SC5309A and SC5310A
2.5 GHz Downconverter
Software API Manual
# 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++ .................................................................................... 5
3. Identifying, Opening and Closing Devices .................................................................. 6
   3.1 Identifying devices on the Host Computer ............................................................... 6
   3.2 Opening and Connecting to a Device ...................................................................... 7
   3.3 Disconnecting from and Closing a Device .............................................................. 7
   3.4 Multiple Devices .................................................................................................... 7
   3.5 Initialize Device ....................................................................................................... 7
4. Configuration Functions ............................................................................................... 8
   4.1 Setting the frequency at the ports .......................................................................... 8
   4.2 Setting the Attenuators ......................................................................................... 8
   4.3 Configuring the Conversion Signal Path ................................................................ 9
   4.3.1 Enabling the RF Preamplifier ........................................................................... 10
   4.4 Configuring the Conversion Gain .......................................................................... 10
   4.5 Setting the Synthesizer modes .............................................................................. 12
   4.6 Configuring the Reference Clock .......................................................................... 12
   4.6.1 Adjustment to the Internal TCXO Clock ........................................................... 13
   4.7 Saving the New Default State of the Device ......................................................... 13
5. Query Functions ........................................................................................................... 13
   5.1 Getting General Device Information ..................................................................... 13
   5.2 Getting the Device Status ..................................................................................... 14
   5.3 Getting Other RF Parameters .............................................................................. 14
   5.4 Retrieving Auto-gain parameters ........................................................................ 15
   5.5 Retrieving the Device Temperature ...................................................................... 15
6. Calibration Functions ................................................................................................... 15
   6.1 Obtaining Calibration Data .................................................................................. 15
   6.1.1 Structured Calibration Data format .................................................................. 15
   6.1.2 Reading Formatted Data ................................................................................ 17
   6.1.3 Reading Raw Calibration Data ........................................................................ 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Configuring the Gain of the Device Using Calibration</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>General Functions</td>
<td>21</td>
</tr>
<tr>
<td>7.1</td>
<td>Writing to the User EEPROM</td>
<td>21</td>
</tr>
<tr>
<td>7.2</td>
<td>Reading from the Calibration and User EEPROMs</td>
<td>21</td>
</tr>
<tr>
<td>7.3</td>
<td>Configuring the Frequency Plan</td>
<td>22</td>
</tr>
<tr>
<td>7.4</td>
<td>Self-Calibration of the LO1 synthesizer</td>
<td>22</td>
</tr>
<tr>
<td>7.5</td>
<td>Write Registers</td>
<td>23</td>
</tr>
<tr>
<td>7.6</td>
<td>Read Registers</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Revision Notes</td>
<td>24</td>
</tr>
</tbody>
</table>
1 Introduction

The SignalCore SC5309A and SC5310A are high dynamic, high performance triple stage super heterodyne RF downconverters covering the frequency range of 100 kHz to 2.5 GHz. These products offer wide conversion bandwidths of up to MHz, 40 MHz, making them ideal for applications involving broad band RF signal conversion such as those in data links, TV and radio band spectrum monitoring, cable TV test, and other test and measurement systems.

This manual serves as a programming guide to those using the Windows™ software API to program these devices for the purpose communicating with them through a host computer via the PXIe, USB, or RS232 bus. The document is structured into sections that describes the generic use of its functions such as searching for available devices, opening a device and obtaining a handle to it, changing the conversion parameters such as RF frequency, attenuation, filter selection and enabling RF amplifiers, obtaining gain correction using calibration data, and putting the device into power standby.

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

The SC5309A is a PXI based product, while the SC5310A is controlled through USB and RS232. Although they are of different platforms and controlled through different buses, their API functions are almost identical, so they are treated together in this document. Each of these three different methods of communication requires its unique set of header files, dynamic linked libraries (DLL), and system level drivers.

We can illustrate below using diagrams the software architectures of the communication methods utilized. At the highest level, where the user application resides, are the user code, and the header file(s) (*.h) and the library file(*lib) for the device. The level below that has the device API DLL and driver DLL (*.dll), which are called by the application level. The lowest level is where the device system driver or the kernel level driver (*.sys) resides. Figure 1 shows the software architecture of the three interface buses, the left-side diagram represents the PXIe software architecture, the middle diagram represents the USB software architecture, and the right-side diagram represents the RS232 software architecture.

![Software Architectures Diagram](image)

2.1 API Function Names and Call type

The function names for an interface are compounded words comprising of the product name, followed by the interface, and ending with the function description such as “sc5310a_usbSetFrequency”. Since the PXIe version has only one interface the interface description is left out such that the previous example would read “sc5309aSetFrequency”. In this document, we will leave out the product and interface description so that “SetFrequency” is used to represent all interfaces. We will use sc5310a_usb and sc5310a_rs232 to explicitly distinguish between the USB and RS232 interfaces. All functions are of call type __stdcall in Windows™.
2.2 Compiling Code in C/C++

The header files are shared between PXI, USB and RS232 interfaces, and not only are they shared between these interfaces for downconverters; they are also shared with the upconverters of the same family, which is code named “albatross”. Thus, many of the headers have the prefix “scialb” in their name. To successfully use these header files to write applications, proper macros are required to be defined for the compiler to properly compile the code.

<table>
<thead>
<tr>
<th>DEVICE/INTERFACE</th>
<th>MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXI</td>
<td>PXI_INT</td>
</tr>
<tr>
<td>USB</td>
<td>USB_INT</td>
</tr>
<tr>
<td>RS232</td>
<td>RS232_INT</td>
</tr>
</tbody>
</table>

In Microsoft Visual Studio, these macros can be entered in as Preprocessor Definitions in the project properties window. This could also be accomplished in GCC with the -D name option, where name is one of the macro words.
3 IDENTIFYING, OPENING AND CLOSING DEVICES

The SC5309A and SC5310A downconverters 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”

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 label is missing or beyond the means to view, there is a function to obtain the current devices connected to the host computer. The SearchDevice() function scans the host computer for SC5309A or SC5310A devices and if they are found, a list containing their serial numbers and the number of devices are returned. The function is declared as:

SCISTATUS SearchDevices(char **serialNumberList,
                         int *numberDevices);

The **serialNumberList is a 2D array of 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;
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 and
 SCI_SN_LENGTH is defined 0x09
*/
status = SearchDevices( serialNumbers,
                         &nDevices
                     );
if(status != SCI_SUCCESS)
    ...error handling, free allocated memory...

It is important to free all allocated memory immediately once they are no longer 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 communicating with the device is to open a connection to it 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 deviceHandle;

Status = DeviceOpen( "<serial number string>",
    &deviceHandle
);

The "<serial number string>" of type char can be substituted by the serialNumber[i] as found in the previous code example. Upon successfully executing this function, the device active LED on the front panel will turn on 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 and once it has executed, the active LED on the front panel will turn off and the HANDLE to the device is no longer valid for further use.

status = DeviceClose( deviceHandle );
deviceHandle = NULL;

3.4 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. 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:
#define RESET_STATE 1;
#define CURRENT_STATE 0;

Status = InitDevice( deviceHandle, 
        RESET_STATE 
    );

In the above example, 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, attenuation, filters, 
and signal path.

4.1 Setting the frequency at the ports

The RF input port frequency can be set by calling the SetFrequency() function while the IF 
output port frequency can be set by calling the SetIfFrequency() function. The RF port 
frequency has a settable upper limit of 2.5 GHz and a lower limit of 0 Hz, while the IF port 
frequency has a settable upper limit of 100 MHz and a lower limit of 0 Hz. Although these are 
fuctional limits, they may not represent the operational performance boundaries of the 
device. Please consult the product hardware manual for more information.

The functions to change frequency may be programmed as:

double rf_frequency = 1.0e9;
double if_frequency = 70e6;

status = SetFrequency( deviceHandle, 
        rf_frequency 
    );

status = SetIfFrequency( deviceHandle, 
        if_frequency 
    );

Note that the RF frequency resolution is 1 Hz, while the IF frequency is 0.5 MHz.

4.2 Setting the Attenuators

These devices have 4 sets of programmable attenuators: there are 2 in the RF input section and 
2 in the final IF output section. Although these attenuators have 0.25 dB step resolution, they
are calibrated to 1 dB steps. Numbers represent these attenuators as defined in the header files:

```c
#define RFATTEN1 0
#define RFATTEN2 1
#define IF3ATTEN1 2
#define IF3ATTEN2 3
```

To set the attenuators to a certain value, we use the function `SetAttenuator()`, and as an example, the following code snippet sets the first RF attenuator to 20 dB and the final IF attenuator to 5.25 dB:

```c
status = SetAttenuator( deviceHandle,
                        RFATTEN1,
                        20.00
); Status = SetAttenuator( deviceHandle,
                          IF3ATTEN2,
                          5.25
);```

### 4.3 Configuring the Conversion Signal Path

These downconverters have configurable filter options and conversion paths, and depending on the option choice, the user needs to properly configure the device prior to setting frequencies and gain (via attenuators and preamplifier). The function `SetSignalPath()` is used to configure the device paths. It requires a structure input containing the configuration parameters in the form:

```c
typedef struct
{
    bool_t rfAmp;
    bool_t if2Filter1;
    bool_t invertSpectrum;
} signalPathParams_t;
```

The product user manual provides details to what each of the structure parameters are, however a brief description is provided here:

- **rfAmp** – the RF preamplifier setting, a value of 0 disables and 1 enables. The parameter only affects the preamplifier if only the `autoCtrlRfAmp` parameter for autogain calculation is disabled; see `SetAutoGain()`. There is another function `SetRfAmp()` that enables and disables the RF preamplifier by overriding the setting of the `SetAutoGain()` function.
• **If2Filter1** – this selects filter#1 instead of the default filter@0. Note, standard product offering does not include filter#1, it is an option. Filter#1 may have a different bandwidth from filter#0, giving the user bandwidth flexibility in some applications.

• **invertSpectrum** – when set the spectrum at the IF port is inverted relative to the RF input.

To set the device to invert the IF spectrum and uses the default IF2 filter, the code is as follows:

```c
signalPathParams_t *pathParameters;

pathParameters->if2Filter1 = 0; /* set the filter to default */
pathParameters->invertSpectrum = 1;

status = SetSignalPath( deviceHandl,
    pathParameters
);
```

### 4.3.1 Enabling the RF Preamplifier

Although the RF preamplifier can be enabled or disabled using the `SetSignalPath()` function, it can also be controlled through the `SetPreamp()` function. The advantage of using this function is that it only acts on the amplifier and not on other signal path components. It will also force the setting of the amplifier, and return it to manual control. The amplifier is enabled by simply writing:

```c
Bool_t preampStatus = 1;

Status = SetPreamp( deviceHandle,
    preampStatus
);
```

### 4.4 Configuring the Conversion Gain

The device has an algorithm to automatically set up the attenuators and amplifier to achieve the desired gain while maintaining the desired dynamic range. The gain of the device is the difference in power between the output and input ports, if most of the gain is applied close to the input of the device, the signal-to-noise dynamic range is improved, however if it is applied later close to the output, the linearity dynamic range is improved. A balance of the gain will also provide a balance between SNR and IMD dynamic ranges.

Since the computation is performed by the onboard MCU, it relies heavily on ideal parameters such as attenuator states in the calculations, and thus the device setting will only be within 2 dB of the desired value. To achieve better gain setting and accuracy calculation, the functions discussed in chapter 6 should be used. Setting the device up for gain requires user input gain-parameters in its structure form:

```c
typedef struct gainParams_s
{
```
Here the members are:

- **rfLevel** – this is the nominal expected level at the RF port, commonly called the reference level when the converter is used in a signal analyzer application.
- **mixerLevel** – this is the nominal level at the first input mixer. Typical values of -20 dBm to -13 dBm are used.
- **ifLevel** – this is the nominal output IF level.
- **linearMode** – There are 6 options:
  0. Calculations will use the mixerLevel to determine the attenuator values; this is default.
  1. Normal mode, a trade-off between noise figure and linearity.
  2. Better noise figure mode.
  3. Best SNR dynamic range mode. Best setting for fundamental tone measurement.
  4. Better linearity
  5. Best linearity
- **autoCtrlRfAmp** – the algorithm determines the state of the RF preamp instead of the status of the RF preamp set manually by SetPreamp() or SetSignalPath().

The following demonstrates how to set the device up to automatically set up the gain as the RF frequency is changed. If its parameters remain unchanged throughout the application, the function below only needs to be run once.

```c
unsigned char loadParams = 1;
unsigned char autoGainEnable = 1;
gainParams_t *gainParams;

gainParams->rfLevel = -10.0;
gainParams->mixerLevel = -20.0;
gainParams->ifLevel = 0.0;
gainParams->linearMode = 0;
gainParams->autoCtrlRfAmp = 1;

status = SetAutoGain( deviceHandle,
    gainParams,
    loadParams,
    autoGainEnable
);```
There are 2 additional parameters used in the above functions; `loadParams` and `autoGainEnable`, and they are defined as:

- **loadParams** – set this to 1 if the device is required to load in a fresh set of `gainParams`. If the gain parameters do not change through the application, then the parameter only needs to be loaded once.
- **autoGainEnable** – this parameter if set to 1 will cause the device to update the attenuators and RF amplifier when frequencies are changed. **NOTE**: If the external API functions are used to compute the automatic setting for the attenuators and hence the gain of the device, this `autoGainEnable` must be set to low (0). We recommend using the API function `CalcAttenValues()` to compute the attenuator settings, and we also recommend calling this function to store the `gainParams` by enabling `loadParams` but disabling `autoGainEnable`. Later in the applications or another application may recall these `gainParams` of the device by calling the function `GetGainCalcParameters();`

### 4.5 Setting the Synthesizer modes

The loop gain of the synthesizer can be changed to shape the phase noise spectral density of the signal. There are 3 options for the loop gain: low, normal, and high. For low levels of close-in carrier phase noise, select **HIGH**. The first local oscillator is an agile VCO based synthesizer whose tuning speed may be improved by enabling fast tuning; consult the hardware manual for more information on these synthesizer modes. The following demonstrates how the settings are done:

```c
Enum LOOPGAIN pllLoopGain = HIGH;

Status = SetSynthMode( deviceHandle,
    pllLoopGain
);
```

### 4.6 Configuring the Reference Clock

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

```c
bool_t lockExtEnable = 1; /*enable locking to external source */
bool_t refOutEnable = 1; /* enable output of reference clk */
bool_t clk100Enable = 0; /* ref out will be 10 MHz, not 100 MHz*/
bool_t pxi10ClkEnable = 0; /* export 10MHz PXI clk – SC5309A only */

status = SetReferenceClock( deviceHandle,
    lockExtEnable,
    refOutEnable,
    clk100Enable,
    pxi10ClkEnable
);```
4.6.1 Adjustment to the Internal TCXO Clock

The device has a TCXO timebase whose frequency accuracy may be adjusted via a DAC. When the device is not locked to an external reference source, it uses its internal TCXO as the reference. The following functions are used to make small incremental adjustments to this clock:

```c
unsigned int tcxoDac = 0x2E0A; /* value range of 0x00 to 0x3FFF */
status = SetReferenceDac( deviceHandle, tcxoDac );
```

4.7 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 `SetAsDefault()` function. Once this function is executed, the current state will be the device reset and power up state. This is done by simply coding as follows:

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

5 QUERY FUNCTIONS

These functions read back data from the device, data 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, etc, can be retrieved from the device using:

```c
deviceInfo_t deviceInfo;
status = GetDeviceInfo( deviceHandle, &deviceInfo );
```

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

```c
typedef struct deviceInfo_s
{
    uint32_t productSerialNumber;
    float firmwareRevision;
    ...
} deviceInfo_t;
```
float HardwareRevision;
uint8_t deviceInterface;
scidate_t calDate;
scidate_t manDate;
} deviceInfo_t;

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,
 &deviceStatus
);

The members of deviceStatus_t will not be explicitly discussed here as there are many of them. Please read the scialbdef.h header file in the Error! Reference source not found. section for details.

5.3 Getting Other RF Parameters

The RF dynamic parameters such as attenuator values, IF frequencies, LO frequencies, and RF frequency can be read back using:

rfParams_t rfParams;

status = GetRfParameters( deviceHandle,
 &rfParams
);

The structure of the rfParams_t is:

typedef struct
{
    double frequency; /* the RF frequency */
    double if1Freq;  /* the first IF freq */
    double if2Freq;  /* the second IF freq */
    double if3Freq;  /* the third IF freq */
    double lo1Freq; /* the first agile LO freq */
    double lo2Freq; /* the second LO freq */
    double lo3Freq; /* the third LO freq */
    attenuator_t atten; /* the values of the attenuators */
5.4 Retrieving Auto-gain parameters

To obtain the current autoGainParameters used to calculate the attenuators values for automatic setting of gain, call the function:

```c
gainParams_t *gainParams;
status = GetGainCalcParameters( deviceHandle,
                                 gainParams);
```

5.5 Retrieving the Device Temperature

The device has an internal temperature sensor that reports temperature back in degree Celsius, and it is read back using:

```c
float deviceTemp;
status = GetTemperature( deviceHandle, 
                        deviceTemp);
```

This temperature can be used in computing the conversion gain of the device, as gain is a temperature dependent parameter.

6 Calibration Functions

These functions utilize the onboard calibration data to compute the conversion gain of the device. The conversion gain varies as the device configuration such as signal path selection, attenuator values, and its temperature changes. To compute the conversion gain accurately would require two sets of information: the device configuration and its calibration data.

6.1 Obtaining Calibration Data

There are a couple of ways to read in data from the device:

1. Read data back from the device in a formatted structure.
2. Read data back as an array of raw bytes and then convert the raw bytes into formatted data.

6.1.1 Structured Calibration Data format

The structure format that holds the calibration data is:

```c
typedef struct calData_s
```
The following are descriptions of each of the struct members:

- **calTemp** – this is the factory calibration temperature, \( T_0 \).
- **tempCoeff** – contains 2 temperature-gain coefficients \( c_1 \) and \( c_2 \).
- **if3ResponseCalFreq** – this is the IF frequency relative gain response with respect to 50 MHz (by default, could be different on custom units).
- **if3ResponseCal** – this is the relative gain at those IF frequencies.

<table>
<thead>
<tr>
<th>If2ResponseCalFreq</th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>...</th>
<th>fN</th>
</tr>
</thead>
<tbody>
<tr>
<td>If3ResponseCal</td>
<td>rg(f0)</td>
<td>rg(f1)</td>
<td>rg(f2)</td>
<td>...</td>
<td>rg(fN)</td>
</tr>
</tbody>
</table>

- **if3Atten1Cal** and **if3Atten2Cal** – contains the relative measured values corresponding to the attenuation states of the two IF3 attenuators at a fixed IF of 50 MHz.

<table>
<thead>
<tr>
<th>Attenuation state</th>
<th>1dB</th>
<th>2dB</th>
<th>3dB</th>
<th>...</th>
<th>30dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF3 Attenu#1</td>
<td>A1_a1</td>
<td>A1_a2</td>
<td>A1_a3</td>
<td>...</td>
<td>A1_a30</td>
</tr>
<tr>
<td>IF3 Attenu#2</td>
<td>A2_a1</td>
<td>A2_a2</td>
<td>A2_a3</td>
<td>...</td>
<td>A2_a30</td>
</tr>
</tbody>
</table>

- **if2Filt1Cal** – This is the relative gain difference with respect to filter#0 when selecting filter#1 bandpass filter.
- **if3InvertGain** – This is the relative gain difference when spectral inversion is selected.
- **rfCalFreq**, **rfAbsGainCal**, **rfAmpGainCal**, **rfAtten1Cal** and **rfAtten2Cal** – the RF response calibration includes the absolute gain of the device as a function of RF frequency under the following conditions:
  - All attenuators are set to 0 dB
  - All filter selections are in their default state of value 0
  - IF3 frequency is set at 50 MHz

*The above conditions are the default and all other configuration measurements are made relative to it. Also included are the RF pre-amplifier gain and the measured*
attenuation values at each state of the two RF attenuators. The data is laid out as (cyan cells are data):

\[
\begin{array}{cccccc}
\text{rfCalFreq} & f_0 & f_1 & f_2 & \ldots & f_N \\
\text{rfAbsGainCal} & g(f_0) & g(f_1) & g(f_2) & \ldots & g(f_N) \\
\text{rfAmpGainCal(f)} & \text{ag}(f_0) & \text{ag}(f_1) & \text{ag}(f_2) & \ldots & \text{ag}(f_N) \\
\text{rfAtten1Cal 1dB} & A_1\_a1(f_0) & A_1\_a1(f_1) & A_1\_a1(f_2) & \ldots & A_1\_a1(f_N) \\
\text{rfAtten1Cal 2dB} & A_1\_a2(f_0) & A_1\_a2(f_1) & A_1\_a2(f_2) & \ldots & A_1\_a2(f_N) \\
\vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\
\text{rfAtten1Cal 30dB} & A_1\_a30(f_0) & A_1\_a30(f_1) & A_1\_a30(f_2) & \ldots & A_1\_a30(f_N) \\
\text{rfAtten2Cal 1dB} & A_2\_a1(f_0) & A_2\_a1(f_1) & A_2\_a1(f_2) & \ldots & A_2\_a1(f_N) \\
\text{rfAtten2Cal 2dB} & A_2\_a2(f_0) & A_2\_a2(f_1) & A_2\_a2(f_2) & \ldots & A_2\_a2(f_N) \\
\vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\
\text{rfAtten2Cal 30dB} & A_2\_a30(f_0) & A_2\_a30(f_1) & A_2\_a30(f_2) & \ldots & A_2\_a30(f_N)
\end{array}
\]

6.1.2 Reading Formatted Data

Memory must be allocated for the members of struct calData_t prior to passing it through the GetCalData() function to retrieve calibration data. The minimum memory size requirement for the arrays are provided in their descriptions above. Their size constants can also be found in the scimjdefs.h header files. The following code snippet demonstrates how formatted data is read from the device:

```c
calData_t *calData;

calData->rfCal = (float**)calloc(RFCALPARAMLEN,sizeof(float*));
for(i = 0; i < RFCALPARAMLEN; i++)
    calData->rfCal[i] = (float*)calloc(RFCALFREQLEN,sizeof(float));

--- likewise allocate memory to the other struct members ---

status = GetCalData( devicehandle,
    calData
);
```

There are two useful functions to handle calibration memory; one to allocate and the other to deallocate. You may call function AllocateCalDataMemory() to allocate the required data, and when data is no longer needed the memory may be deallocated with function DeallocateCalDataMemory(). We can rewrite the previous example as such:

```c
calData_t *calData;

status = AllocateCalDataMemory(calData);
status = GetCalData( devicehandle,
    calData
);

//do something with the data and when data is longer needed
Status = DeallocateCalDataMemory(calData);
```
6.1.3 Reading Raw Calibration Data

Reading the entire calibration data may be longer than what the application desires, so in cases where data needs to be retrieved faster the data could be stored to file ahead of time and read back when it is required. Raw data in bytes may be read from the device and stored at a text file on the host computer. Once the data is read in as a 1D byte array, it will need to be formatted to be useful. There are two functions provided to perform these tasks: one to read in the raw data, the other to convert it to formatted data:

```c
unsigned char *rawData;
calData_t *calData;

rawData = (unsigned char*)calloc(RAWDATALEN, sizeof(char));
status = GetRawCalData( deviceHandle, rawCalData );
//Raw cal data can be stored to file and retrieved quicker.
//This raw data can then to connected to formatted data:
Status = AllocateCalDataMemory(calData);
status = RawToFormatData( rawCalData, calData );
```

Note that RAWDATALEN is the total number of raw data bytes. This data is stored on the calibration EEPROM between addresses 0x3F8 and 0x55FF; see the calibration EEPROM map of the product hardware manual for more details.

6.2 Configuring the Gain of the Device Using Calibration

The SC5309A and SC5310A are broadband devices whose RF conversion gain response varies as a function of frequency, filter selection, and signal path for any given attenuation setting. That is, setting the attenuators to obtain a certain gain value at one frequency does not guarantee that it remains the same at another frequency, especially if the other frequency is over a couple of GHz difference. The user can experimentally determine how the attenuators are to be set as a function of frequency. The resulting gain data can be stored in a table to be read back and applied as frequency is changed. In effect, the user is performing self-calibration on the device and using the calibration data in an application.

These devices come with both RF and IF attenuators so that the user has the freedom to set them accordingly to achieve the desired performance. For more information on how to set these attenuators to achieve the desired performance, consult the hardware manual. SignalCore has algorithms to compute the values of the attenuators such that the device is set...
up for its best desired performance. The function \texttt{CalcAttenValues()} will compute the attenuator values based on user inputs such as frequency, nominal input and output power levels, and linear mode selection. In addition to these user inputs, it also uses calibration data so that the conversion gain is also computed and returned with the attenuator values. If the device is programmed with the calculated attenuator values, the computed gain is that of the device to within margin of error.

Furthermore, if \texttt{gainParams.autoCtrlRfAmp} is set to 1 the function will alter the state of the RF amplifier and return its state in the \texttt{signalPathParams_t} structure.

The following code demonstrates how the \texttt{gainParams_t} structure as discussed in chapter 4.4 is used, along with frequency parameters, calibration data, and temperature, to compute the attenuator values required to configure the device to the calculated gain.

```c

calData_t *calData;

/* read in calibration data to fill up calData, see GetCalData() */

gainParams_t gainParams;

attenuator_t *attenuator; /* receive attenuator values */

float gain; /* receive the computed gain */

gainParams.rfLevel = -10.0;
gainParmas.mixerLevel = -20.0;
gainParams.ifLevel = 0.0;
gainParams.linearMode = 0;
gainParams.autoCtrlRfAmp = 0;

loadParams = 1; allow loading the gainParams into device memory
autoGainEnable = 0; disables the devices ability to override external settings to attenuators and preamplier.

status = CalcAttenValues( rfFrequency,
                        ifFrequency,
                        temperature,
                        gainParams,
                        calData,
                        pathParameters,
                        attenuator,
                        &gain);

/* apply the calculated attenuator values */
status = SetAttenuator( deviceHandle,
                    RFATTEN1,
                    Attenuator->rfAtten1Value
                );

...do the same for the rest of the attenuators...
```
If the user prefers to set the attenuator values and the RF amplifier independently from those calculated by the `CalcAttenValues()` function, the gain of the device may be computed using the `CalcGain()` function, as shown here:

```c
...fill in the attenuator values, set the pathParameters, then call ...

Status = CalcGain( rfFrequency,
    ifFrequency,
    temperature,
    calData,
    pathParameters,
    attenuator,
    &gain
);
```

The computed gain of the device is approximately the difference between the output IF level and the input RF level. The step resolution and accuracies of the attenuators limit the gain values, so although the exact desired gain may not be obtainable, the above 2 functions return a value that is close to the its actual gain of the device for that setting. Notice that in both these functions the gain is not the input parameter to set up the device. Rather, the gain is computed by examining the settings of the device. In many converter applications, it is easier to think in terms of the expected RF level and the required IF level, so configuring the device to meet the input and output requirements makes better sense. Unlike the function `SetAutoGain()`, which is performed on the device MCU, both `CalcAttenValues()` and `CalcGain()` are mathematical functions performed on the host computer because they are computationally complex and “heavy”; it would take much longer to compute with the onboard MCU. The advantage of these two functions over the `SetAutoGain()` function is the improved absolute gain accuracy and computational speed, the disadvantage is that the user needs to call on them for every frequency change. It is important to call `SetAutoGain()` once with its `autoGainEnable` parameter set to FALSE (0) before using these two functions. Doing so will disable the device from overwriting the attenuator and/or amplifier settings when frequency is...
changed. Furthermore, `autoGainEnable` should be set to FALSE whenever manual setting of the attenuators and the RF preamplifier is required.

## 7 General Functions

These functions may be useful for some applications in that they aid in the reading from and writing to the EEPROMs, making minor frequency adjustments to IF1 and IF2, performing synthesizer self-calibration, and writing the registers directly.

### 7.1 Writing to the User EEPROM

These devices have an onboard EEPROM option which is accessible to the user for storing user information such as system specific data and calibration. Data is written one byte at a time.

```c
unsigned char data = 0xED; /* byte data to be written */
unsigned int memAddress = 0x04; /* address from the data */

status = WriteUserEeprom( deviceHandle, memAddress, data);
```

### 7.2 Reading from the Calibration and User EEPROMs

Both calibration and user EEPROM data are read back in the form of a byte array. Selection of the EEPROM, its starting memory address, the length of data to be read back, and an array to receive the data are passed to the `ReadEeprom()` function. The code here demonstrates how to read back the product serial number:

```c
unsigned int startAdd = 0x04;
unsigned int dataLen = 4;
unsigned char receivedBytes[dataLen];

status = ReadEeprom( deviceHandle, CALEEPROM, startAdd, dataLen, receivedBytes);
```

The serial number is an unsigned 32bit numeric and it needs to be converted to a string format of its hexadecimal representation, which is the format that is presented in the literature and used to open a device. Another note we need to make is that data stored in the calibration EEPROM is little endian. The following is a method to convert the data to a string format:
char snString[9]; /* 8 chars + termination */

sprintf(snString, “%X”, *(unsigned int*)receivedBytes);

### 7.3 Configuring the Frequency Plan

There is a function that allows the user to change the frequency of IF1 and IF2, as well as RF and final IF (IF3) frequencies. However, the latter two parameters can be dynamically changed using the `SetFrequency()` and `SetIfFrequency()` functions respectively. Calling the following function will make these the default startup parameters:

```c
status = SetFreqPlanParam( deviceHandle,
    rfFrequency,
    if1Frequency,
    if2FreqFilt#0,
    if2FreqFilt#1,
    if3Frequency,
    lo2PllStepSize,
    lo3PllStepSize
);
```

These parameter values are set at the factory to optimize for the performance of the device, especially for custom filters and their bandwidths. The typical value of IF1 is 3.625 GHz, and IF2 is 305 MHz. Their relationship to the second local oscillator frequency is:

$$IF1 = IF2 + LO2$$

The lo2PllStepSize and lo3PllStepSize parameters are also affected by the choice of filters and final IF3 frequency. Optimized PLL step sizes reduces the magnitude of phase related spurious products.

### 7.4 Self-Calibration of the LO1 synthesizer

The VCO based synthesizer is calibrated at the factory and the calibration is sufficient for the circuitry to maintain lock with the calibration cycle of 3 years. By design, after factory calibration the synthesizer should remain frequency locked for periods over 10 years if its temperature does not deviate from its calibration temperature, typically about 42 °C, by more than ±5 °C. However, the procedure can be run more frequently to ensure that the circuit is always optimized despite changes in component characteristics over time and temperature. Once this function is executed, the program should wait for 5 to 7 seconds for it to complete. Upon a successful calibration, it will update the calibration EEPROM at address 0x1C with 1, otherwise 0.

Note that the following function returns immediately before the calibration procedure is completed.

```c
status = SetSynthSelfCal( deviceHandle );
```
7.5 Write Registers

Direct access to the device configuration registers is performed using the `RegWrite()` function. The parameter `regByte` is the register address, and these addresses are provided in the `scimjregs.h` header file. While the register addresses are found in the header file, their map and definition are provided in the hardware manual. The `instructWord` parameter is unsigned 64-bit data associated with the register. Using this function, the input frequency of the device can be programmed as follows:

```c
unsigned char register = RF_FREQUENCY;
unsigned long long regData = 2000000000000000; // value is in milli-Hz
status = RegWrite( deviceHandle, register, regData );
```

7.6 Read Registers

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

```c
SCISTATUS RegRead( HANDLE deviceHandle, 
    uint8_t regByte, 
    uint64_t instructWord, 
    uint64_t *receivedWord);
```

Here `regByte` is the register address, `instructWord` specifies what returned data associated with the register is requested; the `receivedWord` holds the returned data. Registers that return data are referred to as query registers, and in many of these the parameter `instructWord` is set to 0 (zero) or simply ignored by the device. However, there are others whose `instructWord` requires non-zero input. For example, to obtain the current IF1 frequency `instructWord` is 1, and the code is:

```c
unsigned long long instruct = 1;
unsigned long long receivedData;
status = RegRead( deviceHandle, 
    GET_DEVICE_PARAM, 
    instruct, 
    &receivedData);
# Revision Notes

<table>
<thead>
<tr>
<th>Rev</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Initial Release</td>
<td>08-18-18</td>
</tr>
</tbody>
</table>