SignalCore™
PRESERVING SIGNAL INTEGRITY

SC5506A

25 MHz to 6 GHz RF Signal Source

USB, SPI and RS-232 Interfaces

Operating & Programming Manual

support@signalcore.com

CONTENTS

IMPORTANT INFORMATION

Warranty

This product is warranted against defects in materials and workmanship for a period of three years from the date of shipment. SignalCore will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

Before any equipment will be accepted for warranty repair or replacement, a Return Material Authorization (RMA) number must be obtained from a SignalCore customer service representative and clearly marked on the outside of the return package. SignalCore will pay all shipping costs relating to warranty repair or replacement.

SignalCore strives to make the information in this document as accurate as possible. The document has been carefully reviewed for technical and typographic accuracy. In the event that technical or typographical errors exist, SignalCore reserves the right to make changes to subsequent editions of this document without prior notice to possessors of this edition. Please contact SignalCore if errors are suspected. In no event shall SignalCore be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, SIGNALCORE, INCORPORATED MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF SIGNALCORE, INCORPORATED SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. SIGNALCORE, INCORPORATED WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of SignalCore, Incorporated will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against SignalCore, Incorporated must be brought within one year after the cause of action accrues. SignalCore, Incorporated shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow SignalCore, Incorporated's installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright & Trademarks

International Materials Declarations

SignalCore, Incorporated uses a fully RoHS compliant manufacturing process for our products. Therefore, SignalCore hereby declares that its products do not contain restricted materials as defined by European Union directive 2002/95/EC (EU RoHS) in any amounts higher than limits stated in the directive. This statement is based on the assumption of reliable information and data provided by our component suppliers and may not have been independently verified through other means. For products sold into China, we also comply with the "Administrative Measure on the Control of Pollution Caused by Electronic Information Products" (China RoHS). In the current stage of this legislation, the content of six hazardous materials must be explicitly declared. Each of those materials, and the categorical amount present in our products, are shown below:

組成名稱 **Model Name**	鉛 **Lead (Pb)**	汞 **Mercury (Hg)**	镉 **Cadmium (Cd)**	六价铬 **Hexavalent Chromium (Cr(VI))**	多溴联苯 **Polybrominated biphenyls (PBB)**	多溴二苯醚 **Polybrominated diphenyl ethers (PBDE)**
SC5506A	✓	✓	✓	✓	✓	✓

A ✓ indicates that the hazardous substance contained in all of the homogeneous materials for this product is below the limit requirement in SJ/T11363-2006. An **X** indicates that the particular hazardous substance contained in at least one of the homogeneous materials used for this product is above the limit requirement in SJ/T11363-2006.

CE European Union EMC & Safety Compliance Declaration

The European Conformity (CE) marking is affixed to products with input of 50 - 1,000 VAC or 75 - 1,500 VDC and/or for products which may cause or be affected by electromagnetic disturbance. The CE marking symbolizes conformity of the product with the applicable requirements. CE compliance is a manufacturer's self-declaration allowing products to circulate freely within the European Union (EU).

SignalCore products meet the essential requirements of Directives 2014/30/EU (EMC) and 2014/35/EU (product safety) and comply with the relevant standards. Standards for Measurement, Control and Laboratory Equipment include EN 61326-1:2013 and EN 55011:2009 for EMC, and EN 61010-1 for product safety.

Recycling Information

All products sold by SignalCore eventually reach the end of their useful life. SignalCore complies with EU Directive 2012/19/EU regarding Waste Electrical and Electronic Equipment (WEEE).

Warnings Regarding Use of SignalCore Products

(1) PRODUCTS FOR SALE BY SIGNALCORE, INCORPORATED ARE NOT DESIGNED WITH COMPONENTS NOR TESTED FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED “SYSTEM FAILURES”). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE SOLELY RELIANT UPON ANY ONE COMPONENT DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM SIGNALCORE' TESTING PLATFORMS, AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE SIGNALCORE PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY SIGNALCORE, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF SIGNALCORE PRODUCTS WHENEVER SIGNALCORE PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

GETTING STARTED

Unpacking

All SignalCore products ship in antistatic packaging (bags) to prevent damage from electrostatic discharge (ESD). Under certain conditions, an ESD event can instantly and permanently damage several of the components found in SignalCore products. Therefore, to avoid damage when handling any SignalCore hardware, you must take the following precautions:

- Ground yourself using a grounding strap or by touching a grounded metal object.
- Touch the antistatic bag to a grounded metal object before removing the hardware from its packaging.
- *Never* touch exposed signal pins. Due to the inherent performance degradation caused by ESD protection circuits in the RF path, the device has minimal ESD protection against direct injection of ESD into the RF signal pins.
- When not in use, store all SignalCore products in their original antistatic bags.

Remove the product from its packaging and inspect it for loose components or any signs of damage. Notify SignalCore immediately if the product appears damaged in any way.

Verifying the Contents of your Shipment

Verify that your SC5506A kit contains the following items:

Quantity	Item
1	SC5506A Dual Channel RF Signal Source
1	Software Installation USB Flash Drive (may be combined with other products onto a single drive)

Setting Up and Configuring the SC5506A

The SC5506A is a core module-based RF signal source with all I/O connections and indicators located on the front face of the module as shown in Figure 1. Each location is discussed in further detail below.

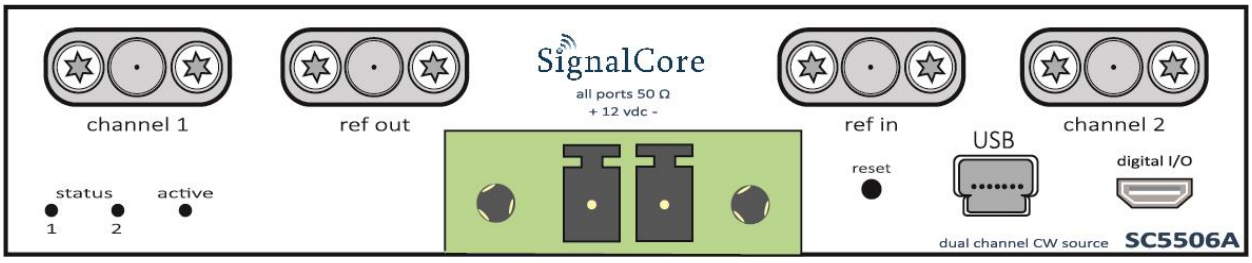


Figure 1. Front view of the SC5506A showing user I/O locations.

Power Connection

Power is provided to the device through a two-position screw terminal block connection as shown in Figure 1. Proper operation of the device requires +12 VDC source and ground return wires capable of delivering a minimum current of 1.5 Amps. The polarity of the connector is shown on the front panel of the RF module, just above the screw terminal block.

Signal Connections

All signal connections (ports) on the SC5506A are SMA-type. Exercise caution when fastening cables to the signal connections. Over-tightening any connection can cause permanent damage to the device.

The condition of your system's signal connections can significantly affect measurement accuracy and repeatability. Improperly mated connections or dirty, damaged or worn connectors can degrade measurement performance. Clean out any loose, dry debris from connectors with clean, low-pressure air (available in spray cans from office supply stores).

If deeper cleaning is necessary, use lint-free swabs and isopropyl alcohol to gently clean inside the connector barrel and the external threads. Do not mate connectors until the alcohol has completely evaporated. Excess liquid alcohol trapped inside the connector may take several days to fully evaporate and may degrade measurement performance until fully evaporated.

Tighten all SMA connections to 5 in-lb max (56 N-cm max).

RF OUT CHANNEL 1	This port outputs the tunable RF signal from channel 1 of the source. The connector is SMA female. The nominal output impedance is 50 Ω.
RF OUT CHANNEL 2	This port outputs the tunable RF signal from channel 2 of the source. The connector is SMA female. The nominal output impedance is 50 Ω.
REF OUT	This port outputs the internal 10 MHz reference clock. The connector is SMA female. This port is AC-coupled with a nominal output impedance of 50 Ω.
REF IN	This port accepts an external 10 MHz reference signal, allowing an external source to synchronize the internal reference clock. The connector is SMA female. This port is AC-coupled with a nominal input impedance of 50 Ω. Maximum input power is +13 dBm.

Communication Connections

The SC5506A uses a mini-USB Type B connector (for USB communication) and a micro-HDMI (for SPI or RS-232 communication, depending on the version ordered) to communicate with the device. The USB port uses the standard USB 2.0 protocol found on most host computers. The pinout of this connector, viewed from the front of the module, is listed in Table 1.

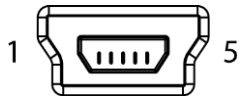


Table 1. Pinout of the SC5506A USB communication connector.

Pin Number	USB Function	Description
1	VBUS	Vcc (+5 Volts)
2	D –	Serial data
3	D +	Serial data
4	ID	Not used
5	GND	Device ground (also tied to connector shell)

The user can also communicate with the device through the micro-HDMI port. Depending on the version ordered, this connector provides either the SPI or RS-232 communication path. The pinout of this connector, viewed from the front of the module, is listed in Table 2.

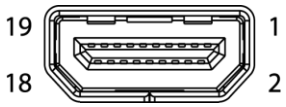


Table 2. Pinout of the SC5506A micro-HDMI connector for either SPI or RS-232 communication.

Pin Number	SPI Function	RS-232 Function
3	MISO	TxD
5	–	–
8	RESET_B	RESET_B
9	MOSI	RxD
11	CS	–
15	SRLDY	–
17	CLK	–
18	SPI MODE	BAUD SELECT
4, 7, 10, 13, 16	GND	GND
1, 2, 8, 12, 14, 19	NC	NC

Reset Button

Depressing this momentary-action push button switch will reset the device to its default state. The SC5506A has the ability to store the current configuration at any point as the default setting. If the factory setting has been overwritten with a saved user configuration, resetting the device will reinitialize the

device to the saved user configuration. Otherwise, resetting the device will restore the factory configuration. Refer to the table in the “Default Startup Mode” section of this manual for default settings.

Indicator LEDs

The SC5506A provides visual indication of important modes. There are three LED indicators on the unit. Their behavior under different operating conditions is shown in Table 3.

Table 3. LED indicator states.

LED	Color	Definition
STATUS	Green	“Power good” and all oscillators phase-locked
STATUS	Orange	Channel powered down
STATUS	Red	One or more oscillators off lock
STATUS	Off	Power fault
ACTIVE	Green/Off	Device is open (green) /closed (off) , this indicator is also user programmable (see register map)

Output Amplitude Control

As shown in Figure 2, the SC5506A source architecture at a high level consists of an output amplitude control section and a frequency synthesis section. The amplitude of the signal is controlled through the use of digital step attenuators (DSAs) and a voltage controlled attenuator (VCA). The DSAs provide the coarse-step tuning over a wide range while the VCA provides fine tune correction to the DSAs. The VCA is part of the automatic level control loop (ALC), which additionally consists of an RF amplifier, a power detector, and an integrator. The ALC loop can be closed or open. In the closed loop mode, the power detector outputs a voltage proportional to the power it detects. This voltage is compared to that of the reference ALC DAC voltage, which in turn is set for some calibrated power level. Voltage error between the detector voltage and the ALC DAC voltage drives the integrator output in the direction that will vary the VCA to achieve the desired output power level. When the ALC control loop is opened, the power detector output voltage is grounded, and the integrator is configured as a voltage buffer that drives the ALC DAC voltage to the VCA. In this mode, the ALC DAC voltage directly drives the VCA with voltage levels that correspond to calibrated output power levels.

There are advantages and disadvantages to either of these two amplitude control modes. On one hand, the open loop mode has an advantage over the closed loop mode when close-in carrier amplitude noise is a concern. ALC loops do introduce some level of amplitude modulated noise onto the carrier signal, and these levels may not be acceptable (although they are generally lower than the phase noise). SignalCore offers the user the option to open the ALC loop to remove any unwanted AM noise that results from closed loop control. Another side effect of the closed loop is that the frequency bandwidth of the ALC loop may slow down amplitude settling. Typically, in order to keep AM noise low and close (in offset frequency) to the signal, the loop bandwidth is also kept low. As a result, the settling time is increased.

On the other hand, a closed loop ALC provides better amplitude control over the entire frequency range. With a temperature-stable ALC DAC, the closed loop will precisely maintain the power at the detector, mitigating errors in the components prior to it in the signal path. Temperature-induced errors in components and abrupt amplitude changes when switching filters in the filter banks contribute to errors in the amplitude of the signal. However, these errors occur before the power detector and are compensated by the feedback loop action. Errors in amplitude are thus confined to the output attenuators, amplifiers, and the loop components. When the loop is opened, amplitude errors resulting from all parts of the amplitude control section as well as the synthesizer section will affect the overall output amplitude accuracy. In particular, when the filters within the filter bank are switched from one to another, the signal experiences abrupt discontinuities in its amplitude which the open loop calibration cannot appropriately account for in its correction algorithm.

Setting of the amplitude control components are performed automatically by the system, although the user may choose to override the ALC DAC value if needed. In Figure 2, the labels in red indicate parameters or devices which the user has direct control over.

Frequency Synthesizer

The synthesizer section of the SC5506A comprises a multiple phase-locked loop architecture whose base frequency reference is a 10 MHz TCXO. The user may choose to phase-lock this base reference to an external source if required. The 100 MHz VCXO is phase-locked to the TCXO for frequency stability. While the TCXO determines the very close-in phase noise, the VCXO phase noise determines the system phase noise in the frequency offset range of approximately 1 kHz to 30 kHz. The 100 MHz VCXO provides the reference signal to the main RF signal synthesizer, which is comprised of three phase-locked loops (PLLs) and a direct digital synthesis (DDS) oscillator. The "fine" PLL provides a tuning resolution of few millihertz over a narrow frequency range, while the "coarse" PLL tunes in steps of a few megahertz over several gigahertz of range. The "main" or summing PLL combines the signals of the "coarse" and "fine" loops into one broad tuning signal with fine tuning capability.

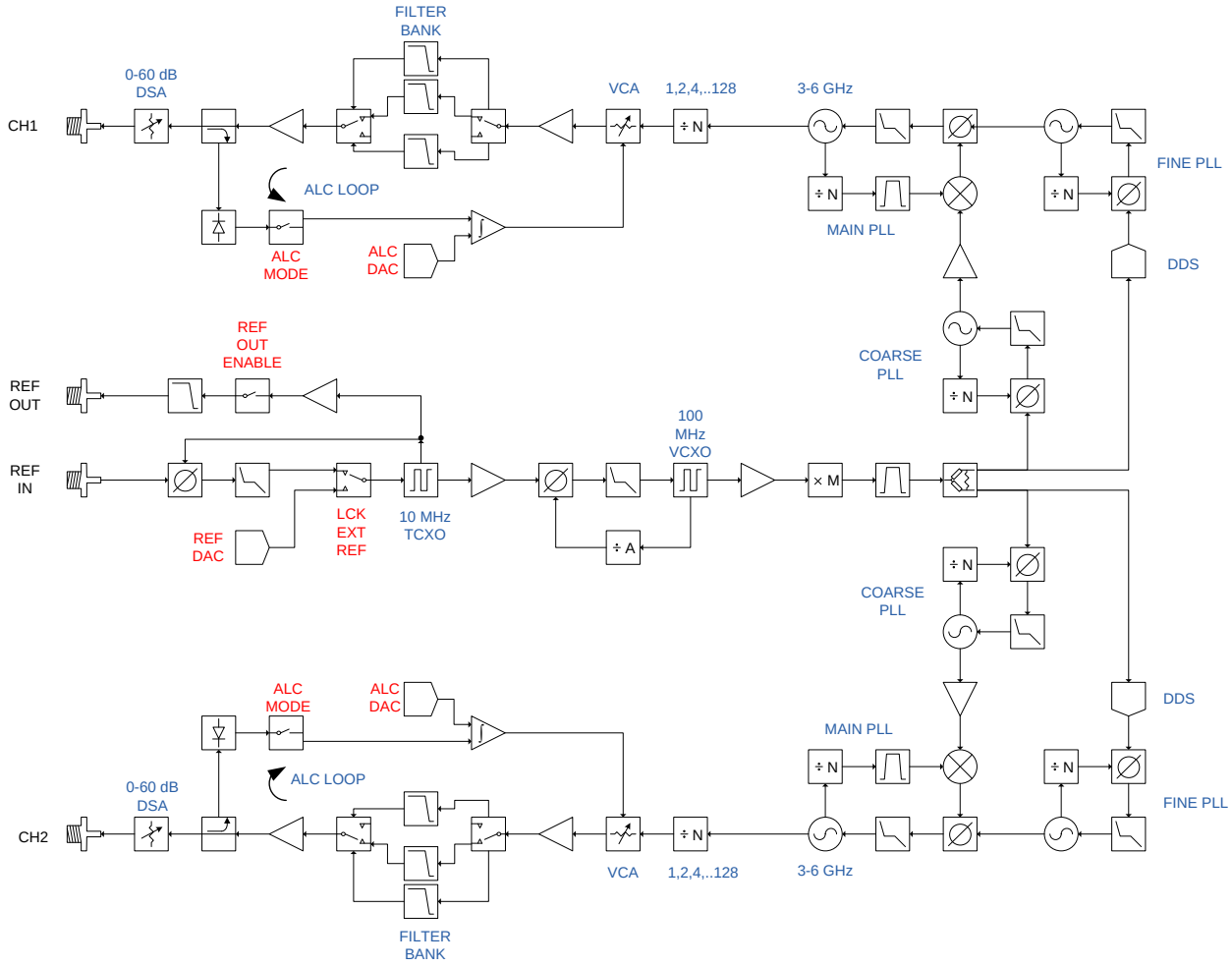


Figure 2. Simplified block diagram of the SC5506A dual channel RF signal source.

Using this multiple loop approach produces signals with low phase noise and low phase spurs, high levels of which exist in single loop architectures such as single fractional-N PLLs. Although a single fractional-N type PLL may provide fine resolution, its large fractional spurs may make it unusable in certain parts of the band - especially at frequency regions close the integer boundaries. A multiple loop architecture allows fine tuning with extremely low phase spurs.

Reference Clock Control

As mentioned above, the primary clock reference for the SC5506A is an onboard 10 MHz TCXO. Should the user require better frequency stability and/or accuracy, this TCXO can be programmed to phase-lock to an external source such as an OCXO or rubidium clock. The device can also be programmed to export its internal 10 MHz clock. To adjust the accuracy of the TCXO as needed (e.g., to correct for long-term accuracy drift), the user may vary the reference DAC voltage by writing the **REFERENCE_DAC_SETTING** register.

Harmonics and Range of Operation

The SC5506A's guaranteed operating frequency range is 100 MHz to 6.0 GHz. However, the device is capable of tuning as low as 25 MHz and as high as 6.15 GHz. Within these out-of-range regions, the amplitude may not be within specification but the frequency accuracy will not be affected. At low frequencies of operation (200 MHz and lower), the harmonics of the signal can potentially be observed as high as -10 dBc at 0 dBm output. This is due to the limited space available for additional filtering in these ranges. At lower frequencies, the large physical size of appropriate filters makes it impossible to accommodate them within the compact form of this device. Furthermore, as the low frequencies are synthesized through frequency dividers, their output waveforms become more "square" than sinusoidal, giving rise to higher odd-order harmonics.

The device is specified to a maximum calibrated level of +10 dBm, although the maximum calibrated output is greater than that in most regions of the spectrum. The accuracy degrades as the amplitude approaches the compression point due to the linear approximation in the correction algorithm. As a general rule however, the lower the tuned frequency the higher the achievable output power.

Channel Standby and RF Enable

The SC5506A has independent standby and output enable features for both channels. The user may wish to place one or both of the channels into standby mode to reduce power consumption and thus lower the operating temperature of the device under the same environmental conditions. Putting one channel into standby also eliminates its signal generation, and as a result the remaining operational channel does not receive any cross-signal contamination, especially when generating low level signals. Although the device has the option to independently disable the RF output of one channel or the other, this feature does not shut down the internal synthesizer, so there still exists the possibility of contaminating signals. They are usually very low but might not be tolerated in some applications. Taking either channel out of standby requires the device to wait for the power rails to settle and all internal components to be reprogrammed, usually on the order of one second.

Disabling the RF output moves the frequency to some very low value so that the step attenuators and the voltage controlled attenuator have the most effective attenuation. Combined, this will push the signal level below -100 dBm. Enabling the RF output is nearly instantaneous as all components remain active even when the RF output is disabled.

Default Startup Mode

The factory power-up state for the device is detailed in Table 4. The default state can be changed to the current state of either channel programmatically, allowing the user to power up the device in the last saved state without having to reprogram.

Table 4. Factory default power-up state.

	CH1	CH2
Frequency	2.0 GHz	2.4 GHz
Power	0.00 dBm	0.00 dBm
RF Output	Enabled	Enabled
ALC Mode	Closed Loop	Closed Loop
Standby	Disabled	Disabled
Auto Level	Enabled	Enabled
Ref Out	Disabled	
Ext Ref Lock	Disabled	

SC5506A Programming Interface

Device Drivers

The SC5506A is programmed by writing to its set of configuration registers, and its status read back through its set of query registers. The user may choose to program directly at register level or through the API library functions provided. These API library functions are wrapper functions of the registers that simplify the task of configuring of the register bytes. The register specifics are covered in the next section. Writing to and reading from the device at the register level through the API involves calls to the **sc5506a_RegWrite** and **sc5506a_RegRead** functions respectively.

For Microsoft Windows™ operating systems, The SC5506A API is provided as a dynamic linked library, *sc5506a.dll*, which is available for 32bit and 64bit operating systems. This API is based on the libusb-1.0 library and therefore it is required to be installed on the system prior to development. The *libusb-1.0.dll* will install along with the *sc5506a.dll*, and along with the header files for development. However for possible newer versions of libusb-1.0, visit http://libusbx.org to check for version updates and downloads. To install the necessary drivers, right click on the *sc5506a.inf* file in the *Win*\ directory and choose install. When the device is connected to a USB port, the host computer should identify the device and load the appropriate driver. For more information, see the *SC5506A_Readme.txt* file in the *Win*\ directory.

For LabVIEW™ support, a full LabVIEW API is provided and is available in the *Win**API**LabVIEW*\ directory. To use the library, copy the "SignalCore" folder in that directory to *%LabVIEW path%**instr.lib* location of your LabVIEW installation directory. The LabVIEW functions are simply VI wrappers around *sc5506a.dll*. Code written purely in G that does not call or depend on external library functions is available to our customers on request. If pure G code SC5506A API functions are implemented, the National Instruments driver wizard packaged with NI-VISA should be used to create a driver for the intended operating system. SignalCore's vendor ID is **0x277C** and the product ID (PID) is **0x0016**.

For other operating systems, users will need to write and compile their own drivers. The device register map provides the necessary information to successfully implement a driver for the SC5506A. Driver code based on libusb-1.0 is available to our customers on request. Should the user require assistance in writing an appropriate API other than that provided, please contact SignalCore for additional example code and hardware details.

Using the Application Programming Interface (API)

The SC5506A API library functions make it easy for the user to communicate with the device. Using the API removes the need to understand register-level details - their configuration, address, data format, etc. Using the API, commands to control the device are greatly simplified. For example, to obtain the device temperature, the user simply calls the function **sc5506A_GetDeviceTemperature**, or calls **sc5506A_SetFrequency** to tune the frequency. The software API is covered in detail in the "Software API Library Functions" section.

SPI and RS-232 Programming

Please see the SPI and RS-232 sections of this manual for interfacing to the device registers using either of these communication methods.

SETTING THE SC5506A: WRITING TO CONFIGURATION REGISTERS

Configuration Registers

The users may write the configuration registers (write only) directly by calling the **sc5506a_RegWrite** function. The syntax for this function is **sc5506a_RegWrite(deviceHandle, registerCommand, instructWord)**. The **instructWord** takes a 64 bit-word. However, it will only send the required number of bytes to the device. These registers are the same for USB, SPI, and RS-232. Table 5 summarizes the register addresses (commands) and the effective bytes of command data.

Table 5. Configuration registers.

Register Name	Register Address	Serial Range	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INITIALIZE	0x01	[7:0]	Open	Open	Open	Open	Open	Open	Open	Mode
SET_SYSTEM_ACTIVE	0x02	[7:0]	Open	Open	Open	Open	Open	Open	Open	Enable "active" LED
RF_FREQUENCY	0x10	[7:0]	Frequency Word (MHz) [7:0]							
		[15:8]	Frequency Word (MHz) [15:8]							
		[23:16]	Frequency Word (MHz) [23:16]							
		[31:24]	Frequency Word (MHz) [31:24]							
		[39:32]	Frequency Word (MHz) [39:32]							
		[47:40]	Open	Open	Open	Open	Open	Open	Open	Channel
RF_POWER	0x11	[7:0]	RF Power Word [7:0]							
		[15:8]	Sign Bit	RF Power Word [14:8]						
		[23:9]								Channel
RF_OUT_ENABLE	0x12	[7:0]	Open	Open	Open	Open	Open	Open	Channel	Mode
AUTO_POWER_DISABLE	0x13	[7:0]	Open	Open	Open	Open	Open	Open	Channel	Mode
RF_ALC_MODE	0x14	[7:0]	Open	Open	Open	Open	Open	Open	Channel	Mode
DEVICE_STANDBY	0x15	[7:0]	Open	Open	Open	Open	Open	Open	Channel	Mode
REFERENCE_MODE	0x16	[7:0]	Open	Open	Open	Open	Open	Open	Ref Out Enable	Lock Enable
USER_EEPROM_WRITE	0x1B	[7:0]	Data [7:0]							
		[15:8]	EEPROM Address [7:0]							
		[23:16]	EEPROM Address [15:8]							
REFERENCE_DAC_SETTING	0x1D	[7:0]	DAC word [7:0]							
		[15:8]	Open	Open	DAC word [13:8]					
STORE_STARTUP_STATE	0x23	[7:0]	Open	Open	Open	Open	Open	Open	Open	Open
SET_ALC_DAC_VALUE	0x24	[7:0]	ALC DAC Word [7:0]							
		[15:8]	ALC DAC Word [13:8]							
		[23:16]	Open	Open	Open	Open	Open	Open	Open	Channel

To write to the device through USB transfers such as bulk transfer, it is important to send data with the register byte first, followed by the most significant bit (MSB) of the data bytes. For example, to set the RF power level of channel 2 to a particular amplitude, the byte stream would be [0x11][0x01][15:8][7:0].

Initializing the Device

INITIALIZE (0x01) - Writing 0x00 to this register will reset the device to the default power-on state. Writing 0x01 will reset the device but leave it in the current state. The user has the ability to define the default startup state by writing to the **STORE STARTUP STATE (0x23)** register, described later in this section.

Setting the System Active LED

SET_SYSTEM_ACTIVE (0x02) - This register simply turns on the front panel "active" LED with a write of 0x01 or turns off the LED with a write of 0x00. This register is generally written when the device driver opens or closes the device.

Setting the RF Frequency

RF_FREQUENCY (0x10) - This register set the RF frequency for each channel. Data is sent as a 40 bit word with the LSB in Hz.

Setting the RF Power

RF_POWER (0x11) - This register sets the RF power level for each channel. The LSB is 1/100th of a dB and absolute magnitude is carried in the first 15 bits, starting with bit 0. The sign bit is indicated on bit 15. Setting bit 15 high implies a negative magnitude. For example, to write 10.05 dBm to the register, the data is simply 1005 (0x03ED). For -10.05 dBm, the data is 33773 (0x83ED).

Setting RF Output Enable

RF_OUT_ENABLE (0x12) - This register enables or disables the RF signal output for each channel. Setting bit 0 low (0) disables RF output. Setting bit 0 high (1) enables RF output.

Disabling the Auto Power Feature

AUTO_POWER_DISABLE (0x13) – When changing frequency on either channel, the device will calculate new settings for the amplitude control components such that the amplitude remains the same as the last setting. If the amplitude is also changed at the new frequency setting, the user has the option to turn off this auto power adjustment. By default, auto power adjustment is enabled. To disable auto power adjustment, set bit 0 of this register high (1).

Setting the RF Automatic Level Control (ALC) Mode

RF_ALC_MODE (0x14) – For each channel independently, writing 0x00 to this register puts the ALC in a closed loop operation. Writing 0x01 will run the ALC in an open loop. See the "Output Amplitude Control" section to understand the differences between the modes.

Setting the Device Standby Mode

DEVICE_STANDBY (0x15) – For each channel, writing 0x01 to this register will power-down the analog/RF circuitry. Writing 0x00 to this register will enable the analog/RF circuitry and the channel will return to its last programmed state.

Setting the Reference Clock

REFERENCE_MODE (0x16) - This register sets the behavior of the reference clock section. Bit 1 enables (1) or disables (0) the output reference signal, and Bit 0 enables (1) or disables (0) the device to phase-lock to an external source. It is important that if the device is not intended to lock externally, the external source connection should be removed from the "ref in" connector. Even with external locking disabled, the presence of a large signal from the external source on the reference input terminal could potentially modulate the internal references, causing a spur offset in the RF signal.

Writing to the User EEPROM

USER_EEPROM_WRITE (0x1B) - There is an onboard 32 kilobyte EEPROM for the user to store data. User data is sent one byte at a time and is contained in the last (least significant) byte of the three bytes of data written to the register. The other two bytes contain the write address in the EEPROM. For example, to write user data 0x22 into address 0x1F00 requires writing 0x1F0022 to this register.

Setting the Reference DAC Value

REFERENCE_DAC_SETTING (0x1D) - The frequency precision of the device's 10 MHz TCXO is set by the device internally and the factory calibrated unsigned 14 bit value is written to the reference DAC on power-up from the EEPROM. The user may choose to write a different value to the reference DAC by accessing this register for example, to correct for long-term accuracy drift.

Storing the Startup State

STORE_STARTUP_STATE (0x23) – Writing to this register will save the current device state as the new default power on (startup) state. All data written to this register will be ignored as only the write command is needed to initiate the save.

Setting the RF ALC DAC Value

SET_ALC_DAC_VALUE (0x24) - Writing a 14 bit control word to the ALC DAC register adjusts output amplitude. This is useful when the user wants to make minute adjustments to the power level.

QUERYING THE SC5506A: WRITING TO REQUEST REGISTERS

The registers to read data back from the device (such as device status) are accessed through the **sc5506a_RegRead** function. The function and parameter format for this command is **sc5506a_RegRead(deviceHandle, registerCommand, instructWord,*dataOut)**. Any instructions in addition to the register call is placed into "instructWord", and data obtained from the device is returned via the pointer value dataOut. The set of request registers are shown in Table 6.

Table 6. Query registers.

Register Name	Register Address	Serial Range	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GET_TEMPERATURE	0x17	[7:0]	Open	Open	Open	Open	Open	Open	Open	Open
GET_DEVICE_STATUS	0x18	[7:0]	Open	Open	Open	Open	Open	Open	Open	Open
USER_EEPROM_READ	0x1A	[7:0]	EEPROM Address [7:0]							
		[15:8]	EEPROM Address [15:8]							
CAL_EEPROM_READ	0x21	[7:0]	EEPROM Address [7:0]							
		[15:8]	EEPROM Address [15:8]							
GET_ALC_DAC_VALUE	0x39	[7:0]	Open	Open	Open	Open	Open	Open	Open	Channel
GET_DEVICE_PARAMS	0x25	[7:0]	Open	Open	Open	Open	Parameter value			

To read from the device using native USB transfers instead of the **sc5506a_RegRead** function requires two operations. First, a write transfer is made to the device **ENPOINT_OUT** to tell the device what data needs to be read back. Then, a read transfer is made from **ENDPOINT_IN** to obtain the data. The number of valid bytes returned varies from 1 to 3 bytes. See the register details below.

Reading the Device Temperature

GET_TEMPERATURE (0x17) - Data returned by this register needs to be processed to correctly represent data in temperature units of degrees Celsius. Data is returned in the first 14 bits [13:0]. Bit [13] is the polarity bit indicating whether it is positive (0x0) or negative (0x1). For an **ENDPOINT_IN** transfer, data is returned in 2 bytes with the MSB first. The temperature value represented in the raw data is contained in the next 13 bits [12:0]. To obtain the temperature ADC code, the raw data should be masked (bitwise AND'ed) with 0x1FFF, and the polarity should be masked with 0x2000. The conversion from 12 bit ADC code to an actual temperature reading in degrees Celsius is shown below:

Positive Temperature (bit 13 is 0) = ADC code / 32

Negative Temperature (bit 13 is 1) = (ADC code – 8192) / 32

It is not recommended to read the temperature too frequently, especially once the SC5506A has stabilized in temperature. The temperature sensor is a serial device located inside the RF module. Therefore, like any other serial device, reading the temperature sensor requires sending serial clock and data commands from the processor. The process of sending clock pulses on the serial transfer line may cause unwanted

spurs on the RF signal as the serial clock could potentially modulate the internal oscillators. Furthermore, once the SC5506A stabilizes in temperature, repeated readings will likely differ by as little as 0.25 ^{O}C over extended periods of time. Given that the gain-to-temperature coefficient is on the order of less than -0.01 dB/^{O}C, gain changes between readings will be negligible.

Reading the Device Status

GET_DEVICE_STATUS (0x18) - This register, summarized in Table 7, returns the device status information such as phase lock status of the PLL, current reference settings, etc. Data is contained in the first three bytes.

Table 7. Description of the status data bits.

Bit	Description
[23]	Channel 1 Auto level enable *(Firmware >= rev 4.0)*
[22]	Channel 2 Auto level enable *(Firmware >= rev 4.0)*
[21]	Channel 1 ALC open
[20]	Channel 2 ALC open
[19]	Channel 1 RF output enable
[18]	Channel 2 RF output enable
[17]	Channel 1 standby enable
[16]	Channel 2 standby enable
[15]	Channel 1 sum PLL status
[14]	Channel 1 coarse PLL status
[13]	Channel 1 fine PLL status
[12]	Reserved
[11]	Channel 2 sum PLL status
[10]	Channel 2 coarse PLL status
[9]	Channel 2 fine PLL status
[8]	Reserved
[7]	VCXO PLL status
[6]	TCXO PLL status
[5]	External reference detected
[4]	Lock to external reference enable
[3]	Reference clock output enable
[2]	Device accessed

Reading the User EEPROM

USER_EEPROM_READ (0x1A) - Once data has been written to the user EEPROM, it can be retrieved by calling this register and using the process outlined next for reading calibration data. The maximum address for this EEPROM is 0x7FFF. A single byte is returned.

Reading the Calibration EEPROM

CAL_EEPROM_READ (0x21) - Reading a single byte from an address in the device EEPROM is performed by writing this register with the address for the instructWord. The data is returned as a byte. The CAL EEPROM maximum address is also 0x7FFF. Reading above this address will cause the device to retrieve data from the lower addresses. For example, addressing 0x8000 will return data stored in address location 0x0000. The calibration EEPROM map is shown in Table 8Table 8. Calibration EEPROM map..

All calibration data, whether floats, unsigned 8-bit, unsigned 16-bit or unsigned 32-bit integers, are stored as flattened unsigned byte representation. A float is flattened to 4 unsigned bytes, so once it is read back it needs to be un-flattened back to its original type. Unsigned values containing more than a single byte are converted (un-flattened) simply by concatenation of the bytes through bit-shifting. The starting lower address contains the least significant byte. Converting to floating point representation is slightly more involved. First, convert the 4 bytes into an unsigned 32-bit integer value through concatenation, and then (in C/C++) type-cast a float pointer to the address of the value. In C/C++, the code would be float Y = *(float *)&X, where X has been converted earlier to an unsigned integer.

An example written in C code would look something like the following:

```
byte_value[4]; // read in earlier
unsigned int uint32_value;
float float32_value;

int count  =  0;
while (count  <  4) {
       uint32_value = unit32_value | (byte_value[count] <<
(count*8));
       count++;
}

float32_value = *(float *)&uint32_value;
```

Reading the RF ALC DAC Value

GET_ALC_DAC_VALUE (0x39) - The user may be interested to obtain the current value of the ALC DAC for the purpose of making minor adjustments to the RF output power level. Data is returned in 2 bytes and only the first 14 bits contain valid data.

Reading the Device Parameters (firmware >= rev 4.0)

GET_DEVICE_PARAMS (0x25) - This register will return either the channel frequency or power level, depending on the input byte value.

0: ch1 frequency	1: ch1 power level	2: ch2 frequency	3: ch2 power level

The frequency value comes back in Hz, occupying 5 returned bytes. The power level come back in 100th of dBm masked by 0x7FFF of the first 2 of the 5 bytes. The mask 0x8000 is the sign bit.

CALIBRATION EEPROM MAP

Table 8. Calibration EEPROM map.

EEPROM ADDRESS (HEX)	NUMBER OF DATA POINTS	TYPE	DESCRIPTION
0	1	U32	Factory information
4	1	U32	Product serial number
8	1	U32	RF module number
C	1	U32	Product manufacture date
10	1	U32	Last calibration date
14	4	NA	Reserved
24	1	F32	Firmware revision
28	1	F32	Hardware revision
2C	40	F32	Reserved
CF	1	U8	Startup state (reference)
D0	32	U8	Startup state (synth)
F0	1	U32	TCXO DAC calibration value
F4	1	F32	Calibration temperature
F8	1	F32	ALC enabled temperature coefficient
FC	1	F32	ALC disabled temperature coefficient
100	128	U8	Ch0 VCO value
180	128	U8	Ch0 VCO tune value
200	128	U8	Ch1 VCO value
280	128	U8	Ch1 VCO tune value
300	1	U8	Reference attenuation value
301	125	U16	Cal frequencies (MHz)
3FB	3875	F32	Ch0 attenuator values
4087	125	F32	Ch0 ALC closed ref RF power
427B	125	U16	Ch0 ALC closed ref DAC value
4375	375	F32	Ch0 ALC closed coefficients (2^{nd} order)
4951	125	F32	Ch0 ALC opened ref RF power
4B45	125	U16	Ch0 ALC opened ref DAC value
4C3F	375	F32	Ch0 ALC opened coefficient
521B	3875	F32	Ch1 attenuator values
8EA7	125	F32	Ch1 ALC closed ref RF power
909B	125	U16	Ch1 ALC closed ref DAC value
9195	375	F32	Ch1 ALC closed coefficients (2^{nd} order)
9771	125	F32	Ch1 ALC opened ref RF power
9965	125	U16	Ch1 ALC opened ref DAC value
9A5F	375	F32	Ch1 ALC opened coefficient

SOFTWARE API LIBRARY FUNCTIONS

SignalCore's philosophy is to provide products to our customers whose lower hardware functions are easily accessible. For experienced users who wish to use direct, low-level control of frequency and gain settings, having the ability to access the registers directly is a necessity. However, others may wish for simpler product integration using higher level function libraries and not having to program registers directly. The functions provided in the SC5506A API dynamic linked library or LabVIEW library are:

- **sc5506a_SearchDevices**
- **sc5506a_OpenDevice**
- **sc5506a_CloseDevice**
- **sc5506a_RegWrite**
- **sc5506a_RegRead**
- **sc5506a_InitDevice**
- **sc5506a_SetStandby**
- **sc5506a_SetFrequency**
- **sc5506a_SetPowerLevel**
- **sc5506a_SetRfOut**
- **sc5506a_SetAlcMode**
- **sc5506a_DisableAutoLevel**
- **sc5506a_SetReferenceClock**
- **sc5506a_SetReferenceDac**
- **sc5506a_WriteUserEeprom**
- **sc5506a_StoreCurrentState**
- **sc5506a_SetAlcDac**
- **sc5506a_GetDeviceStatus**
- **sc5506a_GetTemperature**
- **sc5506a_GetAlcDac**
- **sc5506a_ReadCalEeprom**
- **sc5506a_ReadUserEeprom**
- **sc5506a_GetDeviceInfo**
- **sc5506a_GetDeviceParams** (note: function added in rev2.0 for firmware >= 4.0)

Each of these functions is described in more detail on the following pages. Example code written in C/C++ is located in the *Win\Driver\src* directory to show how these functions are called and used. First, for C/C++ we define the constants and types which are contained in the C header file, *sc5506a.h*. These constants and types are useful not only as an include for developing applications using the SC5506A API, but also for writing device drivers independent of those provided by SignalCore.

Constants Definitions

```
// Parameters for storing data in the onboard EEPROM
#define CALEEPROMSIZE            65536    // bytes
#define USEREEPROMSIZE           32768    // bytes

// Assign channel names
#define CH1                          0
#define CH2                          1

// Attenuator assignments
#define CH1ATTEN0                    0
#define CH1ATTEN1                    1
#define CH2ATTEN0                    2
#define CH2ATTEN1                    3


// Define error codes
#define SUCCESS                      0
#define USBDEVICEERROR              -1
#define USBTRANSFERERROR            -2
#define INPUTNULL                   -3
#define COMMERROR                   -4
#define INPUTNOTALLOC               -5
#define EEPROMOUTBOUNDS             -6
#define INVALIDARGUMENT             -7
#define INPUTOUTRANGE               -8
#define NOREFWHENLOCK               -9
#define NORESOURCEFOUND            -10
#define INVALIDCOMMAND             -11

// Define device registers
#define INITIALIZE                0x01    // initialize the device
#define SET_SYSTEM_ACTIVE         0x02    // set the system “active” light
#define RF_FREQUENCY              0x10    // set the frequency
#define RF_POWER                  0x11    // set power of LO1
#define RF_OUT_ENABLE             0x12    // enable RF output
#define AUTO_PWR_DISABLE          0x13    // disable auto power leveling
#define RF_ALC_MODE               0x14    // select closed (0) or open (1) loop ALC modes
#define DEVICE_STANDBY            0x15    // place the selected channel in standby mode
#define REFERENCE_MODE            0x16    // reference settings
#define GET_TEMPERATURE           0x17    // get the internal temperature of device
#define GET_DEVICE_STATUS         0x18    // read the device status
#define USER_EEPROM_READ          0x1A    // read a byte from the user EEPROM
#define USER_EEPROM_WRITE         0x1B    // write a byte to the user EEPROM
#define REFERENCE_DAC_SETTING     0x1D    // set the reference DAC value
#define CAL_EEPROM_READ           0x21    // read a byte from the calibration EEPROM
#define STORE_STARTUP_STATE       0x23    // store the new default state
#define SET_ALC_DAC_VALUE         0x24    // set the RF ALC DAC value
#define GET_ALC_DAC_VALUE         0x39    // read back the RF ALC DAC value
```

Type Definitions

```
typedef struct deviceInfo_t
{
	unsigned int productSerialNumber;
	unsigned int rfModuleSerialNumber;
	float firmwareRevision;
	float hardwareRevision;
	unsigned int calDate; // year, month, day, hour:&(0xFF000000,0xFF0000,0xFF00,0xFF)
	unsigned int manDate; // year, month, day, hour:&(0xFF000000,0xFF0000,0xFF00,0xFF)
}	deviceInfo_t;


typedef struct //deviceStatus_t
{
	bool ch1AutoLevelEnable;	//added in rev2.0 for Firmware >= 4.0
	bool ch2AutoLevelEnable;	//added in rev2.0 for firmware >= 4.0
	bool ch1AlcOpen;
	bool ch2AlcOpen;
	bool ch1OutEnable;
	bool ch2OutEnable;
	bool ch1StandbyEnable;
	bool ch2StandbyEnable;
	bool ch1SumPllStatus;
	bool ch1CrsPllStatus;
	bool ch1FinePllStatus;
	bool ch2SumPllStatus;
	bool ch2CrsPllStatus;
	bool ch2FinePllStatus;
	bool vcxoPllStatus;
	bool tcxoPllStatus;
	bool extRefDetected;
	bool extRefLockEnable;
	bool refClkOutEnable;
	bool deviceAccess;
} deviceStatus_t;
```

Function Definitions and Usage

Before reading this API section, please note that there are API changes in revision 2.0 or later. The changes consist of the following:

- The API library is renamed from sc5506a.dll to sc5506a_usb.dll for the USB interface to prevent confusion between libraries for USB and RS232.
- Two extra members of the deviceStatus_t struct is added, these report the current status of the Auto leveling function of each channel. These are valid only on devices with firmware revision 4 or higher.
- The device handle type is changed from struct type to HANDLE (*void) type. In the following functions, **deviceHandle *devHandle** should be changed to **HANDLE devHandle** when calling revision 2 or higher API library.
- Function sc5506a_GetDeviceParams() has been added to retrieve current channel frequency and power level. This is valid only on devices with firmware revision 4 or higher.
- The device handle from the sc5506a_OpenDevice() function is passed back via a parameter instead of a returned type.

The functions listed below are found in the **sc5506a.dll** or **sc5506a_usb.dll** dynamic linked library for the Windows™ operating system. These functions are also provided in the LabView library, **sc5506a.llb**. The LabView functions contain context help (Cntrl-H) to help with the input and output parameters.

Function: **sc5506a_SearchDevices**

Definition: **int sc5506a_SearchDevices(char **serialNumberList)**

Output: **char** **serialNumberList (2-D array pointer list)

Description: **sc5506a_SearchDevices** searches for SignalCore SC5506A devices connected to the host computer and **returns** (**int**) the number of devices found, and 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 **sc5506a_OpenDevice** function on how to open a device.

Note: changes to OpenDevice() function.

Function: **sc5506a_OpenDevice** *(API DLL < 2.0)*

Definition: **deviceHandle *sc5506a_OpenDevice(char *devSerialNum)**

Input: **char** * devSerialNum (serial number string)

Return: *deviceHandle (unsigned int number for the deviceHandle)

Description: **sc5506a_OpenDevice** opens the device and turns the front panel “active” LED on if it is successful. It returns a handle to the device for other function calls.

Function: **sc5506a_OpenDevice** *(API DLL > 2.0)*

Definition: **int sc5506a_OpenDevice(char *devSerialNum, HANDLE *devHandle)**

Input: **char** * devSerialNum (serial number string)

Return: **int** (status)

Output: **HANDLE *devHandle** (device handle)

Description: **sc5506a_OpenDevice** opens the device and turns the front panel “active” LED on if it is successful. It returns a handle to the device for other function calls.

Function: **sc5506a_CloseDevice**

Definition: **int sc5506a_CloseDevice(deviceHandle *devHandle)**

Input: **deviceHandle** *devHandle (handle to the device to be closed)

Description: **sc5506a_CloseDevice** closes the device associated with the device handle and turns off the "active" LED on the front panel if it is successful.

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

```
// Declaring
#define MAXDEVICES 50
devicehandle *devHandle; //device handle
// HANDLE devHandle; // API >= 2.0

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

numOfDevices = sc5506a_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 USB 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++;
       }
//** sc5506a_OpenDevice, open device 0
devHandle = sc5506a_OpenDevice(deviceList[0]);
// status = sc5506a_OpenDevice(deviceList[0], &devHandle);  //API >= 2.0

// Free memory
       for(i = 0; i<MAXDEVICES;i++) free(deviceList[i]);
       free(deviceList); // Done with the deviceList
 //
// Do something with the device
// Close the device
status = sc5506a_CloseDevice(devHandle);
```

Function: **sc5506a_RegWrite**

Definition: **int sc5506a_RegWrite(deviceHandle *devHandle, unsigned char commandByte, unsigned long long int instructWord)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned char commandByte (register address)
unsigned long long int instructWord (the data for the register)

Description: **sc5506a_RegWrite** writes the instructWord data to the register specified by the commandByte. See the register map on **Error! Reference source not found.** for more i nformation.

Example: To set the power level to 2.00 dBm:

```
int status = sc5506a_RegWrite(devHandle, RF_POWER, 200);
```

Function: **sc5506a_RegRead**

Definition: **int sc5506a_RegRead(deviceHandle *devHandle, unsigned char commandByte, unsigned long long int instructWord, unsigned int *receivedWord)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned char commandByte (The address byte of the register to write to)
unsigned long long int instructWord (the data for the register)
unsigned int *receivedWord (data to be received)

Description: **sc5506a_RegRead** reads the data requested by the instructWord data to the register specified by the commandByte. See the register map on Table 6 for more information.

Example: To read the status of the device:

```
unsigned int deviceStatus;

int status = sc5506a_RegRead(devHandle,
GET_DEVICE_STATUS,0x00,&deviceStatus);
```

Function: **sc5506a_InitDevice**

Definition: **int sc5506a_InitDevice(deviceHandle *devHandle, bool mode)**

Input: **deviceHandle** *devHandle (handle to the opened device)
bool mode (Set the mode of initialization)

Description: **sc5506A_InitDevice** initializes (resets) the device. Mode = 0 resets the device to the default power up state. Mode = 1 resets the device but leaves it in its current state.

Function: **sc5506a_SetStandby**

Definition: **int sc5506a_SetStandby(deviceHandle *devHandle, unsigned int channel, bool standbyStatus)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name to put into standby)
bool standbyStatus Set to true (1) to set device in standby mode

Description: **sc5506a_SetStandby** puts a channel in standby mode where the power to the analog circuits for that channel are disabled, conserving power.

Function: **sc5506a_SetFrequency**

Definition: **int sc5506a_SetFrequency(deviceHandle *devHandle, unsigned int channel, unsigned long long int frequency)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name of target frequency change)
unsigned long long int frequency (frequency in Hz)

Description: **sc5506a_SetFrequency** sets the channel RF frequency.

Function: **sc5506a_SetPowerLevel**

Definition: **int sc5506a_SetPowerLevel(deviceHandle *devHandle, unsigned int channel, float powerLevel)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name of target power level change)
float powerLevel (set power in dBm)

Description: **sc5506a_SetPowerLevel** sets the value of the desired output power level for the channel.

Function: **sc5506a_SetRfOut**

Definition: **int sc5506a_SetRfOut(deviceHandle *devHandle, unsigned int channel, bool mode)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name of target RF output toggle)
bool mode (disable/enable RF output)

Description: **sc5506a_SetRfOut** enables or disables the RF output on a channel.

Function: **sc5506a_SetAlcMode**

Definition: **int sc5506a_SetAlcMode(deviceHandle *devHandle, unsigned int channel, bool mode)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name for desired ALC mode change)
bool mode (closed or open loop operation of the ALC circuit)

Description: **sc5506a_SetAlcMode** sets the ALC loop to closed or open loop operation for a channel.

Function: **sc5506a_DisableAutoLevel**

Definition: **int sc5506a_DisableAutoLevel(deviceHandle *devHandle, unsigned int channel, bool mode)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int channel (channel name to disable auto-level)
bool mode (defines when to disable auto-level)

Description: **sc5506a_DisableAutoLevel** disables the device from auto adjusting the power level to the current state when frequency is changed. Disabling the auto adjust feature allows the device to return faster after calling **sc5506a_SetPowerLevel**. If the power remains the same for the next tuned frequency, this should not be disabled.

Function: **sc5506a_SetReferenceClock**

Definition: **int sc5506a_SetReferenceClock(deviceHandle *devHandle, bool lockExtEnable, bool refOutEnable)**

Input:
deviceHandle *devHandle (handle to the opened device)
bool lockExtEnable (enables phase locking to an external source)
bool refOutEnable (enables the internal clock to be exported on the "ref out" port)

Description: **sc5506a_SetReferenceClock** configures the reference clock behavior of the device**.**

Function: **sc5506a_SetReferenceDac**

Definition: **int sc5506a_SetReferenceDac(deviceHandle *devHandle, unsigned int dacValue)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int dacValue (14bit value for the reference DAC)

Description: sc5506a_SetReferenceDac set the value of the DAC that tunes the internal reference TXCO. The user may choose to override the value stored in memory for example, to correct for long-term accuracy drift.

Function: **sc5506a_WriteUserEeprom**

Definition: **int sc5506a_WriteUserEeprom(deviceHandle *devHandle, unsigned int memAdd, unsigned char byteData)**

Input:
deviceHandle *devHandle (handle to the opened device)
unsigned int memAdd (memory address to write to)
unsigned char byteData (byte to be written to the address)

Description: **sc5506a_WriteUserEeprom** writes one byte of data to the memory address specified.

Function: **sc5506a_StoreCurrentState**

Definition: **int sc5506a_StoreCurrentState(deviceHandle *devHandle)**

Input:
deviceHandle *devHandle (handle to the opened device)

Description: **sc5506a_StoreCurrentState** stores the current state of the devices as the default power-up state.

Function: **sc5506a_SetAlcDac**

Definition: **int sc5506a_SetAlcDac(deviceHandle *devHandle, unsigned char channel, unsigned int dacValue)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned char channel (channel name of target ALC DAC)
unsigned int dacValue (14 bit value for adjusting the ALC DAC)

Description: **sc5506a_SetAlcDac** writes a value to the ALC DAC to control the RF output level for a channel.

Function: **sc5506a_GetDeviceStatus**

Definition: **int sc5506a_GetDeviceStatus(deviceHandle *devHandle, deviceStatus_t *deviceStatus)**

Input: **deviceHandle** *devHandle (handle to the opened device)
Output: **deviceStatus_t** *deviceStatus (deviceStatus struct)

Description: **sc5506a_GetDeviceStatus** retrieves the status of the device such as phase lock status and current device settings.

Example: Code showing how to use function:

```
deviceStatus_t *devStatus;
devStatus = (deviceStatus_t*)malloc(sizeof(deviceStatus_t));

int status = sc5506a_GetDeviceStatus(devHandle, devStatus);

if(devStatus->vcxoPllLock)
printf("The 100 MHz is phase-locked \n");
else
printf("The 100 MHz is not phase-locked \n");

free(deviceStatus);
```

Function: **sc5506a_GetTemperature**

Definition: **int sc5506a_GetTemperature(deviceHandle *devHandle, float *temperature)**

Input: **deviceHandle** *devHandle (handle to the opened device)
Output: **float** *temperature (temperature in degrees Celsius)

Description: **sc5506a_GetTemperature** retrieves the internal temperature of the device.

Function: **sc5506a_GetAlcDac**

Definition: **int sc5506a_GetAlcDac(deviceHandle *devHandle, unsigned char channel, unsigned int *dacValue)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned char channel (channel name of target ALC DAC)
Output: **unsigned char** *dacValue (the read back byte data)

Description: **sc5506a_GetAlcDac** reads back the current ALC DAC value.

Function: **sc5506a_ReadCalEeprom**

Definition: **int sc5506a_ReadCalEeprom(deviceHandle *devHandle, unsigned int memAdd, unsigned char *byteData)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned int memAdd (EEPROM memory address)

Output: **unsigned char** *byteData (the read back byte data)

Description: **sc5506a_ReadCalEeprom** reads back a byte from a specific memory address of the calibration EEPROM.

Function: **sc5506a_ReadUserEeprom**

Definition: **int sc5506a_ReadUserEeprom(deviceHandle *devHandle, unsigned int memAdd, unsigned char *byteData)**

Input: **deviceHandle** *devHandle (handle to the opened device)
unsigned int memAdd (EEPROM memory address)

Output: **unsigned char** *byteData (the read back byte data)

Description: **sc5506a_ReadUserEeprom** reads back a byte from a specific memory address of the user EEPROM.

Function: **sc5506a_GetDeviceInfo**

Definition: **int sc5506a_GetDeviceInfo(deviceHandle *devHandle, deviceInfo_t *devInfo)**

Input: **deviceHandle** *devHandle (handle to the opened device)

Output: **deviceInfo_t** *devInfo (device info struct)

Description: **sc5506a_GetDeviceInfo** retrieves device information such as serial number, calibration date, revisions, etc.

Function: **sc5506a_GetDeviceParams** (only for software Rev2 or later, firmware rev 4 or later)

Definition: **int sc5506a_GetDeviceParams(deviceHandle *devHandle, deviceParams_t *deviceParams)**

Input: **deviceHandle** *devHandle (handle to the opened device)

Output: **deviceParams _t * deviceParams** (device parameters)

Description: **sc5506a_GetDeviceParams** retrieves device parameters such as frequency and power setting for each channel.

PROGRAMMING THE SERIAL PERIPHERAL INTERFACE (SPI)

The SPI Architecture

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. In other words, 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. $\overline{CS}$ must be asserted for the entire duration of the transfer.

Once a full transfer has been received, the device will proceed to process the command and de-assert low the SERIAL_READY bit. The status of this bit can be read by the host by invoking the SERIAL_READY register (0x04). The SERIAL_READY bit can also be monitored on pin 15 of the SPI interface (digital I/O) connector. While SERIAL_READY is de-asserted low, the device will ignore any incoming commands. It is only ready when the previous command is fully processed and SERIAL_READY is re-asserted high. It is important that the host monitors the SERIAL_READY bit and performs transfers only when it is asserted high to avoid miscommunication.

All data transferred to and from the device are clocked on the falling edge of the clock by default as shown in Figure 3. To clock data in and out on the rising edge of the clock, pin 18 of the digital I/0 connector needs to be grounded. See Table 2 for the connector pin layout. Figure 4 shows a 3 byte SPI transfer initiated by the host; the device is always in slave mode. The CS pin must be asserted low for a minimum period of $5\ \mu s$ before data is clocked in. The clock rate may be as high as 1.0 MHz, however if the external SPI signals do not have sufficient integrity due to cabling problems then the rate should be lowered. SignalCore recommends that the clock rate not exceed 1.0 MHz to ensure proper serial operation. 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. The time required to perform this task is indicated in Figure 4 by T_B, which is the time interval between the end of one byte transfer and the beginning of another. The recommended time delay for T_B is $10\mu s$ or greater. The number of bytes transferred depends on the command. 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 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 to be reset externally. The time required to process a command is also dependent on the command itself. Measured times for command completions are typically between $40\ \mu s$ to $150\ \mu s$ after reception. The user may choose to wait a minimum of $150\ \mu s$ or query the SERIAL_READY bit before sending in another command. The latter is recommended for robustness.

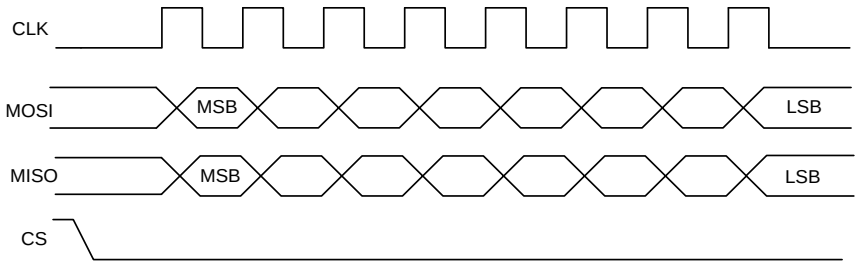
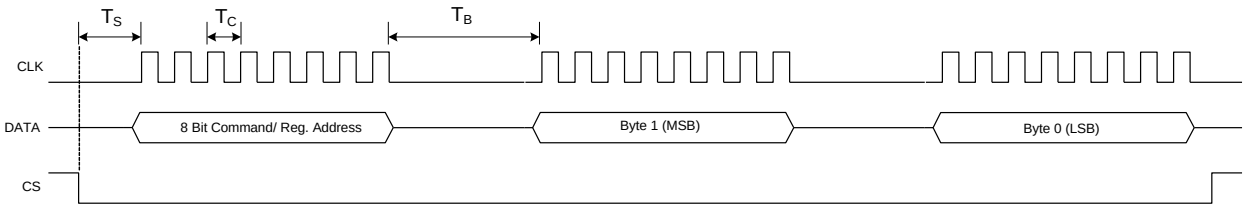


Figure 3. SPI Mode - Data clocked in on falling clock edge.



T_S : Time between CS assertion and first clock cycle >= 5 uS
T_C : Clock period = 1 uS typical (Clock rate depends on signal integrity from host)
T_B : Duration between byte transfer >= 5 uS

Figure 4 SPI timing

Additional SPI Registers

There are two additional registers available for SPI communication as shown in Table 9. Data byte(s) associated with registers can be “zeros” or “ones” - it doesn’t matter which value since the device ignores them. They are only required for clocking out the returned data from the device.

Table 9. Additional SPI registers.

Register Name	Register Address	Serial Range	Bit 7 (MSB)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0 (LSB)
SPI_OUT_BUFFER	0x19	[7:0]	Open							
		[15:8]	Open							
		[23:16]	Open							
		[31:24]	Open							
		[39:32]	Open (> firmware Rev 4)							

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 Table 9. The subsequent bytes contain the data associated with the register. As data from the host is being transferred to the device, data present on its SPI output buffer is simultaneously transferred back, MSB first, via the master-in-slave-out (MISO) line. The data return is invalid for most transfers except for those register commands querying for data from the device. See “Reading the SPI Bus" section below for more information on retrieving data from the device. Figure 5 shows the contents of a single 3 byte SPI command written to the device. Table 5 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”.

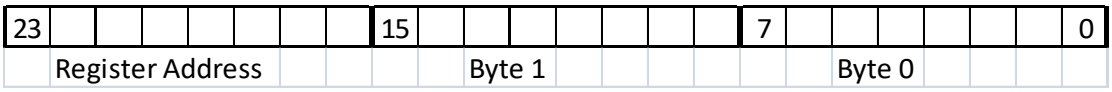


Figure 5. Single 3 byte transfer buffer.

Reading the SPI Bus

Data is simultaneously read back during a SPI transfer cycle. Requested data from a prior command (see Table 10) is available on the device SPI output buffers, and these are transferred back to the user host via the MISO line. To obtain valid requested data would require querying the SPI_OUT_BUFFER, which requires 5 bytes of clock cycles; 1 byte for the device register (0x19) and 4 empty bytes (MOSI) to clock out the returned data (MISO). An example of reading the temperature from the device is shown in Figure 6.

Note: With firmware versions 4.0 or later, the number of bytes read back will extended from 4 to 5 bytes to be able to hold the frequency parameters being read back. Including the 0x19 register, a total of 6 bytes needs to be clock in to properly retrieve the correct values.

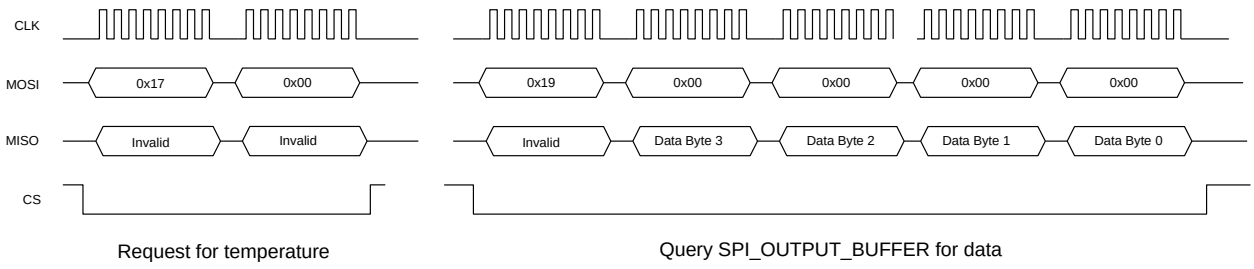


Figure 6. Reading queried data.

In the above example, valid data is present in the last 2 bytes - byte 1 and byte 0. Table 10 shows the valid data bytes associated with the querying register.

Table 10. Valid returned data bytes.

Register Name (Address)	Register Code (Hex)	Byte 4	Byte 3	Byte2	Byte 1	Byte 0
GET_TEMPERATURE	0x17				valid	valid
GET_DEVICE_STATUS	0x18			valid	valid	valid
USER_EEPROM_READ	0x1A					valid
CAL_EEPROM_READ	0x21					valid
GET_ALC_DAC_VALUE	0x39				valid	valid
GET_DEVICE_PARAM	0x25	valid	valid	valid	valid	valid

PROGRAMMING THE RS-232 INTERFACE

The RS-232 version of the SC5506A has a standard interface buffered by an RS-232 transceiver so that it may interface directly with many host devices, such as a desktop computer. The interface connector for RS-232 communication is labeled “Digital I/O” on the front of the panel. Refer to Figure 1 and Table 2 for position and pin-out information. The device communication control set is provided in Table 11 below.

Table 11. RS-232 communication settings.

Baud rate	Rate of transmission. Pin 18 of the Digital IO connector selects the rate. By default if the pin is pulled high or open, the rate is set 56700 at power up or upon HW reset. When the pin is pulled low or grounding it, the rate is set to 115200.
Data bits	The number of bits in the data is fixed at 8.
Parity	Parity is 0 (zero).
Stop bits	1 stop bit.
Flow control	0 (zero) or none.

Writing to the Device via RS-232

It is important that all necessary bytes associated with any one register are fully sent. In other words, if a register requires a total of four bytes (address plus data) then all four bytes must be sent even though the last byte may be a null. The device, upon receiving the first register addressing byte, will wait for all the associated data bytes before acting on the register instruction. Failure to complete the register transmission will cause the device to behave erratically or hang. Information for writing to the configuration registers is provided in Table 5.

When the device receives all the information for a register and finishes performing its instruction, it will return a byte back to the host. Querying this return byte ensures that the prior configuration command has been successfully executed and the device is ready for the next register command. It is important to clear the incoming RX buffer on the host by querying or force flushing it to avoid incoming data corruption of querying registers. The return byte value is 1 for a successful configuration and 0 for an unsuccessful configuration.

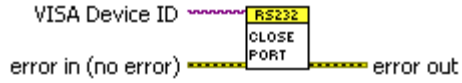
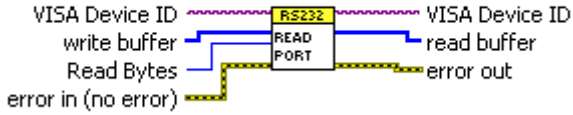
Reading from the Device via RS-232

To query information from the device, the query registers are addressed, and data is returned. Returned data vary in length, which are dependent on the register call. Table 6 contains the query register information. As with the configuration registers, it is important that the data byte(s) associated with the query registers are sent even if they are nulls. The returned data length is also detailed in the “Querying the SC5506A: Writing to Request Registers” section.

Using the LabVIEW Functions and NI-VISA

Functions for RS-232 control are provided in LabVIEW and use NI-VISA. The low level port control functions are unique to RS-232 and are contained in the subdirectory **UsartPort**. These functions are listed in Table 12 below for convenience. All provided LabVIEW VI functions are not protected, so the user may open them to understand how the register calls are made.

Table 12. LabVIEW RS-232 port access functions.

Function	Description
Rs232OpenPort.vi 	Opens the VISA session to the serial port associated with the device. Option to select between 2 Baud rates.
Rs232ClosePort.vi 	Closes the VISA session associated with the serial port.
Rs232WritePort.vi 	Writes the data bytes in the buffer to the opened port. The data buffer is a byte array. Most significant data is sent first.
Rs232ReadPort.vi 	Writes the query register data and reads back the data associated with the register. The buffers are byte arrays. Most significant data is sent and received first. The number of bytes to be read back from the queried register needs to be specified.

CALIBRATION & MAINTENANCE

The SC5506A is factory calibrated and ships with a certificate of calibration. SignalCore strongly recommends that the SC5506A be returned for factory calibration every 12 months or whenever a problem is suspected. The specific calibration interval is left to the end user and is dependent upon the accuracy required for a particular application.

SC5506A calibration data is stored in the RF module (metal housing). Therefore, changing or replacing interface adapters will not affect unit calibration. However, SignalCore maintains a calibration data archive of all units shipped. Archiving this data is important should a customer need to reload calibration data into their device for any reason. SignalCore also uses the archived data for comparative analysis when units are returned for calibration.

Should any customer need to reload calibration data for their SC5506A, SignalCore offers free support through support@signalcore.com. SignalCore will provide a copy of the archived calibration data along with instructions on how to upload the file to the SC5506A.

The SC5506A requires no scheduled preventative maintenance other than maintaining clean, reliable connections to the device as mentioned in the “Getting Started” section of this manual. There are no serviceable parts or hardware adjustments that can be made by the user.

REVISION NOTES

Rev 1.0.0	Initial release.
Rev 1.0.2	Added SPI programming Instructions.
Rev 1.1.0	Added documentation on the usage of pin 18 of the digital IO connector to be used for setting the SPI mode on startup or reset.
Rev 1.1.3	Combined in RS-232 documentation.
Rev 1.1.4	Corrected Table 10.
Rev 1.1.5	Corrected section on SPI communication, corrected for missing references.
Rev 1.1.6	Updated EN and IEC testing standards and EU Directive references, minor rewrites throughout manual.
Rev 1.2.0	Removed legacy SPI information
Rev 2.0.0	Added revision 2 API information to take advantage of firmware revision 4 and later
Rev 2.1.0	Added note for pin 8 in Table 2 as reset_b
Rev 2.1.1	Corrected SPI information to include device with firmware rev 4 or later