TEST EQUIPMENT PLUS

Signal Hound API , version 2012_01_05 thru 2012_09_17

Signal Hound Application Programming Interface

TEST EQUIPMENT PLUS

Signal Hound USBSA-44 Application Programming Interface (API)

Table of Contents

Chapter

1

Introduction

About the Signal Hound API and building applications.

The Signal Hound Application Programming Interface is a tool for software engineers to design custom applications for the Signal Hound. Like the Signal Hound Graphical User Interface (GUI), the API is used to send commands to, and receive data from, the Signal Hound device. But unlike the Signal Hound GUI, you have the flexibility as a programmer to control the Signal Hound at a lower level, and process, log or store data in any format you choose.

DISCLAIMER—This API is provided free of charge, without warranty or support. Software developers may only use this API with a genuine Signal Hound™.

A simple application will send a series of commands to the API. The **first command to the API must be the initialize command**. This takes twenty seconds to execute because it must download a calibration table from the Signal Hound device. While the GUI can store this as a local file and only load it once, the API has no such luxury.

The **second command to the API must be a configuration command**. This selects the attenuator settings, mixer, intermediate frequency, and clock settings.

After these first two commands, the Signal Hound is ready to begin collecting data. A set of functions are available to you, each tailored to get the most out of a particular aspect of the Signal Hound.

A sample application is available to you. A drop-down menu allows you to initialize, configure, and collect data from the Signal Hound through the API. It is written in Visual C++ 6, but may be readily ported to any number of languages.

The API consists of SH_API.dll, SH_API.h, SH_API.lib, and this document. To use, put the DLL you're your application's working directory, insert the header file into your application's source code, include the library file in your project's settings, then build your application. Typically the DLL is installed with your application. The library and header files are used to build your application. If you are familiar with using DLLs this should be a straightforward process. If not, please review your compiler's documentation before proceeding.

Chapter

2

Function Listing

List of Common Functions for the Signal Hound API

Pre-Initialization (For advanced users)

Function: int SHAPI_GetSerialNumbers(unsigned int * pSerNumArray)

Arguments: pSerNumArray points to an array of 8 unsigned integers the user passes to the API

Execution Time: < 1 second

Return values: Count of available serial numbers

Remarks: Call this to determine what Signal Hounds are connected. The serial number may be used, if necessary, to identify model.

Function: void SHAPI_DisableDataPipes()

Remarks: Call this to disable the "realtime" data pipe, for applications which must run without administrator privileges.

Initialization

Functions: int SHAPI_Initialize()
int SHAPI_InitializeNext()

Arguments: deviceNum, a number from 0 to 7, where 0 is the first device to initialize, 1 is the second, and so on.

Execution Time: 4-20 seconds approx
Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
You MUST call this function before any others and wait for it to complete!
SHAPI_Initialize initializes a single Signal Hound USB Interface.
SHAPI_InitializeNext initializes the next Signal Hound for a multiple Signal Hound application. A maximum of 8 Signal Hounds may be initialized.

Advanced users: Devices must be initialized in a single thread. After initialization of all devices is complete, separate threads may be used for each device.

The above functions also loads correction constants from the SA44(B) flash memory into the DLL.

USB-SA124A Additional Initialization

Function: int SHAPI_GetSA124CalData(LPCSTR filename,int deviceNum=0) // (SA 124 only)

Arguments: filename is D[serial number].bin, and is usually located in c:\Program Files\Signal Hound
It is also available over the internet. Call for details.

Remarks:
This loads correction constants for SA124., and is REQUIRED after calling initialization. If you do not know if you are using an SA124, use *int SHAPI_IsSA124(int deviceNum=0).*

Fast Initialization (USB-SA44 and USB-SA44B only)

Functions: int SHAPI_CopyCalTable(unsigned char * p4KTable, int deviceNum=0)
Functions: int SHAPI_InitializeEx(unsigned char * p4KTable, int deviceNum=0)

Arguments:	p4KTable:	Pointer to a buffer of 4096 bytes for the cal table
	deviceNum:	A number from 0 to 7, where 0 is the first device to initialize

Execution Time: 0.2 seconds approx
Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
SHAPI_CopyCalTable fills the 4 KB buffer with calibration data retrieved during SHAPI_Initialize. SHAPI_InitializeEx initializes a Signal Hound with calibration data you supply, saving the time it takes to retrieve this data.

The FIRST time your software encounters a new Signal Hound, call SHAPI_Initialize followed by SHAPI_CopyCalTable. Store this 4 KB table and pass it as an argument the NEXT time your software needs to initialize the Signal Hound (the next time your application is launched) for faster initialization.

Initialization (SA124A)

int SHAPI_GetSA124CalData(LPCSTR filename,int deviceNum=0) // (SA 124 only)
Loads correction constants for SA124. REQUIRED.

Configuration

Function: int **SHAPI_Configure**(double attenVal=10.0, int mixerBand=1, int sensitivity=0, int decimation=1, int IF_Path=0, int ADC_clock=0 , int deviceNum=0)

Function: int **SHAPI_ConfigureFast**(double attenVal=10.0, int mixerBand=1, int sensitivity=0, int decimation=1, int IF_Path=0, int ADC_clock=0 , int deviceNum=0)

Arguments:
attenVal—Attenuator setting. Must be **0.0, 5.0, 10.0, or 15.0** dB. 10 dB is default.

mixerBand—For RF input frequencies below 150 MHz this should always be set to **0**. For RF frequencies above 150 MHz this should always be set to **1**.

Sensitivity—For lowest sensitivity, set to **0**. For highest sensitivity set to **2**.

Decimation—Sample rate is equal to 486.1111 Ksps divided by this number. Must be **between 1 and 16, inclusive**. Part of resolution bandwidth (RBW) calculation.

IF Path—Set to **0** for default 10.7 MHz Intermediate Frequency (IF) path. This path has higher selectivity but lower sensitivity. Set to **1** for 2.9 MHz IF path.

ADC clock—Set to 0 to select the default 23 1/3 MHz ADC clock. Set to 1 to select the for 22 ½ MHz ADC clock, which is useful if your frequency is a multiple of 23 1/3 MHz.

DeviceNum: Specify 0-7 if multiple Signal Hounds used.

Execution Time: 400 msec or less.
Note: **SHAPI_ConfigureFast** skips any settings that have not changed since the last Configure call, and is much faster.

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
This function configures the Signal Hound and prepares it to receive an RF signal.
This function must be called before data is captured.

Slow Sweep

Function: int **SHAPI_GetSlowSweep**(double * dBArray, double startFreq, double stopFreq, int &returnCount, int FFTSize=1024, int avgCount=16, int imageHandling=0 ,int deviceNum=0)

Arguments:
dBArray —Pointer to array of double precision floating point numbers. This is where your data will get stored.

StartFreq—Frequency of first amplitude value returned

StopFreq—Minimum frequency of last amplitude value returned. Due to rounding, several additional values may be returned

returnCount —Count of amplitude values returned.

FFTSize —Size of FFT. This and the decimation setting are used to calculate RBW. May be 16-65536 in powers of 2.

avgCount —Number of FFTs that get averaged together to produce the output. The amount of data captured at each frequency is a product of FFTSize and avgCount. *This product must be a multiple of 512*.

imageHandling —Set to 0 for default, IMAGE REJECTION ON (mask together high side and low side injection). Set to 1 for HIGH SIDE INJECTION. Set to 2 for LOW SIDE INJECTION.

DeviceNum: Specify 0-7 if multiple Signal Hounds used.

Execution Time: [40 + (FFTSize * avgCount * decimation) / 486] msec per slice. The number of slices is equal to decimation * (stop – start) / 201KHz, rounded up.

Return values:
0 for success, otherwise returns error code (see Appendix A)

Remarks:
This function captures an array of data. Data points are amplitude, in dBm. The first data point is equal to the starting frequency. Subsequent data points are spaced by 486.1111 KHz / FFT size / decimation. You may call SHAPI_GetSlowSweepCount to get the size of this array.

External Trigger: When the external trigger is enabled, this function is blocking until a logic high is received. Call this function with image handling set to 1, otherwise you will need a second trigger pulse for the image rejection sweep.

Fast Sweep

Function: int SHAPI_GetFastSweep(double * dBArray, double startFreq, double stopFreq, int &returnCount, int FFTSize=16, int imageHandling=0, int DeviceNum=0)

Arguments:
dBArray —Pointer to array of double precision floating point numbers. This is where your data will get stored.

StartFreq—Frequency of first amplitude value returned. This value is rounded to the nearest 200 KHz.

StopFreq—Frequency of last amplitude value returned. This value is rounded to the nearest 200 KHz.

returnCount —Count of amplitude values returned.

FFTSize —Size of FFT. This is used to calculate RBW. May be 1 or 16-256, in powers of 2.

imageHandling —Set to 0 for default, IMAGE REJECTION ON (mask together high side and low side injection). Set to 1 for HIGH SIDE INJECTION. Set to 2 for LOW SIDE INJECTION.

DeviceNum: Specify 0-7 if multiple Signal Hounds used.

Execution Time: [40 + 1.2 * slice count] msec for large sweeps, up to twice this for small sweeps. The number of slices is equal to (stop – start) / 200KHz, rounded up.

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
This function captures an array of data. Data points are amplitude, in dBm. The first data point is equal to the starting frequency. For FFT size of 1 (raw power only), data points are spaced 200 KHz.
Otherwise data points are spaced 400 KHz / FFT Size.
RBW is based on FFT size only, as decimation is equal to 1.

Prior to calling Fast Sweep, configure as follows:
--DECIMATION MUST BE SET TO 1.
--IF PATH MUST BE SET TO 0.
--ADC CLOCK MUST BE SET TO 0.

Calculate RBW

Function: double SHAPI_GetRBW(int FFTSize, int decimation)

Arguments:

FFTSize —Size of FFT. 8-65536, in powers of 2.

Decimation—Signal Hound Decimation setting

Return values:
RBW in Hz. Equal to 1.6384e6 / decimation / FFTSize;

Remarks:
Returns an approximation of the RBW. For an FFT size of 1024 and decimation of 16, an RBW of 100 Hz is returned.

Get Sweep Count

Functions:
int SHAPI_GetSlowSweepCount(double startFreq, double stopFreq, int FFTSize ,int deviceNum=0);
int SHAPI_GetFastSweepCount(double startFreq, double stopFreq, int FFTSize);

Arguments:

Start & Stop Frequencies, in Hz.
FFTSize —Size of FFT. 16-65536, in powers of 2.
DeviceNum: Specify 0-7 if multiple Signal Hounds used.

Return values:
Sample count. May be used to allocate memory.

Remarks:
Returns the count of double precision floating point values to expect from GetSlowSweep or GetFastSweep.

Cycle the Device (Preset)

Function:
int SHAPI_CyclePort();

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
Takes about 2.5 seconds. MUST INITIALIZE FIRST. Presets the Signal Hound hardware. Useful to restore Signal Hound to a known state.

For multi-device applications, only call this before initializing Signal Hounds, or when preparing to exit the application. Otherwise, behavior is unknown.

Function:
SHAPI_CyclePowerOnExit()

Power cycles all Signal Hounds. Use prior to closing your software to restore the Signal Hound's state to a known condition.

Select External Reference (USB-SA44 and USB-SA44B only)

Function:
int SHAPI_SelectExt10MHz(int deviceNum=0)

deviceNum: Specify 0-7 if multiple Signal Hounds used.

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
Takes about 50 msec. Checks for >0 dBm 10 MHz reference. If present, the external 10 MHz is selected.

Select External Trigger or Sync Out

Function:
void `SHAPI_SyncTriggerMode` (int mode , int deviceNum=0)

Remarks:
Mode may be set to:

SHAPI_EXTERNALTRIGGER	triggers on an external logic high
SHAPI_SYNCOUT	pulses high when data collection begins
SHAPI_TRIGGERNORMAL	triggers immediately

When using an external trigger (3.3V or 5V OK), some functions (slow sweep, measurement receiver) will wait for a logic high before beginning data collection. There is no timeout, so use the external trigger with caution as it will halt operations until a TTL high is received.

Get I / Q Data Packet

Function:
int SHAPI_GetIQDataPacket (int * pIData, int * pQData, double &centerFreq, int size , int deviceNum=0)

Arguments:

pIData—Pointer to the buffer where In-phase channel data will be stored. Values –32768 to +32767. Raw DAC output.

pQData—Pointer to the buffer where Quadrature-phase channel data will be stored. Values –32768 to +32767. Raw DAC output.

CenterFreq—Passed by reference. The API will modify this to the nearest available actual center frequency. You may correct the I/Q data to a specific frequency and phase alignment by applying a linear phase offset.

size —Number of I/Q data pairs to store. Must be multiple of 512, up to 128,000.

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
Changes to selected center frequency, high-side LO injection, no image rejection. Reports SIZE data points at current decimation / clock rates.

For multi-device applications, the currently selected device is used.

External Trigger: When the external trigger is enabled, this function is blocking until a logic high is received.

Operating Over Full Temperature Range

Functions:
float SHAPI_GetTemperature(int deviceNum=0)
int SHAPI_LoadTemperatureCorrections(LPCSTR filename , int deviceNum=0) // (SA44 only)
int SHAPI_GetSA124CalData(LPCSTR filename,int deviceNum=0) // (SA 124 only)
Arguments:

filename —Pointer to the string with the temperature correction data, typically formatted as "D01234567.bin".

Return values:
SHAPI_LoadTemperatureCorrections: "true" for success, "false" for failure.
SHAPI_GetTemperature: internal temperature in °C (32-bit floating point value)

Remarks:

USB-SA44B only!!!

Call LoadTemperatureCorrections to maintain amplitude accuracy when operating at cold or hot temperatures.
Call SHAPI_GetTemperature to read the current temperature and use it for amplitude corrections.

Using the RF Preamplifier (USB-SA44B)

Function:
void SHAPI_SetPreamp(int value)

Arguments:
value = 0 for preamplifier **off**, 1 for preamplifier **on**

Remarks:

USB-SA44B only!!!

For multi-device applications, the currently selected device is used. Turns on or turns off the RF preamplifier. The preamplifier can be used to improve the sensitivity and decrease LO feed-through for sensitive readings. Set the attenuator to ensure the preamplifier input sees less than -25 dBm of input power to avoid overdriving your mixer and distorting your signal. Turn off the preamplifier below 500 KHz.

Function:
int SHAPI_IsPreampAvailable()

Return value:
1 if a preamplifier is available, , e.g. a USB-SA44B
0 if a preamplifier is not available, e.g. a USB-SA44

Remarks:

Tests to see if a preamplifier is available, e.g. is this device a USB-SA44B?

Using Multiple Signal Hounds

Up to 8 Signal Hounds can operate independently through the API

Function:
All functions with "deviceNum" as a parameter

Arguments: Pass deviceNum to the API, to identify which Signal Hound you are communicating with. deviceNum is a number from 0 to 7, where 0 is the first device initialized, 1 is the second, and so on.

Labview / 32-bit Matlab Compatible Functions

Additional functions have been added to support Labview / 32-bit Matlab compatibility. For these functions, passing parameters by reference, the real-time data pipe, and default parameter values have been avoided.

For using a single device, always set your deviceNum to zero.
For documentation on the following functions, see previous sections, but remove the _LVC prefix.
SHAPI_LVC_GetSlowSweepCount may be broken at the moment. You may have to estimate size then use returned count from GetSlowSweep until the next API is released.

```
void SHAPI_LVC_LoadTemperatureCorrections(char * filename, int deviceNum);
void SHAPI_LVC_SetPreamp(int val, int deviceNum);
unsigned int SHAPI_LVC_GetSerNum(int deviceNum);
void SHAPI_LVC_Initialize();
int SHAPI_LVC_GetSA124CalData(char * filename,int deviceNum);
int SHAPI_LVC_Configure(double attenVal, int mixerBand, int sensitivity,
   int decimation, int useIF2_9, int ADCclock, int deviceNum);
int SHAPI_LVC_ConfigureFast(double attenVal, int mixerBand, int sensitivity,
   int decimation, int useIF2_9, int ADCclock, int deviceNum);
int SHAPI_LVC_GetSlowSweepCount(double startFreq, double stopFreq, int FFTSize,int deviceNum);
int SHAPI_LVC_GetSlowSweep(double * dBArray, double startFreq, double stopFreq,
        int * pReturnCount, int FFTSize, int avgCount, int imageHandling, int deviceNum);
int SHAPI_LVC_GetFastSweepCount(double startFreq, double stopFreq, int FFTSize);
int SHAPI_LVC_GetFastSweep(double * dBArray, double startFreq, double stopFreq,
        int * pReturnCount, int FFTSize, int imageHandling, int deviceNum);
int SHAPI_LVC_GetIQDataPacket(int * pI, int * pQ, double * pCenterFreq, int size, int deviceNum);
int SHAPI_LVC_SetupLO(double * pCenterFreq, int mixMode, int deviceNum);
int SHAPI_LVC_StartStreamingData(int deviceNum);
int SHAPI_LVC_StopStreamingData(int deviceNum);
int SHAPI_LVC_GetStreamingPacket(int *bufI, int *bufQ, int deviceNum);
int  SHAPI_LVC_ProcTGSweep(double * dBArray, double startFreq, double stepSize, int count, int
attenval);

int  SHAPI_LVC_SetTGFreqAtten(double freq, unsigned char atten);
```

Function:

int SHAPI_StartStreamingData(int deviceNum=0)
int SHAPI_StopStreamingData(int deviceNum=0)

Arguments: deviceNum. A negative value uses the currently selected device. Otherwise, pass the device number 0-7 you wish to start or stop.

Remarks:
Starts or stops the streaming of I/Q data at the selected frequency. Once you start streaming data, you must stop it before using any functions **except** SHAPI_GetStreamingPacket.

Function:
int SHAPI_GetStreamingPacket(int *bufI, int *bufQ, int deviceNum=0)

Arguments: deviceNum. A negative value uses the currently selected device. Otherwise, pass the device number 0-7 you wish to get data from.
bufI, bufQ: 32-bit integer buffers of size 4096 samples, to receive the unprocessed I/Q data. Values will be -32768 to 32767.

Remarks:
Call this function to get the next chunk of 4096 samples from the receive buffer. Must be called in a timely fashion or data will be lost. Returns when data is received. The decimation rate in SHAPI_Configure controls the sample rate, and can be used to reduce the IF bandwidth and amount of data received.

The sequence for receiving streaming data should be:

1. Initialize
2. Configure
3. Setup LO
4. Start Streaming
5. Repeatedly Get Streaming Packet
6. Stop Streaming

Using the Measurement Receiver

```
MEAS_RCVR_STRUCT:
       // *** INPUTS ***
       double  RFFrequency;                 //RF carrier frequency (Hz)
       double  AudioLPFreq;                 //Audio LowPass Cutoff  (Hz)
       double  AudioBPFreq;                 //Audio BandPass Center (Hz)
       int     UseLPF;                      //Set to non-zero to use audio low-pass filter
       int     UseBPF;                      //Set to non-zero to use audio low-pass filter

       // *** OUTPUTS ***
       double  RFCounter;                   //RF frequency count out (Hz)
       double  AMAudioFreq;                 //AF frequency count out after AM demod (Hz)
       double  FMAudioFreq;                 //AF frequency count out after FM demod (Hz)
       double  RFAmplitude;                 //dB Full Scale.

       double  FMPeakPlus;                  //Peak Positive Modulation, in Hz
       double  FMPeakMinus;                 //Peak Negative Modulation, in Hz
       double  FMRMS;                       //RMS  Modulation, in Hz

       double  AMPeakPlus;                  // In percent
       double  AMPeakMinus;
       double  AMRMS;
```

Function:
int SHAPI_RunMeasurementReceiver (void * pMeasRcvrStruct, int deviceNum=0)

Arguments:
pMeasRcvrStruct —Pointer to the measurement receiver structure, with the RF frequency and filter settings previously set.

Return values:
Ignore the int return value. Your `MEAS_RCVR_STRUCT will be fully populated upon return.`

Remarks:
To use: You must call **SHAPI_Initialize** followed by **SHAPI_Configure** before you call **SHAPI_RunMeasurementReceiver**. It is strongly recommended that you use the 2.9 MHz IF in your **SHAPI_Configure** call. The incidental AM for the 2.9 MHz IF is much lower than the 10.7 MHz, and it is more sensitive.

Keep your **`RFAmplitude`** readings between -45 and -5 dB Full Scale (dBFS) for best accuracy. As you approach 0 dBFS, readings may become inaccurate. Above 0 dBFS readings are meaningless as you are overdriving the ADC.

You may change **sensitivity** and **attenuator** settings **SHAPI_Configure** to change ranges, increasing dynamic range. The practice of taking a reading immediately before changing range, then immediately after changing to calculate an offset works well, and is required for a large dynamic range.

The IF Bandwidth is controlled by the decimation setting in your **SHAPI_Configure** call.
IF Bandwidth = 240 KHz / decmation. Decimations of 1,2,4,8, or 16 are recommended. 64K samples are taken regardless of IF bandwidth, so with decimation set to 16 the function will take about 2 seconds to return.

External Trigger: When the external trigger is enabled, this function is blocking until a logic high is received.

Appendix

A

Error Codes

ERROR_HOUND_NOT_FOUND	100
ERROR_PACKET_HEADER_NOT_FOUND	101
ERROR_WRITE_FAILED	102
ERROR_WRONG_NUM_READ	103
ERROR_READ_TIMEOUT	104
ERROR_DEVICE_NOT_LOADED	105
ERROR_MISSING_DATA	106
ERROR_EXTRA_DATA	107
ERROR_OUT_OF_RANGE	200
ERROR_NO_EXT_REF	201

Appendix B

Complete Function Listing

```
extern "C" __declspec( dllimport ) int SHAPI_GetSerialNumbers(unsigned int * pSerNumArray); //Returns -1 on error
extern "C" __declspec( dllimport ) void SHAPI_DisableDataPipes();

extern "C" __declspec( dllimport ) float SHAPI_GetTemperature(int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_LoadTemperatureCorrections(LPCSTR filename, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SetPreamp(int val, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_IsPreampAvailable(int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetFMFFTPK(int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetAudioFFTSample(int idx, int deviceNum=0);
extern "C" __declspec( dllimport ) unsigned int SHAPI_GetSerNum(int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SyncTriggerMode(int mode, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_CyclePowerOnExit();
extern "C" __declspec( dllimport ) double SHAPI_GetAMFFTPK(int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_ActivateAudioFFT(int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_DeactivateAudioFFT(int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetRBW(int FFTSize, int decimation);
extern "C" __declspec( dllimport ) double SHAPI_GetLastChannelPower(int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetLastChannelFreq(int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SetOscRatio(double ratio,int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_InitializeNext(); //Returns -1 on error
extern "C" __declspec( dllimport ) int SHAPI_GetSA124CalData(LPCSTR filename,int deviceNum=0); //Returns -1 on
error
extern "C" __declspec( dllimport ) int SHAPI_IsSA124(int deviceNum=0); //Returns -1 on error
extern "C" __declspec( dllimport ) int SHAPI_Initialize();
extern "C" __declspec( dllimport ) void SHAPI_WriteCalTable(unsigned char *myTable, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_CopyCalTable(unsigned char * p4KTable, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_InitializeEx(unsigned char * p4KTable, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_SetAttenuator(double attenVal, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_SelectExt10MHz(int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_Configure(double attenVal=10.0, int mixerBand=1, int sensitivity=0,

                    int decimation=1, int useIF2_9=0, int ADCclock=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_ConfigureFast(double attenVal=10.0, int mixerBand=1, int sensitivity=0,

                    int decimation=1, int useIF2_9=0, int ADCclock=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetSlowSweepCount(double startFreq, double stopFreq, int FFTSize,int
deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetSlowSweep(double * dBArray, double startFreq, double stopFreq,

                              int &returnCount, int FFTSize=1024,

                              int avgCount=16, int imageHandling=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetFastSweepCount(double startFreq, double stopFreq, int FFTSize);
extern "C" __declspec( dllimport ) int SHAPI_CyclePort();

extern "C" __declspec( dllimport ) int SHAPI_GetFastSweep(double * dBArray, double startFreq, double stopFreq,

                              int &returnCount, int FFTSize=16, int imageHandling=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetIQDataPacket(int * pIData, int * pQData, double &centerFreq, int size,
int deviceNum=0);
```

```
extern "C" __declspec( dllimport ) int SHAPI_Authenticate(int vendorcode=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_SetupLO(double &centerFreq, int mixMode=1, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_StartStreamingData(int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_StopStreamingData(int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetStreamingPacket(int *bufI, int *bufQ, int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetPhaseStep(int deviceNum=0);
extern "C" __declspec( dllimport ) double SHAPI_GetChannelPower(double cf, int *iBigI, int *iBigQ, int count, int
deviceNum);
extern "C" __declspec( dllimport ) int SHAPI_GetIntFFT(int FFTSize, int *iBigI, int *iBigQ, double * dFFTOut, int
deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_SetupFastSweepLoop(double startFreq, double stopFreq, int &returnCount,
int MaxFFTSize=16, int imageHandling=0, int AvgCount=1, int deviceNum =0);
extern "C" __declspec( dllimport ) int SHAPI_SetupMultiFreqSweepLoop(int count, double * freqbuf, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetMultiFreqIQ(int count, int size, int * pI1, int * pQ1, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_SetupMultiBandSweepLoop(int count, double * freqbuf, int * freqsize, int
deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetMultiBandIQ(int count, int size, int * pI1, int * pQ1, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_GetFSLoopIQSize(double startFreq, double stopFreq, int MaxFFTSize=16);
extern "C" __declspec( dllimport ) int SHAPI_GetFSLoopIQ(int * pI1, int * pQ1, int * pI2, int * pQ2, double startFreq,
double stopFreq, int MaxFFTSize=16, int imageHandling=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int SHAPI_ProcessFSLoopData(double * dBArray, int * pI1, int * pQ1, int * pI2, int *
pQ2, double startFreq, double stopFreq,

                                        int &returnCount, int FFTSize=16,int MaxFFTSize=16, int imageHandling=0, int deviceNum=0);
extern "C" __declspec( dllimport ) int  SHAPI_RunMeasurementReceiver(void * LPStruct, int deviceNum=0);
extern "C" __declspec( dllimport ) int  SHAPI_ProcTGSweep(double * dBArray, double startFreq, double stepSize, int
count, int attenval, double TBMultiplier=1.0, int deviceNum=0); // Fill
extern "C" __declspec( dllimport ) int  SHAPI_SetTGFreqAtten(double freq, unsigned char atten); // Fill
extern "C" __declspec( dllimport ) void  SHAPI_TGSyncOff(int deviceNum=0);
extern "C" __declspec( dllimport ) int  SHAPI_BBSPSweep(double * pData, int startfreq, int stopfreq, int stepfreq, int
deviceNum=0); //in MHz
extern "C" __declspec( dllimport ) int  SHAPI_ProcessData(double * pAmpData, double * pFreqData, int * pIData, int *
pQData, double ctrfreq, int datacount, int fftsz , int deviceNum=0); //in MHz
extern "C" __declspec( dllimport ) int  SHAPI_ProcessLoopPatch(double * pAmpData, double * pFreqData, int *
pIDataH, int * pQDataH,int * pIDataL, int * pQDataL, double ctrfreq, int fftsz,

                                                              bool rejectImage, int deviceNum=0 ); // max fftsz 256;
extern "C" __declspec( dllimport ) int  SHAPI_BuildIQFIR(double * pReal, double * pImag); // max fftsz 256;
extern "C" __declspec( dllimport ) int  SHAPI_ProcIQ(double freq, double * dIout, double * dQout, int * pI, int * pQ, int
* pIold, int * pQold, double * pFIRRe, double * pFIRIm, int deviceNum=0); // max fftsz 256;
extern "C" __declspec( dllimport ) int SHAPI_GetBBPowerReading(double freq, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SetEFCDAC(int efcdac, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_Set36MHzGain(int gain, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_Set36MHzDacs(int lingain, int loggain, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SetSA124Band23Xover(int myXover, int deviceNum=0);
extern "C" __declspec( dllimport ) void SHAPI_SetSA124ClockOut(int val, int deviceNum=0);
```