = Power
	//         2 = Modulation charateristics
	//         4 = Drift and carrier offset
	//         8 = In - band emissions
	//         256 = Waveform
	// Arg11: RSSI threshold for packets to analyse in dBm
	//        Only packets greater than this threshold will be analysed
	// Arg12: Mask of PDU types to analyse
```

```
//        Packets with PDUs not in the mask will not be analysed
//        1 = ADV_IND
//        64 = ADV_SCAN_IND
// Arg13: False = ignore address in received packets
//        True = check address in received packets
// Arg14: False = address to check against is public
//        True = address to check against is random
//        Ignored if Arg13 is False
// Arg15: Address to check against as an ascii string representing octets in hex
//        Ignored if Arg13 is False

const unsigned char ScanA[] = "AABBCCDDEEFF";
char CmpAddr[13];
sprintf_s(CmpAddr, 13, "%012llX", advA );

// Do each channel sequentially so that we get statistics on each channel
// This also checks that the DUT can respond on each channel

if (sapphire_startScan(-10.0f, -10.0f, -0.5f, 7, 0, false, ScanA, 10, 0, 500, 0xF, rssiThreshold, 1 + 64,
      true, advArandom, (const unsigned char *) CmpAddr) != SAPPHIRE_NO_ERROR) getError();

err = WaitForSingleObject(sem, 30 * 1000);
if (err != WAIT_OBJECT_0) printf("Error waiting for script to terminate\n");
getError();

if (sapphire_stopAdvScan() != SAPPHIRE_NO_ERROR) getError();

// Recover the address of the device which was tested
// unnecessary if CmpAddr was set in sapphire_startScan call

advA = MacAddr & 0xFFFFFFFFFFFF;
advArandom = (MacAddr & 0x1000000000000) ? true : false;

printf("Test results for device %012llX\n", advA);

// Output power results

float x[5 * 6];

for (int chan = 0; chan < 3; chan++) {

      uint64_t chanFilter;
      if (chan == 0) {
            printf("Results for channel 0\n");
            chanFilter = 0x0000000001;
```

```
        }
        else if (chan == 1) {
              printf("Results for channel 12\n");
              chanFilter = 0x0000001000;
        }
        else {
              printf("Results for channel 39\n");
              chanFilter = 0x8000000000;
        }

        // Poll results table
        // Arg1: measurement, addition of 0x80000000 recovers off-air measurements
        // (not relevant for power or inband emissions)
        // Arg2: channel mask filter
        // Arg3: phy mask filter
        // Arg4: power mask filter
        sapphire_pollResultsTable(0, chanFilter, 0x2, 0xFFFFFFFF, x);
        printTable(PowerHeadings, x, 2);

        //Output modulation results

        sapphire_pollResultsTable(1 + 0x80000000, chanFilter, 0x2, 0xFFFFFFFF, x);
        printTable(ModulationHeadings, x, 6);

        // Output drift results

        sapphire_pollResultsTable(2 + 0x80000000, chanFilter, 0x2, 0xFFFFFFFF, x);
        printTable(DriftHeadings, x, 5);

        // Output inband emissions results in off - air mode
        // Results may not be useful if connection to DUT is over-the-air

        sapphire_pollResultsTable(3, chanFilter, 0x2, 0xFFFFFFFF, x);
        printTable(InBandHeadings, x, 4);

}

// Do a sensitivity search

// There can be no more than 32 power steps

float startPwr = -85.0f;
float stepPwr = -1.0f;
float stopPwr = startPwr +15.0f * stepPwr;
```

```
            // To reduce test time, listen on all 3 advertising channels simultaneously
            // and just measure PER from the first 50 scan requests issued

            if (sapphire_startScan(startPwr, stopPwr, stepPwr, 7, 1, false, ScanA, 50, 0, 5000, 0x0, rssiThreshold,
                        1 + 64, true, advArandom, (const unsigned char *)CmpAddr) != SAPPHIRE_NO_ERROR) getError();

            err = WaitForSingleObject(sem, 30 * 1000);
            if (err != WAIT_OBJECT_0) printf("Error waiting for script to terminate\n");
            getError();

            if (sapphire_stopAdvScan() != SAPPHIRE_NO_ERROR) getError();

            // Read back number of packets vs power

            float        c1[32];
            float        c12[32];
            float        c39[32];
            int32_t      n;

            sapphire_pollResultsPlot(0x80000011, 2, 0x0000000001, 0x2, 0xFFFFFFFF, &n);
            sapphire_getResultsPlotFloat(0, n / 12, c1);
            sapphire_pollResultsPlot(0x80000011, 2, 0x0000001000, 0x2, 0xFFFFFFFF, &n);
            sapphire_getResultsPlotFloat(0, n / 12, c12);
            sapphire_pollResultsPlot(0x80000011, 2, 0x8000000000, 0x2, 0xFFFFFFFF, &n);
            sapphire_getResultsPlotFloat(0, n / 12, c39);
            printf("                          Packet error rate\n");
            printf("Power (dBm)   Channel 0    Channel 12   Channel 39\n");
            for (int32_t k = 0; k < n / 12; k++) {
                  printf("%6.1f         %6.1f       %6.1f       %6.1f\n", startPwr, c1[k], c12[k], c39[k]);
                  startPwr = startPwr + stepPwr;
            }

      }

      getError();

      sapphire_disconnect();

      printf("All done\n");
      getchar();

      return 0;
}
```

6.2.5 Simplied phy test report generation

A test script can be run, and a report generated using a single call to the C dll. The generated report is either a .txt, .csv or .html file, depending on the file extension specified for the output file. The contents of the file ReportHdr (if specified) will be inserted at the top of the report file. This permits customisation of the report file on a production line.

```c
#include "sapphire.h"

#pragma comment(lib, "libSapphire.lib")

char TestScript[] = "D:\\LE_1M_production_fast_short.sta";
char ReportHdr[] = "";
char TestReport[] = "D:\\report.html";

#define nLoss(3)
float  frqs[nLoss] = { 2402.0f , 2441.0f , 2480.0f };
float  loss[nLoss] = { 1.0f , 1.1f , 1.2f };

int main()
{

	char *s = sapphire_testWrapper(
		0,				// Serial number of TLF3000 to use.
		false,				// IO voltage to be supplied by TLF3000
		5,				// com port on host to use, set to -1 if DUT is connected directly to TLF3000
		0,				// Transport : 0 = direct test mode, 1 = H4, 2 = H5, 3 = BCSP
		19200,				// Baud rate
		false,				// HW flow control
		false,				// SW flow control
		1,				// Stop bits
		0,				// Parity : 0 = none, 1 = even, 2 = odd
		TestScript,			// Name of file containing the test script.
		TestReport,			// Name of file to receive test results.
		ReportHdr,			// Name of file containing header to be inserted at the top of the test results.
		1,				// Receiver port to use. 1 = Rx/Tx, 0 = Monitor In
		0.0f,				// Front-end attenuation to be used in TLF3000 in dB
		nLoss,				// Number of points describing cable loss
		frqs,				// Frequencies in MHz at which cable loss is defined
		loss,				// Cable loss in dB at each of the defined frequencies
		true,				// Run to completion (false = abort on first fail)
		20000				// Timeout for test script execution in ms
	);

	sapphire_disconnect();

	return 0;
}
```

6.2.6 Generating a test report for a device which is advertising

This script is similar to that in section TBD. However, in this instance a test report is generated by calling sapphire_results_advscan. The generated report is either a .txt, .csv or .html file, depending on the file extension specified for the output file. The contents of the file ReportHdr (if specified) will be inserted at the top of the report file. This permits customisation of the report file on a production line.

```
char TestReport[] = "D:\\TestReport.html";
char *ReportHdr = nullptr;

HANDLE sem;
int32_t bestRSSI;
uint64_t advA;
uint64_t MacAddr;
bool advArandom;
const int32_t rssiThreshold = -40;

void ScanEnd(std::vector<unsigned char> d) {
	uint64_t MacAddr;
	deserialise(d, Pad<4>(), MacAddr);
	ReleaseSemaphore(sem, 1, NULL);
}

void SniffAdvA(std::vector<unsigned char> d) {
	uint32_t	dummy;
	uint64_t	addr;
	uint32_t	chan;
	int32_t		rssi;
	deserialise(d, dummy, addr, chan, rssi);
	bool found = bestRSSI > rssiThreshold;
	if (rssi > bestRSSI) {
		advA = addr & 0xFFFFFFFFFFFF;
		advArandom = (addr & 0x1000000000000) ? true : false;
		bestRSSI = rssi;
		printf("Found device with %s address of %012llX with RSSI %d\n", advArandom ? "random" : "public", advA,
rssi);
	}
	if ((bestRSSI > rssiThreshold) && (!found)) ReleaseSemaphore(sem, 1, NULL);
}

void EnvData(std::vector<unsigned char> d) {}

void PwrData(std::vector<unsigned char> d) {}

void dataCallback(uint8_t *data, uint32_t len) {
	std::vector<unsigned char> d;
	for (uint32_t k = 0; k < len; k++) d.push_back(data[k]);
	uint16_t len_lsb;
	uint8_t len_msb;
	uint8_t type;
	deserialise(d, len_lsb, len_msb, type);
	switch (type) {
```

```
        case 7: ScanEnd(d); break;
        case 66: SniffAdvA(d); break;
        case 193: EnvData(d); break;
        case 194: PwrData(d); break;
        default: printf("Callback %d\n", type);  break;
        }
}

void getError() {
        char err[1024];
        sapphire_getError(err, 1024);
        while (err[0]) {
                printf("%s\n", err);
                sapphire_getError(err, 1024);
        }
}

int main()
{

        sapphire_search_result_t* results;

        while (true) {
                int N;
                sapphire_search(&results, &N);
                for (int i = 0; i < N; i++) {
                        printf("Found: %d\n", results[i].serial_number);
                }
                if (N) {
                        // This simply connects to the first TLF3000 which is found
                        printf("Connecting\n");
                        sapphire_connect(results[0].handle);
                        break;
                }
                std::this_thread::sleep_for(500ms);
        }

        // Register callback for asynch data channel
        sapphire_set_data_callback((sapphire_data_callback_t)&dataCallback);

        // Enter Adv/Scan mode
        if (sapphire_setMode(3) != SAPPHIRE_NO_ERROR) getError();

        sem = CreateSemaphoreA(NULL, 0, 1, "Done");
```

```
// Select input port
// 0 - MONITOR IN
// 1 - Tx / Rx

if (sapphire_setRxPort(1) != SAPPHIRE_NO_ERROR) getError();

// Set rx frontend attenuation in units of 0.5dB
if (sapphire_setRxAtten(0) != SAPPHIRE_NO_ERROR) getError();

// Sniff for advertising devices
bestRSSI = -120;
if (sapphire_sniffAdvA(true) != SAPPHIRE_NO_ERROR) getError();

DWORD err = WaitForSingleObject(sem, 1000);
bool haveAdvA = true;
if (err == WAIT_TIMEOUT) {
      printf("Failed to find advertising device\n");
      haveAdvA = false;
}
else if (err != WAIT_OBJECT_0) {
      printf("Error waiting for sniff\n");
}
getError();

if (haveAdvA) {

      if (sapphire_sniffAdvA(false) != SAPPHIRE_NO_ERROR) getError();

      // Start scanning
      // Arg0 : Initial power in power sweep
      // Arg1 : Final power in power sweep
      // Arg2 : Step size for power sweep
      // Arg3 : Mask of channels to advertise on
      //       1 = channel 0
      //       2 = channel 12
      //       4 = channel 39
      //       To help ensure DUT sees advertising packet recommend value of 7, ie all channels
      // Arg4: 0 = cycle round the channels in the channel mask sequentially
      //       1 = transmit on all the channels in the channel mask concurrently
      // Arg5: TxAdd bit for header
      //       False = AdvA is public
      //       True = AdvA is random
      // Arg6: ScanA as an ascii string representing octets in hex
      // Arg7 : Number of advertising packets to respond to
      //       0 = respond forever
```

```
// Arg8: 0 = run to completion
//       1 = stop on test failure
// Arg9: Timeout in ms to prevent hanging if no advertising packets seen
// Arg10 : Mask of data to collect
//        1 = Power
//        2 = Modulation charateristics
//        4 = Drift and carrier offset
//        8 = In - band emissions
//        256 = Waveform
// Arg11: RSSI threshold for packets to analyse in dBm
//        Only packets greater than this threshold will be analysed
// Arg12: Mask of PDU types to analyse
//        Packets with PDUs not in the mask will not be analysed
//        1 = ADV_IND
//        64 = ADV_SCAN_IND
// Arg13: False = ignore address in received packets
//        True = check address in received packets
// Arg14: False = address to check against is public
//        True = address to check against is random
//        Ignored if Arg13 is False
// Arg15: Address to check against as an ascii string representing octets in hex
//        Ignored if Arg13 is False

const unsigned char ScanA[] = "AABBCCDDEEFF";
char CmpAddr[13];
sprintf_s(CmpAddr, 13, "%012llX", advA);

// Do each channel sequentially so that we get statistics on each channel
// This also checks that the DUT can respond on each channel

if (sapphire_startScan(-70.0f, -80.0f, -1.0f, 7, 0, false, ScanA, 10, 0, 500, 0xF, rssiThreshold, 1 + 64,
             true, advArandom, (const unsigned char *)CmpAddr) != SAPPHIRE_NO_ERROR) getError();

err = WaitForSingleObject(sem, 30 * 1000);
if (err != WAIT_OBJECT_0) printf("Error waiting for script to terminate\n");
getError();

if (sapphire_stopAdvScan() != SAPPHIRE_NO_ERROR) getError();

// Generate a report
char *msg = sapphire_results_advscan(TestReport, ReportHdr);
printf("%s\n", msg);

// Get MAC address of device tested
char mac[128];
```

```
            if (sapphire_getMAC(mac, 128) != SAPPHIRE_NO_ERROR)
                  getError();
            else
                  printf("%s\n", mac);
      }

      getError();

      sapphire_disconnect();

      printf("All done\n");
      getchar();

   return 0;
}
```

6.2.7 Simplified procedure for testing advertising device

An advertising device can be tested and a report generated using an single call to the C dll. The generated report is either a .txt, .csv or .html file, depending on the file extension specified for the output file. The contents of the file ReportHdr (if specified) will be inserted at the top of the report file. This permits customisation of the report file on a production line.

```
#include "sapphire.h"

#pragma comment(lib, "libSapphire.lib")

char ReportHdr[] = "";
char TestReport[] = "D:\\junk.html";

#define OVER_THE_AIR

#define nLoss(3)

#ifdef OVER_THE_AIR
float  frqs[nLoss] = { 2402.0f , 2441.0f , 2480.0f };
float  loss[nLoss] = { 0.0f , 0.0f , 0.0f };
float   propLoss = 40.0f;
bool   rxPort = 0;
#else
float  frqs[nLoss] = { 2402.0f , 2441.0f , 2480.0f };
float  loss[nLoss] = { 1.0f , 1.1f , 1.2f };
float   propLoss = 0.0f;
bool   rxPort = 1;
#endif

int main()
{

      char *s = sapphire_scanWrapper(
            0,                          // Serial number of TLF3000 to use.
            TestReport,                 // Name of file to receive test results.
            ReportHdr,                  // Name of file containing header to be inserted at the top of the test results.
            rxPort,                     // Receiver port to use. 1 = Rx/Tx, 0 = Monitor In
            0.0f,                       // Front-end attenuation to be used in TLF3000 in dB
            nLoss,                      // Number of points describing cable loss
            frqs,                       // Frequencies in MHz at which cable loss is defined
            loss,                       // Cable loss in dB at each of the defined frequencies
            10000,                      // Timeout for running tests in ms
            "AABBCCDDEEFF",             // ScanA address to use
            false,                      // ScanA address is public
            true,                       // Sniff to find advertiser to test
            1000,                       // Length of time to sniff in ms, if this is 0 then the first advertiser found
                                        // is used, if it is non zero then the advertiser with the biggest RSSI is used
                                        // ignored if SniffAdvertiser is false
            1000,                       // Timeout for the sniff operation in ms - ignored if SniffAdvertiser is false
            "0123456789AB",             // AdvA address of advertiser to test - ignored if SniffAdvertiser is true
            false,                      // AdvA is public - ignored if SniffAdvertiser is true
```

```
            -70.0f + propLoss,        // Power at DUT for testing receiver sensitivity
            20,                       // Number of packets to send for testing receiver sensitivity
            -20 - propLoss,           // RSSI threshold which packets must exceed if they are to be analysed
            true,                     // Send scan requests to ADV_IND packets received from advertiser
            true,                     // Send scan requests to ADV_SCAN_IND packets received from advertiser
            -5.0f - propLoss,         // Lower test limit for DUT transmit power
            10.0f - propLoss          // Upper test limit for DUT receive power
      );

      // Get MAC address of device tested
      char mac[128];
      if (sapphire_getMAC(mac, 128) == SAPPHIRE_NO_ERROR) printf("%s\n", mac);

      printf("%s\n", s);

      getchar();

      sapphire_disconnect();

      return 0;

}
```

6.3 Library reference

6.3.1 Overview

This section lists the C dll library commands which are available for the Sapphire application.

6.3.2 sapphire_setMode

sapphire_setMode() sets the operating mode of the Sapphire application:

sapphire_error_t **sapphire_setMode**(int mode)

mode defines the operating mode:

Value	Operating mode
0	Phy level tester
1	Signal generator
2	Signal analyzer
3	Advertise/scan
> 3	Illegal

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.3 sapphire_startScript

sapphire_startScript() instructs the Sapphire application to prepare for the download of a new test script. A test script can only be started when the phy level tester is not running. This command can be issued irrespective of which mode the Sapphire application is operating in. Any existing but incomplete test script download will be aborted.

sapphire_error_t **sapphire_startScript**(bool overwrite, bool flash, const char *fileName);

overwrite is a Boolean. If True, then if a test script file of the same name already exists it will be overwritten. If False, then if a test script file of the same name already exits the command will fail.

flash is a Boolean. If True, then the test script file will be placed in non-volatile FLASH memory. If False, then the test script file will be placed in volatile RAM. Currenty only RAM is supported.

testScriptName is a string containing the name of the test script file. Valid test script filenames must start with an alphabetic character. The remainder of the file name must be composed from the characters a-z, A-Z, 0-9 and _.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.4 sapphire_contScript

sapphire_contScript() appends a new line to a test script previously opened using startScript but not yet closed with an *sapphire_endScript*. A test script can only be added to when the phy level tester is not running. This command can be issued irrespective of which mode the Sapphire application is operating in.

Each line in the test script is a sequence of ASCII characters defining a test to be performed and its associated parameters. The test script is case insensitive. Full details of the test script format can be found in section TBD.

sapphire_error_t **sapphire_contScript**(const char *line)

line is a character string containing the next line to be appended to the test script file. It is not necessary to include an end of line character.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.5 sapphire_endScript

sapphire_endScript() signifies that the entire content of a test script has been download. This can only be called after sapphire_startScript() has successfully be executed. This command cannot be executed if the phy level tester is running. This command can be issued irrespective of which mode the Sapphire application is operating in.

sapphire_error_t **sapphire_endScript**()

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.6 sapphire_runScript

sapphire_runScript() instructs the phy level tester to start executing a test script. This command can only be executed when the Sapphire application is in phy level test mode. It cannot be executed whilst a test script is being downloaded, i.e. after a *sapphire_startScript()* command and prior to an *sapphire_endScript()* command.

sapphire_error_t **sapphire_runScript**(uint32_t repeats, bool runToEnd, bool limitsOnly, bool reorder,bool flash, const char *script)

repeat specifies the number of times the testScript will be executed.

runToEnd is a Boolean. If set, then all the tests specified in the test script will be performed. If not set, then the test script will terminate as soon as a test failure is detected.

onlyLimits is a Boolean. If set, the receiver tests will terminate as soon as sufficient packets have been sent to determine whether the test will pass or fail the PER limit. This will reduce the accuracy of the returned PER value but will dramatically reduce test time.

reorder is a Boolean. If this flag is set, then the test script will be reordered so that the transmitter tests are performed first followed by the receiver tests. If this flag is not set, then the tests will be performed in the order specified in the test script.

flash is a Boolean. If True, then the test script file to be run resides in non-volatile FLASH memory. If False, then the test script file to be run resides in volatile RAM. Currenty only RAM is supported.

script is a string containing the name of the test script file to be run. Valid test script filenames must start with an alphabetic character. The remainder of the file name must be composed from the characters a-z, A-Z, 0-9 and _.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.7 sapphire_testWrapper

sapphire_testWrapper() provides a convenient mechanism for running a test script contained in a .sta file and generating a test report. The test report can be in .txt, .csv or .html format.

```
char *sapphire_testWrapper ( uint32_t TFL3000SerialNumber, bool VIOext, int32_t port,
                             uint32_t transport, uint32_t baud, bool hwFlow, bool swFlow,
                             uint32_t stopBits, uint32_t parity , char *TestScript, char *TestReport,
                             char *ReportHdr, bool rxPort, float rxAtten, uint32_t nLoss, float *frqs,
                             float *loss, bool RunToCompletion, uint32_t ScriptTimeOut )
```

TLF3000SerialNumber is the serial number of the *TLF3000* to use for the test. If a zero serial number is specified, then the first *TLF3000* unit discovered will be used.

VIOext If True, then the IO voltage will be supplied externally. If False, then the *TLF3000* will supply a 3.3V IO voltage. This parameter is ignored if the DUT is not connected directly to the *TLF3000*.

port The number of the comport on the host computer to which the DUT is attached. If the DUT is connected directly to the *TLF3000* then **port** should be a negative number.

tansport method to be used to communicate with the DUT. Possible values are:

0	Direct test mode
1	H4 UART
2	H5 UART
3	BCSP UART

baudrate contains the baud rate used to communicate with the DUT. Supported baud rates for a DUT connected directly to the *TLF3000* unit are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. Automatic baud detection will occur if this field is set to zero. Automatic baud rate detection is only available for H5 and BCSP transports.

hwFlow the type of hardware flow control which will be used in communicating with the DUT. Possible values are:

0	No hardware flow control
1	RTS/CTS

swFlow indicates the type of software flow control which will be used in communicating with the DUT. Possible values are:

0	No software flow control
1	XON/XOFF

stopBsits is the number of stop bits to be used when communicating over UART. Possible values are:

1	1 stop bit
2	2 stop bits

parity determines the parity bits to be used when communicating over UART. Possible values are:

0	None
1	Odd
2	Even

TestScript is the name of the .sta file containing the test script to be run. The .sta file can be built and exported from the GUI.

TestReport is a character string containing the name of the file to receive the test report. The filename should include the suffix .txt, .csv or .html. The format of the output file is governed by the filename suffix.

ReportHeader is character string referencing a text file. The contents of this file will placed at the start of the test report file. This enables the report to be customised.

rxPort is the receiver port to use for reception and can be one of the following values:

0	Monitor In
1	Tx/Rx port

rxAtten contains the receiver frontend attenuation in units of dB. The permissible attenuator range is 0 to 31.5 dB.

nLoss is the number of frequency points at which the cable loss is specified.

frqs is an array of length **n** of frequencies in MHz at which the cable loss is specified.

loss is an array of length n of cable losses in dB. The cable loss must be between 0 and 25dB.

RunToCompletion If False, then testing will terminate if a test fails. If True, all tests will be performed.

ScriptTimeOut If the run time of the tests exceeds **ScriptTimeOut** ms, then the script will be aborted.

Returns a pointer to a character string. The string will normally either contain the text “Passed” or “Failed”. If an error occurred, then the text will contain an error message.

6.3.8 sapphire_setCableLoss

sapphire_setCableLoss() informs the Sapphire application of the cable loss between the Tx/Rx port of the *TLF3000* unit and the DUT. The loss is specified at 2.4 GHz. The Sapphire application requires knowledge of this loss to compensate its transmit power and to calibrate its received signal strength measurements. The same loss is also used to compensate the out-of-band CW blocker. Since the cable will vary between 24 MHz and 6 GHz, this compensation can only be approximate.

sapphire_error_t **sapphire_setCableLoss**(float dB)

dB is the cable loss in dB. The cable loss must be between 0 and 25dB.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.9 sapphire_setCableLosses

sapphire_setCableLosses() informs the Sapphire application of the cable loss between the Tx/Rx port of the *TLF3000* unit and the DUT as a function of frequency. The Sapphire application requires knowledge of this loss to compensate its transmit power and to calibrate its received signal strength measurements.

sapphire_error_t **sapphire_setCableLosses**(uint32_t n, float *frq, float *dB)

n is the number of frequency points at which the cable loss is specified.

frq is an array of length **n** of frequencies in MHz at which the cable loss is specified.

dB is an array of length n of cable losses in dB. The cable loss must be between 0 and 25dB.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.10 sapphire_setDUTProperties

sapphire_setDUTProperties() informs the Sapphire application of the DUT's supported features. This information is only required for the phy test mode. This command can be executed at any time.

```
sapphire_error_t sapphire_setDUTProperties(uint32_t role, bool advExt,
                    bool dataExt, uint32_t maxAdvOctet, uint32_t maxRxOctet,
                    uint32_t maxRxTime, uint32_t maxTxOctet, uint32_t maxTxTime,
                    bool _2Mbps, bool coded, bool stable, bool RxAoA2,
                    bool TxAoA2, bool RxAoD2, bool TxAoD2, bool RxAoA1,
                    bool TxAoA1, bool RxAoD1, bool TxAoD1, uint16_t maxCTElen,
                    uint16_t numAnt, uint16_t maxPatt)
```

role indicates which role the DUT supports. This determines whether both transmit and receive tests can be done, and if so, on which channels. Possible values are:

Value	Role
0	Central
1	Peripheral
2	Broadcaster
3	Observer

advExt is a Boolean indicating whether extended advertising features are supported.

dataExt is a Boolean indiciating whether extended data lengths are supported.

maxAdvOctet is the maximum number of advertising octets.

maxRxOctet is the maximum number of octets in a packet payload which the DUT can receive.

maxRxTime is the maximum packet length that can be received in units of µs.

maxTxOctet is the maximum number of octets in a packet payload which the DUT can transmit.

maxTxTime is the maximum packet length that can be transmitted in units of µs.

_2Mbps is a Boolean indicating whether 2Mbps is supported.

coded is a Boolean indicating whether coded phys are supported.

stable is a Boolean indicating whether stable modulation index is supported.

RxAoA2 is a Boolean indicating whether the receiver can receive an AoA CTE with 2µs slots.

TxAoA2 is a Boolean indicating whether the transmitter can transmit an AoA CTE with 2µs slots.

RxAoD2 is a Boolean indicating whether the receiver can receive an AoD CTE with 2µs slots.

TxAoD2 is a Boolean indicating whether the transmitter can transmit an AoD CTE with 2µs slots.

RxAoA1 is a Boolean indicating whether the receiver can receive an AoA CTE with 1µs slots.

TxAoA1 is a Boolean indicating whether the transmitter can transmit an AoA CTE with 1µs slots.

RxAoD1 is a Boolean indicating whether the receiver can receive an AoD CTE with 1µs slots.

TxAoD1 is a Boolean indicating whether the transmitter can transmit an AoD CTE with 1µs slots.

maxCTElen is the maximum supported CTE length in 8µs slots

numAnt is the number of antenna attached to the DUT.

maxPatt is the maximum length switch pattern the DUT supports.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.11 sapphire_setDUTPort

sapphire_setDUTPort informs the Sapphire application which comport the DUT is attached to. This information is required by the Sapphire application when operating in phy level test mode. This command can be executed at anytime.

sapphire_error_t **sapphire_setDUTPort**(int32_t port)

port is the number of the comport to which the DUT is connected. If this number is positive, then it is interpreted as a comport on the host machine. If this number if negative, then it implies that the DUT is connected to the DIO connector on the rear of the *TLF3000* unit.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.12 sapphire_setDUTComms

sapphire_setDUTComms informs the Sapphire application how it should communicate with DUT. This information is required by the Sapphire application when operating in phy level test mode. This command can be executed at anytime.

sapphire_error_t **sapphire_setDUTComms**(uint32_t transport, uint32_t baudrate, bool hw_flow, bool sw_flow, uint32_t stop_bits, uint32_t parity, bool crc)

tansport method to be used to communicate with the DUT. Possible values are:

0	Direct test mode
1	H4 UART
2	H5 UART
3	BCSP UART

baudrate contains the baud rate used to communicate with the DUT. Supported baud rates for a DUT connected directly to the *TLF3000* unit are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. Automatic baud detection will occur if this field is set to zero. Automatic baud rate detection is only available for H5 and BCSP transports.

hw_flow the type of hardware flow control which will be used in communicating with the DUT. Possible values are:

0	No hardware flow control
1	RTS/CTS

sw_flow indicates the type of software flow control which will be used in communicating with the DUT. Possible values are:

0	No software flow control
1	XON/XOFF

stop_bits is the number of stop bits to be used when communicating over UART. Possible values are:

1	1 stop bit
2	2 stop bits

parity determines the parity bits to be used when communicating over UART. Possible values are:

0	None
1	Odd
2	Even

crc specifies whether a CRC should be appended to the packet when using H5 or BCSP. Possible values are:

0	No CRC
1	16 bit CCITT CRC

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.13 sapphire_setPhyTestDIO

sapphire_setPhyTestDIO() defines how the digital output lines should be programmed when the Sapphire application is in the phy tester mode. The digital output lines can be used to show when the tester is running and whether tests have passed or failed. This command can be executed at any time. If a digital output line is programmed to respond to both the phy tester running and the pass/fail status, then the pass/fail status takes priority.

sapphire_error_t **sapphire_setPhyTestDIO**(uint8_t run_bits, uint8_t run_mask, uint8_t pass_bits, uint8_t pass_mask)

run_bits is an 8 bit mask determining the polarity of the 8 digital output lines when the phy tester is running. A bit set to '1' indicates that the corresponding digital output line should be set high whilst the phy tester is running. A bit set to '0' indicates that the corresponding digital output line should be set low whilst the phy tester is running.

run_mask is an 8 bit mask indicating which bits in the *RunMask* are relevant and which should be ignored. A bit set to '1' indicates that the corresponding bit in *RunBits* should be acted on. A bit set to '0' indicates that the corresponding bit in *RunBits* should be ignored.

pass_bits is an 8 bit mask determining how the 8 digital output lines should be set to indicate pass or fail. A bit set to '1' indicates that the corresponding digital output line should be set high if all tests have passed. A bit set to '0' indicates that the corresponding digital output line should be set low if all tests have passed.

pass_mask is an 8 bit mask indicating which bits in the *PassMask* are relevant and which should be ignored. A bit set to '1' indicates that the corresponding bit in *PassBits* should be acted on. A bit set to '0' indicates that the corresponding bit in *PassBits* should be ignored.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.14 sapphire_setWanted

sapphire_setWanted() programs the wanted signal when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until startStopSigGen is invoked.

```
sapphire_error_t  sapphire_setWanted(bool On, uint32_t Chan, double Pwr,
                        uint32_t Phy, uint32_t AA, uint32_t Seed, bool Whiten,
                        uint32_t PayloadType, uint32_t PayloadLen,uint32_t PktInterval, uint16_t NumPkts,
                        double Ramp, double Before, double After, uint32_t DIOmask,
                        uint32_t NumDistort, double *distortions )
```

On is a Boolean to indicate whether the wanted signal generator should be started. If this flag set, then the wanted signal generator is started, otherwise the wanted signal generator is halted if it is already running.

Chan is the RF channel number for the wanted signal. Valid channels are in the range 0 to 39.

Pwr is the power of the wanted signal in units of dBm. This should be in range -170 dBm to 0 dBm.

Phy is the phy to be used by the wanted signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

AA is the access address to be used in the transmitted packets.

Seed is the CRC seed to be used in the transmitted packets.

Whiten is a Boolean which specifies whether the transmission should be whitened.

PayloadType defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31
7	`11110000` in transmission order
8	`10101010` in transmission order
9	`11111111`
10	`00000000`
11	`00001111` in transmission order
12	`01010101` in transmission order

PayloadLen is the length of the wanted signal packet payload in octets. This must be in the range 0 to 255.

PktInterval is the interval between the start of one packet and the start of the subsequent packet in units of µs.

NumPkts is the number of packets to be transmitted. If this value is specified as 0, then the wanted signal transmitter will continue to generate packets until it is terminated. The maximum value which can be specified is 635535.

Ramp is the time for the power ramp up and down in units of µs

Before is the length of unmodulated carrier between the power ramp and the first symbol of the packet in units of µs.

After is the length of unmodulated carrier between the last symbol of the packet and the power ramp down in units of µs.

DIOmask is a mask indicating which digital IO lines should be toggled high when the wanted signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when a packet is being transmitted. Only lines 2 to 7 can be specified.

NumDistort is the number of different groups of 5 entries in the distortion table.

Distortions contains groups of 5 int16 which described the distortions to be applied:

1. *int16_t : CarrierOff*. The carrier offset in units of kHz. Valid range is -500 kHz to +500kHz for 2Mbps uncoded and -250 kHz to +250 kHz for all other phy.
2. *uint16_t : ModIndex*. The modulation index in units of 0.0001. Valid range is 0.4 to 0.6.
3. *uint16_t : DriftMag*. The drift magnitude in units of kHz. Valid range is -156 kHz to +156 kHz for 2 Mbps uncoded and -78 kHz to +78 kHz for all other phy.
4. *uint16_t : DriftRate*. The drift rate in units of Hz. Valid range is 0 to 2400 Hz.
5. *int16_t : SymbolTime*. The symbol timing error in units of ppm. Valid range is -100 to +100.

Each group of distortions is applied to 50 transmitted packets. Once all the groups of distortions have been exhausted, the list is rewound, and the first group reused. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *DriftMag* and whose frequency is determined by *DriftRate*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.15 sapphire_setWantedCTE

sapphire_setWantedCTE() programs a wanted signal with a CTE when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until sapphire_startStopSigGen is invoked.

```
sapphire_error_t  sapphire_setWantedCTE(bool On, uint32_t Chan, double Pwr, uint32_t Phy,
                                        uint32_t AA, uint32_t Seed, bool Whiten, uint32_t PayloadType,
                                        uint32_t PayloadLen, uint32_t PktInterval, uint16_t NumPkts,
                                        double Ramp, double Before, double After, uint32_t DIOmask,
                                        uint32_t NumDistort, double *distortions, bool HasCTE, bool CTEaod,
                                        bool CTE2us, uint32_t CTEslots, bool CTEinfo, bool CTEext, double Delay,
                                        uint32_t NumAnt, bool PattB, double *Amps, double *Phis,
                                        uint8_t *AntCodes)
```

On is a Boolean to indicate whether the wanted signal generator should be started. If this flag set, then the wanted signal generator is started, otherwise the wanted signal generator is halted if it is already running.

Chan is the RF channel number for the wanted signal. Valid channels are in the range 0 to 39.

Pwr is the power of the wanted signal in units of dBm. This should be in range -170 dBm to 0 dBm.

Phy is the phy to be used by the wanted signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

AA is the access address to be used in the transmitted packets.

Seed is the CRC seed to be used in the transmitted packets.

Whiten is a Boolean which specifies whether the transmission should be whitened.

PayloadType defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31
7	11110000 in transmission order
8	10101010 in transmission order
9	11111111
10	00000000
11	00001111 in transmission order
12	01010101 in transmission order

PayloadLen is the length of the wanted signal packet payload in octets. This must be in the range 0 to 255.

PktInterval is the interval between the start of one packet and the start of the subsequent packet in units of µs.

NumPkts is the number of packets to be transmitted. If this value is specified as 0, then the wanted signal transmitter will continue to generate packets until it is terminated. The maximum value which can be specified is 635535.

Ramp is the time for the power ramp up and down in units of µs

Before is the length of unmodulated carrier between the power ramp and the first symbol of the packet in units of µs.

After is the length of unmodulated carrier between the last symbol of the packet and the power ramp down in units of µs.

DIOmask is a mask indicating which digital IO lines should be toggled high when the wanted signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when a packet is being transmitted. Only lines 2 to 7 can be specified.

NumDistort is the number of different groups of 5 entries in the distortion table.

Distortions contains groups of 5 int16 which described the distortions to be applied:

1. *int16_t : CarrierOff*. The carrier offset in units of kHz. Valid range is -500 kHz to +500kHz for 2Mbps uncoded and -250 kHz to +250 kHz for all other phy.
2. *uint16_t : ModIndex*. The modulation index in units of 0.0001. Valid range is 0.4 to 0.6.
3. *uint16_t : DriftMag*. The drift magnitude in units of kHz. Valid range is -156 kHz to +156 kHz for 2 Mbps uncoded and -78 kHz to +78 kHz for all other phy.
4. *uint16_t : DriftRate*. The drift rate in units of Hz. Valid range is 0 to 2400 Hz.
5. *int16_t : SymbolTime*. The symbol timing error in units of ppm. Valid range is -100 to +100.

Each group of distortions is applied to 50 transmitted packets. Once all the groups of distortions have been exhausted, the list is rewound, and the first group reused. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *DriftMag* and whose frequency is determined by *DriftRate*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets.

HasCTE is True is the packet has a CTE. If this parameter is false, all successive parameters are ignored.

CTEaod is True if the CTE is AoD and False if the CTE is AoA.

CTE2us is True if the CTE has 2µs switching slots and False if the CTE has 1µs switching slots.

CTEslots is the number of 8µs slots in the CTE.

CTEinfo is True then the CTEinfo within the packet is set to the opposite of **CTEaod**. If it is false, then the CTEinfo field within the packet reflects the setting of **CTEaod**.

CTEext indicates that an external antenna array has been connected to the *TLF3000* for AoD. This parameter is ignored for AoA.

Delay is the delay in µs between the control lines to the external antenna array changing and the antenna array switching. This parameter is ignored is **CTEext** is False.

NumAnt is the number of antenna in the array. This parameter is ignored for AoA.

PattB specifies the antenna switching pattern. If True, then antenna switching pattern B is used. If False, then antenna switching pattern A is used.

Amps is an array of length of length **NumAnt** indicating the relative amplitude to transmit on each simulated antenna in units of dB. This parameter is ignored if **CTEext** is True or **CTEaod** is False.

Phis is an array of length of **NumAnt** indicating the phase to transmit on each simulated atenna in units of degrees. This parameter is ignored if **CTEext** is True or **CTEaod** is False.

AntCodes is an array of length **NumAnt** indicating the code to be placed on the DIO lines to select each antenna. This parameter is ignored if **CTEext** is False or **CTEaod** is False.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.16 sapphire_setInterferer

sapphire_setInterferer() programs the modulated interferer signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until sapphire_startSigGen is invoked.

```
sapphire_error_t  sapphire_setInterferer(bool on, uint32_t MHz, double pwr, uint32_t mod,
                                         uint32_t payload, uint32_t mask, bool cont, uint32_t octets,
                                         uint32_t period )
```

on is a Boolean to indicate whether the modulated interferer signal generator should be started. If this flag is set, then the modulated interferer signal generator is started, otherwise the modulated interferer signal generator is halted if it is already running.

MHz is the carrier frequency for the interferer signal in MHz. Valid range is 2392 MHz to 2488 MHz inclusive.

pwr is the power of the modulated interferer signal in dBm. Valid range is -170dBm to 0dBm.

mod is the phy to be used by the modulated interferer signal. Valid options are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

payload defines the contents of the wanted signal's payload. Valid options are:

0	PRBS9
1	PRBS11
2	PRBS15
3	PRBS20
4	PRBS23
5	PRBS29
6	PRBS31

masl is a mask indicating which digital IO lines should be toggled high when the interferer signal is active. Bits 0 to 7 in the mask correspond to digital output lines 0 to 7. If a bit is set to '1', then the corresponding digital output line will be set high when the interferer is being transmitted. Only lines 2 to 7 can be specified.

cont is a Boolean which indicates whether the interferer transmission is continuous (True) or packetised (False).

octets is the length of the inteferer signal packet payload in octets. This must be in the range 0 to 255. This parameter is ignored if **cont** is True.

period is the interval between the start of one packet and the start of the subsequent packet in units of µs. This parameter is ignored if **cont** is True.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.17 sapphire_setInBandCW

sapphire_setInBandCW() programs the in-band CW signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until sapphire_startSigGen is invoked.

sapphire_error_t **sapphire_setInBandCW**(bool on, uint32_t src, uint32_t MHz, double pwr)

on is a Boolean to indicate whether the in-band CW generator should be turned on or off. If set, then the in-band CW generator is turned on, otherwise the in-band CW generator is turned off.

src indicates which of the two in-band CW sources is being programmed. Valid values are 0 and 1.

MHz is the frequency of the in-band CW signal in Hz. Valid range is 2,395,000,000 Hz to 2,485,000,000 Hz inclusive.

pwr is the power of the in-band CW signal in units of dBm. Valid range is -170 dBm to 0 dBm.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.18 sapphire_setOutOfBandCW

sapphire_setOutOfBandCW programs the out-of-band CW signal generator when the Sapphire application is in signal generator mode. This command can only be executed when the Sapphire application is in signal generator mode. There will be no output from the signal generator until sapphire_startSigGen is invoked.

sapphire_error_t **sapphire_setOutOfBandCW**(bool on, uint32_t MHz, double pwr)

on is a Boolean to indicate whether the out-of-band CW generator should be turned on or off. If set, then the out-of-band CW generator is turned on, otherwise the out-of-band CW generator is turned off.

MHz is the frequency of the out-of-band CW signal in MHz. Valid range is 24 MHz to 6,000 MHz.

pwr is the power of the out-of-band CW signal in units of dBm. Valid range is -50dBm to -28 dBm.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.19 sapphire_startSigGen

sapphire_startSigGen determines when the signal generator is active. This is a global enable for the signal generator output; it overrides the individual on fields for the wanted signal, modulated interferer, in-band CW, AWGN and out-of-band CW. This command can only be executed when the Sapphire application is in signal generator mode.

sapphire_error_t **sapphire_startSigGen**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.20 sapphire_stopSigGen

sapphire_stopSigGen will stop all output from the signal generator. It will not affect the programming of the various signal sources, ie the wanted signal, modulated interferer, in-band CW and out-of-band CW. This command can only be executed when the Sapphire application is in signal generator mode.

sapphire_error_t **sapphire_stopSigGen**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.21 sapphire_signalAnalyser

sapphire_signalAnalyser() starts or stops the signal analyser. It also determines various run parameters for the signal analyser. This command can only be executed when the Sapphire application is in signal analyser mode.

sapphire_error_t **sapphire_signalAnalyser**(bool run, uint32_t aa, uint64_t chanMask,

```
                                    uint32_t testMask, uint32_t pktMask, uint32_t phyMask,
                                    uint32_t pktlenMask , int32_t rssi, bool deWhiten, bool offAir,
                                    uint32_t overSample, bool stopOnFail)
```

run is a Boolean which indicates whether the signal analyser is to be started. If set., the signal analyser will be started, otherwise it will be stopped.

aa contains the 32 bit access address for the packets to be analysed.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers.

testMask is a bit mask indicating which types of analysis should be performed.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

pktMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *PktMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

phyMask is a mask containing the phys which will be analysed. Only packets which match the selected phys will be analysed. The meaning of the bits in this mask are:

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

pktlenMask this mask contains the packet length filter. Only packets which pass the packet length filter will analysed. There are 32 packet length groups. The first length group corresponds to packets with payloads in the range 0 to 7 octets, the second group to packets with payloads of 8 to 15 octets, etc. If a bit is '1', then packets whose payloads fall within the the corresponding packet length group will be

analysed. If a bit is '0', then packets whose payloads fall within the the corresponding packet length group are ignored.

rssi Only packet with an RSSI greater than **rssi** will be analysed. **rssi** is in units of dBm.

deWhiten is a Boolean to indicate whether the packets to be analysed have been whitened. If deWhiten is set then the packets are whitened, otherwise the packets are unwhitened. The signal analyser needs to know whether the packet has been whitened in order to extract the correct length from the packet header.

overSample is the oversampling ratio to be applied to raw captured waveform data. This only applies to the raw waveform data; all transmitter tests are performed with data that has been 32x oversampled. Valid values are:

0	4x oversampling
1	8x oversampling
3	16x oversampling
7	32x oversampling

offAir if True then the off-air mode will be used. This enables packets whose payloads do not conform to those in the Bluetooth LE phy test specification to be analysed.

stopOnFail is a Boolean to indicate how the signal analyser should respond when a limit failure is detected. If clear, then the signal analyser ignores the limit failure and continues to run. If set, then the signal analyser will complete the analysis of the current failing packet and then halt.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.22 sapphire_startScan

startScan() instructs the *TLF3000* to look for advertising events and issue scan requests in reply. The *TLF3000* then looks to see if a scan response was received, indicating that the DUT received the scan request. The power, modulation, carrier frequency offset, drift and in-band emissions are simultaneously calculated. The *TLF3000* can be instructed to gradually reduce its transmit power so that the sensitivity of the DUT can be established.

```
sapphire_error_t sapphire_startScan(float startdBm, float stopdBm, float stepdBm, uint16_t chanMask,
                                    bool concurrent, bool TxAdd, const unsigned char* scanAddr,
                                    uint16_t numPkt, uint16_t stopCond, float timeoutms, uint16_t capture,
                                    float rssiThreshdBm, uint8_t pduMask, bool checkAddrIn,
                                    bool randAddrIn, const unsigned char* cmpAddr)
```

startdBm the power at which the *TLF3000* should send its first **numPkt** packets in units of dBm.

stopdBm the power at which the *TLF3000* should send its last **numPkt** packets in units of dBm. Note that the *TLF3000* will step down from **startdBm** in steps of **stepdBm** until **stopdBm** is reached. Hence the final packets may be sent at a power level slightly higher than **stopdBm**.

stepdBm is the power reduction between each set of **numPkt** in units of dBm. Not more than 32 steps are permitted.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers. Only advertising channels should be used.

concurrent. If this is False, then the *TLF3000* will test each of the 3 advertising channels in turn. If this is True, then the *TLF3000* will test which ever advertising channel the next advertising packet is received on.

TxAdd if this is True then the ScanAddr used by the *TLF3000* is random address, otherwise it is a public address.

scanAddr is the ScanAddr used by the *TLF3000*. This argument is a 12 element character string containing the ScanAddr in hex.

numPkt If **concurrent** is False, then **numPkt** is the number of packets tested on each advertising channel. If **concurrent** is True, then **numPkt** is the total number of packets tested.

stopCond If True, the scan will terminate as soon as a test limit fails. If False, the tests will run to completion, even if a test fails.

timeoutms If the time taken to issue **numPkt** scan requests exceeds **timeoutms** then the current test will be aborted and the test test started. The units of the timeout are ms.

capture is a bit mask indicating which types of analysis should be performed. It is recommended that raw waveform data is not collected unless necessary since this slows down the processing of advertising packets.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

rssiThreshdBm Only received packets which have a signal strength greater than or equal to **rssiThreshdBm** will be processed. The units of the parameter are dBm.

pduMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *pduMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

checkAddrIn If True, then the *TLF3000* will only issue scan requests to the advertising device whose address is **cmpAddr**. If False, the *TLF3000* will issue scan requests to any advertising device.

randAddrIn If True, then **cmpAddr** is a random address, otherwise **cmpAddr** is a public address. Ignored if **checkAddr** is False.

cmpAddr is the advertising address of the device which *TLF3000* should issue scan requests to if **checkAddrIn** is True. This argument is ignored otherwise. This argument is a 12 element character string containing the address in hex.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.23 sapphire_startAdv

startAdv() instructs the *TLF3000* to start sending advertising packets and listen for scan requests. The power, modulation, carrier frequency offset, drift and in-band emissions of the scan requests are calculated. The *TLF3000* can be instructed to gradually reduce its transmit power so that the sensitivity of the DUT can be established.

```
sapphire_error_t sapphire_startAdv(float startdBm, float stopdBm, float stepdBm, uint16_t chanMask,
                                   bool concurrent, uint32_t advPDU, bool ChSel, bool TxAdd, bool RxAdd,
                                   const unsigned char* AdvA, const unsigned char* TargetA,
                                   const unsigned char* PayLoad, uint16_t numPkt, uint16_t period,
                                   uint16_t stopCond, uint16_t capture, float rssiThreshdBm,
                                   uint8_t pduMask, bool checkAddrIn, bool randAddrIn,
                                   const unsigned char* cmpAddr)
```

startdBm the power at which the *TLF3000* should send its first **numPkt** packets in units of dBm.

stopdBm the power at which the *TLF3000* should send its last **numPkt** packets in units of dBm. Note that the *TLF3000* will step down from **startdBm** in steps of **stepdBm** until **stopdBm** is reached. Hence the final packets may be sent at a power level slightly higher than **stopdBm**.

stepdBm is the power reduction between each set of **numPkt** in units of dBm. Not more than 32 steps are permitted.

chanMask is a 40 bit mask indicating which RF channels the signal analyser should examine. A '1' in the mask indicates that the corresponding LE RF channel should be analysed, a '0' in the mask indicates that

the corresponding LE RF channel should be ignored. The mask references RF channel numbers, not LE channel numbers. Only advertising channels should be used.

concurrent. If this is False, then the *TLF3000* will transmit on each of the 3 advertising channels in turn. If this is True, then the *TLF3000* will transmit on all 3 advertising channels simultaneously.

advPDU is the value of the advertising PDU which the *TLF3000* should transmit.

ChSel is the value of the chSel field in the packet header which the *TLF3000* transmits.

TxAdd is the value of the TxAdd field in the packet header which the *TLF3000* transmits.

RxAdd is the value of the RxAdd field in the packet header which the *TLF3000* transmits.

AdvA is the advertising address of the *TLF3000*. This argument is a 12 element character string containing the AdvA address in hex.

TargetA is the TargetA address used in the *TLF3000* transmitted packets. This argument is ignored if advPDU does not require a TargetA address. This argument is a 12 element character string containing the TargetA address in hex.

Payload is the advertising data to be included in the *TLF3000* transmitted packets. This argument is a character string of even length. Each pair of characters represents the hex value of one octet of the payload.

numPkt If **concurrent** is False, then **numPkt** is the number of advertising packets transmitted on each channel. If **concurrent** is True, then **numPkt** is the total number of advertising packets transmitted.

period is the interval between transmitted advertising packets in units of µs.

stopCond If True, advertising will terminate as soon as a test limit fails. If False, the tests will run to completion, even if a test fails.

capture is a bit mask indicating which types of analysis should be performed. It is recommended that raw waveform data is not collected unless necessary since this slows down the processing of advertising packets.

Bit position	Bit mask	Analysis
0	0x0001	Measure power
1	0x0002	Measure modulation characteristics
2	0x0004	Measure carrier offset & drift
3	0x0008	Measure in-band emissions
4	0x0010	Measure CTE
8	0x0100	Collect raw waveform data

rssiThreshdBm Only received packets which have a signal strength greater than or equal to **rssiThreshdBm** will be processed. The units of the parameter are dBm.

pduMask determines which packets are to be analysed. The lowest 4 bits of the (dewhitened) packet header are compared against this mask to determine whether the packet should be analysed. The lowest 4 bits of the packet header are converted into a bit index in the range 0 to 15. If *pduMask* has a '1' at the bit location signified by the bit index then the packet is analysed, otherwise the packet is ignored.

checkAddrIn If True, then the *TLF3000* will only analyse scan requests to the scanning device whose address is **cmpAddr**. If False, the *TLF3000* will analyse scan requests from any device.

randAddrIn If True, then **cmpAddr** is a random address, otherwise **cmpAddr** is a public address. Ignored if **checkAddr** is False.

cmpAddr is the ScanA address of the device which *TLF3000* should analyse scan requests from if **checkAddrIn** is True. This argument is ignored otherwise. This argument is a 12 element character string containing the address in hex.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.24 sapphire_stopAdvScan

stopAdvScan terminates a pervious advertising or scanning session started on the *TLF3000*.

```
sapphire_error_t sapphire_stopAdvScan()
```

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.25 sapphire_sniffAdvA

sapphire_sniffAdvA() instructs the *TLF3000* to start or stop sniffing advertising addresses. Any advertising devices found are reported through a callback.

```
sapphire_error_t sapphire_sniffAdvA( bool run )
```

run If True, then the *TLF3000* will sniff for advertising devices and report their AdvA address and RSSI. If False, *TLF3000* will cease sniffing.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.26 sapphire_getMAC

sapphire_getMAC returns the AdvA address of the last device tested in advertise/scan mode.

```
sapphire_error_t sapphire_getMAC( char *mac , uint32_t l )
```

mac is a character string to receive the AdvA address of the last device tested as a hex string.

l is the maximum length of the character string **mac**.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.27 sapphire_results_advscan

sapphire_results_advscan generates a test report for the last device tested in advertise/scan mode.

char ***sapphire_results_advscan**(char *TestReport, char *ReportHdr)

TestReport is a character string containing the name of the file to receive the test report. The filename should include the suffix .txt, .csv or .html. The format of the output file is governed by the filename suffix.

ReportHeader is character string referencing a text file. The contents of this file will placed at the start of the test report file. This enables the report to be customised.

Returns a pointer to a character string which will normally be either “Passed” or “Failed”. If an error occurred then a pointer to a character string describing the error will be returned.

6.3.28 sapphire_scanWrapper

sapphire_scanWrapper provides a convenient mechanism of testing a DUT which is advertising and generating a test report.

```
char *sapphire_scanWrapper ( uint32_t TFL3000SerialNumber, char *TestReport, char *ReportHdr,
                             bool rxPort, float rxAtten, uint32_t nLoss, float *frqs, float *loss,
                             uint32_t ScriptTimeout, const char *scanA, bool ScanArandom,
                             bool SniffAdvertiser, uint32_t SniffDuration, uint32_t SniffTimeout,
                             const char *AdvA, bool AdvArandom, float txPwr, uint32_t nPkt,
                             int32_t threshRSSI, bool ADV_IND, bool ADV_SCAN_IND, float minPwr,
                             float maxPwr );
```

TLF3000SerialNumber is the serial number of the *TLF3000* to use for the test. If a zero serial number is specified, then the first *TLF3000* unit discovered will be used.

TestReport is a character string containing the name of the file to receive the test report. The filename should include the suffix .txt, .csv or .html. The format of the output file is governed by the filename suffix.

ReportHeader is character string referencing a text file. The contents of this file will placed at the start of the test report file. This enables the report to be customised.

rxPort is the receiver port to use for reception and can be one of the following values:

0	Monitor In
1	Tx/Rx port

rxAtten contains the receiver frontend attenuation in units of dB. The permissible attenuator range is 0 to 31.5 dB.

nLoss is the number of frequency points at which the cable loss is specified.

frqs is an array of length **n** of frequencies in MHz at which the cable loss is specified.

loss is an array of length n of cable losses in dB. The cable loss must be between 0 and 25dB.

ScriptTimeOut If the run time of the tests exceeds **ScriptTimeOut** ms, then the script will be aborted.

scanA is the ScanA used by the *TLF3000*. This argument is a 12 element character string containing the ScanAddr in hex.

ScanARandom if this is True then the ScanA used by the *TLF3000* is random address, otherwise it is a public address.

SniffAdvertiser If True, then the *TLF3000* will sniff for advertising devices. The device with the highest RSSI will be selected for testing. If False, the device to be tested will be termined by **AdvA**.

SniffDuration If **SniffAdvertiser** is True, this is the length of time the *TLF3000* will sniff for advertising devices in units of ms. If this is zero, then the first advertising device found will be used.

SniffTimeout The maximum time the More *TLF3000* ph30 will sniff for advertising devices in units of ms. If no advertising devices are found within this period the test will be aborted.

AdvA The advertising address of the DUT to test. This is ignored if **SniffAdvertiser** is True.

AdvARandom Indicates whether the address specified by **AdvA** is random or public.

txPwr The power that the *TLF3000* should send advertising requests to the DUT in units of dBm.

nPkt The number of advertising requests the *TLF3000* should issue on each advertising channel.

threshRSSI Only packets which have a signal strength greater than **threshRSSI** will be analysed by the *TLF3000*.

ADV_IND If True, the *TLF3000* will respond to ADV_IND packets from the DUT. If False, the *TLF3000* will ignore ADV_IND packets from the DUT.

ADV_SCAN_IND If True, the *TLF3000* will respond to ADV_SCAN_IND packets from the DUT. If False, the *TLF3000* will ignore ADV_SCAN_IND packets from the DUT.

minPwr The lower test limit for output power measurements from the DUT.

maxPwr The upper test limit for output power measurements from the DUT.

Returns a pointer to a character string. The string will normally either contain the text “Passed” or “Failed”. If an error occurred, then the text will contain an error message.

6.3.29 sapphire_pollResultsTable

sapphire_pollResultsTable() returns results in a tabular form when operating in signal analyser mode or advertise/scan mode.

For non-CTE tests, the results contributing to the returned table are filtered by:

1. An RF channel filter
2. A phy filter
3. A packet length group filter

Any packets which satisfy all the filters will contribute to the statistics in the returned table.

For CTE measurements, the filtering operating has to be modified to prevent packets with different CTE configurations being combined. A search is performed to find the last received packet which satisfies:

1. An RF channel filter
2. A phy filter
3. A CTE type filter
4. A CTE length filter

The most recent packet which satisfies these filters is used to determine the CTE type and length which will be used to populate the table. All packets with the same CTE type and length, and which also satisfy the RF channel and phy filters, will contribute to the statistics in the returned table.

sapphire_error_t **sapphire_pollResultsTable**(uint32_t measurement, uint64_t chanMask, uint32_t modulation, uint32_t pktlen, float *res)

measurement is the measurement set for which tabular results are requested. Possible values are:

Meas	Measurement set	Collection mode for signal analyser
0x00000000	Output power	Standard test packets
0x00000001	Modulation characteristics	Standard test packets
0x00000002	Carrier offset & drift	Standard test packets
0x00000003	In-band emissions	Standard test packet & Off-air
0x00000004	#rxed packets in advertise/scan	N/A
0x00000005	CTE – absolute phases	Standard test packet & Off-air
0x40000005	CTE – differential phases	Standard test packet & Off-air
0x80000000	Output power	Off-air mode
0x80000001	Modulation characteristics	Off-air mode
0x80000002	Carrier offset & drift	Off-air mode
0x80000003	In-band emissions	Standard test packet & Off-air
0x80000004	#rxed packets in advertise/scan	N/A
0x80000005	CTE – absolute phases	Standard test packet & Off-air
0xC0000005	CTE – differential phases	Standard test packet & Off-air

The *Meas* types of 0x00000004 and 0x80000004 are equivalent and are only used in advertise/scan mode. When advertising, this *Meas* type returns the number of received scan requests or connection requests. When scanning, this *Meas* type returns the number of received scan responses.

chanMask is a 40 bit mask indicating containing the RF channel filter. Only packets which pass the RF channel filter results will contribute to the statistics in the returned table. A '1' in the mask indicates that the corresponding LE RF channel results should be included in the table, a '0' in the mask indicates that the corresponding LE RF channel results should be ignored.

modulation is a 4 bit mask containing the phy filter. Only packets which pass the phy filter will contribute to the statistics in the returned table. If a bit is '1', then the results for the corresponding phy scheme will contribute to the statistics in the returned table. If a bit is '0', then the results for the corresponding phy scheme are ignored.

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

pktlen. For non-CTE measurements, this mask contained the packet length filter. Only packets which pass the packet length filter will contribute to the statistics in the returned table. There are 32 packet length groups. The first length group corresponds to packets with payloads in the range 0 to 7 octets, the second group to packets with payloads of 8 to 15 octets, etc. If a bit is '1', then the results for the

corresponding packet length group will contribute to the statistics in the returned table. If a bit is '0', then the results for the corresponding packet length group are ignored.

For CTE measurements, the top 8 bits contain the supplemental type filter. During the initial search for the most recent packet, only packets which are consistent with the CTE type filter will be considered. If a bit is set to '1', then packets with the corresponding phy are included in the search. If a bit is set to '0', then packets with the corresponding phy are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Type
24	0x01000000	AoA
25	0x02000000	AoD, 1 µs slots
26	0x04000000	AoD, 2 µs slots

The lower 19 bits contain a filter for the CTE length. During the initial search for the most recent packet, only packets which are consistent with the CTE length filter will be considered. If a bit is set to '1', then packets with the corresponding CTE length are included in the search. If a bit is set to '0', then packets with the corresponding CTE length are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Length
0	0x00000001	16 µs
1	0x00000002	24 µs
2	0x00000004	32 µs
…	…	…
18	0x00040000	160 µs

res contains the requested table of results. The first 5 values in table correspond to the first row in the table, the second 5 values to the second row, etc. The tables returned for the various measurement sets are:

1. Output power

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					

2. Modulation characteristics

Quantity	Packet Count	Minimum	Maximum	Average	Current
$\Delta F1_{max}$					NaN
$\Delta F1_{avg}$					
$\Delta F2_{max}$					NaN
$\Delta F2_{avg}$					
$\Delta F2_{avg}$ / $\Delta F1_{avg}$					
$\Delta F2_{max\ 99.9\%}$					

3. Carrier offset and drift

Quantity	Packet Count	Minimum	Maximum	Average	Current
F_0					
F_n					NaN
$\lvert F_1 - F_0 \rvert$					
$\lvert F_0 - F_n \rvert$					NaN
$\lvert F_n - F_{n-5} \rvert$					NaN

4. In-band emissions

Quantity	Packet Count	Minimum	Maximum	Average	Current
$F_{tx} \pm$ 2MHz					
$F_{tx} \pm$ (3+n)MHz					
Exceptions					

5. # Received packets

Quantity	Packet Count	Minimum	Maximum	Average	Current
#rxed pkts					

This table is only available in advertising/scan mode. This table refers to the number of received scan or connection requests when advertising or the number of received scan responses when scanning. All the entries in the table are identical.

6. CTE- absolute phases

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					
Fs_i					
Fs_1-F_p					
Fs_i-F_0					
Fs_i-Fs_j					
$P_{ref,dev} / P_{ref,ave}$					
$P_{n,dev} / P_{n,ave}$					
Φ_0					
Φ_1					
….					
Φ_N					

Φ_o to Φ_7 correspond to the eight measurements made at 1µs intervals throughout the reference period. Φ_8 to Φ_N refer to phases measured in the subsequent sampling slots. The supplemental is processed by using the first 7µs of the reference period to estimate a starting phase and a frequency offset. The frequency offset is then removed from the supplemental. The starting phase is then subtracted from the frequency compensated supplemental prior to the phases being sampled.

7. CTE- differential phases

Quantity	Packet Count	Minimum	Maximum	Average	Current
P_{avg}					
$Pk - P_{avg}$					
Fs_i					
Fs_1-F_p					
Fs_i-F_0					
Fs_i-Fs_j					
$P_{ref,dev} / P_{ref,ave}$					
$P_{n,dev} / P_{n,ave}$					
$\Phi_1 - \Phi_o$					
$\Phi_2 - \Phi_1$					
….					
$\Phi_N - \Phi_{N-1}$					

Φ_o to Φ_7 correspond to the eight measurements made at 1µs intervals throughout the reference period. Φ_8 to Φ_N refer to phases measured in the subsequent sampling slots. The supplemental

is processed by using the first 7µs of the reference period to estimate a starting phase and a frequency offset. The frequency offset is then removed from the supplemental. The starting phase is then subtracted from the frequency compensated supplemental prior to the phases being sampled.

Some current values are vector quantities rather than scalar quantities, for example, F_n, the average frequency over 10 bits in the drift calculation. The current values for these vector quantities are returned as NaN, as shown in the tables above.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.30 sapphire_pollResultsPlot

sapphire_polltResultsPlot() instructs Sapphire to capture data ready for reading back in a subsequent command,

sapphire_error_t **sapphire_pollResultsPlot**(uint32_t plotThing, uint32_t plotMethod, uint64_t chanMask, uint32_t modulation, uint32_t pktlen, int32_t *n)

To access plot data, a sequence of two commands is required:

1. *pollPlotResults*. This command determines whether data is available, and if so, stashes it within the Sapphire application.
2. *getPlotResults*. This command reads back sections of the stashed data.

Multiple *getPlotResults* commands can be issued to retrieve different sections of the stashed data.

plotThing. The measured quantity for which the plot is requested. Possible values are:

Quantity	Measurement	Notes
0x00000000	P_{avg}	Standard test packets
0x00000001	$Pk - P_{avg}$	Standard test packets
0x00000002	$\Delta F1_{max}$	Standard test packets
0x00000003	$\Delta F1_{avg}$	Standard test packets
0x00000004	$\Delta F2_{max}$	Standard test packets
0x00000005	$\Delta F2_{avg}$	Standard test packets
0x00000006	$\Delta F2_{avg}/\Delta F1_{avg}$	Standard test packets
0x00000007	$\Delta F2_{max}$ *99% percentile*	Standard test packets
0x00000008	F_0	Standard test packets
0x00000009	F_n	Standard test packets
0x0000000A	$F_0 - F_n$	Standard test packets
0x0000000B	$F_1 - F_0$	Standard test packets
0x0000000C	$F_{n+5} - F_n$	Standard test packets
0x0000000D	P_{tx} *@ ±2 MHz offset*	-
0x0000000E	P_{tx} *@ ≥±3 MHz offset*	-
0x0000000F	*# in-band emission exceptions*	-
0x00000010	*Max in-band exception*	-
0x00000011	*# rxed packets*	-
0x00000012	P_{avg} *CTE*	Standard test packets
0x00000013	$P_k - P_{avg}$ *CTE*	Standard test packets
0x00000014	Fs_i	Standard test packets
0x00000015	$Fs_1 - Fs_p$	Standard test packets
0x00000016	$Fs_i - F_0$	Standard test packets
0x00000017	$Fs_i - Fs_j$	Standard test packets
0x00000018	$P_{ref,dev} / P_{ref,ave}$	Standard test packets
0x00000019	$P_{n,dev} / P_{n,ave}$	Standard test packets
0x0000001A	*CTE* Φ_0	Standard test packets
0x0000001B	*CTE* Φ_1	Standard test packets
…	…	Standard test packets
0x0000006B	*CTE* Φ_{81}	Standard test packets
0x4000001A	*CTE* $\Phi_1 - \Phi_0$	Standard test packets
0x4000001B	*CTE* $\Phi_{2\_}\Phi_1$	Standard test packets
…	…	Standard test packets
0x4000006A	*CTE* $\Phi_{81} - \Phi_{80}$	Standard test packets
0x80000000	P_{avg}	Off-air mode
0x80000001	$Pk - P_{avg}$	Off-air mode
0x80000002	$\Delta F1_{max}$	Off-air mode

0x80000003	$\Delta F1_{avg}$	Off-air mode
0x80000004	$\Delta F2_{max}$	Off-air mode
0x80000005	$\Delta F2_{avg}$	Off-air mode
0x80000006	$\Delta F2_{avg}/\Delta F1_{avg}$	Off-air mode
0x80000007	$\Delta F2_{max}$ *99% percentile*	Off-air mode
0x80000008	F_0	Off-air mode
0x80000009	F_n	Off-air mode
0x8000000A	$F_0 - F_n$	Off-air mode
0x8000000B	$F_1 – F_0$	Off-air mode
0x8000000C	$F_{n+5} – F_n$	Off-air mode
0x80000012	P_{avg} *CTE*	Off-air mode
0x80000013	P_k - P_{avg} *CTE*	Off-air mode
0x80000014	Fs_i	Off-air mode
0x80000015	$Fs_1 – Fs_p$	Off-air mode
0x80000016	$Fs_i – F_0$	Off-air mode
0x80000017	$Fs_i – Fs_j$	Off-air mode
0x80000018	$P_{ref,dev} / P_{ref,ave}$	Off-air mode
0x80000019	$P_{n,dev} / P_{n,ave}$	Off-air mode
0x8000001A	*CTE* Φ_0	Off-air mode
0x8000001B	*CTE* Φ_1	Off-air mode
…	…	Off-air mode
0x8000006B	*CTE* Φ_{81}	Off-air mode
0xC000001A	*CTE* Φ_1- Φ_0	Off-air mode
0xC000001B	*CTE* $\Phi_{2\_}\Phi_1$	Off-air mode
…	…	Off-air mode
0xC000006A	*CTE* Φ_{81}- Φ_{80}	Off-air mode
0x000003E8	ACP spectra	-
0x000003E9	Amplitude – stashed packet	-
0x000003EA	FM deviation – stashed packet	-
0x000003EB	CTE phase – stashed packet	-
0x000003EC	IQ data – stashed packet	-
0x000003ED	FM deviation – CTE – stashed packet	-
0x000003EF	Amplitude – CTE – last packet received	-
0x000007D1	Amplitude – last packet received	-
0x000007D2	FM deviation – last packet received	-
0x000007D3	CTE phase last packet received	-
0x000007D4	IQ data – last packet received	-
0x000007D5	FM deviation – CTE – last packet received	-
0x000007D6	Amplitude – CTE - last packet received	-

Some quantities can be estimated when the packet payload does not contain a standard test packet sequence. For these quantities it is necessary to specify whether the 'Standard test packet' or 'off-air' measurement should be returned.

plotMethod. Describes the type of plot which is requested. Possible values are:

0x00000000	Quantity vs RF channel
0x00000001	Quantity vs phy
0x00000002	Quantity vs packet length group
0x00000003	Quantity as a histogram
0x00000004	Quantity vs time or frequency (in-band emissions)

chanMask. A 40 bit mask indicating which RF channels' results should be include in the plot. A '1' in the mask indicates that the corresponding LE RF channel results should be included in the plot, a '0' in the mask indicates that the corresponding LE RF channel results should be ignored. This mask is ignored when *Method* equals 0.

modulation. This is a mask indicating which phy schemes should be included in the plot. If a bit is '1', then the results for the corresponding phy are included in the plot. If a bit is '0', then the results for the corresponding phy are ignored. This mask is ignored when *Method* equals 1.

Bit Position	Bit Mask	Phy
0	0x01	2 Mbps, uncoded
1	0x02	1 Mbps, uncoded
2	0x04	1 Mbps, coded, S = 2
3	0x08	1 Mbps, coded, S = 8

pktLen. For non-CTE meausrements this mask indicates which packet length groups should be included in the results plot. There are 32 packet length groups. The first packet length group includes packets with payload lengths in the range 0 to 7 octets, the second group contains packets with payload lengths of 8 to 15 octets, etc. If a bit is '1', then the results for the corresponding packet length group are included in the plot. If a bit is '0', then the results for the corresponding packet length group are ignored. This mask is ignored when *Method* equals 2.

For CTE measurements, the top 8 bits contain the supplemental type filter. During the initial search for the most recent packet, only packets which are consistent with the CTE type filter will be considered. If a bit is set to '1', then packets with the corresponding phy are included in the search. If a bit is set to '0', then packets with the corresponding phy are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Type
24	0x01000000	AoA
25	0x02000000	AoD, 1 µs slots
26	0x04000000	AoD, 2 µs slots

The lower 19 bits contain a filter for the CTE length. During the initial search for the most recent packet, only packets which are consistent with the CTE length filter will be considered. If a bit is set to '1', then packets with the corresponding CTE length are included in the search. If a bit is set to '0', then packets with the corresponding CTE length are ignored. The meaning of the bits within the filter are:

Bit Position	Bit Mask	Supplemental Length
0	0x00000001	16 µs
1	0x00000002	24 µs
2	0x00000004	32 µs
…	…	…
18	0x00040000	160 µs

n is the number of bytes of data which are available for reading back.

If the parameter **plotMethod** has been set to read back data vs RF channel, phy or packet length group, then three sequences of values are returned. The first sequence corresponds to the minimum value of the quantity, the second sequence to the average value of the quantity and the third sequence to the maximum value of the quantity.

The number of samples available to be read back is:

Method	Number of returned samples	Bytes available to read
0	3*40 = 120	480
1	3*4 = 12	48
2	3*32	128

For *plotMethod* = 0, the first 40 samples correspond to the minimum value observed over RF channels 0 to 39. The next 40 samples correspond to the average value observed over RF channels 0 to 39. The final 40 samples correspond to the maximum value observed over RF channels 0 to 39.

For *plotMethod* = 1, the first 4 samples correspond to the minimum value observed for each modulation scheme. The order of the modulation schemes is:

1. 2 Mbps, uncoded
2. 1 Mbps, uncoded
3. 1 Mbps, coded, S = 2
4. 1 Mbps, coded, S = 8

The next 4 samples correspond the average value observed for each modulation scheme. The final 4 samples correspond to the maximum value observed for each modulation scheme.

For *plotMethod* = 2, the first N samples correspond the minimum value observed for each of the N packet length groups which have been defined. The next N samples correspond the average value observed for each of the N packet length groups. The final N samples correspond to the maximum value observed for each of the N packet length groups.

The units of the returned quantities are:

Measurement	**Units**
P_{avg}	dBm
Pk - P_{avg}	dB
$\Delta F1_{max}$	kHz
$\Delta F1_{avg}$	kHz
$\Delta F2_{max}$	kHz
$\Delta F2_{avg}$	kHz
$\Delta F2_{avg} / \Delta F1_{avg}$	Dimensionless
$\Delta F2_{max}$ 99% percentile	kHz
F_0	kHz
F_n	kHz
F_0 - F_n	kHz
$F_1 - F_0$	kHz
$F_{n+5} - F_n$	kHz
F_{tx} @ ±2MHz offset	dBm
F_{tx} @ ≥±3MHz offset	dBm
# in-band emission exceptions	Dimensionless
Max in-band exception	dBm
P_{avg} *CTE*	dBm
P_k - P_{avg} *CTE*	dBm
Fs_i	kHz
$Fs_1 - Fs_p$	kHz
$Fs_i - F_0$	kHz
$Fs_i - Fs_j$	kHz
$P_{ref,dev} / P_{ref,ave}$	dB
$P_{n,dev} / P_{n,ave}$	dB
Φ_n	°
Φ_n- Φ_{n-1}	°

If **plotMethod** is set to 3 then a histogram of the specified quantity will be collected. Each histogram is composed of 128 floating point samples. These form a histogram with equally spaced bins. The lower and upper edges of the first and last bins are:

Quantity	**Lower edge of 1st bin**	**Upper edge of last bin**
P_{avg}	-100 dBm	+28 dBm
$Pk - P_{avg}$	0 db	6.4 dB
$\Delta F1_{max}$	-512 kHz	+512 kHz
$\Delta F1_{avg}$	0 kHz	+512 kHz
$\Delta F2_{max}$	-512 kHz	+512 kHz
$\Delta F2_{avg}$	0 kHz	+512 kHz
$\Delta F2_{avg}/\Delta F1_{avg}$	0	1.28
$\Delta F2_{max}$ *99% percentile*	0	512 kHz
F_0	-256 kHz	+256 kHz
F_n	-256 kHz	+256 kHz
$F_0 - F_n$	-128 kHz	+128 kHz
$F_1 - F_0$	-64 kHz	+64 kHz
$F_{n+5} - F_n$	-64 kHz	+64 kHz
F_{tx} *@ ±2MHz offset*	-100 dBm	+28 dBm
F_{tx} *@ ≥±3MHz offset*	-100 dBm	+28 dBm
# in-band emission exceptions	0	128
Max in-band exception	-100 dBm	+28 dBm
P_{avg} *CTE*	-100 dBm	+28 dBm
$P_k - P_{avg}$ *CTE*	0 db	6.4 dB
Fs_i	-256 kHz	+256 kHz
$Fs_1 - Fs_p$	-32 kHz	+32 kHz
$Fs_i - F_0$	-32 kHz	+32 kHz
$Fs_1 - Fs_p$	-32 kHz	+32 kHz
$P_{ref,dev} / P_{ref,ave}$	0	1
$P_{n,dev} / P_{n,ave}$	0	1
Φ_n	-180°	+180°
$\Phi_n - \Phi_{n-1}$	-180°	+180°

If **plotMethod** is set to 4, then the specified quantity is collected as a function of time or frequency (in-band emission spectra).

For those quantities which are collected as a function of time, the saved data consists of 24 bytes of meta data followed by the requested quantity as a series of floating point numbers. The meta data is composed of 6 uint32_t:

1. *uint32_t : MSB*. MSB of timestamp of packet P_0 location in units of 250 ns since the Unix epoch of 1970-01-01 00:00:00 + 0000 (UTC).

2. *uint32_t : LSB*. LSB of timestamp of packet P_0 location in units of 250 ns since the Unix epoch of 1970-01-01 00:00:00 + 0000 (UTC).
3. *uint32_t : N.* The number of samples available.
4. *uint32_t : RF Chan*. The LE RF channel number on which the packet was collected.
5. *uint32_t : Phy*. phy of the packet for which the data was collected. Possible values are:

0	2 Mbps, uncoded
1	1 Mbps, uncoded
2	1 Mbps, coded, S = 2
3	1 Mbps, coded, S = 8

6. *uint32_t : Len*. For non-CTE quantities this istThe number of octets in the payload of the packet for which the data was collected. For CTE quantities the lower 24 bits contain the number of slots in the CTE whilst the upper 8 bits contain the CTE type.

Amplitude, FM deviation and IQ data commence 100µs prior to the starts of the packet and continue for 100µs past the end of the packet.

In standard test packet collection mode:

1. The first samples of $\Delta F1_{max}$ and $\Delta F2_{max}$ correspond to bit 4 of the payload. The interval between samples is 1 bit. Samples which are not valid contain NaN.
2. The first samples of *Fn* correspond to the average starting at bit 1 of the payload. The interval between samples is 10 bits. These samples are used to derive F_0 - F_n and $F_{n+5} – F_n$.

In off-air mode:

1. The first samples of $\Delta F1_{max}$ and $\Delta F2_{max}$ correspond to bit 1 of the packet. The interval between samples is 1 bit. Samples which are not valid contain NaN.
2. The first samples of *Fn* correspond to the average starting at bit 1 of the payload. The interval between samples is 16 bits. Samples which are not valid contain NaN. These samples are used to derive F_0 - F_n and $F_{n+5} – F_n$.
3. F_1 is taken as the value of F_n that is closest to the averaging period used to calculate F_1 in the standard test packet mode.

The units of the returned quantities are:

Measurement	**Units**
P_{avg}	dBm
Pk - P_{avg}	dB
$\Delta F1_{max}$	kHz
$\Delta F1_{avg}$	kHz
$\Delta F2_{max}$	kHz
$\Delta F2_{avg}$	kHz
$\Delta F2_{avg} / \Delta F1_{avg}$	Dimensionless
$\Delta F2_{max}$ 99% percentile	kHz
F_0	kHz
F_n	kHz
F_0 - F_n	kHz
$F_1 - F_0$	kHz
$F_{n+5} - F_n$	kHz
F_{tx} @ ±2 MHz offset	dBm
F_{tx} @ ≥± 3MHz offset	dBm
# in-band emission exceptions	Dimensionless
Max in-band exception	dBm
P_{avg} *CTE*	dBm
P_k - P_{avg} *CTE*	dBm
Fs_i	kHz
$Fs_1 - Fs_p$	kHz
$Fs_i - F_0$	kHz
$Fs_i - Fs_j$	kHz
$P_{ref,dev} / P_{ref,ave}$	dB
$P_{n,dev} / P_{n,ave}$	dB
Φ_n	°
$\Phi_n - \Phi_{n-1}$	°
Amplitude	dBm
FM deviation	kHz

If **plotMethod** is set to 4 and the quantity is an in-band emission quantity then both 100 kHz resolution and 1MHz resolution spectra are collected.

The data which can be read back represents in-band emissions as a function of frequency. Results are available for both the final 1MHz resolution spectrum and the 100kHz resolution spectrum which was summed to generate the 1MHz spectrum. This provides greater visibility of what is dominating the in-band emissions.

If the **ChanMask** contains a single channel, then the available data includes the average as well as the minimum, maximum and current values of the spectra. If the **ChanMask** contains more than one channel, then the average spectra are not available.

The available data can contain the following fields:

1. *81*float : Min_1MHz*. The lowest recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
2. *81*float : Max_1MHz*. The highest recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
3. *81*float : Curr_1MHz*. The last recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz.
4. *81*float : Avg_1MHz*. The average recorded value of the in-band emissions as a function of frequency from 2401MHz to 2481 MHz. This field is only present if the *ChanMask* contained a single channel.
5. *810*float : Min_100kHz*. The lowest recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
6. *810*float : Max_100kHz*. The highest recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
7. *810*float : Curr_100kHz*. The last recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz.
8. *810*float : Avg_100kHz*. The average recorded values of the 100 kHz spectrum summed to generate the in-band emissions spectrum as a function of frequency from 2400.550 MHz to 2481.450 MHz. This field is only present if the *ChanMask* contained a single channel.

All spectra are in units of dBm.

n is the number of bytes of data which are available to be read back, or zero if no data is available.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.31 sapphire_getResultsPlotInt16

sapphire_getResultsPlotInt16() reads back the data which was captured using a previous sapphire_polltResultsPlot() command. This command should be used when the captured data consisted of an array of int16.

sapphire_error_t **sapphire_getResultsPlotInt16**(uint32_t offset, uint32_t n, int16_t *res)

offset is the offset from the start of the buffer of the data to be read back (see sapphire_pollResultsPlot). The offset is in units of 2 bytes.

n is the number of int16 to be read back from the buffer.

res is the data read back from the buffer.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.32 sapphire_getResultsPlotFloat

sapphire_getResultsPlotFloat() reads back the data which was captured using a previous sapphire_polltResultsPlot() command. This command should be used when the captured data consisted of an array of float.

sapphire_error_t **sapphire_getResultsPlotFloat**(uint32_t offset, uint32_t n, float *res)

offset is the offset from the start of the buffer of the data to be read back (see sapphire_pollResultsPlot). The offset is in units of 4 bytes.

n is the number of float to be read back from the buffer.

res is the data read back from the buffer.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.33 sapphire_clearResults

sapphire_clearResults() erases all test results accumulated in the Sapphire application by the signal analyser mode or the advertise/scan mode. This command can be executed at any time.

sapphire_error_t **sapphire_clearResults**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.34 sapphire_setRxAtten

sapphire_setRxAtten() sets the receiver frontend attenuation. This command can be executed at any time.

sapphire_error_t **sapphire_setRxAtten**(uint32_t atten)

atten contains the receiver frontend attenuation in units of 0.5 dB. The permissible attenuator range is 0 to 31.5 dB.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.35 sapphire_setRxPort

sapphire_setRxPort() determines whether the Monitor In port or the Tx/Rx port should be used for reception. The Monitor In port is has a noise figure of +6 dB and so is ideally suited for performing off-air

measurements. The Tx/Rx port has a noise figure of +46 dB but can handle signals as large as +27 dBm. It is therefore ideally suited for conducted measurements. This command can be executed at any time.

sapphire_error_t **sapphire_setRxPort**(uint32_t port)

port is the receiver port to use for reception and can be one of the following values:

0	Monitor In
1	Tx/Rx port

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.36 sapphire_setDIOVolts

sapphire_setDIOVolts() determines whether the digital IO voltage is supplied by *TLF3000* or is supplied by an external source. If *TLF3000* supplies the digital IO voltage then it is fixed at 3.3 V. The *TLF3000* is able to provide up to 500 mA on the 3v3 supply. If an external voltage is used for the digital IO then it must be in the range 0.8 V to 3.6 V. This command can be executed at any time.

sapphire_error_t **sapphire_setDioVolts**(bool volts)

volts determines whether the IO voltage is supplied internally or externally:

0	3.3 V IO supplied by *TLF3000*
1	0.8 V to 3.6 V IO supplied externally

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.37 sapphire_getError

sapphire_getError() returns the oldest error message from the Sapphire error queue. Once read, this eror message is removed from the error queue.

sapphire_error_t **sapphire_getError**(char* err , uint32_t maxlen)

err is a string containing the oldest message of the Sapphire error queue. If the queue is empty, then an empty string is returned.

maxlen is the maximum length of the string which can be returned in **err**.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.38 sapphire_search

sapphire_search returns a list of the *TLF3000* devices connected to the host computer.

sapphire_error_t **sapphire_search**(sapphire_search_result_t** results, int* N)

results is a list of devices which are connected to the host computer. The serial number of each device can be obtained from `results[i].serial_number.`

N is the number of devices which have been found.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.39 sapphire_connect

sapphire_connect() connects to a device discovered by **sapphire_search** and launches the Sapphire application.

sapphire_error_t **sapphire_connect**(sapphire_handle_t* handle)

handle is a handle to the device as returned by **sapphire_search**.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.40 sapphire_disconnect

sapphire_disconnect() will cause the Sapphire application to exit and control return to the *TLF3000* supervistor program,

sapphire_error_t **sapphire_disconnect**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.41 sapphire_setDataCallback

sapphire_setDataCallback adds an asynchronous data callback to the Sapphire application. This is used to report asynchronous data such as test results or the termination of a command.

sapphire_error_t **sapphire_set_data_callback**(sapphire_data_callback_t cb)

cb the name of the callback to be used.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.42 sapphire_getFriendlyName

sapphire_getFriendlyName() will return the friendly name of the *TLF3000* unit.

sapphire_error_t **sapphire_friendly_name**(char* name, unsigned maxlen)

name is a string containging the friendly name of the *TLF3000* unit.

maxlen is the maximum length of the string which can be returned in **name**.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.43 sapphire_getSerialNumber

sapphire_getSerialNumber() will return the serial number of the *TLF3000* unit.

sapphire_error_t **morpehlp_serial_number**(uint32_t* serial)

serial is an integer containing the serial number of the *TLF3000* unit.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.44 sapphire_DUTclose

sapphire_DUTclose() closes any serial communications to the DUT. This can be used on a production line to close the communications with one DUT allowing another DUT to be inserted and the tests re-run.

sapphire_error_t **sapphire_DUTClose**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.45 sapphire_stashExe

sapphire_stashExe() instructs the *TLF3000* to place the Sapphire image into RAM disc so that it can be quickly retrieved after another application has been run.

sapphire_error_t **sapphire_stashExe**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.46 sapphire_swapExe

sapphire_swapExe() tells Sapphire which application is going to be run next on the *TLF3000*. This application will be started when the Sapphire application is suspended.

sapphire_error_t **sapphire_swapExe**(unsigned int num)

num is the application number of the next application to be run.

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.47 sapphire_suspend

sapphire_suspend() suspends execution of the Sapphire application and changes execution to the application specified in the last **sapphire_swapExc()** command.

sapphire_error_t **sapphire_suspend**(void)

Returns SAPPHIRE_NO_ERROR if the command succeeds.

6.3.48 sapphire_resume

sapphire_resume() should be called after another application has swapped execution to the Sapphire application. This causes the resumption of the Sapphire application.

```
sapphire_error_t sapphire_resume(void)
```

Returns SAPPHIRE_NO_ERROR if the command succeeds.

7 Test Script Format

7.1 Overview

When running in phy level test mode, the tests which are performed are specified via a test script. A test script is a named ASCII file which is downloaded to the Sapphire application. Multiple test scripts can reside within the Sapphire application simultaneously. These test scripts can reside in either RAM or FLASH (not currently implemented). Test scripts residing in RAM will be lost when the unit is powered down. Test scripts held in FLASH will persist between power cycles of the unit.

7.2 General Format

Test scripts are held as ASCII files. The test script is case insensitive. There should be no whitespace within the test script.

Each line within the file represents a new test to be run. This is composed of:

1. A test header to indicate which test is to be run
2. A test body to indicate the parameters for that test

The test header and test body are separated by the character ‘,’.

Fields within the test body are separated by the character ‘,’.

7.3 Test header

The first 3 letters of each line must be either:

1. ‘TRM’ to represent a transmitter test
2. ‘RCV’ to represent a receiver test

If the 4th letter of the line is ‘/’, then the next letters must be one of the following, indicating a subtest:

1. ‘PS’ to indicate a transmitter CTE power stability test
2. ‘ASI’ to indicate a transmitter CTE antenna switching integrity test
3. ‘IQC’ to indicate a receiver IQ coherency test
4. ‘IQDR’ to indicate a receiver IQ dynamic range test

Immediately following the transmit/receive letters is a test number to indicate which test should be performed. The values for transmitter tests (TRM) are:

Test number	Test description	Phy
1	Output power	Uncoded, 1 Mbps
3	In-band emissions	Uncoded, 1 Mbps
5	Modulation characteristics	Uncoded, 1 Mbps
6	Carrier frequency offset & drift	Uncoded, 1 Mbps
7	In-band emissions	2 Mbps
9	Modulation characteristics	Stable, uncoded, 1 Mbps
10	Modulation characteristics	2 Mbps
11	Modulation characteristics	Stable, 2 Mbps
12	Carrier frequency offset & drift	2 Mbps
13	Modulation characteristics	LE Coded, S=8
14	Carrier frequency offset & drift	LE Coded, S=8
90	Output power	Uncoded, 2Mbps
91	Output power	LE Coded, S=2
92	Output power	LE Coded, S=8
93	In-band emissions	LE Coded, S=2
94	In-band emission	LE Coded, S=8

The values for transmitter CTE power stability tests (TRM/PS) are:

Test number	Test description	Phy
1	Tx power stability, AoD	Uncoded, 1 Mbps, 2µs slots
2	Tx power stability, AoD	Uncoded, 1 Mbps, 1µs slots
3	Tx power stability, AoD	Uncoded, 2 Mbps, 2µs slots
4	Tx power stability, AoD	Uncoded, 2 Mbps, 1µs slots

The values for transmitter CTE antenna switching integrity tests (TRM/ASI) are:

Test number	Test description	Phy
5	Antenna switching integrity	Uncoded, 1 Mbps, 2µs slots
6	Antenna switching integrity	Uncoded, 1 Mbps, 1µs slots
7	Antenna switching integrity	Uncoded, 2 Mbps, 2µs slots
8	Antenna switching integrity	Uncoded, 2 Mbps, 1µs slots

And for receiver tests (RCV):

Test number	Test description	Phy
1	Receiver sensitivity	Uncoded, 1 Mbps
3	C/I & receiver selectivity	Uncoded, 1 Mbps
4	Blocking	Uncoded, 1 Mbps
5	Intermodulation	Uncoded, 1 Mbps
6	Maximum input signal	Uncoded, 1 Mbps
7	PER report integrity	Uncoded, 1 Mbps
8	Receiver sensitivity	2 Mbps
9	C/I & receiver selectivity	2 Mbps
10	Blocking	2 Mbps
11	Intermodulation	2 Mbps
12	Maximum input signal	2 Mbps
13	PER report integrity	2 Mbps
14	Receiver sensitivity	Stable, uncoded, 1Mbps
15	C/I & receiver selectivity	Stable, uncoded, 1 Mbps
16	Blocking	Stable, uncoded, 1 Mbps
17	Intermodulation	Stable, uncoded, 1 Mbps
18	Maximum input signal	Stable, uncoded, 1Mbps
19	PER report integrity	Stable, uncoded, 1 Mbps
20	Receiver sensitivity	Stable, 2 Mbps
21	C/I & receiver selectivity	Stable, 2 Mbps
22	Blocking	Stable, 2 Mbps
23	Intermodulation	Stable, 2 Mbps
24	Maximum input signal	Stable, 2 Mbps
25	PER report integrity	Stable, 2 Mbps
26	Receiver sensitivity	LE coded, S=2
27	Receiver sensitivity	LE coded, S=8
28	C/I & receiver selectivity	LE coded, S=2
29	C/I & receiver selectivity	LE coded, S=8
30	PER report integrity	LE coded, S=2
31	PER report integrity	LE coded, S=8
32	Receiver sensitivity	Stable, LE coded, S=2
33	Receiver sensitivity	Stable, LE coded, S=8
34	C/I & receiver selectivity	Stable, LE coded, S=2
35	C/I & receiver selectivity	Stable, LE coded, S=8
36	PER report integrity	Stable, LE coded, S=2
37	PER report integrity	Stable, LE coded, S=8

The values for receiver CTE IQ coherency tests (RCV/IQC) are:

Test number	Test description	Phy
1	IQ sample coherency, AoD	Uncoded, 1 Mbps, 2µs slots
2	IQ sample coherency, AoD	Uncoded, 1 Mbps, 1µs slots
3	IQ sample coherency, AoD	Uncoded, 2 Mbps, 2µs slots
4	IQ sample coherency, AoD	Uncoded, 2 Mbps, 1µs slots
5	IQ sample coherency, AoA	Uncoded, 1 Mbps, 2µs slots
6	IQ sample coherency, AoA	Uncoded, 2 Mbps, 2µs slots

The values for transmitter CTE antenna switching integrity tests (TRM/ASI) are:

Test number	Test description	Phy
7	IQ samples dynamic range, AoD	Uncoded, 1 Mbps, 2µs slots
8	IQ samples dynamic range, AoD	Uncoded, 1 Mbps, 1µs slots
9	IQ samples dynamic range, AoD	Uncoded, 2 Mbps, 2µs slots
10	IQ samples dynamic range, AoD	Uncoded, 2 Mbps, 1µs slots
11	IQ samples dynamic range, AoA	Uncoded, 1 Mbps, 2µs slots
12	IQ samples dynamic range, AoA	Uncoded, 2 Mbps, 2µs slots

The format of the remainder of the test script line depends on which test has been specified.

7.4 Test Body

The format of the test body varies depending on the test specified in the test header.

7.4.1 Test body for transmitter tests without CTE

All transmitter tests without CTE have the same test body which takes the format:

"*payload_lengths,rf_channels,num_packets*"

payload_lengths is a numeric field specifying the range of payload lengthss for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is a broadcast device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels

shown below will be assumed:

Test #	**Test description**	**Phy**	**Central Peripheral**	**Broadcaster**
TRM1	Output power	Uncoded, 1 Mbps	0,19,39	0,12,39
TRM3	In-band emissions	Uncoded, 1 Mbps	2,19,37	0,12,39
TRM5	Modulation characteristics	Uncoded, 1 Mbps	0,19,39	0,12,39
TRM6	Carrier frequency offset & drift	Uncoded, 1 Mbps	0,19,39	0,12,39
TRM8	In-band emissions	Uncoded, 2 Mbps	2,19.37	0,12,39
TRM9	Modulation characteristics	Stable, uncoded, 1 Mbps	0,19,39	0,12,39
TRM10	Modulation characteristics	Uncoded, 2 Mbps	0,19,39	0,12,39
TRM11	Modulation characteristics	Stable, uncoded, 2 Mbps	0,19,39	0,12,39
TRM12	Carrier frequency offset & drift	Uncoded, 2 Mbps	0,19,39	0,12,39
TRM13	Modulation characteristics	LE Coded, S=8	0,19,39	0,12,39
TRM14	Carrier frequency offset & drift	LE Coded, S=8	0,19,39	0,12,39

num_packets is a numeric integer field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default number of packets shown below will be assumed:

Test #	*Test description*	*Phy*	*# Packets*
TRM1	Output power	Uncoded, 1 Mbps	1
TRM3	In-band emissions	Uncoded, 1 Mbps	max[1 , 2.5 ms/packet_duration]
TRM5	Modulation characteristics	Uncoded, 1 Mbps	10 + 10
TRM6	Carrier frequency offset & drift	Uncoded, 1 Mbps	10
TRM8	In-band emissions	Uncoded, 2 Mbps	max[1 , 2.5 ms/packet_duration]
TRM9	Modulation characteristics	Stable, uncoded, 1 Mbps	10 + 10
TRM10	Modulation characteristics	2 Mbps	10 + 10
TRM11	Modulation characteristics	Stable, uncoded, 2 Mbps	10 + 10
TRM12	Carrier frequency offset & drift	2 Mbps	10
TRM13	Modulation characteristics	LE Coded, S=8	10 + 10
TRM14	Carrier frequency offset & drift	LE Coded, S=8	10

7.4.2 Test body for transmitter tests with CTE

All transmitter tests with CTE have the same test body which takes the format:

"*payload_lengths,rf_channels,num_packets,CTE_len,num_antenna,antenna_IDs*"

payload_lengths is a numeric field specifying the range of payload lengthss for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is a broadcast device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels shown below will be assumed:

Test #	Test description	Phy	Central Peripheral	Broadcaster
TRM15	Output power with CTE	Uncoded, 1 Mbps	0,19,39	0,12,39
TRM16	Carrier frequency offset & drift with CTE	Uncoded, 1 Mbps	0,19,39	0,12,39
TRM17	Carrier frequency offset & drift with CTE	Uncoded, 2 Mbps	0,19,39	0,12,39
TRM/PS1	Tx power stability, AoD	Uncoded, 1 Mbps, 2µs slots	0,19,39	0,12,39
TRM/PS2	Tx power stability, AoD	Uncoded, 1 Mbps, 1µs slots	0,19,39	0,12,39
TRM/PS3	Tx power stability, AoD	Uncoded, 2 Mbps, 2µs slots	0,19,39	0,12,39
TRM/PS4	Tx power stability, AoD	Uncoded, 2 Mbps, 1µs slots	0,19,39	0,12,39
TRM/PS5	Antenna switching integrity	Uncoded, 1 Mbps, 2µs slots	0,19,39	0,12,39
TRM/PS6	Antenna switching integrity	Uncoded, 1 Mbps, 1µs slots	0,19,39	0,12,39
TRM/PS7	Antenna switching integrity	Uncoded, 2 Mbps, 2µs slots	0,19,39	0,12,39
TRM/PS8	Antenna switching integrity	Uncoded, 2 Mbps, 1µs slots	0,19,39	0,12,39

num_packets is a numeric integer field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default number of packets shown below will be assumed:

Test #	*Test description*	*Phy*	*# Packets*
TRM15	Output power with CTE	Uncoded, 1 Mbps	1
TRM16	Carrier frequency offset & drift with CTE	Uncoded, 1 Mbps	10
TRM17	Carrier frequency offset & drift with CTE	Uncoded, 2 Mbps	10
TRM/PS1	Tx power stability, AoD	Uncoded, 1 Mbps, 2µs slots	1
TRM/PS2	Tx power stability, AoD	Uncoded, 1 Mbps, 1µs slots	1
TRM/PS3	Tx power stability, AoD	Uncoded, 2 Mbps, 2µs slots	1
TRM/PS4	Tx power stability, AoD	Uncoded, 2 Mbps, 1µs slots	1
TRM/PS5	Antenna switching integrity	Uncoded, 1 Mbps, 2µs slots	1
TRM/PS6	Antenna switching integrity	Uncoded, 1 Mbps, 1µs slots	1
TRM/PS7	Antenna switching integrity	Uncoded, 2 Mbps, 2µs slots	1
TRM/PS8	Antenna switching integrity	Uncoded, 2 Mbps, 1µs slots	1

CTE_len is a numeric integer field specifying the number of 8µs slots in the CTE. If this field is not specified, then it will default to 20.

num_antenna is a numeric integer field specifying the number of antenna. If this field is not specified, then it will default to 2.

antenna_IDs is a numeric field specifying the antenna IDs to be used. If this field is not specified, then it will default to 0:*num_antenna*-1.

7.4.3 Test body for receiver sensitivity and maximum input signal

Receiver sensitivity and maximum input signal level tests shared the same test body format:

'*payload_lengths,rf_channels,num_packets,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates*'

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels

which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels shown below will be assumed.

Test #	Test description	Phy	Central Peripheral	Observer
RCV1	Receiver sensitivity	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV6	Maximum input signal	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV8	Receiver sensitivity	Uncoded, 2 Mbps	0,19,39	0,12,39
RCV12	Maximum input signal	Uncoded, 2 Mbps	0,19,39	0,12,39
RCV14	Receiver sensitivity	Stable, uncoded, 1Mbps	0,19,39	0,12,39
RCV18	Maximum input signal	Stable, uncoded, 1 Mbps	0,19,39	0,12,39
RCV20	Receiver sensitivity	Stable, uncoded, 2 Mbps	0,19,39	0,12,39
RCV24	Maximum input signal	Stable, uncoded, 2 Mbps	0,19,39	0,12,39
RCV26	Receiver sensitivity	LE coded, S=2	0,19,39	0,12,39
RCV27	Receiver sensitivity	LE coded, S=8	0,19,39	0,12,39
RCV32	Receiver sensitivity	Stable, LE coded, S=2	0,19,39	0,12,39
RCV33	Receiver sensitivity	Stable, LE coded, S=8	0,19,39	0,12,39

num_packets is a numeric field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default value of 1500 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then the default values shown below will be assumed:

Test #	Test description	Phy	Wanted signal
RCV1	Receiver sensitivity	Uncoded, 1 Mbps	-70 dBm
RCV6	Maximum input signal	Uncoded, 1 Mbps	-10 dBm
RCV8	Receiver sensitivity	Uncoded, 2 Mbps	-70 dBm
RCV12	Maximum input signal	Uncoded, 2 Mbps	-10 dBm
RCV14	Receiver sensitivity	Stable, uncoded, 1 Mbps	-70 dBm
RCV18	Maximum input signal	Stable, uncoded, 1 Mbps	-10 dBm
RCV20	Receiver sensitivity	Stable, uncoded, 2 Mbps	-70 dBm
RCV24	Maximum input signal	Stable, uncoded, 2 Mbps	-10 dBm
RCV26	Receiver sensitivity	LE coded, S=2	-75 dBm
RCV27	Receiver sensitivity	LE coded, S=8	-82 dBm
RCV32	Receiver sensitivity	Stable, LE coded, S=2	-75 dBm
RCV33	Receiver sensitivity	Stable, LE coded, S=8	-82 dBm

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified then:

1. Receiver sensitivity tests assume the following:

Entry	Carrier offset
1	100 kHz
2	19 kHz
3	-3 kHz
4	1 kHz
5	52 kHz
6	0 kHz
7	-56 kHz
8	97 kHz
9	-25 kHz
10	-100 kHz

2. Maximum input signal tests assume no carrier frequency offset.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then the following assumptions are made:

1. For receiver sensitivity test with standard modulation indices a list of 10 entries is assumed:

Entry	Modulation index
1	0.450
2	0.480
3	0.460
4	0.520
5	0.530
6	0.540
7	0.470
8	0.500
9	0.450
10	0.550

2. For receiver sensitivity tests with stable modulation indices a list of 10 entries is assumed:

Entry	Modulation index
1	0.495
2	0.498
3	0.496
4	0.502
5	0.503
6	0.504
7	0.497
8	0.500
9	0.495
10	0.505

3. For maximum input signal level tests a value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified (or 10 if no carrier frequency offsets were specified)
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, then the following assumptions are made:

1. For receiver sensitivity tests a list of 10 entries is assumed:

Entry	Symbol Timing Error
1	-50 ppm
2	-50 ppm
3	+50 ppm
4	+50 ppm
5	+50 ppm
6	-50 ppm
7	-50 ppm
8	-50 ppm
9	-50 ppm
10	+50 ppm

2. For maximum input signal level tests a value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified (or 10 if no carrier frequency offsets were specified)
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then the following assumptions are made:

1. For receiver sensitivity tests a value of 50 kHz is assumed
2. For maximum input signal level tests a value 0 kHz is assumed

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified (or 10 if no carrier frequency offsets were specified)
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then the following assumptions are made:

1. For receiver sensitivity tests a value of 1250 Hz is assumed
2. For maximum input signal level tests a value 0 Hz is assumed

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified (or 10 if no carrier frequency offsets were specified)
2. One, in which case the specified drift rate is used for all transmitted packets

7.4.4 Test body for C/I and receiver selectivity tests

Receiver C/I and selectivity tests have a test body in the form:

"*payload_lengths,rf_channels,num_packets,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,interferer_offsets,image_hi_lo,ci_levels_lo,search_range,ci_level_hi*"

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels shown below will be assumed:

Test #	Test description	Phy	Central/Peripheral	Observer
RCV3	C/I & receiver selectivity	Uncoded, 1 Mbps	2,19,37	0,12,39
RCV9	C/I & receiver selectivity	Uncoded, 2 Mbps	2,19,37	0,12,39
RCV15	C/I & receiver selectivity	Stable, uncoded, 1 Mbps	2,19,37	0,12,39
RCV21	C/I & receiver selectivity	Stable, uncoded, 2 Mbps	2,19,37	0,12,39
RCV28	C/I & receiver selectivity	LE coded, S=2	2,19,37	0,12,39
RCV29	C/I & receiver selectivity	LE coded, S=8	2,19,37	0,12,39
RCV34	C/I & receiver selectivity	Stable, LE coded, S=2	2,19,37	0,12,39
RCV35	C/I & receiver selectivity	Stable, LE coded, S=8	2,19,37	0,12,39

num_packets is a numeric field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default value of 1500 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then the default values shown below will be assumed:

Test #	Test description	Phy	Wanted signal
RCV3	C/I & receiver selectivity	Uncoded, 1 Mbps	-67 dBm
RCV9	C/I & receiver selectivity	Uncoded, 2 Mbps	-67 dBm
RCV15	C/I & receiver selectivity	Stable, uncoded, 1 Mbps	-67 dBm
RCV21	C/I & receiver selectivity	Stable, uncoded, 2 Mbps	-67 dBm
RCV28	C/I & receiver selectivity	LE coded, S=2	-72 dBm
RCV29	C/I & receiver selectivity	LE coded, S=8	-79 dBm
RCV34	C/I & receiver selectivity	Stable, LE coded, S=2	-72 dBm
RCV35	C/I & receiver selectivity	Stable, LE coded, S=8	-79 dBm

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified then a value of 0 kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, then a value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign

of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a value of 1250 Hz is assumed.

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift rate is used for all transmitted packets

interferer_offsets is a numeric field containing the frequency offsets of the interferer from the wanted signal in units of MHz which will be tested. Where the wanted signal frequency plus the interferer offset lies outside the range 2400 MHz to 2483 MHz, then the test will be silently ignored.

If not interferer offsets are specified, then the following assumptions are made:

Test #	Test description	Phy	Interferer Offsets
RCV3	C/I & receiver selectivity	Uncoded, 1 Mbps	-80:1:+81 MHz
RCV9	C/I & receiver selectivity	Uncoded, 2 Mbps	-80:2:80 MHz
RCV15	C/I & receiver selectivity	Stable, uncoded, 1 Mbps	-80:1:+81 MHz
RCV21	C/I & receiver selectivity	Stable, Uncoded, 2 Mbps	-80:2:80 MHz
RCV28	C/I & receiver selectivity	LE coded, S=2	-80:1:+81 MHz
RCV29	C/I & receiver selectivity	LE coded, S=8	-80:1:+81 MHz
RCV34	C/I & receiver selectivity	Stable, LE coded, S=2	-80:1:+81 MHz
RCV35	C/I & receiver selectivity	Stable, LE coded, S=8	-80:1:+81 MHz

image_hi_lo is a numeric field containing the RF channels on which high-side mix is applied by the receiver, i.e. the image frequency is located above the wanted signal frequency. The phy level tester requires this information to correctly set the interferer level on the image frequency and adjacent channels. If this field is absent, then it is assumed that all RF channels are low-side mix, i.e. the image frequency is located below the wanted signal frequency.

The phy level tester assumes that the image frequency is fixed for all RF channels. The image frequency is specified in the *Set DUT Type* host command.

ci_levels_lo is a numeric array containing the relative level of the wanted signal to the interferer in units of dB.

The number of entries in the list of C/I values must be equal to either:

1. the number of entries of in the interferer offset list, or the number of entries in the default interferer offset list is not interferer offsets were specified,
2. one, in which case the same C/I value will be used for all interferer offsets.

The C/I values should be specified on the assumption that the receiver utilities low-side mixing, i.e. the image frequency lies below the wanted signal frequency.

If no C/I levels are specified, then the following assumptions are made, where n is 1 for 1 Mbps phys and 2 for 2 Mbps phys:

Test #	Phy	f_{rx}	f_{rc}±n	f_{rx}±2n	f_{rc}±n(3+m)	f_{image}	f_{iamge}±2n
RCV3	Uncoded, 1 Mbps	+21 dB	+15 dB	-17 dB	-27 dB	-9 dB	-15 dB
RCV9	Uncoded, 2 Mbps	+21 dB	+15 dB	-17 dB	-27 dB	-9 dB	-15 dB
RCV15	Stable, uncoded, 1 Mbps	+21 dB	+15 dB	-17 dB	-27 dB	-9 dB	-15 dB
RCV21	Stable, uncoded, 2 Mbps	+21 dB	+15 dB	-17 dB	-27 dB	-9 dB	-15 dB
RCV28	LE coded, S=2	+17 dB	+11 dB	-21 dB	-31 dB	-13 dB	-19 dB
RCV29	LE coded, S=8	+12 dB	+6 dB	-26 dB	-36 dB	-18 dB	-24 dB
RCV34	Stable, LE coded, S=2	+17 dB	+11 dB	-21 dB	-31 dB	-13 dB	-19 dB
RCV35	Stable, LE coded, S=8	+12 dB	+6 dB	-26 dB	-36 dB	-18 dB	-24 dB

f_{rx} is the frequency of the wanted signal in MHz and f_{image} is the frequency of the receiver image in MHz.

search_range is a numeric field containing the range of interferer levels to test over in units of dB. The nominal interferer level is set by the wanted signal level minus the C/I level. However, the phy lvl tester can further vary the interferer level by addition of the values specified in the search range. This provides a means of determining the ultimate C/I performance of a device for a given wanted signal level. If no search range is specified, then a default value of 0 dB is assumed.

ci_levels_hi is identical to *ci_levels_lo* except that the values are specified on the assumption that the receiver utilities high-side mixing, i.e. the image frequency lies above the wanted signal frequency.

7.4.5 Test body for receiver blocking tests

Receiver blocking tests have a test body in the form:

'*payload_lengths,rf_channels,num_packets,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,blocker_frequencies,blocker_levels,search_range*'

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then a default channel of 12 will be assumed.

num_packets is a numeric field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default value of 1500 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then a default value of -67 dBm will be assumed.

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified then a value of 0 kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, then a value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a value of 1250 Hz is assumed.

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift rate is used for all transmitted packets

Blocker_frequencies is a numeric field containing the blocker frequencies which will be tested in units of MHz.

If no interferer offsets are specified, then the following blocker frequencies are assume:

1. 30 : 10 : 2000 MHz
2. 2003 : 3 : 2399 MHz
3. 2484 : 3 : 2997 MHz
4. 3000 : 25 : 6000 MHz

blocker_levels is a numeric array containing the absolute level of the blocker in units of dB.

The number of entries in the list of blocker levels must be equal to either:

1. the number of entries of in the blocker frequency list, 624 if no blocker frequencies were specified,
2. one, in which case the same blocker level will be used for all blocker frequencies.

If no blocker levels are specified, then the following assumptions are made:

< 2000 MHz	2000-2399 MHz	2484-2999 MHz	>= 3000 MHz
-30 dBm	-35 dBm	-35 dBm	-30 dBm

search_range is a numeric field containing the range of blocker levels to test over in units of dB. The nominal blocker level is set by the blocker level field, however, the phy lvl tester can further vary the blocker level by addition of the values specified in the search range. This provides a means of determining the ultimate blocking performance of a device for a given wanted signal level. If no search range is specified, then a default value of 0 dB is assumed.

7.4.6 Test body for receiver intermodulation tests

Receiver intermodulation tests have a test body in the form:

'*payload_lengths,rf_channels,num_packets,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,intermodulation_N,interferer_levels*'

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the following default channels will be assumed

Test #	Test description	Phy	Central/Peripheral	Observer
RCV5	Intermodulation	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV11	Intermodulation	Uncoded, 2 Mbps	0,19,39	0,12,39
RCV17	Intermodulation	Stable, uncoded, 1 Mbps	0,19,39	0,12,39
RCV23	Intermodulation	Stable, uncoded, 2 Mbps	0,19,39	0,12,39

num_packets is a numeric field specifying the number of packets over which the test will be conducted. If this field is not specified, then the default value of 1500 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then a default value of -64dBm will be assumed.

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified then a value of 0kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, then a value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign

of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a value of 1250 Hz is assumed.

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift rate is used for all transmitted packets

Intermodulation_N is a numeric field containing the separation of the two interfering signals. This is in units of 1 MHz of 1 Mbps phys and units of 2 MHz for 2 Mbs phys. If not separation of the two interfering signals is specified, then a default value of 3 is assumed.

interferer_levels is a numeric array containing the absolute levels of the two interferer signals in units of dBm. The two interferer signals will always have the same amplitude. If not interferer levels are specified then a default value of -50 dBm is assumed.

For a 1 Mbps phy, the interferers are a CW signal plus a continuous 1 Mbps signal modulated by PRBS15 data. For a 2 Mbps phy, the interferers are a CW signal plus a continuous 2 Mbps signal modulated by PRBS15 data. The CW interferer is always placed closer to the wanted signal than the modulated interferer.

7.4.7 Test body for receiver PER integrity tests

Receiver PER integrity tests have a test body in the form:

‘*payload_lengths,rf_channels,num_packets,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,repeat_count*’

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels

which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the following default channels will be assumed:

Test #	Test description	Phy	Central/Peripheral	Observer
RCV7	PER report integrity	Uncoded, 1 Mbps	19	12
RCV13	PER report integrity	Uncoded, 2 Mbps	19	12
RCV19	PER report integrity	Stable, uncoded, 1 Mbps	19	12
RCV25	PER report integrity	Stable, uncoded, 2 Mbps	19	12
RCV30	PER report integrity	LE coded, S=2	19	12
RCV31	PER report integrity	LE coded, S=8	19	12
RCV36	PER report integrity	Stable, LE coded, S=2	19	12
RCV37	PER report integrity	Stable, LE coded, S=8	19	12

num_packets is a numeric field specifying the number of packets over which the test will be conducted. A value of 0 in this field signifies that the phy level tester should random select an even number of packets in the range 100 to 1500 inclusive. If no number of packets is specified, then a default value of 0 is assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then a default value of -30 dBm will be assumed.

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified then a value of 0kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, then a value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a value of 1250 Hz is assumed.

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified (or one if no carrier frequency offsets were specified)
2. One, in which case the specified drift rate is used for all transmitted packets

repeat_count is a single integer denoting how many times the PER integrity test should be performed. If no repeat count is specified, then a default value of 3 is assumed.

7.4.8 Test body for receiver IQ sample coherency tests

All receiver IQ sample coherency tests have the same test body format:

'*payload_lengths,rf_channels,num_iq,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,CTE_len,num_antenna,antenna_IDs,antenna_phis*'

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels shown below will be assumed.

TBD HERE

Test #	**Test description**	**Phy**	**Central Peripheral**	**Observer**
RCV/IQC1	IQ sample coherency, AoD, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC2	IQ sample coherency, AoD, 1µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC3	IQ sample coherency, AoD, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC4	IQ sample coherency, AoD, 1µs slots	Uncoded, 2 Mbps	0,19,39	0,12,39
RCV/IQC5	IQ sample coherency, AoA, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC6	IQ sample coherency, AoA, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39

num_iq is a numeric field specifying the number of IQ samples which the test will be conducted. If this field is not specified, then the default value of 10000 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then a value of -70dBm will be assumed.

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified a single value of 0 kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a single value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, a single value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a single value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a single value of 0 Hz is assumed

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified drift rate is used for all transmitted packets

CTE_len is a numeric integer field specifying the number of 8µs slots in the CTE. If this field is not specified, then it will default to 20.

num_antenna is a numeric integer field specifying the number of antenna. If this field is not specified, then it will default to 2.

antenna_IDs is a numeric field specifying the antenna IDs to be used. If this field is not specified, then it will default to 0:*num_antenna*-1. Only the first antenna ID is used for AoD tests.

antenna_phis is a numeric field specifying the phase in degrees which the *TLF3000* should transmit on to simulate each antenna when performing AoD measurements. If this parameter is not specified, then all phases default to 0. This field is not used for AoA measurements.

7.4.9 Test body for receiver IQ sample dynamic range tests

All receiver IQ dynamic rante tests have the same test body format:

'*payload_lengths,rf_channels,num_iq,wanted_pwrs,carrier_offsets,mod_indices,symbol_timings,drift_magnitudes,drift_rates,CTE_len,num_antenna,antenna_IDs,antenna_phis,antenna_amps*'

payload_lengths is a numeric field specifying the range of payload lengths for which the test will be conducted. Any payload lengths which exceed the maximum supported payload length of the DUT will be silently ignored. If this field is absent, then the maximum supported payload length for the DUT will be used.

The maximum supported payload length for the DUT may be:

1. Specified using the *Set DUT Type* command
2. Determine by interrogating the DUT if the *interrogate DUT* flag is set in the *Run Test Script* command.

rf_channels is a numeric field specifying the range of RF channels for which the test will be conducted. If the DUT is an observer device which does not support advertising extensions then any RF channels which are not 0, 12 or 39 will be silently ignored. If this field is not specified, then the default channels shown below will be assumed.

TBD HERE

Test #	Test description	Phy	Central Peripheral	Observer
RCV/IQC1	IQ sample coherency, AoD, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC2	IQ sample coherency, AoD, 1µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC3	IQ sample coherency, AoD, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC4	IQ sample coherency, AoD, 1µs slots	Uncoded, 2 Mbps	0,19,39	0,12,39
RCV/IQC5	IQ sample coherency, AoA, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39
RCV/IQC6	IQ sample coherency, AoA, 2µs slots	Uncoded, 1 Mbps	0,19,39	0,12,39

num_iq is a numeric field specifying the number of IQ samples which the test will be conducted. If this field is not specified, then the default value of 10000 will be assumed.

wanted_pwrs is a numeric field specifying the wanted signal level powers over which the test will be conducted in units of dBm. If this field is not specified, then a value of -70dBm will be assumed.

carrier_offsets is a numeric field specifying the carrier frequency offsets to be applied in units of kHz. The first carrier frequency offset is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of carrier frequency offsets is exhausted, the list is rewound and the first entry reused. If no carrier frequency offsets are specified a single value of 0 kHz is assumed.

mod_indices is a numeric array containing the modulation indices to be used. The first modulation index is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of modulation indices offsets is exhausted, the list is rewound and the first entry reused.

If no modulation indices are specified, then a single value of 0.5 is assumed.

The number of modulation indices must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified modulation index is used for all transmitted packets

symbol_timings is a numeric array containing the symbol timing errors to be used in units of ppm. The first symbol timing error is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of symbol timing errors is exhausted, the list is rewound and the first entry reused.

If no symbol timing errors are specified, a single value of 0 ppm is assumed.

The number of symbol timing errors must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified symbol timing error is used for all transmitted packets

drift_magnitudes is a numeric array containing the drift magnitudes to be used in units of kHz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign of the amplitude of the drift is inverted on alternate packets. The first drift magnitude is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift magnitudes is exhausted, the list is rewound and the first entry reused.

If no drift magnitudes are specified, then a single value of 0 kHz is assumed.

The number of drift magnitudes must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified drift magnitude is used for all transmitted packets

drift_rates is a numeric array containing the drift rates to be used in units of Hz. The drift is applied as a sinusoidal disturbance on the carrier frequency whose amplitude is specified by *drift_magnitudes* and whose frequency is determined by *drift_rates*. The drift is always zero at the start of the packet. The sign

of the amplitude of the drift is inverted on alternate packets. The first drift rate is applied to the first 50 transmitted packets, the second to the next 50 packets, etc. Once the list of drift rates is exhausted, the list is rewound and the first entry reused.

If no drift rates are specified, then a single value of 0 Hz is assumed

The number of drift rates must either be:

1. Equal to the number of carrier frequency offsets specified
2. One, in which case the specified drift rate is used for all transmitted packets

CTE_len is a numeric integer field specifying the number of 8µs slots in the CTE. If this field is not specified, then it will default to 20.

num_antenna is a numeric integer field specifying the number of antenna. If this field is not specified, then it will default to 2.

antenna_IDs is a numeric field specifying the antenna IDs to be used. If this field is not specified, then it will default to 0:*num_antenna*-1. Only the first antenna ID is used for AoD tests.

antenna_phis is a numeric field specifying the phase in degrees which the *TLF3000* should transmit on to simulate each antenna when performing AoD measurements. If this parameter is not specified, then all phases default to 0.

antenna_amps is a numeric field specifying the relative amplitude in dB which the *TLF3000* should transmit on to simulate each antenna when performing AoD measurements. If this parameter is not specified, then all relative amplitudes default to {3,-5,-10,0,0 …. 0}.

8 Connectors.

8.1 Front Panel

8.1.1 Monitor In Port

Function	High sensitivity 2.4 GHz RF input Suitable for radiated measurements
Connector type	SMA
Noise figure	6 dB typ
IP3 @ max sensitivity	+7 dBm typ
SNR in 1MHz bandwidth	80 dB typ
Maximum input signal	27 dBm
Maximum usable signal	-10 dBm typ
Frequency range	2401-2481 MHz
Impedance	50 Ω
Coupling	AC
Maximum DC voltage	50 V

8.1.2 Tx/Rx port

Function	Low sensitivity 2.4GHz RF input and signal generator output Suitable for conducted measurements
Connector type	SMA
Impedance	50 Ω
Coupling	AC
Maximum DC voltage	50 V

8.1.3 Receiver Specification

Noise figure	46 dB typ
IP3 @ max sensitivity	+47 dBm typ
SNR in 1MHz bandwidth	80 dB typ
Maximum input signal	27 dBm
Maximum usable signal	27 dBm typ
Frequency range	2401-2481 MHz

8.1.4 Video port

Function	Not used in Sapphire application
Connector type	SMA
Impedance	50 Ω
Coupling	AC
Maximum DC voltage	5 V

8.1.5 External Clock Input

Function	External clock input Permits internal RF, sampling and timestamp clocks to be locked to an external reference
Connector type	SMA
Maximum input signal	-10 dBm
Minimum input signal	+20 dBm
Frequency range	10 MHz
Impedance	50 Ω
Coupling	AC
Maximum DC voltage	50 V

8.1.6 Reference Clock Output

Function	Reference clock output Provides a reference clock which can be used to synchronise other test equipment
Connector type	SMA
Output signal	-2 dBm
Frequency	10 MHz
Impedance	50 Ω
Coupling	AC
Maximum DC voltage	5 V

8.2

8.3 Rear Panel

8.3.1 Digital IO

Function	Digital input and output
Connector type	Hirose LX60-20S

Logic levels when IO voltage is internal 3.3V	
Logic input high	2.0 V (min)
Logic input low	0.8 V (max)
Logic output high	2.4 V (min)
Logic output low	0.55 V (max)
Output current	±24 mA

Pin	**Name**	**Direction**	**Special function**
1	Vio	I/O	Max 500 mA from internal 3.3 V supply. External voltage range 0.8 V to 3.6 V.
2	Vio	I/O	
3	GPI #0	I	
4	GPI #1	I	
5	GPI #2	I	
6	GPI #3	I	
7	GPI #4	I	
8	GPI #5	I	
9	GPI #6	I	UART RTS
10	GPI #7	I	UART Rx
11	GPO #0	O	
12	GPO#1	O	
13	GPO#2	O	
14	GPO#3	O	
15	GPO#4	O	
16	GPO#5	O	
17	GPO#6	O	UART CTS
18	GPO#7	O	UART Tx
19	Gnd	-	
20	Gnd	-	

8.3.2 USB Connector

Function	USB connection to host
Connector type	Micro-USB
Speed rating	High speed
VBUS load	2.2 µF, > 10 kΩ

8.3.3 Ethernet Connector

Function	Ethernet connection to host
Connector type	RJ45
Speed	10 / 100 / 1000

8.3.4 Power Connector~

Function	DC power input
Connector type	2.5 mm jack
Input voltage	12 V DC
Power	10 W typ (application dependent)
Reverse polarity protection	Yes
Over voltage protection	Yes
Under voltage protection	Yes

8.4 Connections for Sapphire Operating Modes

Function	**Phy Test**	**Signal Generator**	**Signal Analyser**	**Advertise/Scan**
RF Tx to DUT	Tx/Rx	Tx/Rx	-	Tx/Rx
RF Rx from DUT	Tx/Rx or Monitor In	-	Tx/Rx or Monitor In	Tx/Rx or Monitor In
UART Rx	GPI 7	-	-	-
UART Tx	GPO 7	-	-	-
UART RTS	GPI 6	-	-	-
UART CTS	GPO 6	-	-	-
IO voltage	Ext Vio or 3v3 Int	Ext Vio or 3v3 Int	Ext Vio or 3v3 Int	Ext Vio or 3v3 Int

9

10 Uncertainty

Quantity	Uncertainty (typical)
Absolute RF power (in-band)	±1.2 dB
Relative RF power (in-band)	±1 dB
Relative RF power (out-of-band)	±3 dB
Absolute frequency (internal reference)	±5 kHz ±2.5 kHz/year
Absolute frequency (external reference)	±1 kHz
Relative frequency	±500 Hz

11

12 Indicators

12.1 Front Panel

12.1.1 Status

The right hand LED indicates the status of the unit. When power is applied this will light YELLOW indicating that the internal power supplies are good. Once the unit has booted and is ready for host commands it will turn GREEN.

12.1.2 RF Overload

The left hand LED indicates whether an RF overload is present. The RF overload may be either on a receiver input or the signal generator output. The *Input/Output Power* messages can provide further details on the source of the overload.

The both the Monitor In port and the Tx/Rx port are protected for input signals up to a maximum level of 27 dB. Input signals beyond this will destroy the unit. If a receive overload condition is present, the results of the unit are uncertain. If the Monitor In port is in use, then swapping to the less sensitive Tx/Rx port may alleviate the RF overload condition.

12.1.3 Flash Update

When the Flash is being updated, the Status LED flash will flash between YELLOW and RED. Do not remove power from the unit whilst the Flash is being updated.