SC5360B Manual

9.3 GHz Dual Channel Phase Coherent RF Downconverter

Core Module with USB and SPI

© 2015 SignalCore, Inc.

support@signalcore.com

Table of Contents

Theory and Operation ... 3
LO Generation ... 3
Device Standby ... 4
Communication Interfaces ... 4
SPI Interface ... 4
USB Interface ... 5
Auxiliary Connections ... 5
Device Registers ... 7
Serial Peripheral Interface (SPI) ... 12
Writing the SPI Bus ... 13
Reading the SPI Bus ... 14
USB Interface ... 15
Writing the Device Registers Directly ... 15
Reading the Device Registers Directly ... 15
USB Driver API ... 16
API Description ... 16
Example Code ... 20
LabVIEW support ... 20
Revision Notes ... 22

THEORY AND OPERATION

The SC5360B is a dual channel phase coherent X-band RF downconverter with each channel having dual conversion stages as indicated in Figure 1. The SC5360B was designed to detect very low level signals, typically less than -50 dBm. With no attenuation engaged, the maximum gain of each channel is about 63 dB. The 2 attenuators in each channel has 30 dB range with 1 dB step resolution, which allow a total gain variation of 60 dB. The input noise figure at maximum gain is typically less than 5 dB, putting the input noise density at almost -170 dBm/Hz.

Each conversion stage has a bandpass filter to minimize cross-talk between the 2 local oscillators, as cross-talk leads to possible intermodulation products being produced in-band. The final 140 MHz IF bandpass filter sets the IF bandwidth of the channel. To ensure further cross-talk isolation, each IF stage circuitry is contained in an EMI gasket lined cavity. EMI gasket lined cavities not only help intra-channel isolation, but also inter-channel isolation.

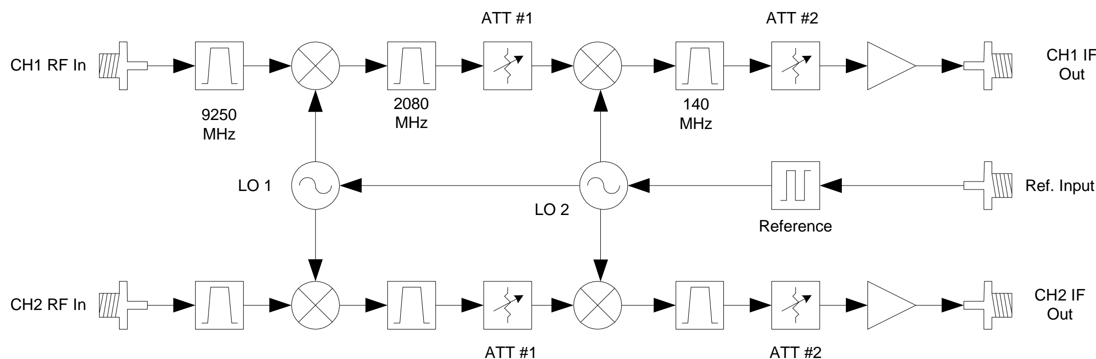


Figure 1. Simplified block diagram of the SC5360B.

LO GENERATION

The SC5360B local oscillator (LO) synthesizers reference an internal 100 MHz VCXO or an external reference source directly or indirectly. The device can be setup via its registers to directly pass an external 100 MHz to the LO synthesizers or phase lock the VCXO to it; see Figure 2 for details. When the device is set to reference the internal VCXO, its frequency accuracy is control via the reference DAC. On power up, if the internal VCXO is used, a factory calibrated DAC value sets the frequency accuracy to within 2.5 ppm. If the user needs to make fine adjustments, it could be done by writing a new value to the DAC via the REFERENCE_DAC register.

While the device is not programmed to phase lock to an external reference, connecting an external reference to the input port will force the LO to bypass the internal VCXO and use the external source directly. If the user wants to use the external source to drive the LO directly, ensure that its frequency is 100 MHz and its amplitude is in the 0-5 dBm range for best results.

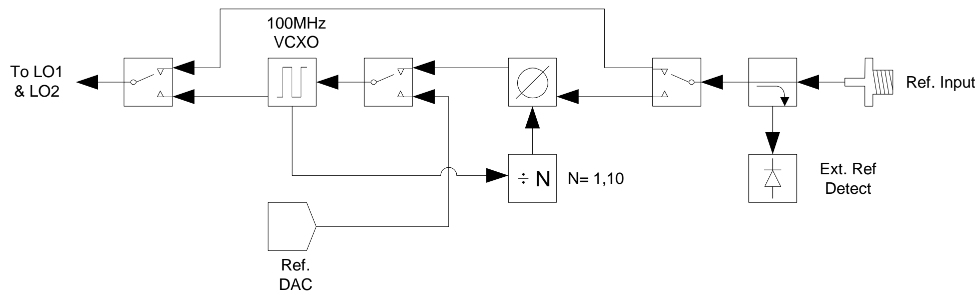


Figure 2 Reference section

The internal VCXO can be programmed to phase lock to the external source upon detection of its presence, this is an indirect use of the external source. This behavior is set via the REFERENCE register. There is also the option to phase lock to an external 100 MHz or 10 MHz.

The local oscillators synthesizers are based on single fractional-N type phase lock loops so to keep the fractional spurs down, only large step sizes are used; 5 MHz and 1 MHz. LO 2 is fixed while LO 1 is tunable over 600 MHz at 5 MHz or 1 MHz. The user may decide which step size to use in an application.

DEVICE STANDBY

Each section of the downconverter has an ultra low noise linear regulator, providing not just low supply noise but further signal isolation that could otherwise leak through common supply lines. Most of these regulators can be turn-off to reduce power consumption by invoking the DEVICE_STANDBY register (see the Device Registers section). On standby, the device power consumptions drops by more then 75%, and it will begin to cool down. When a device comes out of standby, it takes about a second to assume its last state. However, it may take up to 20 minutes to stabilize its temperature and hence its gain and frequency, if it does not use an external reference source.

COMMUNICATION INTERFACES

The SC5360B has both USB and SPI or RS-232 communication interfaces. While the USB interface is always active, only either SPI or RS-232 can be installed. Although both SPI and RS-232 uses the same connector, there are physical hardware differences internally between them so both cannot reside together. The choice of SPI or RS-232 must be set at the factory.

SPI Interface

The SC5360B uses a 9-pin micro-D subminiature connector for SPI communication with the device through a 4-wire serial peripheral interface. The pinout of this male connector, viewed from the RF connector side as shown below, is listed in Table 1. In addition to the 4-wire SPI lines, is the SerialReady line that indicates whether the device is ready to accept data or not. Detailed SPI read and write operations are discussed in detail in the *Serial Peripheral Interface (SPI)* section.

Table 1 Pinout of the SC5360B SPI communication connector

Pin Number	SPI Function	Description
1	SerialReady	Indicates when SPI is ready
2	MISO	Master Input - Slave Output (output from slave)
3	MOSI	Master Output Slave Input (output from master)
5	GND	Signal Ground
7	$\overline{CS}$	Chip Select (active low, output from master)
9	CLK	Serial Clock (output from master)

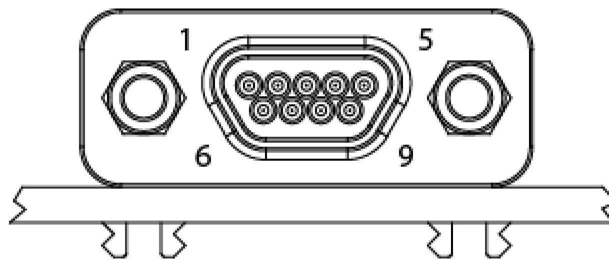


Figure 3 SPI/RS-232 connector

The SPI interface has 2 modes of operation; Modes 0 and 1. The mode is selected via pin 12 of the auxiliary connector; if pin 12 is left open or pulled high to 3.3V on power-up or reset, mode 1 is selected and is the default. Pulling it down to ground will put it into mode 0 upon power-up or reset. See the *Serial Peripheral Interface (SPI)* section and for more details.

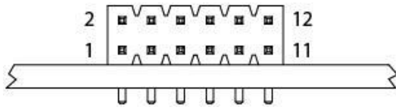
USB Interface

The SC5360B uses uses a mini-USB Type B connector for USB communication with the device using the standard USB 2.0 protocol found on most host computers. More information on the use of the USB interface and its software API are provided in the *USB Interface* section.

AUXILIARY CONNECTIONS

The SC5360B provides access to certain internal logic lines through a 2 mm pitch, 12-position header located underneath the “REF IN” port. The pinout of this header, viewed from the board edge is shown below.

Table 2 Auxiliary connector pinout



Pin	Function	Description
1,3,5,7,9,11	GND	Device ground
2	$\overline{\text{SYS RESET}}$	Resets device back to default settings. Normal pin state is high. Pin low then high will reset the device. 3.3VDC CMOS logic
4	Reserved	
6	PLL LOCK STATUS	Pin high indicates "power good" and all oscillators are phased-locked. Max current is 20 mA @ 3.3 VDC
8	Over Temp Status	Pin goes high if temperature of the device exceeds 75 ^{0}C.
10	Reserved	
12	SPI Mode	High = mode 1 Low = mode 0

DEVICE REGISTERS

Communication to the SC5360B is performed by writing to and reading from its set of control and query registers respectively. The control registers are used to set/configure the device, hence a one-way communication. The query registers on the other hand request the device to perform an operation with the expectancy of returned results, hence a two-way communication. The table below lists the device registers and provides the necessary details for each of them.

Table 3. Register 0x01 INITIALIZE (1 Byte write)

Bit	Type	Name	Width	Description
[7:0]	WO	Reserved (write zeros)	8	Initialize the device to the power up state

Table 4. Register 0x02 SET_SYSTEM_ACTIVE (1 Byte write)

Bit	Type	Name	Width	Description
[0]	WO	Activate the “active” LED	1	1 = turns the active LED on 0 = turns the active LED off
[7:1]	WO	Reserved	7	Set all bits to 0.

Table 5. Register 0x05 DEVICE_STANDBY (1 Byte write)

Bit	Type	Name	Width	Description
[0]	WO	Enable Standby	1	1 = Puts the device into standby mode. All power to the analog circuitry will be powered down, conserving power. 0 = Takes the unit out of standby, returning it to its previous state.
[7:1]	WO	Reserved	7	Set all bits to 0.

Table 6. Register 0x10 RF_FREQUENCY (4 Bytes write)

Bit	Type	Name	Width	Description
[31:0]	WO	Tuning Word	32	Sets the RF center frequency in MHz

Table 7. Register 0x12 ATTENUATION (2 Bytes write)

Bit	Type	Name	Width	Description
[7:0]	WO	Attenuation Value	8	Set the value of the attenuator in dB, valid values are 0 to 30.
[8]	WO	Attenuator Number	1	0 = IF Attenuator #1 1 = IF Attenuator #2

[9]	WO	Channel	1	0 = Channel 1 1 = Channel 2
[15:10]	WO	Reserved	6	Set all to zeros

Table 8. Register 0x14 REFERENCE_MODE (1 Byte write)

Bit	Type	Name	Width	Description
[0]	WO	Reference Mode	1	0 = The device will use its internal 100 MHz VCXO as the base reference. If an external 100 MHz (> 0 dBm) is connected to the input reference port and is detected by the device, the internal 100 MHz will power down and the external reference will drive the local oscillator synthesizers directly. Only 100 MHz external reference sources can drive the LO synthesizers directly. 1 = The device VCXO will phase lock to a detected external 100 MHz reference signal.
[2]	WO	Reference Frequency	1	0 = 100 MHz. 1 = 10 MHz.
[7:1]	WO	Reserved	6	Set all to zeros

Table 9. Register 0x15 REFRENCE_DAC (2 Bytes write)

Bit	Type	Name	Width	Description
[13:0]	WO	Reference DAC value	14	Adjust the center frequency of the internal VCXO
[15:14]	WO	Reserved	2	Set to zeros

Table 10. Register 0x16 IF_MONITOR (1 Byte write)

Bit	Type	Name	Width	Description
[0]	WO	Enable monitor port	1	1 = enables the IF monitor ports 0 = disables the IF monitor ports
[7:1]	WO	Reserved	7	Set to zeros

Table 11. Register 0x17 GAIN (2 Bytes write)

Bit	Type	Name	Width	Description

[13:0]	WO	Absolute gain value	14	The desired gain required by the device channel. The device will compute the attenuator settings automatically to set the device gain closest to the desired value.
[14]	WO	sign	1	0 = positive gain value 1 = negative gain value
[15]	WO	channel	1	0 = channel 1 1 = channel 2

Table 12. Register 0x18 TUNE_RESOLUTION (1 Byte write)

Bit	Type	Name	Width	Description
[0]	WO	Tune resolution mode	1	0 = Sets the device to tune in 5 MHz step 1 = Sets the device to tune in 1 MHz step
[7:1]	WO	Reserved	7	Set to zeros

Table 13. Register 0x19 CURRENT_STATE (1 Byte write, 4 Bytes read)

Bit	Type	Name	Width	Description
[2:0]	WO	Mode	3	0 = Stores the current configuration as default power-up state. 1 = Places the current RF frequency in MHz onto the output buffer to be read back by host. Return type is unsigned int. 2 = Places the current tune resolution in MHz onto the output buffer to be read back by host. Return type is unsigned int. 3 = Places the computed conversion gain of channel 1 onto the output buffer to be read back by host. Return type is float. 4 = Places the computed conversion gain of channel 2 onto the output buffer to be read back by host. Return type is float. 5 = Places the attenuator values onto the output buffer to be read back by host. Return type is unsigned int. The 4 bytes are contain the attenuator values as follow: [Ch2Atten2][Ch2Atten1][Ch1Atten2][Ch1Atten1]
[7:3]	WO	Reserved	5	Set to zeros
[31:0]	RO	Request data on buffer	32	Buffer data available on the USB endpoint and also on the SPI output buffer Register 0x1D

Table 14. Register 0x1A DEVICE_STATUS (1 Byte write, 2 Bytes read)

Bit	Type	Name	Width	Description
[7:0]	WO	Set to zeros	8	Places the device status data onto the output buffer to be read back by host.
[15:0]	RO	Read data (unsigned)	16	[15] Reference PLL status [14] LO1 PLL status [13] LO2 PLL Status [12] OverTemp Status. If the temperature of the device exceeds 75 degC, this bit is set to 1. [11] External reference detected status. If an external reference source of significant power is detected at the reference input port, this bit is set to 1. [10] Reference lock enabled. [9] IF monitor port enabled. [8] Device in standby mode. [7:0] reserved Data is on the first 2 return bytes on USB – USB should only read 2 bytes. Data is on the last 2 of the 4 bytes in the SPI output buffer.

Table 15. Register 0x1B TEMPERATURE (1 Byte write, 4 Bytes read)

Bit	Type	Name	Width	Description
[7:0]	WO	Set to zeros	8	Places the temperature data onto the output buffer to be read back by host.
[31:0]	RO	Temperature data	32	Returned type is float. Type cast to convert from unsigned int to float.

Table 16. Register 0x1C DEVICE_INFO (1 Byte write, 4 Bytes read)

Bit	Type	Name	Width	Description
[2:0]	WO	Info Type	3	Writing this register will place the requested contents into the output buffer. Contents are immediately available for USB read. The contents occupy effectively four bytes. In the case of SPI, contents are transferred to the serial output buffer, so a second query to the SERIAL_OUT_BUFFER register is required to transfer its contents and also to clear the output buffer. 0 = Obtains the product serial number 1 = Obtains the hardware revision

				2 = Obtains the firmware revision 3 = Obtains the manufacture date 4 = Obtains the calibration date (if cal available)
[7:3]	WO	Reserved	5	Set to zeros
[31:0]	RO	Requested Data	32	Data for the requested parameter: Product Serial Number – 32-bit unsigned Hardware Revision – typecast to 32-bit float Firmware Revision – typecast to 32-bit float Manufacture Date – unsigned 32-bit [31:24] Year (last two digits) [23:16] Month [15:8] Day [7:0] Hour Calibration Date – unsigned 32-bit [31:24] Year (last two digits) [23:16] Month [15:8] Day [7:0] Hour

Table 17. Register 0x1D SPI_OUTPUT_BUFFER (4 Bytes write, 4 Bytes read)

Bit	Type	Name	Width	Description
[39:0]	WO	Serial Out Buffer	40	Set all bits to 0. Use of this register is only available for the SPI interface.
[39:0]	RO	Request Data	40	The data clocked back are the contents requested by the 0x19, 0x1A, 0x1B or 0x1C registers.

Registers 0x19, 0x1A, 0x1B, and 0x1C are query or read-back registers. With SPI, the write-only portion of the register must be written first followed by reading register 0x1D to read back the requested data. See the Serial Peripheral Interface (SPI) section for details.

SERIAL PERIPHERAL INTERFACE (SPI)

The SPI interface is implemented using 8-bit length physical buffers for both the input and output, hence they need to be read and cleared before consecutive bytes can be transferred to and from them. The process of clearing the SPI buffer and decisively moving it into the appropriate register takes CPU time, so a time delay is required between consecutive bytes written to or read from the device by the host. The chip-select pin ($\overline{CS}$) must be asserted low before data is clocked in or out of the product. Pin $\overline{CS}$ must be asserted for the entire duration of a register transfer.

Once a full transfer has been received, the device will proceed to process the command and de-assert low the SRDY pin. The status of this pin may be monitored by the host because when it is de-asserted low, the device will ignore any incoming data. The device SPI is ready when the previous command is fully processed and SRDY pin is re-asserted high. It is important that the host either monitors the SRDY pin or waits for 500 us between register writes.

Register writes are accomplished in a single write operation. Register buffer lengths vary depending on the register; they vary in lengths of 2 to 5 bytes, with the first byte being the register address, followed by the data associated with that register. The ($\overline{CS}$) pin must be asserted low for a minimum period of 1 μs (T_S, see Figure 4) before data is clocked in, and must remain low for the entire register write. The clock rate may be as high as 5.0 MHz (T_C = 0.2 μs), however if the external SPI signals do not have sufficient integrity due trace issues then the rate should be lowered.

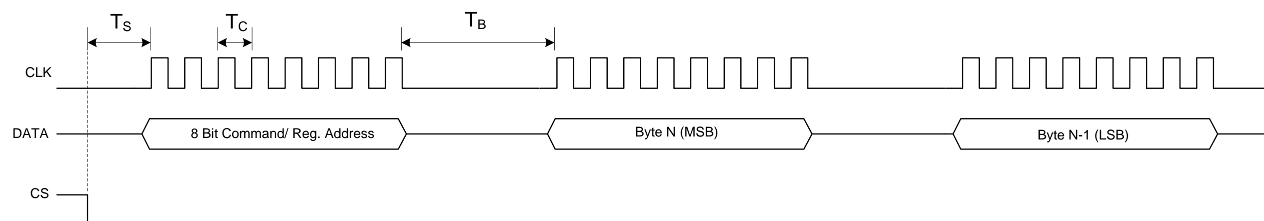


Figure 4 SPI Timing

As mentioned above, the SPI architecture limits the byte rate due to the fact that after every byte transfer the input and output SPI buffers need to be cleared and loaded respectively by the device SPI engine. Data is transferred between the buffers and the internal registers. The time required to perform this task is indicated by T_B, which is the time interval between the end of one byte transfer and the beginning of another. The recommended minimum time delay for T_B is 5 μs for write only registers, and 7 μs for query registers. The number of bytes transferred depends on the register. It is important that the correct number of bytes is transferred for the associated device register, because once the first byte (MSB) containing the device register is received, the device will wait for the desired number of associated data bytes. The device will hang if an insufficient number of bytes are written to the register. In order to clear the hung condition, the device will need an external hard

reset. The time required to process a command is also dependent on the command itself. Measured times for command completions are between 40 μs to 150 μs after reception.

There are two selectable modes of SPI operation available on the device. Leaving pin 12 of the *auxiliary connector* open or pulled high (3.3V), mode 1 (see Figure 5)is enabled at power-up or upon device reset. Jumping the pin to ground will enable mode 0 (see Figure 6) at power-up or upon reset. Once the mode is set, which typically takes a second after power-up, logic on pin 12 will no longer affect the device. In the default mode (mode 1) serial data in and out of the device are clocked on the falling edge of the SPI clock while the CS line is asserted active low. In mode 0, both data in and out are clocked on the rising edge of the SPI clock. Both modes require that the most significant bit (msb) is written first. The recommended clock rate for mode 0 is between 200 kHz and 5 MHz, and for mode 1 is between 100 Hz and 5 MHz.

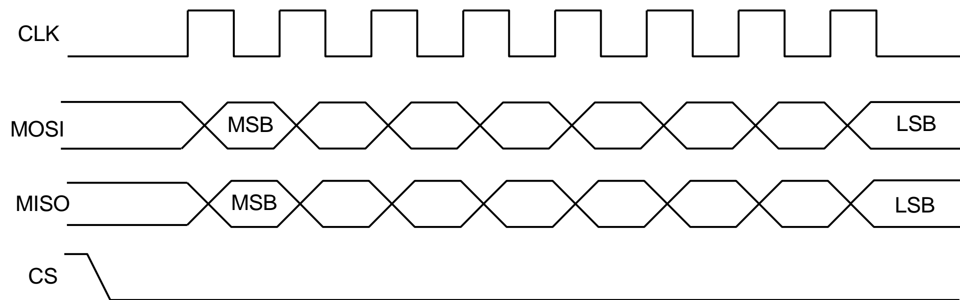


Figure 5 SPI Mode 1 – data clocked in/out on falling clock edge

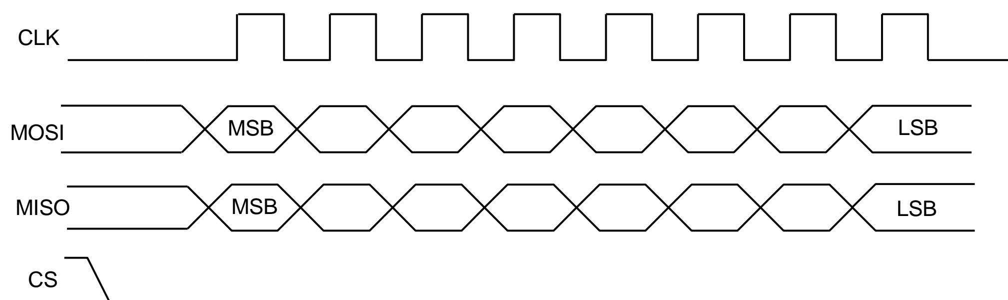


Figure 6 SPI Mode 0 – data clocked in/out on rising clock edge

WRITING THE SPI BUS

The SPI transfer size (in bytes) depends on the register being targeted. The MSB byte is the command register address as noted in the Device Registers section. The subsequent bytes contain the data associated with the register. As data from the host is being transferred to the device via the SDI (MOSI) line, data present on its SPI output buffer is simultaneously transferred back, MSB first, via the SDO (MISO) line. The data return is invalid for most transfers except for those registers querying for

data from the device. See *Reading the SPI Bus* section below for more information on retrieving data from the device. Figure 7 shows the contents of a single 3 byte SPI command written to the device. The Device Registers section provides information on the number of data bytes and their contents for an associated register. There is a minimum of 1 data byte for each register even if the data contents are "zeros".

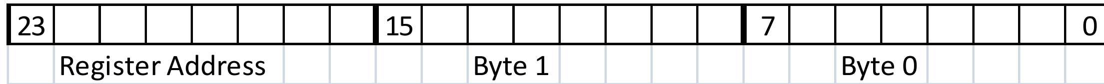


Figure 7 Write 3 bytes of data

READING THE SPI BUS

Data is simultaneously read back during a SPI transfer cycle. Requested data from a prior command is available on the device SPI output buffers, and these are transferred back to the user host via the SDO pin. To obtain valid requested data would require querying the SERIAL_OUT_BUFFER, which requires 5 bytes of clock cycles; 1 byte for the device register (0x1D) and 4 empty bytes (MOSI) to clock out the returned data (MISO). An example of reading the device temperature from the device is shown in Figure 8.

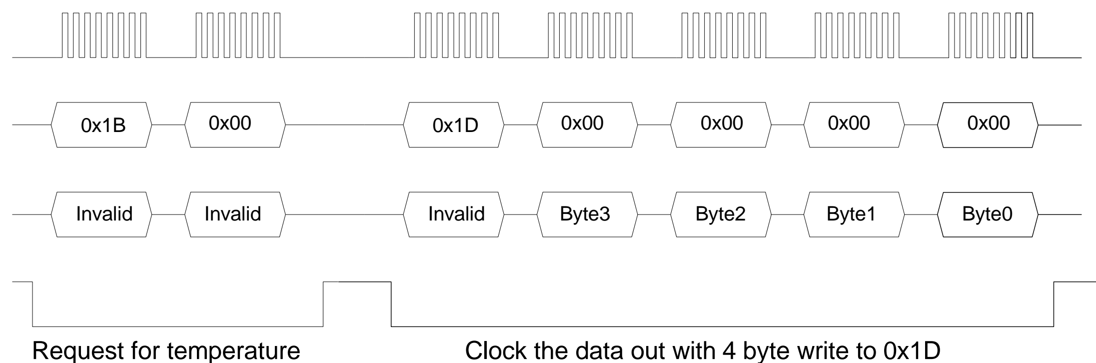


Figure 8 Example for reading back temperature data

In the above example, valid data is present in 4 bytes, however not all queries result in 4 valid bytes. Table 18 shows the valid data bytes associated with the querying register.

Table 18 Valid returned data

Register (Address)	Reg Code	Byte 3	Byte 2	Byte 1	Byte 0
CURRENT_STATE	0x19	Valid	Valid	Valid	Valid
DEVICE_STATUS	0x1A	Invalid	Invalid	Valid	Valid
TEMPERATURE	0x1B	Valid	Valid	Valid	Valid
DEVICE_INFO	0x1C	Valid	Valid	Valid	Valid

USB INTERFACE

The SC5360B USB interface is USB 2.0 compliant running at *Full Speed*, capable of 12 Mbits per second transfer rates. The interface supports three transfer or endpoint types:

- Control Transfer
- Interrupt Transfer
- Bulk Transfer

The endpoint addresses are provided in the C-language header file and are listed below:

```
//  Define SignalCore USB Endpoints
#define  SCI_ENDPOINT_IN_INT          0x81
#define  SCI_ENDPOINT_OUT_INT         0x02
#define  SCI_ENDPOINT_IN_BULK         0x83
#define  SCI_ENDPOINT_OUT_BULK        0x04

// Define for Control Endpoints
#define  USB_ENDPOINT_IN              0x80
#define  USB_ENDPOINT_OUT             0x00
#define  USB_TYPE_VENDOR              (0x02 << 5)
#define  USB_RECIP_INTERFACE          0x01
```

The buffer lengths are sixty-four bytes for all endpoint types. The user should not exceed this length or the device may not respond correctly. This information is provided to aid custom driver development on host platforms other than those that are supported by SignalCore.

WRITING THE DEVICE REGISTERS DIRECTLY

Device register for the SC5360B vary between two bytes and five bytes in length. The most significant byte (MSB) is the command register address that specifies how the device should handle the subsequent configuration data. The configuration data likewise needs to be ordered MSB first, that is, transmitted first. Input and output buffers of 5 bytes long are sufficient on the host. To ensure that a register instruction has been fully executed by the device, reading a byte back from the device will confirm that because the device will only return data upon full execution of the instruction, although this is not necessary.

READING THE DEVICE REGISTERS DIRECTLY

Valid data is only available to be read back after writing one of the query registers such as 0x19, 0x1A, 0x1B, and 0x1C. As soon as one of these registers is written, data is available on the device to be read back. When reading the device data, the MSB is returned as the first byte. Read only the valid number of return data as specified for each register; they vary between 2 to 4 bytes.

USB DRIVER API

The SC5360B USB driver provided by SignalCore is based on libusb-1.0 (www.libusb.org) and its API library is available for the Windows™ and Linux™ operating systems. Source code for both platforms is available upon request by emailing support@signalcore.com. The API functions are nothing more than register wrappers called through the USB bulk transfer function. The C/C++ API library functions are summarized in the table below and each function description is provided in the API description section.

Function	Description
sc5360b_SearchDevices	Finds all the SC5360B Devices connected to the host
sc5360b_OpenDevice	Opens a USB session for the device
sc5360b_CloseDevice	Closes a USB session for the device
sc5360b_InitDevice	Initialize the device to power-up state
sc5360b_SetFrequency	Sets the device frequency for single fixed tone mode
sc5360b_SetGain	Set the channel desired gain
sc5360b_SetAttenuator	Set the attenuator value
sc5360b_SetTuneResolution	Sets the frequency steps to 1 MHz or 5 MHz
sc5360b_SetMonitorPort	Enables or disable the IF monitor ports
sc5360b_SetClockReference	Set of the reference clock behavior
sc5360b_SetReferenceDac	Adjust the internal clock frequency accuracy via a DAC
sc5360b_ConfigCurrentState	Reads or store the current configuration of the device
sc5360b_GetDeviceStatus	Reads the devices status such as PLLs
sc5360b_GetTemperature	Reads the device temperature
sc5360b_GetDeviceInfo	Reads the device info such as serial number, HW rev, etc.
sc5360b_SetDeviceStandby	Sets the device in standby mode
sc5360b_RegRead	Query registers
sc5360b_RegWrite	Configure registers

API DESCRIPTION

The API functions are contained in the **sc5360b.dll** for Windows™ operating systems, or libsc5360b.so.1.0 for Linux™ operating systems. For other operating systems or embedded systems, source code is available for compilation by emailing support@signalcore.com. Information provided below represents the contents of the C/C++ header file, sc5360b.h, but are expanded here, and listed for convenience.

Function: **sc5360b_SearchDevices**
Definition: int sc5360b_SearchDevices(char **serialNumberList)
Output: char **serialNumberList (2-D array pointer list)
Description: sc5360b_SearchDevices searches for SignalCore SC5360B devices connected to the host computer and returns (int) the number of devices found. It also populates the char array with their serial numbers. The user can use this information to open

specific device(s) based on their unique serial numbers. See sc5360b_OpenDevice function on how to open a device.

Function: **sc5360b_OpenDevice**
Definition: usbHandle_t *sc5360b_OpenDevice(char *devSerialNum)
Input: char * devSerialNum (serial number string)
Return: sc5360b_deviceHandle_t* (pointer
Description: sc5360b_OpenDevice opens the device and returns a handle pointer for access.

Function: **sc5360b_CloseDevice**
Definition: int sc5360b_CloseDevice(usbHandle_t *devHandle)
Input: usbHandle_t *devHandle (handle to the device to be closed)
Description: sc5360b_CloseDevice closes the device associated with the device handle.

Example: Code to exercise the functions that open and close the device:

```
// Declaring
#define MAXDEVICES 50
usbHandle_t  *devHandle; //device handle
int numOfDevices; // the number of device types found
char **deviceList;  // 2D to hold serial numbers of the devices found
int status; // status reporting of functions

deviceList = (char**)malloc(sizeof(char*)*MAXDEVICES); // 50 serial numbers to search
for (i=0;i<MAXDEVICES; i++) // allocate 8 char for each device
	deviceList[i] = (char*)malloc(sizeof(char)*SCI_SN_LENGTH); // SCI SN has 8 char

numOfDevices = sc5360b_SearchDevices(deviceList); //searches for SCI for device type
if (numOfDevices == 0)
{
	printf("No signal core devices found or cannot not obtain serial numbers\n");
	for(i = 0; i<MAXDEVICES;i++) free(deviceList[i]);
	free(deviceList);
	return 1;
}
printf("\n There are %d SignalCore %s SC5360B devices found. \n \n",  //
					numOfDevices, SCI_PRODUCT_NAME);
	i = 0;
	while ( i < numOfDevices)
	{
		printf("    Device %d has Serial Number: %s \n", i+1, deviceList[i]);
		i++;
	}
// sc5360b_OpenDevice, open device 0
devHandle = sc5360b_OpenDevice(deviceList[0]);
// Free memory
for(i = 0; i<MAXDEVICES;i++) free(deviceList[i]);
free(deviceList); // Done with the deviceList
 //
// Do something with the device
//
status = sc5360b_CloseDevice(devHandle); // Close the device
```

Function: **sc5360b_InitDevice**
Definition: int sc5360b_SetRfMode(usbHandle_t *devHandle)
Input: usbHandle_t *devHandle (handle to the opened device)
Description: sc5360b_InitDevice sets device to the power-up state.

Function: **sc5360b_SetFrequency**
Definition: int sc5360b_SetFrequency(usbHandle_t *devHandle,
unsigned int frequency)
Input: usbHandle_t *devHandle (handle to the opened device)
unsigned int frequency (frequency in MHz)
Description: sc5360b_SetFrequency sets the single fixed tone RF frequency.

Function: **sc5360b_SetGain**
Definition: int sc5360b_SetGain(usbHandle_t *devHandle,
unsigned char channel, int desiredGain)
Input: usbHandle_t *devHandle (handle to the opened device)
unsigned char channel (device channel)
int desiredGain (See document for bit info)
Description: sc5360b_SetGain sets desired gain of the selected channel. The device will compute and set the attenuator combination to set device gain closest to the desired. One could set the attenuators directly too.

Function: **sc5360b_SetAttenuator**
Definition: int sc5360b_SetAttenuator(usbHandle_t *devHandle, unsigned char channel,
unsigned char attenuator, unsigned char attenvalue)
Input: usbHandle_t *devHandle (handle to the opened device)
unsigned char channel (device channel)
unsigned char attenuator (the attenuator)
unsigned char attenValue (attenuation value)
Description: sc5360b_SetAttenuator will set the value of the select attenuator. Attenuator #1 is in the first IF stage, and attenuator #2 is in the final stage. As a general rule, to increase linearity, set more attenuation in #1. To keep the noise figure low, increase more attenuation in #2.

Function: **sc5360b_SetTuneResolution**
Definition: int sc5360b_SetTuneResolution(usbHandle_t *devHandle, bool reslnMode)
Input: usbHandle_t *devHandle (handle to the opened device)
bool reslnMode (5 or 1 MHz)
Description: sc5360b_SetTuneResolution sets the tuning step to either 5 MHz (0) or 1 MHz (1).

Function: **sc5360b_SetMonitorPort**
Definition: int sc5360b_SetMonitorPort(usbHandle_t *devHandle,
bool mode)
Input: usbHandle_t *devHandle (handle to the opened device)
bool mode (disable/enable)
Description: sc5360b_SetMontiorPort enables or disables the IF monitor ports.

Function: **sc5360b_SetClockReference**
Definition: int sc5360b_ SetClockReference(usbHandle_t *devHandle,
bool refFreqSelect, bool lockExtEnable)
Input: usbHandle_t *devHandle (handle to the opened device)
bool refFreqSelect (Select either 100 MHz or 10 MHz)
bool lockExtEnable (internal/direct clocking or lock to external ref.)
Description: sc5360b_SetClockReference set up how external references are utilized. If lockExtEnable is set to 0, the device will use its internal VCXO to clock the LO synthesizers. If an external source of 100 MHz is connected, it will drive the LO synthesizers directly and the internal VCXO is disabled. If the lockExtEnable is set to 1, the internal VCXO will attempt to lock to the external device. In phase lock mode, the reference frequency may be 100 MHz or 10 MHz.

Function: **sc5360b_SetReferenceDac**
Definition: int sc5360b_SetReferenceDac(usbHandle_t *devHandle,
unsigned int dacValue)
Input: usbHandle_t *devHandle (handle to the opened device)
unsigned int dacValue (DAC value 0-0x3FFF)
Description: sc5360b_ SetReferenceDac sets the value of the DAC that controls the VCXO frequency accuracy when the internal VCXO is used.

Function: **sc5360b_SetDeviceStandby**
Definition: int sc5360b_SetDeviceStandby(usbHandle_t *devHandle,
bool standbyMode)
Input: usbHandle_t *devHandle (handle to the opened device)
bool standbyEnable (enable the device to go in standby mode)
Description: sc5360b_SetDeviceStandby will turn off most analog circuitry, reducing power consumption, stops any current sweeps, and resets the triggers when standbyMode is set to 1. Setting to 0 will take the device out of standby.

Function: **sc5360b_ConfigCurrentState**
Definition: int sc5360b_ConfigCurrentState(usbHandle_t *devHandle, unsigned char mode, deviceState_t *currentState)
Input: usbHandle_t *devHandle (handle to the opened device)
unsigned char mode (read or write)
Output: unsigned int *currentState (current state of device)
Description: sc5360b_ConfigCurrentState stores the current configuration such as RF frequency, attenuation, etc with mode = 1. The current configuration is read and passed back via the currentState_t structure.

Function: **sc5360b_GetDeviceStatus**
Definition: int sc5360b_GetDeviceStatus(usbHandle_t *devHandle, deviceStatus_t *deviceStatus)
Input: usbHandle_t *devHandle (handle to the opened device)
deviceStatus_t *deviceStatus (current device status)
Description: sc5360b_GetDeviceStatus gets the current device status such as the PLL lock status, reference clock configuration, etc .

Function: **sc5360b_GetDeviceInfo**
Definition: int sc5360b_ListStartFrequency(usbHandle_t *devHandle, deviceInfo_t *deviceInfo)
Input: usbHandle_t *devHandle (handle to the opened device)
Output: deviceInfo_t *deviceInfo (device information)
Description: sc5360b_GetDeviceInfo obtains the device information such as serial number, hardware revision, firmware revision, and manufactured date.

Function: **sc5360b_GetTemperature**
Definition: int sc5360b_GetTemperature(usbHandle_t *devHandle, float *temperature)
Input: usbHandle_t *devHandle (handle to the opened device)
Output: float *temperature (frequency)
Description: sc5360b_GetTemperature obtains the device temperature.

EXAMPLE CODE

Code examples in C/C++ demonstrate how the API simplifies programming the device. Both source code and precompiled 32-bit and 64-bit example executables are provided with the software package.

LABVIEW SUPPORT

A LabVIEW USB API is also provided for development on that platform. The API calls the sc5360b.dll and is simply a wrapper of the C/C++ API. The executable SoftFrontPanel.exe was

developed in LabVIEW using the USB API, and its source code is also available in the application subfolder of the LabVIEW functions folder.

REVISION NOTES

Rev 1.0 Original document

© 2015 SignalCore Inc. Information furnished by SignalCore is believed to be accurate and reliable. However, no responsibility is assumed by SignalCore for its use, nor for any infringements of patents or other rights of third parties that may result from its use. Specifications are subject to change without notice. No license is granted by implication or otherwise under any patent or patent rights of SignalCore. All rights reserved. Trademarks and registered trademarks are the property of their respective owners. The word "SignalCore", its logo, and the words "preserving signal integrity" are registered trademarks of SignalCore Incorporated.

Phone: 512 501 6000
Fax: 512 501 6001
Email: info@signalcore.com