Programming Manual

SC5319A & SC5320A

20 GHz to 40 GHz RF Downconverter

Preliminary Rev 1.0

www.signalcore.com

Table of Contents

1 Introduction

The SC5319A and SC5320A are C to K broadband single stage downconverters, with input RF range from 20 GHz to 40 GHz, external LO frequency range from 10 GHz to 20 GHz, and output IF range from 100 MHz to 5 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 SC5319A and SC5320A 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. Contact SignalCore for Linux code using the USB interface. 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 effectively use a function.

2 Driver Architecture

The SC5319A is a PXIe based product, while the SC5320A is controlled through USB and RS232 or SPI. The code name for this product is MockingJay III, shortened to mj3, and this is the name prefix used for the function names of the API.

The software flow to writing a user application is shown in Table 1. At the highest level, where the user application resides, are the user code and header file(s) (*.h) for the device API. The level below that has the device API DLL and Library, which are called by the application level. The lowest level is where the device system driver or the kernel level drivers (*.sys) resides.

Table 1. Software Architectures

	Files	Interface
User application	userapp.c	
Headers	sci_errors.h sci_types.h mj3_defs.h mj3_functions.h	All
API	mj3.dll mj3.lib	All
Driver	Libusb-1.0.dll Winusb.sys	USB
	scipcioxi.sys	PXI

2.1 API Function Names and Call Type

All three interfaces will use the same API. The function names are prefixed with mj3_, such as “mj3_SetFrequency”. All functions are of call type __cdecl for win32 in Windows™.

2.2 Compiling Code in C/C++

All necessary header files must be included to successfully compile application code that uses the API. Table 2 list all these header files.

Table 2 API header files

Header files	Description
sci_types.h	Defined types used by the API
sci_errors.h	API error types
mj3_defs.h	API macros and structures
mj3_functions.h	API functions header
mj3_regs.h	List of device registers
mj3_advance.h	Advance functions (not commonly used)

The first 4 header files listed in Table 2 are necessary for all programs that use the API. The mj3_reg.h file is helpful if one decides to program the device using register calls. The mj3_advance.h lists lower-level functions that the primary functions listed in mj3_functions.h uses. The brief comments above the function names provide usage information. These mj3_function.h function prototypes are listed in appendix A.

Table 3 Library files and DLL

Header files	Description
mj3.lib	Required for all interfaces
mj3.dll	
libusb-1.0.lib	Required only for USB
libusb-1.0.dll	

3 Identifying, Opening, and Closing Devices

The SC5319A and SC5320A downconverters are identified by their unique serial numbers. This serial number is passed to the `mj3_OpenDevice()` function as a string in order to open a connection to the device and retrieve a device handle. 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 `mj3_SearchDevices()` function scans the host computer for SC5319A or SC5320A devices. If found, a list containing the number of devices and their corresponding serial numbers is returned. The function is declared as:

```
SCISTATUS mj3_SearchDevices(sci_comm_interface_t interface, char
**serial_number_list, int *number_devices);
```

The first parameter informs the function to search a particular interface: PCI_INT (0), USB_INT (1), or RS232_INT(2). The `**serial_number_list` is a list of serial number strings for PXI and USB interfaces, but it is a list of COM ports for RS232. The parameter `*number_devices` 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.
  SCI_SN_LENGTH is defined 0x09.
*/
  Status = mj3_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.

```
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 `mj3_DeviceOpen()` function. The function returns a HANDLE to the device that is needed by subsequent function calls to the device.

```
SCISTATUS status;
HANDLE deviceHandle;
Uint8_t baudrate = 0;
Status = mj3_DeviceOpen(USB_INT, "<serial number string>",
baudrate, &deviceHandle);
```

The "`<serial number string>`" of `type char` can be substituted by the `serialNumber[i]` as found in the previous code example or typed in directly. Upon successfully executing this function, the device **active LED** on the front panel will turn green. When `mj3_DeviceOpen()` is executed, it will retrieve the current device's rf parameters and hold it in host memory to be used by other functions but it does not apply any changes to the state of the device; its working state remains unchanged by this function. Note that the parameter `baudrate` is the rate for RS232 communication and the valid values are 0 and 1, denoting rates of 57600 and 115200 respectively. This parameter is ignored for PXIe and USB interfaces.

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 `mj3_DeviceClose()` function. Once it is executed, the **active LED** on the front panel will turn off, all device data will be freed, and the HANDLE to the device will no longer be valid for further use.

```
status = mj3_DeviceClose(deviceHandle);
deviceHandle = NULL;
```

3.4 Multiple Devices

Multiple devices may be opened simultaneously within one application. The `mj3_DeviceOpen()` function must be called for each of the devices using their respective interfaces and 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. When a device is "opened" by an instance, it cannot be accessed by another instance until `mj3_DeviceClose()` is called.

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 = mj3_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. Controlling the LO determines what the IF will be for a given RF or what the RF will be for a given IF. The relationship between LO, RF and IF is provided here:

$$LO = RF \pm IF$$

From the equation, another way of looking at it is, for a given LO, there can be two RF values that will result in the same IF. If RF < LO, we get the lower sideband where the IF spectrum is inverted. Choosing RF > LO results in a non-inverted spectrum at the IF with the upper sideband.

There are 2 ways to set the LO frequency: set it using RF, IF, and the selected sideband or set it directly. The first method sets the LO as follows:

$$LO = RF - IF \quad if\ upper\ sideband$$

$$LO = RF + IF \quad if\ lower\ sideband$$

To set the device to convert the RF to IF for a selected sideband, first set the sideband using the function `mj3_SetSideband()`. This is an important step. Not only will it help to properly compute the LO frequency, but it also switches in the proper RF hybrid circuit to suppress the unwanted sideband signal. Next call the `mj3_SetIfFrequency()` and `mj3_SetRfFrequency()`, the order is not important. To tune to the next RF, simply call `mj3_SetRfFrequency()`**;** the other functions do not need to be called if their values remain unchanged. The internal LO frequency is computed and set based on these function calls. The following code demonstrates this.

```
double rf_frequency = 23.5e9; // 23.5 GHz
double if_frequency = 1.25e9; // 1.25 GHz
uint8_t sideband = 0; //upper sideband

status = mj3_SetSideband(dev_handle, sideband);
status = mj3_SetIfFrequency(dev_handle, if_frequency);
status = mj3_SetRfFrequency(dev_handle, rf_frequency);
/* do something, wait, etc */
status = mj3_SetRfFrequency(dev_handle, rf_frequency + 100e6);
     //increment to downconvert 23.60 GHz
```

The internal LO frequency is obtained by fetching its value with the `mj3_FetchRfParameters()` function, which is discussed in detail in section 5. Note that this LO frequency is the value at the mixer port and is double the frequency of the internal synthesizer or an external source.

4.2 Configuring the Conversion Signal Path

The device has the option to bypass conversion by directing the input RF signal directly to the IF output port. The function `mj3_SetBypassConversion()` will enable/disable this behavior. When bypass is enabled, the internal synthesizer will go into standby mode, the attenuators will be set to the highest attenuation level, and the internal IF will be terminated to ground (disabled).

In the conversion path there are configurable gain adjustments via the step attenuators, rf pre-amplifier, programmable IF filter bandwidths, and conversion sideband options. Registers and corresponding API functions to control each of these choices are provided and are discussed next.

4.2.1 Enabling the RF Preamplifier

The RF preamplifier can be enabled or disabled using the `mj3_SetRfPreamp()` function and its usage is shown in the following code:

```
uint8_t preampEnalbe = 1;
Status = mj3_SetRfPreamp(deviceHandle, preampEnable);
```

4.2.2 Setting the Attenuators

These devices have 2 programmable attenuators: one in the RF input section, and the other in the IF output section. The RF attenuator has 0.5 dB step resolution with a maximum attenuation value of 31.5 dB, while the IF attenuator has 0.25 dB step resolution with a maximum attenuation value of 31.75 dB. Numbers that represent these attenuators ,as defined in the header files, are:

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

To set the attenuators to a certain value, use the function `mj3_SetAttenuator()`. As an example, the following code snippet sets the first RF attenuator to 10 dB and the final IF attenuator to 5.25 dB.

```
status = SetAttenuator(deviceHandle, RFATTEN, 10.00);
status = SetAttenuator(deviceHandle, IFATTEN, 5.25);
```

4.2.3 Setting the IF Filter Bandwidth

The IF filter is a programmable 4-bit lowpass filter bank, with 16 levels of bandwidths. At the lowest value the bandwidth is about 3 GHz, and at the highest value the bandwidth is about 5 GHz. The function `mj3_SetIfFilter()` is called to set the desired bandwidth, and its usage is:

```
uint8_t filterBw = 10;
status = mj3_SetIfFilter(deviceHandle, filterBw);
```

4.2.4 Setting the IF Output

The IF output can be disabled or enabled, and when it is disabled, its signal is terminated to ground. If conversion bypass is enabled, setting the IF output to enable will be ignored because the device

cannot have both the bypass signal and the IF signal be at the output at the same time. The controlling function is `mj3_SetIfOutput()`.

4.2.5 Setting the IF Sideband

This function properly switches in the proper 90^{O} hybrid to select either the lower or upper sideband signal. Its selection also affects how the LO frequency is computed for a given RF and IF, as seen earlier in section 4.1. The function `mj3_SetSideband()` performs this task.

4.2.6 Automatic Configuration of the Conversion Gain

The user may manually configure the conversion gain (loss) of the converter device by adjusting the RF and IF attenuators, and enabling or disabling the RF preamplifier. The gain can be approximated by using the following equation:

$$Gain = Gain_{max} + Gain_{rfamp} - Atten_{RF} - Atten_{IF}$$

$Gain_{max}$ and $Gain_{rfamp}$ can be approximated from datasheet specifications or read back using the `mj3_FetchMaxGain()`function, see section 5. Care must be exercised when setting the attenuators and enabling the RF preamplifier, because the converter's linearity and signal sensitivity depends on their configuration. For example, having more attenuation in the IF section helps with reducing the gain and maintaining good signal sensitivity but it could degrade the device linearity. On the other hand, having more attenuation in the RF section improves linearity but degrades signal sensitivity. Enabling the RF preamplifier should only be done to improve the sensitivity when lower-level signals (< 30 dBm or less) are expected.

Another way to configure the device for gain is to use the automatic conversion gain function provided, which computes better gain accuracy (typical within 1.0 dB) because it uses frequency dependent calibrated data that is stored in the device memory. The function takes in user parameters and uses them to compute and set the attenuators and optionally the RF preamplifier to best achieve the linearity requirement and achieve the desired gain. These parameters, listed in a type-defined structure called `auto_conv_params_t`, are shown here:

```
typedef struct
{
float rf_level;           //expected rf level input
float mixer_level;        //max desired level at mixer input
float if_level;           //expected if level output
uint8_t linearity_mode;      //linear modes: 0=mixer level control,
            1=balance of snr and linearity, 2=better snr, best snr,
            better linearity, best linearity
uint8_t auto_amp_ctrl;       //allows the preamplifier state to be
            changed to achieve the desired results
uint8_t hw_auto_conv;     //hardware does the computation to set
            the device to achieve the desired results. Not
            recommended when using the API and should be set to 0.
} auto_conv_params_t;
```

When this structure, with its members being defined, is passed to the `mj3_SetAutoConversion()` function, the attenuator values are computed and set, and the RF preamplifier enabled or disabled accordingly. This function **must be called** when a new RF, IF or LO frequency, or a combination of these frequencies is changed to obtain the new gain value and have device properly configured at these new frequencies automatically. An example to use this function is listed below.

```
auto_conv_params_t conv_params;
float conv_gain;

conv_params.rf_level = -10; // -10 dbm expected input level
conv_params.mixer_level = -20; // max desired rf level at the mixer,
matters only if linearity_mode = 0
conv_params.if_level = 0; // 0 dbm desired output level
conv_params.linearity_mode = 1; // select a balance of sensitivity
and linearity
conv_params.auto_amp_cntrl = 1; // allows the algorithm to decide
on the state of the rf preamp
conv_params.auto_conv_enable = 1;

status = setautoconversion(devicehandle, &conv_params, &conv_gain);
```

The last parameter `auto_conv_enable`, when enabled, will continuously calculate, and apply attenuator values and preamplifier state to keep the gain as constant as possible, when frequency parameters, such as RF frequency and IF frequency, are changed.

4.3 Selecting the LO source

The converter has the option to use the internal synthesizer or an external source as the local oscillator (LO) by using the function mj3_SetLoSource(); If the source is selected internal, the internal synthesizer will turn on and get out of its standby mode, and if it is selected external the internal synthesizer will go into standby mode. When the internal synthesizer is selected, its signal is also routed out via the LO in/out port at half the frequency of the LO frequency.

4.3.1 Setting the Internal Synthesizer Mode

The `synth_mode_t` defined structure contained the synthesizer mode options:

```
typedef struct
{
  uint8_t lock_mode;
  uint8_t loop_gain;
  uint8_t auto_spur_suppress;
} synth_mode_t;
```

The lock_mode specifies whether the synthesizer uses either the (0) harmonic phase locking method or the (1) traditional integer-N locking method in the coarse frequency synthesis. The loop gain of the synthesizer can be changed to shape the phase noise spectral density of the signal. The default

(0) is the higher gain loop and other (1) is the lower gain loop. The auto_spur_suppress option causes the synthesizer to bounce from harmonic to integer-N locking method to lower spur levels of the specified LO frequency. The following code demonstrates how the settings are written.

```
synth_mode_t synthMode;
synthMode.lock_mode = 0; //harmonic mode
synthMode.loop_gain = 0; //normal gain
synthMode.auto_spur_suppress = 0; // keep the mode on harmonic for
better phase noise

status = mj3_setSynthMode(devicehandle, &synthMode);
```

4.3.2 Self-Calibrating the Internal Synthesizer

Over temperature and time, the resonance frequencies of the oscillators of the internal synthesizer may drift far enough that at certain frequency regions the phase lock loops may not be able to properly achieve lock, causing frequency errors. Should these occur, the internal self-calibration procedure may help to resolve this issue. The function to start the procedure is mj3_SynthSelfCalibrate(). It is important that no other communication takes place when the procedure has started to reduce error. The LO status LED flash red/amber/green during the calibration procedure and returns green after a few seconds when it had completed.

4.4 Configuring the Reference Mode

The reference configuration does 2 or 3 things, depends on the product. For a product with USB control the two parameters are `lock_external` and `ref_dir`, and for a PXIe product there is the additional `pxi10_enable`. The function to configure the reference is mj3_SetReferenceMode(), which takes all three parameters, however `pxi10_enable` is only used if the interface is PXI. The `lock_external` parameter causes the device to lock to an external 10 MHz reference source, while `ref_dir` sets the Reference port to either receive an external source signal or put out the device internal reference signal. An example below shows how this function is used:

```
uint8_t lockExt = 1; // enable locking to external source
uint8_t pxi10_enable = 0;
uint8_t ref_dir = 0; //set to receive

status = mj3_SetReferenceMode(deviceHandle, ref_dir, pxi1_enable,
lockext);
```

4.4.1 Adjustment to the Internal OCXO Clock

The device has a OCXO 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 OCXO as the reference. The current value of the DAC can be retrieve with the `mj3_FetchRfParameters()` function. Adjustments for this value adjusts the OCXO clock, calibrating its frequency.

```
uint16_t ocxoDac = 0x2E0A; /* value range of 0x00 to -x3FFF */
status = mj3_SetReferenceAdjust(deviceHandle, ocxoDac);
```

4.5 Saving the New Default State of the Device

The current operating state of the device, including the new OCXO DAC value as discussed above, can be stored as the device default by calling the `mj3_SetDefaultState()` function. Once this function is executed, the current state will be the device reset and power up default state. This is done by using the following code.

```
status = mj3_SetDefaultState(deviceHandle);
```

4.6 Register Write

Direct access to the device configuration registers is performed using the `mj3_RegWrite()` function.

```
mj3_RegWrite(HANDLE dev_handle,
             uint8_t reg_byte,
             uint64_t instruct_word);
```

The parameter `reg_byte` is the register address, and these addresses are provided in the `mj3_regs.h` header file. While the register addresses are found in the header file, their map and definition are provided in the Hardware Manual. The `instruct_word` parameter is an unsigned 64-bit data associated with the register. Using this function, the input RF frequency of the device can be programmed as follows:

```
uint8_t register = RF_FREQUENCY;
uint64_t reg_data = 28000000000; //28 GHz

status = RegWrite(deviceHandle,
                  register,
                  reg_data
                 );
```

Using the mj3_RegWrite() function to the change the device RF frequency will affect the device only but will not register with the API so it is not advised to directly access the device unless necessary, especially when other API functions are used.

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 = mj3_FetchDeviceInfo(deviceHandle,
                             &deviceInfo);
```

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

```
typedef struct device_info_t
{
   uint32_t product_serial_number;
   float hardware_revision;
   float firmware_revision;
   uint8_t device_interfaces;
   struct date
   {
      uint8_t year; // Add in the millennia 2000
      uint8_t month;
      uint8_t day;
      uint8_t hour;
   } man_date;
} device_info_t;
```

`Device_interfaces: 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.

```
device_status_t deviceStatus;

status = mj3_FetchDeviceStatus(deviceHandle,
                               &deviceStatus
                              );
```

The members of `device_status_t` will not be explicitly discussed here as there are many of them. Please read the `mj3_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 = mj3_FetchRfParameters(deviceHandle, &rfParams);
```

The structure of the `rfParams_t` is as follows.

```
typedef struct
{
  double rf_frequency;    // rf port frequency
  double if_frequency;    // if port frequency
  double lo_frequency;    // lo frequency
  uint16_t reference_dac;   //Reference DAC adjustment value
  conv_path_t path_params;   //conversion path information
  attenuator_t atten; // attenuators
  auto_conv_params_t conv_params; //auto_conversion parameters
} rf_params_t;
```

For details of the `rf_params_t` members, refer to the `mj3_defs.h` header file.

5.4 Retrieving the Device Temperature

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

```
float deviceTemp;
status = mj3_FetchTemperature(deviceHandle, &deviceTemp);
```

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

5.5 Retrieving the Conversion Gain

The calibrated conversion gain of the current device settings can be obtained with the following:

```
float convGain;
status = mj3_FetchConvGain(deviceHandle, &ConvGain);
```

5.6 Retrieving the Calibration Data

Should there be a need to view the calibration data used in the compute the calibrated conversion gain, the following does it.

```
cal_data_t *cal_data= (cal_data_t*)malloc(sizeof(cal_data_t));
status = = mj3_FetchCalData(deviceHandle, &cal_data);
```

5.7 Register Read

Directly requesting data from the device is performed using `mj3_RegRead()`. The function has the following form (from the `mj3_functions.h` header file).

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

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

```
uint64_t instruct = 1;
uint64_t receivedData;
double if_frequency;

status = mj3_RegRead(deviceHandle,
                  GET_DEVICE_PARAM,
                  instruct,
                  &received_data);

if_frequency = (double)received_data;
```

6 Advance Functions

These lower-level functions listed in the mj3_advance.h header file are used by the functions described earlier in sections 3 to 5, and have the prefix `mj3adv_`. They are listed here for the sake of completeness and usefulness for the user.

6.1 Reading the EEPROM

This device has an onboard EEPROM that contains device specific data and calibration. The following reads all the calibration data into a buffer.

```
uint32_t cal_start_add = CALDATAMEMADD;
uint32_t cal_len = CALDATALEN;
uint8_t eeprom = EEPROM0;
uint8_t    *byte_data_buf    =    (uint8*)malloc(sizeof(uint8_t)*
CALDATALEN);
status =mj3adv_ReadEeprom(deviceHandle, eeprom, cal_start_add,
            cal_len, byte_data_buf);
```

Another example is to read back the product serial number located at the EEPROM address 0x04.

```
uint32_t start_add = 0x04;
uint32_t dataLen = 4;
uint8_t received_bytes[dataLen];
status = ReadEeprom(deviceHandle, EEPROM0, start_add,
                    dataLen, received_bytes);
```

The serial number is a 4 byte number 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”, *(uint32_t*)received_bytes);
```

6.2 Storing the Auto Conversion Parameters

The function `mj3adv_SaveAutoConvParams()` is executed when the `mj3_DeviceClose()` function is called. It stores the auto conversion parameters into the device EEPROM memory so that on powerup or when the device instance is opened, they read into host memory. The user may call this function anytime to ensure these parameters are stored.

6.3 Obtaining the Auto Conversion Computed Attenuator Values

The function `mj3adv_AutoCalcAttenValues()` calculates the values of the attenuators used to set the device using the same auto-conversion algorithm in function `mj3_SetAutoConversion()`, however unlike `mj3_SetAutoConversion()` it does not apply the values to the device.

6.4 Fetch Raw Calibration Data

The function `mj3adv_FetchRawCalData()` reads the entire data buffer in bytes. It does the same job as illustrated in the `mj3adv_ReadEeprom()` example of section 6.1 above.

6.5 Converting Raw Data to Formatted Calibration Data

Raw calibration data is converted to formatted calibration data before it can be used to compute for conversion gain by calling `mj3adv_RawToCalData()`.

6.6 Allocating and Deallocating Memory for the Calibration Data

Data needs a space in memory so that it can be used by the application instance, and the function `mj3adv_AllocateCalDataMemory()` simplifies the task. When calibration data is no longer needed, it can be deallocated using `mj3adv_DeallocateCalDataMemory()`.

6.7 Handling Calibration Data Example

It is not necessary for the user to create another set of calibration data for the user application in addition to that which is already in API memory. However, should the user want to do this the following is an example code.

```
uint8_t* raw_data = (uint8_t*)malloc((CALDATALEN + 8) * sizeof(uint8_t));
cal_data_t* cal_data = (cal_data_t*)malloc(sizeof(cal_data_t));


status = mj3adv_FetchRawCalData(deviceHandle,  raw_data);

status = mj3adv_AllocateCalDataMemory(cal_data);

status = mj3adv_RawToCalData(raw_data, cal_data);

free(rawData);

//do something…

mj3adv_DeallocateCalDataMemory(cal_data);
free(cal_data);
```

Appendix A – mj3_functions.h

```
/*
 * Header functions for MockingJay III converters
 *
 *
 * Copyright (c) 2021 SignalCore Inc.
 *
 * Rev 1.0.0
*/

#ifndef MOCKINGJAY3_FUNCTIONS_H__
#define MOCKINGJAY3_FUNCTIONS_H__

#ifdef __cplusplus
extern "C"
{
#endif

/* Export Function Prototypes */


/* Function to find the serial numbers of all SignalCore device with the same product ID
   return:      The number of product devices found
   input:    comm_interface enum {USB, RS232, PXI}
   output:      2-D array (or pointers) to pass out the list serial numbers for devices found
   Example, calling function could declare:
      char **serial_number_list;
      sNList = (char**)malloc(sizeof(char*)*50); // 50 serial numbers
      for (i=0;i<50; i++)
         searchNumberList[i] = (char*)malloc(sizeof(char)*SCI_SN_LENGTH);
   and pass searchNumberList into the function.
*/
SCISTATUS __cdecl mj3_SearchDevices(sci_comm_interface_t comm_interface,
      char **serial_number_list, int *number_devices);

/* Same as above for LabVIEW calls
*/
SCISTATUS __cdecl mj3_SearchDevicesLV(sci_comm_interface_t comm_interface,
      char *serial_number_list, int *number_devices);

/* Function opens the target device.
   return:      pointer to device handle
   input:       -comm_interface enum {USB, RS232, PXI}
             -rate ignore for PXI and USB. For RS232 0=>baud=57600, 1=>baud=115200
             -devSerialNum is the product serial number. Product number is available on
             the product label.
*/
SCISTATUS __cdecl mj3_OpenDevice(sci_comm_interface_t comm_interface,
      char *devSerialNum, uint8_t rate, PHANDLE dev_handle);

/* Function  closes the device associated with the handle.
   return:      error code
   input:       device handle
*/
SCISTATUS __cdecl mj3_CloseDevice(HANDLE dev_handle);

/*    Register level access function prototypes
   ============================================================================================
*/

/*    Writing the register with via the USB device handle allocated by OpenDevice
   return: error code
   input: commandByte contains the target register address, eg 0x10 is the frequency register
   input: instructWord contains necessary data for the specified register address
*/
SCISTATUS __cdecl mj3_RegWrite(HANDLE dev_handle,
                     uint8_t command_byte,
```

```
                    uint64_t instruct_word);

/*    Reading the register with via the USB device handle allocated by OpenDevice
  input: commandByte contains the target register address, eg 0x10 is the frequency register
  input: instructWord contains necessary data for the specified register address
  output: receivedWord is the return data request through the commandByte and instructWord
*/
SCISTATUS __cdecl mj3_RegRead(HANDLE dev_handle,
                    uint8_t command_byte,
                    uint64_t instruct_word,
                    uint64_t *received_word);

/*    Product configuration wrapper function prototypes
  ================================================================================*/
/* Initializes the device
  return: error code
  input:       Mode   0: The device initializes to the power up state
                      1: The device reprograms all internal components to the current device
                         state
*/
SCISTATUS __cdecl mj3_InitDevice(HANDLE dev_handle, uint8_t mode);

/* Sets the device RF port frequency
return: error code
input:   frequency in Hz. If outside of this range, the return is OUTOFRANGE.
*/
SCISTATUS __cdecl mj3_SetRfFrequency(HANDLE dev_handle, double freq);

/* Sets the device RF port frequency
return: error code
input:   frequency in Hz. If outside of this range, the return is OUTOFRANGE.
*/
SCISTATUS __cdecl mj3_SetIfFrequency(HANDLE dev_handle, double freq);

/* Sets the device RF port frequency
return: error code
input:   frequency in Hz. If outside of this range, the return is OUTOFRANGE.
*/
SCISTATUS __cdecl mj3_SetLoFrequency(HANDLE dev_handle, double freq);

/* Sets the RF1 Synth pll mode. harmonic is best for phase noise
return: error code
input:   low_freq_mode: RF synthesizer to generate freqs down to 0) 25 MHz, 1) 50 MHz. Lower freq is
from DDS
      low_loop_gain: 0 normal loop gain, 1 low loop gain. Low loop gain generally gives better spur
suppression
      lock_mode: 0 = harmonic (default), 1=fractN
*/
SCISTATUS __cdecl mj3_SetSynthMode(HANDLE dev_handle, synth_mode_t *synth_mode);

/**   Sets the LO source (internal or external)
return: error code
input:   lo_select 0:internal 1:external (internal LO powers down)
*/
SCISTATUS __cdecl mj3_SetLoSource(HANDLE dev_handle, uint8_t lo_select);

/**   Sets the RF Preamplifier
return: error code
input:   enable
*/
SCISTATUS __cdecl mj3_SetRfPreamp(HANDLE dev_handle, uint8_t enable);

/**   Sets the RF or IF attenuators
return: error code
input:   attenuator 0 RF; 1 IF
      atten_value dBm
*/
SCISTATUS __cdecl mj3_SetAttenuator(HANDLE dev_handle,
```

```
                        uint8_t attenuator,
                        float atten_value);

/**   Sets to bypass conversion, RF to IF port directly
return: error code
input:   enable
*/
SCISTATUS __cdecl mj3_SetBypassConversion(HANDLE dev_handle, uint8_t enable);

/**   Enable IF path to output port
return: error code
input:   enable
*/
SCISTATUS __cdecl mj3_SetIfOutput(HANDLE dev_handle, uint8_t enable);

/**   Select the IF filter
return: error code
input:   enable
*/
SCISTATUS __cdecl mj3_SetIfFilter(HANDLE dev_handle, uint8_t filter_value);

/**   Select the lower or upper conversion sideband
return: error code
input:   side_band 0 = upper sideband, 1 = lower sideband
*/
SCISTATUS __cdecl mj3_SetSideband(HANDLE dev_handle, uint8_t side_band);


/**   Puts the device circuitry for into power standby mode.
return: error code
input: pwrdown_mode
      0: Take device out off power standby. If the device was in standby,
      the device will be reprogrammed to the previous state. The device
      channel needs about a second to stabilize.
      1: powers down only the LO section
      2: All power to the analog (LO and RF path) will be off
*/
SCISTATUS __cdecl mj3_SetDeviceStandby(HANDLE dev_handle, uint8_t cnvtr_standby, uint8_t
synth_standby);

/** set the reference clock behavior
return: error code
input:   lock_eternal: 1 locks the 100 MHz reference to the external 10 MHz source
      pxi10_enable: 1 exports PXI 10 MHz clock, value is ignore in USB devices
      ref dir: 1 force the internal ref out, 0 accepts an external reference
*/
SCISTATUS __cdecl mj3_SetReferenceMode(HANDLE dev_handle, uint8_t ref_dir,
      uint8_t pxi10_enable, uint8_t lock_external);

/** manually adjust the internal reference clock dac to adjust for frequency accuracy
return: error code
input: dac value (max 0x3FFF)
*/
SCISTATUS __cdecl mj3_SetReferenceAdjust(HANDLE dev_handle, uint16_t dac_value);

/**   Store the current state of the signal source into EEPROM as the default startup state
return: error code
input:   none
*/
SCISTATUS __cdecl mj3_StoreDefaultState(HANDLE dev_handle);

/**   Automatically sets up the device to achieve conversion according to autoconv parameters
* and returns the gain
return: error code
input:   conv_params struct
output: conversion gain
*/
SCISTATUS __cdecl mj3_SetAutoConversion(HANDLE dev_handle,
          auto_conv_params_t *conv_params, float *conv_gain);
```

```
/**   Self Cal of the VCO dac values of the harmonic loop and sum loop
return: error code
input:   cal_vco_select 0 = coarse vco, 1 = sum vco
*/
SCISTATUS __cdecl mj3_SynthSelfCalibrate(HANDLE dev_handle, uint8_t cal_vco_select);


/* Product Export Query (Read) function prototypes */
/*------------------------------------------------------------------------------------------------- */

/* Function retrives the rf parameters such as rf1&2 frequencies and other rf sweep components
return:  error code
output:  rf_parameters
*/
SCISTATUS __cdecl mj3_FetchRfParameters(HANDLE dev_handle, rf_params_t *rf_params);

/* Function retrives current temperature of the device
return:  error code
output:  temperature
*/
SCISTATUS __cdecl mj3_FetchTemperature(HANDLE dev_handle, float *temp);

/* Function retrives the device status - PLL locks status, ref clk config, etc see deviceStatus_t type
return:  error code
output:  device status
*/
SCISTATUS __cdecl mj3_FetchDeviceStatus(HANDLE dev_handle, device_status_t *device_status);

/* Function fetches the device Info
return:  error code
output:  deviceInfo      device information structure
*/
SCISTATUS __cdecl mj3_FetchDeviceInfo(HANDLE dev_handle, device_info_t *device_info);

/* Function fetches the conversion gain of the device with its current setup
return:  error code
output:  conversion gain
*/
SCISTATUS __cdecl mj3_FetchConvGain(HANDLE dev_handle, float* conver_gain);

/* Function fetches the max gain (preamp off) and the relative gain of the preamp of the device with
its current setup
return:   error code
output:   max_gain
      preamp_gain
*/
SCISTATUS __cdecl mj3_FetchMaxGain(HANDLE dev_handle, float* max_gain, float* preamp_gain);

/* Function fetches the device calibration data stored in memory
return:  error code
output:  pointer to calibration data
*/
SCISTATUS __cdecl mj3_FetchCalData(HANDLE dev_handle, cal_data_t **cal_data);

# ifdef __cplusplus
}
# endif

# endif  /* MOCKINGJAY3_FUNCTIONS_H__ */
```

Revision Table

Revision	Revision Date	Description
0.1	06/22/2020	Document Created
0.2	04/12/2021	First Preliminary version
0.3	011/12/2021	Edited for API changes
1.0	04/14/2022	Edited for release