

# SignalCore™

PRESERVING SIGNAL INTEGRITY



## Programming Manual

SC5520A & SC5521A UHFS

160 MHz to 40 GHz CW Signal Source

Rev 1.2

[www.signalcore.com](http://www.signalcore.com)

## Table of Contents

1	Introduction .....	3
2	Driver Architecture .....	4
2.1	API Function Names and Call Type .....	4
2.2	Compiling Code in C/C++ .....	4
3	Identifying, Opening, and Closing Devices .....	5
3.1	Identifying Devices on the Host Computer .....	5
3.2	Opening and Connecting to a Device .....	6
3.3	Disconnecting from and Closing a Device .....	6
3.4	Opening Multiple Devices .....	6
3.5	Initialize Device .....	6
4	Configuration Functions .....	6
4.1	Setting the Frequency at the Output Port .....	7
4.2	Setting the Offset Phase .....	7
4.3	Setting the Synthesizer Mode .....	7
4.4	Setting the RF Mode .....	8
4.5	Setting the List Mode Configuration .....	8
4.6	Functions to setup List Mode .....	9
4.7	Setting the RF Amplitude .....	9
4.7.1	Setting the Power Level .....	9
4.7.2	Enabling the Output port .....	9
4.7.3	Improving Calibrated Level .....	10
4.7.4	Disabling Automatic Leveling .....	10
4.7.5	Manual Amplitude Control .....	10
4.8	Configuring the Reference Clock .....	10
4.9	Saving the New Default State of the Device .....	10
5	Query Functions .....	11
5.1	Getting General Device Information .....	11
5.2	Getting the Device Status .....	11
5.3	Getting Other RF Parameters .....	11
5.4	Retrieving the Device Temperature .....	12
6	General Functions .....	12
6.1	Self-calibration of internal synthesizers .....	12

6.2 Write Registers ..... 12

6.3 Read Registers ..... 13

7 Appendix ..... 13

7.1 Definitions of types ..... 13

Revision Table..... 16

# 1 Introduction

The SC5520A and SC5521A are high performance signal generators (PSG) in a compact size. For the rest of this document they will be addressed as UHFS unless explicitly called out by their product name. The UHFS frequency range of its generated signal is from 160 MHz to 40 GHz. For more information about its operation and hardware features, see the UHFS hardware manual.

This manual serves as a programming guide for those using the Windows™ software API to program these devices for the purpose of communicating with them through a host computer via the PXIe, USB or RS232 bus. This document is structured into sections that describe the generic use of the product's functions, such as searching for available devices, opening a device, changing the frequency generation parameters, setting power level, and putting the device into power standby.

This manual will explain each function in detail, including the purpose of the function and what its parameters mean. Wherever applicable, snippets of C/C++ code are provided as examples on how to properly use a function.

LabVIEW™ VIs provided have the same function names and parameters, so this manual may serve as a reference for development in LabVIEW™.

## 2 Driver Architecture

The SC5520A is a PXIe (PCI express) based product, while the SC5521A is controlled through USB and RS232. Although a single API called 'sc5520a\_uhfs' is used for all three communication interfaces, each method of communication requires a unique set of system or kernel level drivers.

The software architectures of the communication methods are illustrated in the following table. The left column represents the PXIe software architecture, the middle column represents the USB software architecture, and the right column represents the RS232 software architecture.

Table 1. Software Architectures

PXIe	USB	RS232
userapp.c	userapp.c	userapp.c
sc5520a_uhfs.h sc5520a_uhfs.lib	sc5521a_uhfs.h sc5521a_uhfs.lib	sc5521a_uhfs.h sc5521a_uhfs.lib
sc5520a_uhfs.dll scipcioxi.dll	sc5521a_uhfs.dll libusb-1.0.dll	sc5521a_uhfs.dll kernel32.dll
scipcioxi.sys	winusb.sys	serial.sys

At the highest level, where the user application resides, are the user code, header file(s) (.h), and library file (.lib) for the device. The next level has the device API DLL and driver DLL (.dll), both called by the applicated level. The last level is where the device system driver, or the kernel level driver, (.sys) resides.

### 2.1 API Function Names and Call Type

The function names for an interface are compounded words comprising of the product name first and ending with the function description such as "sc5520a\_uhfsSetFrequency". In this document, all function descriptions will leave out the product name description so that "SetFrequency" is used to represent all interfaces. All functions are of call type `__cdecl` in Windows™.

### 2.2 Compiling Code in C/C++

All necessary header files must be included to compile user written applications. The following table shows the necessary files.

SignalCore headers	C/C++ headers
sci_uhfs_defs.h	stdint.h
sci_uhfs_regs.h	Windows.h
sc5520a_uhfs .h	
sci_types.h	
sci_errors.h	

All functions and their descriptions are found in the `sc5520a_uhfs.h` header file. This following subsections provides further usage descriptions.

### 3 Identifying, Opening, and Closing Devices

The PSG PXIe and USB interfaces are identified by their unique serial numbers. This serial number is passed to the `OpenDevice()` function as a string in order to open a connection to the device. The string consists of 8 HEX format characters such as `100E4FC2`. However, the RS232 interface is assumed to be connected to a serial port of the host and is identified from the port name such as `COM1`, `COM2`, etc.

#### 3.1 Identifying Devices on the Host Computer

The serial number is found on the product label, attached to the outer body of the product. However, if the serial number cannot be found, there is a function to obtain the current devices connected to the host computer. The `SearchDevices()` function scans the host computer for converter devices. If found, a list containing its serial number is returned. The function is declared as;

```
SCISTATUS SearchDevices(sciCommInterface_t commInterface,
                        char **serialNumberList,
                        int *numberDevices);
```

The first parameter `commInterface` is an enumeration of {`PCI_INT`, `USB_INT`, `RS232_INT`}, the `**serialNumberList` is a 2D array format [`number of devices`, `serial number length + 1`], and `*numberDevices` is the number of devices detected and available for connection.

The following code snippet demonstrates how to prepare to call this function.

```
SCISTATUS status;

Sci_comm_interface_t comm_interface = USB_INT;
char **serialNumbers;
int i, nDevices;
serialNumbers = (char**)malloc(sizeof(char*)*MAXDEVICES);
    for (i=0;i<MAXDEVICES; i++)
        serialNumbers[i] = (char*)malloc(sizeof(char)*SCI_SN_LENGTH);
/*
    MAXDEVICES is the number of devices to allocate memory for.
    SCI_SN_LENGTH is defined as 0x09.
*/
    Status = SearchDevices(comm_interface, serialNumbers, &nDevices);
    if(status != SCI_SUCCESS)
        ...error handling, free allocated memory...
```

It is important to free all allocated memory immediately once it is not in use. The following code lines show how to deallocate the memory used to hold the serial numbers.

```
for(i=0;i<MAXDEVICES; i++)
    free(serialNumbers[i]);
free(serialNumbers);
```

### 3.2 Opening and Connecting to a Device

The first step to communicate with the device is to open a connection from the host computer. The following code is an example of how this is done using the `DeviceOpen()` function. The function returns a HANDLE to the device that must be used by subsequent function calls to the device.

```
SCISTATUS status;
HANDLE device_handle;
uint8_t baudrate = 1; // rate: 0 = 57600 1 = 115200
sciCommInterface_t commInterface = RS232_INT;
Status = DeviceOpen(commInterface, COM1, baudrate, &device_handle);
```

Note, serial baud rate of the device defaults to 115200 if the baud rate pin is left unconnected or pulled high. The `COM1` of type `char` for this example accesses the device through the serial port. Upon successfully executing this function, the device **active LED** on the front panel will turn green. This `DeviceOpen()` call does not apply any other changes to the device; its working state remains unchanged by the command.

### 3.3 Disconnecting from and Closing a Device

When the device is no longer in use, the application should disconnect it from the host computer. This is done by using the `DeviceClose()` function. Once it has executed, the **active LED** on the front panel will turn off, and the HANDLE to the device will no longer be valid for further use.

```
status = DeviceClose(device_handle);
deviceHandle = NULL;
```

### 3.4 Opening Multiple Devices

Multiple devices may be opened simultaneously within one application. The `DeviceOpen()` function must be called for each of the devices using their respective serial numbers / serial ports. The HANDLE returned by each call is unique to each device and must be used for subsequent calls only on the device from which it is returned.

### 3.5 Initialize Device

To initialize the device to its reset state or power-up state, use the following code example.

```
#define RESET_STATE 1;
#define CURRENT_STATE 0;
Status = InitDevice(device_handle, RESET_STATE);
```

In the example above, if the value `0` or `CURRENT_STATE` is written, the device will reprogram all the hardware to its current state; that is, the state does not change, but the hardware components are refreshed.

## 4 Configuration Functions

These functions set the device configuration parameters such as frequency and amplitude.

#### 4.1 Setting the Frequency at the Output Port

Setting the frequency at the output is simply writing the frequency value to the `SetFrequency()` function:

```
double rf_freq = 15.2e9;
SetFrequency(dev_handle, rf_freq);
```

#### 4.2 Setting the Offset Phase

When the frequency of the signal changes, its phase with respect to the reference clock is indeterministic. However, as the signal is settled at a particular frequency, its phase can be changed by writing the `sc5520a_uhfsSetSignalPhase()` function. Upon a change in frequency, the phase initial point returns to 0;

```
float phase = 10.0;
sc5520a_uhfsSetSignalPhase(dev_handle, phase);
```

The phase range varies with frequency nominally by design according to the table below

Frequency	Phase (deg)
≥ 5 GHz	0 to 360
1.25 GHz to < 2.5 GHz	0 to 180
625 MHz to < 1.25 GHz	0 to 90
375 MHz to < 625 MHz	0 to 45
312.5 MHz to < 375 MHz	0 to 22.5
160MHz to < 312.5 MHz	0 to 11.25

#### 4.3 Setting the Synthesizer Mode

There are various modes that the synthesizer can be configured to work, and the function `SetSynthMode()` sets the device to operate with the following parameters:

- `lock_mode` values are 0 or 1 representing Harmonic and Fractional, respectively. This `lock_mode` indicates the harmonic or fractional generator is used by the final phase-lock loop of the synthesizer. The harmonic generator will provide the best phase noise; however, frequencies close to the boundaries of 100 MHz multiples may have higher spurious signal levels, especially at boundary frequencies > 4.3 GHz. The fractional generator does not result in final boundary spurs; however, the phase noise is degraded from frequency offsets < 100 KHz.
- `loop_gain` values are 0 or 1 representing Normal and Low. Low gain will generally lower the PLL loop BW, providing better suppression of spurs > 100 KHz offset from the carrier at the expense of an increase in phase noise for offsets < 100 kHz. This has noticeable effect when `lock_mode` is Harmonic (0).
- `disable_spur_suppress` only takes effect when `lock_mode` is Harmonic (0). If it is enabled, the synthesizer will attempt to suppress spurs by 1) lowering loop gain and/or 2) ping-ponging



between lock modes automatically, especially around boundary spurs. If it is disabled (1), the synthesizer will not run an internal algorithm to try suppressing spurs.

#### 4.4 Setting the RF Mode

The device can be set to list and stationary single tone operation using function `SetRfMode()`. Setting the `rf_mode` value to 1 configures the device for sweep mode, resulting in the device not being responsive to frequency change request, but only responding to software or hardware triggers. How the device responds to these triggers depends on the list mode configuration. See the subsection that follows.

#### 4.5 Setting the List Mode Configuration

List mode behavior is set by calling the `ListModeConfig()` function. The list mode structure is explained below.

```
typedef struct list_mode_t
{
    uint8_t sweep_mode;
    uint8_t sweep_dir;
    uint8_t tri_waveform;
    uint8_t hw_trigger;
    uint8_t step_on_hw_trig;
    uint8_t return_to_start;
    uint8_t trig_out_enable;
    uint8_t trig_out_on_cycle;
} list_mode_t;
```

- `Sweep_mode` indicates whether (0) the frequency points are read from the list manual uploaded to the device or (1) based on the start, stop, and step frequencies calculations.
- `Sweep_dir` determines whether the frequency points start from the beginning of the list (start frequency) or from the end of the list (stop frequency) represented by values 0 or 1, respectively.
- `tri_waveform` values 0 and 1 are represented by a sawtooth or triangular sweep. A sawtooth sweep is one that upon reaching the end of the list, immediately returns to the beginning to complete a cycle. A triangular sweep is one that upon reaching the end, traces the steps backward to the beginning to complete a cycle.
- `hw_trigger`, when set to 1, uses the external pin of the IO connector to trigger the list mode behavior. Software trigger is used when the value is set to 0. Hardware trigger occurs on a low going transition, a 1 to 0 transition.
- `step_on_hw_trig` will start the sweep of the frequency points whose interval is based on the sweep dwell time set by function `SweepDwellTime()` if its value is zero. If the value is 1, a single frequency change occurs on each hardware trigger. For time critical applications, hardware trigger stepping is recommended.
- `return_to_start`, when set to 1, will always return the frequency pointer to the start upon completion of cycles, otherwise it will remain at the final frequency point.
- `trig_out_enable`, when set to 1, enables the trigger output pin on the I/O connector.

- `Trig_out_on_cycle`, when set to 1, will pulse the trigger output pin on the completion of each cycle, otherwise it will pulse on each frequency step.

## 4.6 Functions to setup List Mode

The following are functions to set up the frequency points, dwell time, and cycles for the list/sweep behavior. All frequencies are in Hertz (Hz), and dwell times are in 500  $\mu$ s; that is, a value of 1 is 500  $\mu$ s, 2 is 1 ms, etc. For if cycle count is set to 0, the sweep will loop continuously until a trigger is detected or when list mode is disabled through `SetRfMode()`. The functions are:

```
SweepStartFreq(HANDLE dev_handle, double freq);
SweepStopFreq(HANDLE dev_handle, double freq);
SweepStepFreq(HANDLE dev_handle, double freq);
SweepDwellTime(HANDLE dev_handle, uint32_t dwell_time);
ListCycleCount(HANDLE dev_handle, uint32_t cycle_count);
ListBufferPoints(HANDLE dev_handle, uint32_t list_points);
```

The `ListBufferPoints()` function only applies to the manually loaded list. The points should be less than or equal to the number of frequency points loaded.

The list of frequency points and their corresponding amplitude levels are loaded to the device by calling the `ListBufferWrite(HANDLE dev_handle, double *freq, float *level, int len)` function. The parameter length must be  $\leq$  to the length of the frequency and level buffers. Once the list buffers are uploaded, the buffer points can be dynamically changed with the `ListBufferPoints()` function if needed.

The `ListBufferWrite()` discussed here loads the data to a RAM buffer in memory. This buffer can be stored permanently to EEPROM that can be retrieved upon power up of the device. Transferring data to and from memories between the EEPROM and the RAM buffer is done by calling the `ListBufferTransfer()` function. A value of 0 for the `transfer_mode` will move data from RAM to EEPROM and a value of 1 will do the opposite.

When the list mode is configured for software triggering, calling the function `ListSoftTrigger()` will trigger the behavior of the list mode.

## 4.7 Setting the RF Amplitude

### 4.7.1 Setting the Power Level

The RF power level in dBm is set using function `SetPowerLevel()`:

```
float rf_level = 4; // 4 dBm
SetPowerLevel(dev_handle, rf_level);
```

### 4.7.2 Enabling the Output port

The output signal can be enabled by calling the `SetOutputEnable()`. This function simply maximizes the attenuation level and sets the signal to some frequency whose leakage is minimal. The internal oscillators are fully operational so leakages may still appear at the output port. This allows the device to enable and put out a signal in a relatively short period of time, typically from disable to enable would take less than 10 ms. To completely turn the oscillators off to eliminate LO

leakages, call the `SetDeviceStandby()` function. Furthermore, putting the device into standby mode powers down most analog functions, except for the reference circuitry that keeps the OCXO active so that the frequency is stable when the device is taken off standby; cold start of an OCXO causes frequency drift.

#### 4.7.3 Improving Calibrated Level

If the amplitude accuracy requires very fine adjustment to the output amplitude, incremental values can be written directly to the amplitude adjustment DAC by calling `SetLevelDacValue()`. To find out the current level DAC value, call the `FetchLevelDacValue()` function (See the Query Functions section). Increasing the DAC value lowers the amplitude.

#### 4.7.4 Disabling Automatic Leveling

When frequency changes, the device needs to compute the parameters for the ALC DAC and output attenuators to set the amplitude accurately at the new frequency. The computational time plus component setup time can be as long as 350 $\mu$ s, increasing the switching time between frequency changes. The raw amplitude variation between 2 frequencies that are less than 100 MHz apart is typically less than 1 dB, so in applications where this is tolerable the automatic leveling control should be disabled to increase switch speed. The function to enable and disable automatic leveling is `SetAutoLevelDisable()`.

#### 4.7.5 Manual Amplitude Control

In applications such as driving a mixer, where automatic leveling is not required, the amplitude can be coarsely set manually by programming the internal leveling DAC and attenuator. This can be done by calling the `SetLevelDacValue()` and `SetRfAtten()` respectively. The attenuator provides effectively 20 to 25 dB of adjustment, while the level DAC provides another 15 dB.

### 4.8 Configuring the Reference Clock

The configuration of the device reference clock behavior is performed using the following function:

```
uint8_t pxi10Enable = 1; \\ Export PXI-10MHz (valid only in PXIe)
uint8_t select_high = 0; \\ Export 10 MHz instead of 100 MHz
uint8_t lock_external = 1; \\ Lock to external 10 MHz reference clock

SetReferenceMode(dev_handle, pxi10Enable, select_high, lock_external);
```

The accuracy of the internal 10 MHz OCXO reference can be adjusted finely by changing its control voltage via a DAC, which can be written to using function `SetReferenceDacValue()` that accepts a 16 bit value. This new value can be stored as the default using the function `StoreDefaultState()`, which will be discussed ahead.

### 4.9 Saving the New Default State of the Device

The current operating state of the device, including the new DAC value as discussed above, can be stored as the device default by calling the `StoreDefaultState()` function. Once this function is executed, the current state will be the device reset and power up state. This is done by using the following code.

```
status = SetAsDefault(deviceHandle);
```

## 5 Query Functions

These functions read back data from the device such as the current device configuration, operating status, temperature, and other general device information.

### 5.1 Getting General Device Information

Information such as the product hardware revision, serial number, and more can be retrieved from the device using the following code:

```
device_info_t device_info;
status = FetchDeviceInfo(device_handle, &device_info);
```

The `device_info_t` structure has the following members (see header files for more info):

```
typedef struct device_info_t
{
    uint32_t product_serial_number;
    float hardware_revision;
    float firmware_revision;
    uint8_t device_interface;
    struct date
    {
        uint8_t year; // year
        uint8_t month;
        uint8_t day;
        uint8_t hour;
    } man_date;
} device_info_t;

device_interface - 0 = PXI/PXIe, 1=USB&SPI, 2=USB&RS232
```

### 5.2 Getting the Device Status

The phase lock loop status of each of the internal synthesizers and the operational configuration such as the signal path configuration, reference configuration, and local oscillator power status can be obtained by passing the `deviceStatus_t` structure into the following function:

```
deviceStatus_t deviceStatus;
status = GetDeviceStatus(deviceHandle, &device_status);
```

The members of `device_status_t` will not be explicitly discussed here as there are many of them. Please see the `sci_uhfs_def.h` header file in the Appendix section for details.

### 5.3 Getting Other RF Parameters

The RF dynamic parameters such as frequency, offset phase, and power level can be read back using the following code:

```
device_rf_params_t device_rf_params;
status = GetRfParameters(device_handle, &device_rf_params);
```

The structure of the `device_rf_params_t` is as follows:

```
typedef struct device_rf_params_t
{
    double frequency;           //current ch#1 rf frequency
    double sweep_start_freq;    //sweep start frequency
    double sweep_stop_freq;     //sweep stop frequency ( > start_freq)
    double sweep_step_freq;     //sweep step frequency
    uint32_t sweep_dwell_time;  //dwell time at each frequency
    uint32_t sweep_cycles;      //number of cycle to sweep/list
    uint32_t buffer_points;     //current number of list buffer points
    float rf_phase_offset;      //offset_phase value
    float power_level;          //current output power level
    float atten_value;          //current rf attenuator value
    uint16_t level_dac_value;   //current value of the level DAC
} device_rf_params_t;
```

## 5.4 Retrieving the Device Temperature

The device has an internal temperature sensor that reports temperature back in degrees Celsius:

```
float device_temp;
status = GetTemperature(device_handle, device_temp);
```

This temperature can be used to monitor the internal temperature of the device to ensure it is not outside the recommended range.

# 6 General Functions

## 6.1 Self-calibration of internal synthesizers

The PSG has 2 main internal synthesizers with very wide band oscillators. The frequencies of these oscillators are generally outside the capture range of their PLL phase detectors and therefore need pre-tune voltages to bring the frequency inside their capture range. Thus, a calibration has to be performed for each synthesizer to figure out what the pre-tune voltages are for various regions of frequencies. The 2 synthesizers are the coarse harmonic and the main synthesizers. To calibrate the internal synthesizers, call the function `SynthSelfCalibrate(device_handle, vco_select)` where `vco_select` is either 0 or 1 representing the coarse and sum synthesizers, respectively. Over time and changes in temperature, the raw VCO frequencies may drift, so it is recommended to run the function to re-align the pre-tune voltages.

## 6.2 Write Registers

Direct access to the device configuration registers is performed using the `RegWrite()` function. The parameter `reg_byte` is the register address, and these addresses are provided in the `sci_uhfs_regs.h` header file. While the register addresses are found in the header file, their map and definition are provided in the hardware manual. The `instruct_word` parameter is unsigned 64-bit data

associated with the register. Using this function, the 3.2 GHz input frequency of the device can be programmed as follows:

```
uint8_t reg_byte = RF_FREQUENCY; // RF_FREQUENCY = 0x10
uint64_t instruct_word = 3,200,000,000,000; // in mHz

status = RegWrite( deviceHandle, reg_byte, instruct_word);
```

### 6.3 Read Registers

Directly requesting data from the device is performed using `RegRead()`. The function has the following form (from the `sc5520a_uhfs_functions.h` header file):

```
SCISTATUS RegRead(HANDLE device_handle,
                  uint8_t reg_byte,
                  uint64_t instruct_word,
                  uint64_t *received_word);
```

Here `reg_byte` is the register address, `instruct_word` specifies what returned data associated with the register is requested while the `received_word` holds the returned data. Registers that return data are referred to as query registers, and in many of these the parameter `instruct_word` is set to 0 (zero) or simply ignored by the device. However, there are others whose `instruct_word` requires non-zero input. For example, to obtain the current frequency, `instruct_word` is 0 for register address `GET_RF_PARAMETER (0x20)` and the code is:

```
uint64_t instruct_word = 0;
uint64_t received_data;
double frequency;

status = RegRead( deviceHandle, GET_RF_PARAMETERS,
                  instruct_word, &received_data);
frequency = (double)received_data * 0.0001; // convert from mHz to Hz
```

## 7 Appendix

### 7.1 Definitions of types

```
enum LOOPGAIN /* pll loop gain*/
{
    LOW,
    MEDIUM, /* factory default*/
    HIGH
};

typedef enum referenceFreq
{
    _1MHz,
    _5MHz,
    _10MHz,
    _100MHz
} referenceFreq_t;
```

```

typedef enum device_interface
{
    PCI_INT = 0, \
    USB_INT, \
    RS232_INT
} device_interface_t;

/*
 * Structure types to hold data
 */

typedef struct device_info_t
{
    uint32_t product_serial_number;
    float hardware_revision;
    float firmware_revision;
    uint8_t device_interfaces;
    struct date
    {
        uint8_t year; // year
        uint8_t month;
        uint8_t day;
        uint8_t hour;
    } man_date;
} device_info_t;

typedef struct list_mode_t
{
    uint8_t sweep_mode;           // 0 uses list for buffer, 1 calculates using
stop-start-step
    uint8_t sweep_dir;          // 0 start/beginning to stop/end, 1 stop/end to
start/beginning
    uint8_t tri_waveform;       // 0 sawtooth, 1 triangular
    uint8_t hw_trigger;         // 0 soft trigger expected, 1 hard trigger expected
    uint8_t step_on_hw_trig;    // 0 trigger to sweep through list, 1 stepping on
ever trigger (on hard trigger only)
    uint8_t return_to_start;    // if 1, frequency returns to start frequency after
end of cycle(s)
    uint8_t trig_out_enable;    // 1 enable a trigger pulse at the trigger on pin
    uint8_t trig_out_on_cycle; // 0 trigger out on every frequency change, 1
trigger on cycle complete
} list_mode_t;

typedef struct
{
    uint8_t sum_pll_ld;         //lock status of main pll loop
    uint8_t crs_pll_ld;        //lock status of coarse offset pll loop (used only
for harmonic mode)
    uint8_t fine_pll_ld;       //lock status of the dds tuned fine pll loop
    uint8_t crs_ref_pll_ld;    //lock status of the coarse reference pll loop
    uint8_t crs_aux_pll_ld;    //lock status of the auxiliary coarse pll loop (used
only for IntN or FracN mode)

```

```

    uint8_t ref_100_pll_ld;    //lock status of the 100 MHz VCXO pll loop
    uint8_t ref_10_pll_ld;    //lock status of the master 10 MHz TCXO pll loop
} pll_status_t;

typedef struct
{
    uint8_t rf1_lock_mode;    //synthesizer lock mode for chn#1: 0 = use harmonic
    circuit, 1 = fracN circuit
    uint8_t rf1_loop_gain;    //Changing the loop gain of the sum pll. 0 = normal,
    1 = low. low gain helps suppress spurs and far out phase noise, but increase the
    close in phase.
    uint8_t device_access;    //if a session has been open for the device
    uint8_t device_standby;    //indicates chn#1 standby
    uint8_t auto_pwr_disable; //indicates power adjustment is performed when
    frequency is changed.
    uint8_t output_enable;    //indicates chn#1 RF output
    uint8_t ext_ref_lock_enable; //indicates that 100 MHz VCXO is set to lock to
    an external source
    uint8_t ext_ref_detect;    //indicates external source detected
    uint8_t ref_out_select;    //indicates the reference output select: 0=10 MHz,
    1=100MHz
    uint8_t list_mode_running; //indicates list/sweep is triggered and currently
    running
    uint8_t rf_mode;          //indicates chn#1 rf mode set: 0=fixed tone state,
    1=list/sweep mode state
    uint8_t over_temp;        //indicates if the temperature of the devices has
    exceeded ~75degC internally
    uint8_t harmonic_ss;      //hamonic spur suppression state
    uint8_t pci_clk_enable;    //PXI 10 MHz clock enable
} operate_status_t;

typedef struct
{
    list_mode_t list_mode;    //list mode parameters
    operate_status_t operate_status; //operating parameters
    pll_status_t pll_status;    //pll status
} device_status_t;

typedef struct device_rf_params_t
{
    double frequency; //current ch#1 rf frequency
    double sweep_start_freq; //sweep start frequency
    double sweep_stop_freq; //sweep stop frequency ( > start_freq)
    double sweep_step_freq; //sweep step frequency
    uint32_t sweep_dwell_time; // dwell time at each frequency
    uint32_t sweep_cycles; // number of cycle to sweep/list
    uint32_t buffer_points; // current number of list buffer points
    float rf_phase_offset; // phase offset
    float power_level; // current ch power level
    float atten_value; // current rf attenuator value
    uint16_t level_dac_value; // current level DAC value
} device_rf_params_t;

```



## Revision Table

Revision	Revision Date	Description
0.1	11/24/2019	Document Created
1.0	03/17/2020	Initial Release
1.1	04/22/2020	Grammatical edits
1.2	04/16/2021	Updated Section 4.2 function name

