Programming Manual

SC5317A* & SC5318A

6 GHz to 26.5 GHz RF Downconverter

www.signalcore.com

*SC5317A coming soon
# 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 Multiple Devices .............................................................................................................................. 6
   3.5 Initialize Device ............................................................................................................................... 6
4 Configuration Functions ............................................................................................................................ 6
   4.1 Setting the Frequency at the Ports .................................................................................................... 7
   4.2 Setting the Attenuators ................................................................................................................... 8
   4.3 Configuring the Conversion Signal Path .......................................................................................... 8
      4.3.1 Enabling the RF Preamplifier .................................................................................................. 9
   4.4 Setting the Synthesizer Modes ........................................................................................................ 9
   4.5 Configuring the Reference Clock ................................................................................................... 10
      4.5.1 Adjustment to the Internal TCXO Clock ................................................................................. 10
   4.6 Saving the New Default State of the Device .................................................................................. 10
   4.7 Setting the First LO Drive Path ...................................................................................................... 10
5 Query Functions ......................................................................................................................................... 10
   5.1 Getting General Device Information ............................................................................................... 10
   5.2 Getting the Device Status ............................................................................................................... 11
   5.3 Getting Other RF Parameters ........................................................................................................ 11
   5.4 Retrieving the Device Temperature ................................................................................................. 12
6 Calibration Functions ............................................................................................................................ 12
   6.1 Obtaining Calibration Data ............................................................................................................ 12
      6.1.1 Structured Calibration Data Format ....................................................................................... 12
      6.1.2 Reading Formatted Data ......................................................................................................... 13
      6.1.3 Reading Raw Calibration Data ............................................................................................... 14
   6.2 Configuring the Gain of the Device Using Calibration .................................................................... 14
7 General Functions ...................................................................................................................................... 16
7.1 Writing to the User EEPROM ................................................................. 16
7.2 Reading from the Calibration and User EEPROMs .................................. 16
7.3 Self-Calibration of the Synthesizer .......................................................... 17
7.4 Write Registers ..................................................................................... 17
7.5 Read Registers ..................................................................................... 18
Revision Table.............................................................................................. 19
1 Introduction

The SC5317A and SC5318A are C to K broadband single stage downconverters, with input RF range from 6 GHz to 26.5 GHz, external LO frequency range from 6 GHz to 14 GHz, and output IF range from 50 MHz to 3 GHz. These modules feature an internal synthesized local oscillator, RF preamplifier, and variable gain control, making them compact and versatile modules. With the option for an external LO signal, the SC5317A and SC5318A may be configured for SISO applications or paired together with multiple units for MIMO applications such as ground-based satellite communications, point-to-point radio, and test instrument systems.

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 conversion parameters, obtaining gain correction using calibration data, and putting the device into power standby.

This manual will explain each function in detail, including the purpose of the function 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 SC5317A is a PXIe based product, while the SC5318A is controlled through USB and RS232. Each method of communication requires its unique set of header files, dynamic linked libraries (DLL), and system 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>
<thead>
<tr>
<th>Device/Interface</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down Converter</td>
<td>DOWNCONVERTER</td>
</tr>
<tr>
<td>PXI</td>
<td>PXI_DEVICE_TYPE</td>
</tr>
<tr>
<td>USB</td>
<td>USB_DEVICE_TYPE</td>
</tr>
<tr>
<td>RS232</td>
<td>RS232_DEVICE_TYPE</td>
</tr>
</tbody>
</table>

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 applied 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, followed by the interface, and ending with the function description such as “sc5318a_usbSetFrequency”. In this document, all function descriptions will leave out the product and interface description so that “SetFrequency” is used to represent all interfaces. All functions are of call type __stdcall in Windows™.

2.2 Compiling Code in C/C++

The header files are shared between USB and RS232 interfaces. To successfully use the header files to write applications, proper macros must be defined prior to compilation of the code.
Identifying, Opening, and Closing Devices

The SC5317A and SC5318A 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 serial number cannot be found, there is a function to obtain the current devices connected to the host computer. The SearchDevice() function scans the host computer for converter devices. If found, a list containing its serial number is returned. The function is declared as;

```c
SCISTATUS SearchDevices(char **serialNumberList,
                         int *numberOfDevices);
```

The **serialNumberList is a 2D array format [number of devices, serial number length + 1], and *numberOfDevices is the number of devices detected and available for connection.

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

```c
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.
   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 it is not in use. The following code lines show how to deallocate the memory used to hold the serial numbers.

```c
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.

```c
SCISTATUS status;
HANDLE deviceHandle;
Status = DeviceOpen("<serial number string>",
                   &deviceHandle);
```

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.

```c
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 the following code example.

```c
#define RESET_STATE 1;
#define CURRENT_STATE 0;
Status = InitDevice(deviceHandle,
                     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, attenuation, filters, and signal path.
4.1 Setting the Frequency at the Ports

These devices are single stage converters with one LO. There are 2 methods of setting the frequency of the LO for the proper conversion of an RF signal down to IF:

1. Set the values of RF and IF signals; set converted signal to spectrally inverted if needed.
2. Directly write the value of LO.

To choose how the LO is set, use the function SetLoSource():

```c
SetLoSource(HANDLE deviceHandle, 
    uint8_t externalLo, 
    uint8_t loMode, 
    uint8_t loDoubler, 
    uint8_t extLoSelect)
```

- **ExternalLo** – This parameter must be set to 0 to activate and power on the device’s internal LO. Setting it to 1 will indicate to the device that an external LO source is being used. This will power down the internal LO and deactivate any functions that try to use it.
- **loMode** – When `externLo` is set to 0 this mode will activate and if it is set to 0 the LO frequency will be calculated based on the RF and IF values. When set to 1, the LO frequency is directly programmed; see below.
- **loDoubler** – When `externLo` is set to 1, it controls the frequency doubler.
- **extLoSelect** – When `externLo` is set to 1, the parameter selects whether the LO port selected is the front (0) or rear (1) port.

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 26.5 GHz and a lower limit of 6.0 GHz, while the IF port frequency has a settable upper limit of 3500 MHz and a lower limit of 50 Hz. Although these are functional limits, they may not represent the operational performance boundaries of the device. Consult the product user manual for more information.

The functions to change frequency may be programmed as follows.

```c
double rf_frequency = 12.8e9;
double if_frequency = 1.2e9;

status = SetLoSource(deviceHandle, 0, 0, 0, 0);
status = SetFrequency(deviceHandle, 
    rf_frequency);
status = SetIfFrequency(deviceHandle, 
    if_frequency);
```

When LO is selected in `loMode`,

```c
double lo_frequency = 11.6e9;
status = SetLoSource(deviceHandle, 0, 1, 0, 0);
```
status = SetLoFrequency(deviceHandle, lo_frequency);

4.2 Setting the Attenuators

These devices have 5 sets of programmable attenuators, including 2 in the RF input section, 2 in the final IF output section, and 1 in the external IF2 input section. The sets of RF attenuators, the external IF2 attenuator, and the first of the final IF attenuators have 1 dB step resolution, while the second of the final 2 IF attenuators has 0.25 dB step resolution. Numbers represent these attenuators as defined in the header files. The numbers are listed here for reference.

```
#define RFATTEN 0
#define RFATTEN 1
```

Notice that IF1 and IF2 do not have any attenuators in the current hardware and are ignored. To set the attenuators to a certain value, use the function SetAttenuator(). As an example, the following code snippet sets the first RF attenuator to 20 dB and the final IF attenuator to 5.25 dB.

```
status = SetAttenuator(deviceHandle, RFATTEN, 20.00);
status = SetAttenuator(deviceHandle, IFATTEN, 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). As an example, for wide bandwidth of 320 MHz, the 3rd conversion stage is bypassed and IF2 is directly switched to the output IF3 port, such that the IF3 frequency is fixed at 1.25 GHz. With this configuration setting, programming the final IF frequency has no effect. The function SetSignalPath() is used to configure the device paths. It requires a structure input containing the configuration parameters in the following form.

```
typedef struct
{
    uint8_t bypassConverter;
    uint8_t rfAmpEnable;
    uint8_t ifOutEnable;
    uint8_t invertSpectrum;
} signalPathParams_t;
```

A brief description of what each structure parameter does is provided below. For more details, refer to the product user manual.
**Configuration Functions**

- **bypassConverter**: Enabling this will switch the RF port directly to the IF port, bypassing conversion.

- **rfAmpEnable**: The RF preamplifier setting; a value of 0 disables it and a value of 1 enables it.

- **ifOutEnable**: The IF output port may be disabled, especially when the direct path is selected. Disabling this port will attenuate the output signal by more than 50 dB.

- **invertSpectrum**: When the conversion is not bypassed, the spectrum at the IF port can be set to inverted relative to the RF input. This will affect the value of the LO frequency when RF and IF values are used.

To set the device to invert the conversion and also enable the RF preamplifier, use the following code.

```c
signalPathParams_t *pathParameters;
pathParameters->rfAmpEnable = 1; /* set the pre-amplifier */
pathParameters->bypassConverter = 0;
pathParameters->ifOutEnable = 1;
pathParameters->invertSpectrum = 1;

status = SetSignalPath(deviceHandle, 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. The amplifier is enabled by writing the following code.

```c
uint8_t preampStatus = 1;
Status = SetPreamp(deviceHandle, preampStatus);
```

### 4.4 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, medium, and high. For low levels of close-in carrier phase noise, select **HIGH**. The first local oscillator is an agile YIG-based synthesizer whose tuning speed may be improved by enabling fast tuning. Consult the user manual for more information on these synthesizer modes. The following code demonstrates how the settings are written.

```c
Enum LOOPGAIN pllLoopGain = NORMAL;

Status = SetSynthMode(deviceHandle, pllLoopGain);
```
4.5 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 */

status = SetReferenceClock(deviceHandle,
    lockExtEnable,
    pxi10ClkEnable, //only on PXIe
);
```

4.5.1 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 -x3FF */
status = SetReferenceDac(deviceHandle,
    tcxoDac);
```

4.6 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 using the following code.

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

4.7 Setting the First LO Drive Path

The first synthesized local oscillator (LO1) drives by default the first mixer of the conversion path; however, it can be switched to drive an external device such as a mixer. To do this, the first parameter of the enabling function is set to 1. See the following code.

```c
unsigned char loPath = 1;
status = SetLoPath(deviceHandle,
    loPath);
```

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.

```c
deviceInfo_t deviceInfo;
status = GetDeviceInfo(deviceHandle,
```
The `deviceInfo_t` structure has the following members (see header files for more info).

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

- `deviceInterface` – 0 = unassigned, 1 = PXI/PXIe, 2=USB&SPI, 3=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.

```c
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 `scimjdef.h` header file 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 the following code.

```c
rfParams_t rfParams;
status = GetRfParameters(deviceHandle, &rfParams);
```

The structure of the `rfParams_t` is as follows.

```c
typedef struct {
    double frequency; /* the RF frequency */
    double if1Freq; /* the first IF freq*/
    double loFreq; /* the first IF freq for wide IF */
    attenuator_t atten; /* the values of the attenuators */
    signalPathParams_t rfPath /* the signal path config */
} rfParams_t;
```
5.4 Retrieving the Device Temperature

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

```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 with the device configuration such as signal path selection, attenuator values, and its temperature. 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 as follows.

```c
typedef struct calData_s
{
    float calTemp;
    float *tempCoeff;
    float *ifCalFreq;
    float *ifRelGainCal;
    float **ifAttenCal;
    float *bpRfCalFreq;
    float *bpAbsGain;
    float *rfCalFreq;
    float *rfNonInvAbsGainCal;
    float *rfInvAbsGainCal;
    float *rfAmpGainCal;
    float **rfAttenCal;
} calData_t;
```

The following are descriptions of each of the `struct` members:

- **RF Cal params**  
The RF response calibration includes the absolute gains of the device as a function of RF frequency under the following conditions.  
1. All attenuators are set to 0 dB
Amplifier is off
IF frequency is set at 1000 MHz

The RF preamplifier gain and the measured attenuation values at each state of the two RF attenuators is included. The data array is laid out as follows.

<table>
<thead>
<tr>
<th>rfCalFreq</th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>...</th>
<th>fN</th>
</tr>
</thead>
<tbody>
<tr>
<td>noninvert gain(f)</td>
<td>nig(f0)</td>
<td>nig(f1)</td>
<td>nig(f2)</td>
<td>...</td>
<td>nig(fN)</td>
</tr>
<tr>
<td>InvertAbsGain(f)</td>
<td>ig(f0)</td>
<td>ig(f1)</td>
<td>ig(f2)</td>
<td>...</td>
<td>ig(fN)</td>
</tr>
<tr>
<td>rfAmpGain(f)</td>
<td>ag(f0)</td>
<td>ag(f1)</td>
<td>ag(f2)</td>
<td>...</td>
<td>ag(fN)</td>
</tr>
<tr>
<td>rfAtten 1dB(f)</td>
<td>A_a1(f0)</td>
<td>A_a1(f1)</td>
<td>A_a1(f2)</td>
<td>...</td>
<td>A_a1(fN)</td>
</tr>
<tr>
<td>rfAtten 2dB(f)</td>
<td>A_a2(f0)</td>
<td>A_a2(f1)</td>
<td>A_a3(f2)</td>
<td>...</td>
<td>A_a2(fN)</td>
</tr>
<tr>
<td>ifAttten 30dB(f)</td>
<td>A_a30(f0)</td>
<td>A_a30(f1)</td>
<td>A_a30(f2)</td>
<td>...</td>
<td>A_a30(fN)</td>
</tr>
</tbody>
</table>

IF Cal Params

Contains the relative measured values corresponding to the attenuation states of the attenuators as a function of IF frequency. The array has the following layout.

<table>
<thead>
<tr>
<th>ifCalFreq</th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>...</th>
<th>fN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifRelgain(f)</td>
<td>g(f0)</td>
<td>g(f1)</td>
<td>g(f2)</td>
<td>...</td>
<td>g(fN)</td>
</tr>
<tr>
<td>ifAtten 1dB(f)</td>
<td>A_a1(f0)</td>
<td>A_a1(f1)</td>
<td>A_a1(f2)</td>
<td>...</td>
<td>A_a1(fN)</td>
</tr>
<tr>
<td>ifAtten 2dB(f)</td>
<td>A_a2(f0)</td>
<td>A_a2(f1)</td>
<td>A_a3(f2)</td>
<td>...</td>
<td>A_a2(fN)</td>
</tr>
<tr>
<td>ifAttten 30dB(f)</td>
<td>A_a30(f0)</td>
<td>A_a30(f1)</td>
<td>A_a30(f2)</td>
<td>...</td>
<td>A_a30(fN)</td>
</tr>
</tbody>
</table>

ConvBpCal

This is the calibration of the direct RF to IF3 path, bypassing the conversion stage altogether. The path does go through the IF3 Atten #2 and the final IF amplifier. The array format is as follows.

<table>
<thead>
<tr>
<th>bpFreq</th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>...</th>
<th>fN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpAbs Gain(f)</td>
<td>g(f0)</td>
<td>g(f1)</td>
<td>g(f2)</td>
<td>...</td>
<td>g(fN)</td>
</tr>
</tbody>
</table>

tempCoeff

This 1D array holds the 2nd order temperature coefficients that are used to compensate for gain when the operating temperature drifts away from the factory calibration temperature. Gain variations are not only a function of temperature, but also a function of frequency, so 3 regions were measured. The format is as follows.

<table>
<thead>
<tr>
<th>Bands</th>
<th>6-13 GHz</th>
<th>13-20 GHz</th>
<th>20-26 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coeff1</td>
<td>C1</td>
<td>C2</td>
<td>C1</td>
</tr>
</tbody>
</table>

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
The following code snippet demonstrates how formatted data is read from the device.

```c
scimjdefs.h header files. The following code snippet demonstrates how formatted data is read from
the device.

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 ---

/* or simply use the memory location function
   AllocateCalDataMemory(calData);
*/
status = GetCalData(devicehandle,
    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 as 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 and the other to convert it to
formatted data.

unsigned char *rawData;
rawData = (unsigned char*)calloc(RAWDATALEN, sizeof(char));

status = GetRawCalData(deviceHandle,
    rawCalData
);
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 0x298 and 0x55FF. See the calibration EEPROM map of the product
hardware for more details.

6.2 Configuring the Gain of the Device Using Calibration

The SC5317A and SC5318A 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. Consult the user manual for more information on how to set these attenuators to achieve desired performance. SignalCore has algorithms to compute the values of the attenuators such that the device is set up for its best desired performance. The function CalcAttenValue() 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.

If gainParams.autoCtrlRfAmp is set to 1, the function will alter the state of the RF amplifier and return its state in the signalPathParams_t structure.

The following code demonstrates how the gainParams_t structure is used, along with frequency parameters, calibration data, and temperature, to compute the attenuator values required to configure the device to the calculated gain.

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;
gainParams.mixerLevel = -20.0;
gainParams.ifLevel = 0.0;
gainParams.linearMode = 0;
gainParams.autoCtrlRfAmp = 1;
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...

/* use SetSignalPath if using configure or reconfigure */
status = SetSignalPath(deviceHandle, pathParameters);

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 below.

...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 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 is the best way to approach it.

7 General Functions

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

7.1 Writing to the User EEPROM

This device has 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.

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 below 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 32-bit integer 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. Note that data is stored in the calibration EEPROM as little endian. The following is a method to convert the data to a string format.

```c
char snString[9]; /* 8 chars + termination */
sprint(snString, "%X", *(unsigned int*)receivedBytes);
```

### 7.3 Self-Calibration of the Synthesizer

**This function is not currently implemented** The synthesizer is calibrated at the factory and the calibration is sufficient for the circuitry to maintain lock with the calibration cycle of 2 years. By design, after factory calibration, the synthesizer should remain frequency locked for periods over 10 years, but only if its temperature does not deviate from its calibration temperature (typically about 42°C, ±5°C). 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 7 to 10 seconds 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);
```

### 7.4 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 an 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 = 2000000000;
status = RegWrite(deviceHandle, register, regData);
```
7.5 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

);
```

In the above snippet, `regByte` is the register address, `instructWord` specifies what returned data associated with the register is requested, and `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` as 1, see the following code.

```c
unsigned long long instruct = 1;
unsigned long long receivedData;

status = RegRead(deviceHandle,
                 GET_DEVICE_PARAM,
                 instruct,
                 &receivedData

);
```
## Revision Table

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>06/22/18</td>
<td>Document Created</td>
</tr>
<tr>
<td>1.0</td>
<td>09/12/18</td>
<td>First Released Version</td>
</tr>
<tr>
<td>1.1</td>
<td>09/24/18</td>
<td>Edited for clarity</td>
</tr>
</tbody>
</table>