Programming Manual

SC5317A & SC5318A

6 GHz to 26.5 GHz RF Downconverter

Rev 2.0

www.signalcore.com

Table of Contents

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 13.5 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 to the software API with a focus on the Windows^{TM} operating system using the C/C++ programming languages. It is also helpful for other languages such as LabVIEW and Python, that call on the API DLL. The API is common for both the SC5317A and SC5318A, and also handles three of the 4 methods of communication interfaces: the PXIe, USB, and RS232. It does not handle communication through the SPI bus. This document is structured into sections that describe the generic use of the product's functions such as searching for available devices, opening and closing a device, querying the device parameters, configuring the device, 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 effectively use a function.

2 API

The 2nd revision of the software API is different from previous version and its sub-versions. This later version uses a single API to handle all three communication interfaces, name PXI, USB, and RS232, instead of three separate API, one for each interface, as in the previous version. The SC5317A is a PXIe based product, while the SC5318A is controlled through USB and RS232. The internal project name for this product family is "Mockingjay II" so the API library is named "**mj2**" accordingly. Table 1 shows the files needed to write a user application. See the C/C++ example in the software installation directory.

Table 1. Software Architectures

User Level	API
1. userapp.c : user main code 2. sci_types.h 3. sci_errors.h 4. mj2_def.h : definitions 5. mj2_regs.h : registers 6. mj2_advance.h : optional API functions 7. mj2_functions.h : all API functions	1. mj2.dll : API 2. mj2_visa.dll : for PXI using NI-VISA driver

2.1 API Function Header Files

The API function names are prefixed with mj2_, which will be dropped in this document, and the function prototypes are listed in the mj2_function.h header file.

The `mj2_advance.h` file contains functions that provide more information about the device calibration data and other initial device parameters stored in device memory.

The `regs.h` files contains all the registers for direct read and write to the device registers via the `RegRead()` and `RegWrite()` functions.

The `mj2_def.h` file contains definitions of constants, types, and structures that are used by the API.

In this version of the API, functions that deal with calibrated data has changed to make the API easier to interface with other programming languages such as LabVIEW, Python, and MATLAB. In the previous version, calibration data was required to be passed from function to function, whose structure was unrealizable in other languages other than C/C++. In this version we made the data hidden from the user and accessible to functions internally.

2.2 Using the NI-VISA API

During the installation process of the software, the user may select to use one of two kernel level drivers for their SC5317A PXI device, namely SCIPCI and NI-VISA drivers. If the NI-VISA driver is chosen, the SC5317A will use NI-VISA to communicate with the instrument via the `mj2_visa.dll` API. Furthermore, when the NI-VISA driver is chosen, the device will appear on NI_MAX and discoverable from a VISA resource name object.

3 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 serial number string consists of 8 HEX format characters such as `100E4FC2`. With NI-VISA, the number string is the VISA resource name with maximum character length of 32.

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 SC5317A or SC5318A devices. If found, a list containing the number of devices and their corresponding serial numbers is returned. The function is declared as:

```
SCISTATUS SearchDevices(sci_comm_interface_t comm_interface, char
**serialNumberList, int *numberDevices);
```

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 `comm_interface` is either PXI (0), USB (1) or RS232 (2). 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+1));
/*
  MAXDEVICES is the number of devices to allocate memory for.
  SCI_SN_LENGTH is defined 0x08.
*/
  Status = SearchDevices(USB, serialNumbers, &nDevices);
  if(status != SCI_SUCCESS)
  ...error handling, free allocated memory...
```

Allocated memory for device serial numbers may be free when no longer used, 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(USB, "<sn or resource string>",
                     &deviceHandle
                     );
```

The "`<sn or resource 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 green. This `DeviceOpen()` call does not apply any other changes to the device; its working state remains unchanged by the command. The device handle is returned by the function on completion. This handle is then used by other functions to target this particular device and not any other devices of the same type. Once a device is opened, it will not be available for another session trying get access to it until the device session is closed.

3.3 Disconnecting from and Closing a Device

When the device is no longer in use, the application should disconnect from it. 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, and the device is available for another program or new session to open.

```
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.

```
#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()`:

```
SetLoSource(HANDLE deviceHandle,
uint8_t external_lo,
uint8_t lo_mode,
uint8_t lo_doubler,
uint8_t ext_lo_path)
```

- `external_lo` – 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 `external_lo` 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.
- `lo_doubler` – When `externalLo` is set to 1, it controls the frequency doubler.
- `ext_lo_path` – When `externalLo` is set to 1, this parameter selects whether the LO port selected is the front (0) or rear (1) port.

An important point to note is, if the external LO is selected, the device will turn off the internal synthesizer and as a result the power consumption is reduced significantly.

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. Please consult the product Hardware Manual for more information.

The functions to change frequency may be programmed as follows.

```
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`,

```
        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:

```
        #define   RFATTEN   0
        #define   IFATTEN   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. 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 a 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   bypass_conversion;
          uint8_t   rf_amp_enable;
          uint8_t   if_out_enable;
        uint8_t   invert_spectrum;
    } signal_path_params_t;
```

A brief description of each structure parameter's function is provided below. For more details, refer to the product Hardware Manual.

`bypassConverter`	Enabling this will switch the RF port directly to the IF port, bypassing conversion. If this is enabled, the internal synthesizer is turned off.
`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.

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

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

4.4 Configuring the RF amplifier

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.

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

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, 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 Hardware Manual for more information on these synthesizer modes. The following code demonstrates how the settings are written.

```
Enum LOOPGAIN pllLoopGain = NORMAL;

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

4.6 Configuring the Reference Clock

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

```
bool_t lockExtEnable = 1; /* enable locking to external source */

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

4.7 Adjusting the Internal TCXO clock

The device has a TCXO time-base, 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:

```
unsigned int tcxoDac = 0x2E0A; /* value range of 0x00 to -x3FF */
status = Set Reference Dac(deviceHandle,
                             tcxoDac);
```

4.8 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.

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

4.9 Set Device on Standby

The function `SetStandby()` places the device in power standby ther by reducing the power consumption when it is not in active use.

```
uint8_t stdby = 1;  // enable
status = SetStandby(deviceHandle, stdby);
```

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.

```
deviceInfo_t deviceInfo;

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

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

```
typedef struct deviceInfo_s
{
  uint32_t    product_sn;
  float       fw_rev;
  float       hw_rev;
  uint8_t     dev_interface;
  scidate_t   cal_date;
  scidate_t   man_date;

} deviceInfo_t;
```

`dev_interface: 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 `device_status_t` structure into the following function.

```
Device_status_t deviceStatus;

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

The members of `device_status_t` will not be explicitly discussed here as there are many of them. Please read the `mj2_defs.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.

```
rf_params_t rfParams;

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

The structure of the `rf_params_t` is as follows.

```
typedef struct
{
     double rf_freq;            /* the RF frequency */
     double if_freq;            /* the first IF freq */
     double lo_freq;            /* the first IF freq for wide IF */
     attenuator_t  atten;       /* the values of the attenuators */
     signal_path_params_t  path_params; /* signal path config */
} rf_params_t;
```

5.4 Retrieving the Device Temperature

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

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

In applications, it is recommended to query the temperature of the device periodically. If the ambient temperature does not change significantly (< 2 ^{0}C), polling the temperature every 15-30 mins should be sufficient. When temperature is polled, a copy of it is retained in memory for gain correction calculations, which is discussed in the next section.

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 temperature changes. To compute the conversion gain accurately would require two sets of information: the device configuration and its calibration data. These functions are not required in in version 2 or higher of the software but are included for those that may be interested in using the calibration data outside of the API.

6.1 Obtaining Calibration Data

There are two 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.

```
typedef struct
{
    float cal_temp;
   float *temp_coeff;
    float *if_cal_freq;
   float *if_rel_gain_gal;
    float **if_atten_cal;
    float *bp_rf_cal_freq;
    float *bp_abs_gain_cal;
    float *rf_cal_freq;
    float *rf_usb_gain_cal;
    float *rf_lsb_gain_cal;
    float *rf_rmp_gain_cal;
    float **rf_atten_cal;
} cal_data_t;
```

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

`RF Calibration Parameters`

The RF response calibration includes the absolute gain of the device as a function of RF frequency under the following conditions:

1. All attenuators are set to 0 dB
2. Amplifier is off
3. 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:

Rf_cal_freq	f0	f1	f2	...	fN
(usb)noninvert_gain(f)	nig(f0)	nig(f1)	nig(f2)	...	nig(fN)
(lsb)Invert_gain(f)	ig(f0)	ig(f1)	ig(f2)	...	ig(fN)
rfAmpGain(f)	ag(f0)	ag(f1)	ag(f2)	...	ag(fN)
rfAtten 1dB(f)	A_a1(f0)	A_a1(f1)	A_a1(f2)	...	A_a1(fN)
rfAtten 2dB(f)	A_a2(f0)	A_a2(f1)	A_a3(f2)	...	A_a2(fN)
:	:	:	:	...	:
rfAtten 30dB(f)	A_a30(f0)	A_a30(f1)	A_a30(f2)	...	A_a30(fN)

IF Calibration Parameters

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:

If_cal_freq	f0	f1	f2	...	fN
ifRelgain(f)	g(f0)	g(f1)	g(f2)	...	g(fN)
ifAtten 1dB(f)	A_a1(f0)	A_a1(f1)	A_a1(f2)	...	A_a1(fN)
ifAtten 2dB(f)	A_a2(f0)	A_a2(f1)	A_a3(f2)	...	A_a2(fN)
:	:	:	:	...	:
ifAtten 30dB(f)	A_a30(f0)	A_a30(f1)	A_a30(f2)	...	A_a30(fN)

Bypass Calibration Parameters

This is the calibration of the direct RF to the IF 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:

bp_cal_freq	f0	f1	f2	...	fN
bp_abs_gain_cal(f)	g(f0)	g(f1)	g(f2)	...	g(fN)

Temperature Coefficients

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. For this reason, 3 regions were measured. The format is as follows:

Bands	6 GHz - 13 GHz		13 GHz - 20 GHz		20 GHz - 26 GHz	
Coeff1	C1	C2	C1	C2	C1	C2

6.1.2 Reading Formatted Data

Memory must be allocated for the members of `struct cal_data_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 `mj2_defs.h` header files. The following code snippet demonstrates how formatted data is read from the device:

```
cal_data_t *calData;
calData->rf_gain_amp=(float**)calloc(RFCALFREQLEN,sizeof(float*));
calData->rf_atten_cal
=(float**)calloc(CalATTENSTEPS,sizeof(float*));

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

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

/* or simply use the memory location function
  status = mj2adv_AllocateCalDataMemory(calData);
*/
status = GetCalData(    devicehandle,
                        calData
                      );
/*!
--- After cal data has been used and not required anymore its memory
can be free with : */

/*
status = mj2adv_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 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 = mj2adv_GetRawCalData( deviceHandle,
                              rawCalData
                           );
      status = mj2adv_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 Automatic Configuration 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 in the API to compute the values of the attenuators such that the device is set up for its best desired performance. The function `CalcAttenValues()` will compute the necessary attenuator values using calibration data to meet user input requirements such as frequency, nominal input and output power levels, and linear mode selection that were set using `SetGainCalcParams()`. If the device is set with the calculated attenuator values (see `SetAttenuator()`function) the gain of the device is calculated using the `CalcGain()` function.

If `gain_params.auto_amp_ctrl` is set to 1, the `CalcAttenValues()` function may alter the state of the RF amplifier state to meet the user requirements and return its state in the `rf_params_t` structure when function `GetRfParameters()`is called.

The following code demonstrates how the `gain_params_t` structure is used, along with frequency parameters, to compute the attenuator values required to configure the device to the calculated gain.

```
gain_params_t gainParams;
signal_path_params_t pathParams;
attenuator_t *attenuator; /* receive attenuator values */

float gain; /* receive the computed gain */
/*
  Set the pathPaths if needed…
*/

gainParams.rf_level = -10.0;
gainParams.mixer_level = -20.0
gainParams.if_level = 0.0;
gainParams.linear_mode = 0; //using the mixer level for linearity
gainParams.auto_amp_ctrl = 1;

status = SetSignalPath(deviceHandle, pathParameters);
status = SetGainCalcParams(deviceHandle, &gainParams);
status = CalcAttenValues(deviceHandle, attenuator);
```

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

        /* next get the rf amplifier state using GetRfParameters()
        ** and then call SetPreamp() to set its state.

        /* calculate the gain */
        status = CalcGain(deviceHandle, &gain);
```

The `CalcGain()` function computes the expected gain of the current device configuration. Whenever an attenuator is changed, or the RF and IF values, or amplifier status, this function may be used to return the conversion gain of the state of the device.

The computed `gain` of the device is, approximately, the difference between the output IF level and the input RF level. The step resolution and errors of the attenuators limit how accurate the gain can be set, so `CalcGain()` function returns a value that is reflects will with the actual gain of the device for that setting. Notice, in the above example the gain value is not an 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 a suitable approach.

6.2.1 Fully Automated Setting

The example above requires the user to compute the attenuators and set them explicitly, as well as the RF pre-amplifier, which is not explicitly shown. The user would need to repeat this step to obtain the required attenuator values and pre-amplifier state whenever parameters such as RF or IF frequency values are changed in order to maintain a proper desired IF level.

An additional function was added in this version to automatically set the device whenever RF and IF frequencies, RF and IF levels, or linearity mode are changed. This function is called `SetAutoConversion()`, which is similar `SetGainCalcParams()`that it takes in the `gain_params_t` but also with an additional argument call `auto_conv_enable`. This argument if set to 1 turns on the described functionality. As an example, when the RF frequency is changed, the device automatically adjusts the attenuators to keep the gain relatively constant. `CalcGain()` may be called to obtain a more precise gain at the new RF frequency setting. The usage of the function is exemplified in the LabVIEW program (Software Front Panel) that is located in the product installation folder.

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.

```
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.

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

7.3 Self-Calibration of the Synthesizer

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 of more than 10 years, but only if its temperature does not deviate from its calibration temperature (typically about 42°C, ±5°C). The procedure may be run more frequently to ensure the circuit is always optimized despite changes in component characteristics over time and temperature. Once this function is executed, the user application should wait for 7 to 10 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.

```
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 `scimj2regs.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.

```
unsigned char register = RF_FREQUENCY;
unsigned long long regData = 20000000000000; //20 GHz (in mHz)
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 `mj2functions.h` header file).

```
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.

```
unsigned long long instruct = 1;
unsigned long long receivedData;
double if_freq;
status = RegRead(deviceHandle,
                   GET_DEVICE_PARAM,
                   instruct,
                   &receivedData);
If_freq = (double)receivedData * 0.001; // mHz to Hz
```

Revision Table

Revision	Revision Date	Description
0.1	06/22/2018	Document Created
1.0	09/12/2018	First Released Version
1.1	09/24/2018	Edited for clarity
1.2	08/17/2020	Edited formatting to match other Programming Manuals
1.3	9/07/2022	Corrected definition from header file
2.0	12/5/2022	Updated with version 2 of the software API