SignalCore™

PRESERVING SIGNAL INTEGRITY

Datasheet

SC801A & SC802A

0.625 / 1.25 GHz to 10 / 20 GHz

Rev 1.1

www.signalcore.com

Table of Contents

1 Definition of Terms

The following terms are used throughout this datasheet to define specific conditions:

Specification (spec)	Defines expected statistical performance within specified parameters which account for measurement uncertainties and changes in performance due to environmental conditions. Protected by warranty.
Typical Data (typ)	Defines the expected performance of an average unit without specified parameters. Not protected by warranty.
Nominal Values (nom)	Defines the average performance of a representative value for a given parameter. Not protected by warranty.
Measured Values (meas.)	Defines the expected product performance from the measured results gained from individual samples.

Specifications are subject to change without notice. For the most recent product specifications, visit www.signalcore.com.

2 Description

The SC801A and SC802A are part of the nanoSynth® family from our nanoCircuits® line of products. This fully integrated broadband CW signal synthesizer contains the same proprietary core synthesis architecture that is used in other modular SignalCore devices and is designed into a rugged and miniature surface mountable package measuring 2.75” x 1.75” x 0.31”. The SC801A output frequency range is 625 MHz to 10.0 GHz, while the SC802A range is from 1.25 GHz to 20.0 GHz. Tuning at 1 mHz resolution, their multiple loop architecture improves phase noise over those that use single loop integer-N or fractional-N by 15 dB to 20 dB. The typical measured phase noise at 10 kHz offset from a 10 GHz carrier is less than-115 dBc/Hz. Additionally, these devices have frequency sweep and list sweep up to 2048 points. Communicating to the devices is flexible as they have 3 built in communications interfaces: USB, SPI, and UART (RS232).

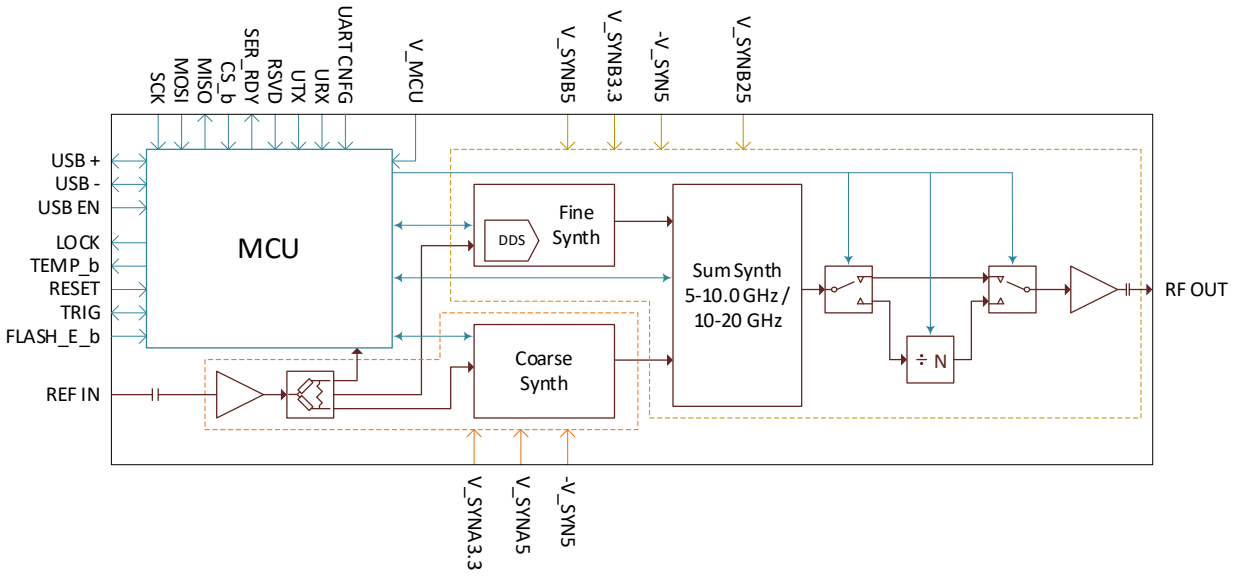


Figure 1. SC801A / SC802A Functional Block Diagram

Product Features

- 625 MHz to 20 GHz
- 1 mHz frequency tuning Resolution
- Phase Noise < -115 dBc/Hc at 10 KHz Offset at 10 GHz
- Rugged and miniature 2.75” x 1.75” x 0.31” SMT package
- Frequency sweep/list mode
- USB, UART and SPI

Applications

- Automated device / IC testers
- Test and measurement equipment
- Wireless communication equipment
- Frequency converter local oscillator
- Digital data converter clock source
- Quantum computing
- Network equipment

3 Specifications

3.1 Spectral Specifications

RF Output Range

SC801A	625 MHz to 10.0 GHz
SC802A	1.25 GHz to 20.0 GHz

Tuning

Resolution	1 mHz
Speed (settled to 0.1 ppm)	< 500 us

Phase Noise[1] (dBc/Hz) typ. Add 5 dB for max.

Offset	1 GHz	5 GHz	10 GHz1	20 GHz
100 Hz	-100	-86	-80	-74
1 kHz	-123	-110	-105	-99
10 kHz	-132	-121	-117	-112
100 kHz	-133	-122	-118	-114
1 MHz	-130	-120	-114	-112
10 MHz	-150	-140	-134	-133
100 MHz	-153	-152	-150	-148

Sideband Phase Spurs[2]

< 1 MHz	< 50 dBc typical
1 MHz to 5 MHz	< 60 dBc typical

Reference Frequency 200 MHz

Harmonics & Spurs

2nd Harmonics	< -15 dBc typical
Sub Harmonics	< -60 dBc typical
Far Out Spurs	< -65 dBc typical

[1] The phase noise < 100 kHz offset is dependent on the external 100 MHz reference source. The phase noise specifications are based on a reference signal with the following typical phase noise levels normalized to 100 MHz.

Offset	*100 Hz*	*1 KHz*	*10 kHz*	*100 kHz*	*1 MHz*
dBc/Hz	*-120*	*-150*	*-168*	*-170*	*-170*

[2] Phase spurs may rise as high as -45 dBc at some frequencies when operating in harmonic generation mode. Set the loop gain to low or changing the generation mode to Int-N may reduce them.

3.2 Amplitude Specifications

Output Power

1.25 GHz	+7 dBmtypical
5.0 GHz	+7 dBmtypical
10.0 GHz	+5 dBmtypical
20.0 GHz	+ 0 dBm typical

Reference Power

Minimum	< +3 dBm typical
Typical	< +7 dBm typical
Maximum	< +10 dBm typical

3.3 Electrical Specifications

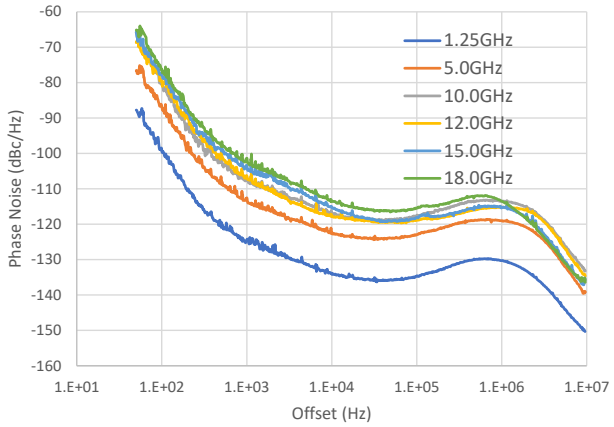
Voltage and Current					
	Parameter	Minimum	Typical	Maximum	Unit
	V_SYNx3.3	3.15	3.3	3.5	V
	V_MCU	3.2	3.3	3.4	V
	V_SYNx5.0	4.95	5.0	5.3	V
	-V_SYN5.0	-5.1	-5.0	-4.9	V
	V_SYN25.0	24	25	27	V
	I_SYN3.3	1.110	1.120	1.140	A
	I_SYN5.0	.770	.775	.785	A
	-I_SYN5.0	-.020	-.026	-.024	A
	I_SYN25.0	.020	.022	.024	A
	Power dissipation	8.00	8.25	8.50	W
	Low input logic	-0.3		0.8	V
	High input logic	2.0		3.6	V
	Low output logic	0.0		0.4	V
	High output logic	2.9		3.3	V

Absolute Maximum Ratings

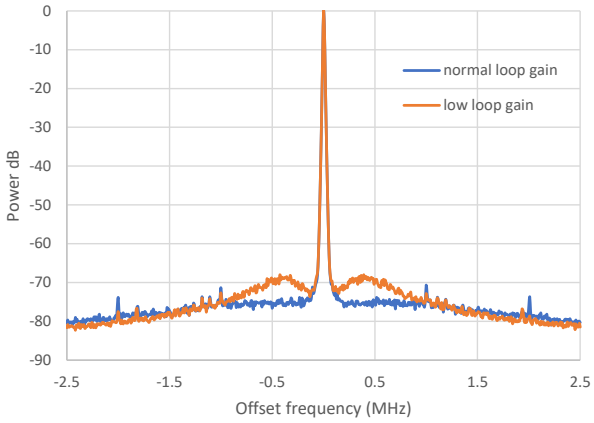
Continuous Power Dissipation	10 W
Storage Temperature	-20 to 90 ^{0}C
Operating Temperature	0 to +80 ^{0}C

3.5 Measured Data

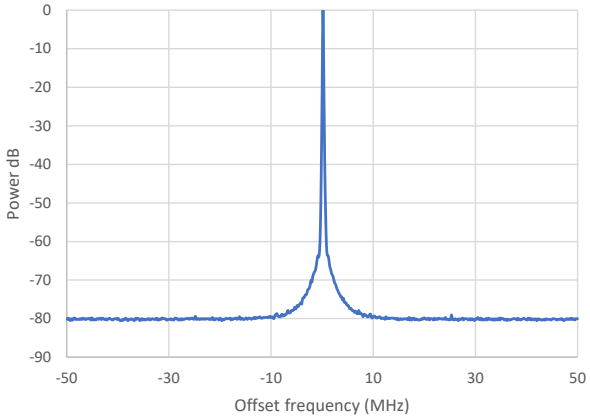
3.5.1 Phase noise, normal loop gain



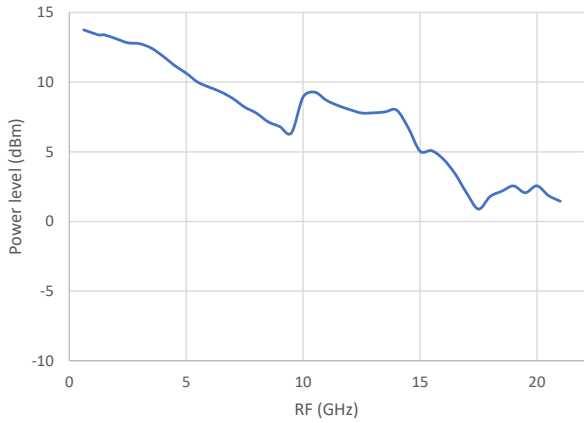
3.5.2 CF = 15 GHz, Span 5 MHz



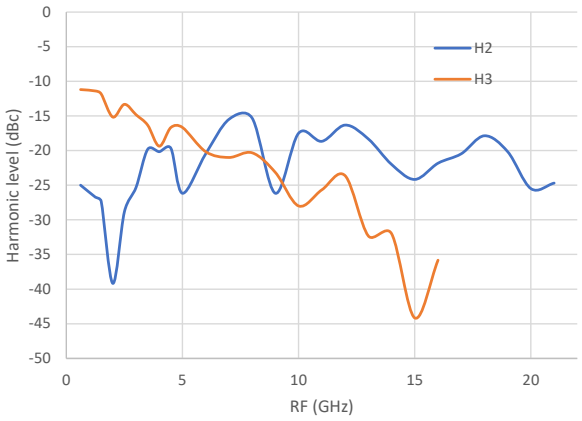
3.5.3 CF = 15 GHz, Span 100 MHz



3.5.4 Output Power Level



3.5.5 Harmonics

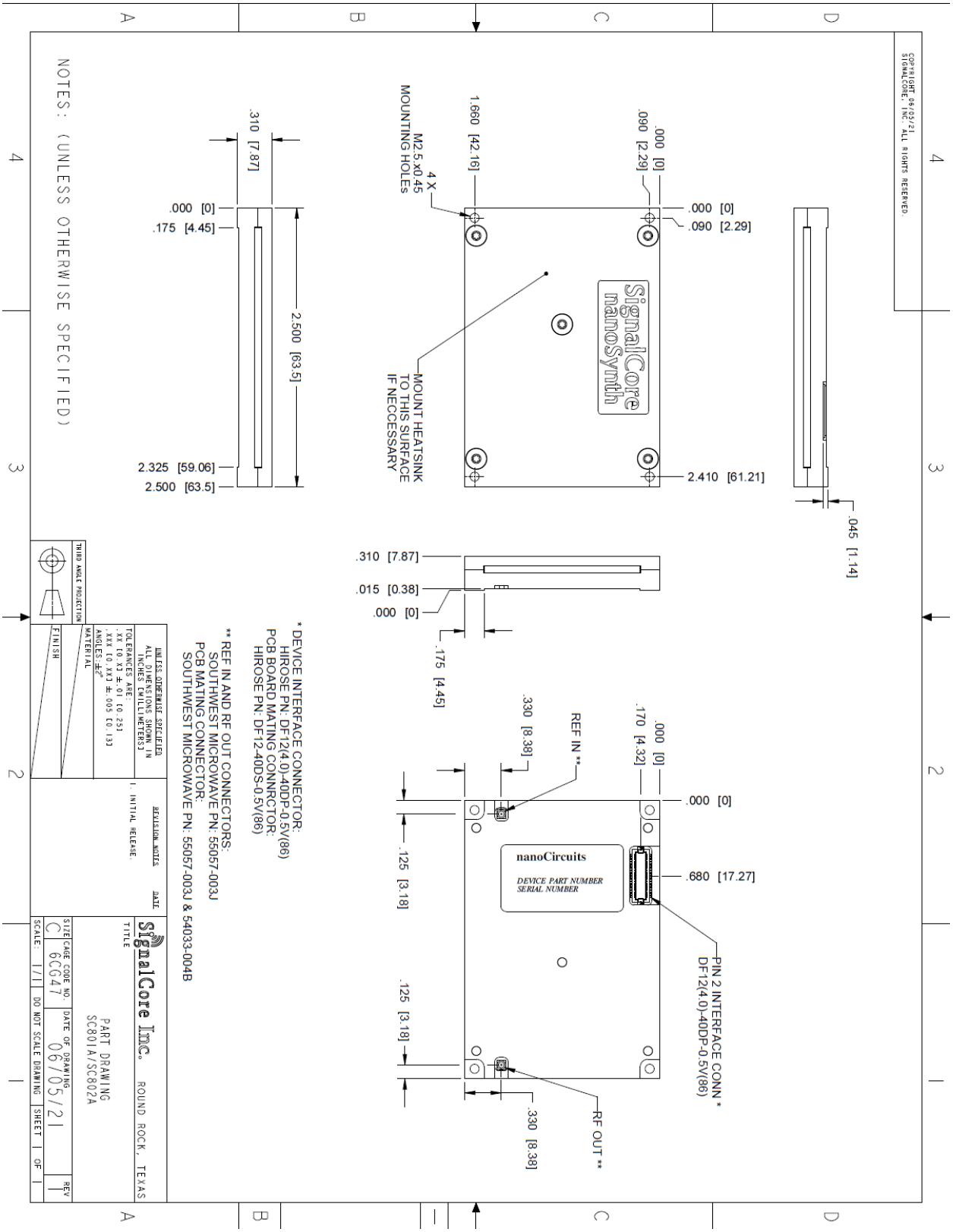


3.5.6 Sub-harmonics @ CF= 10-20 GHz

3.6 40 Pin Connector Description

Pin Number	Function	Description
6,7,15,18,27,34,40	GND	Must be connected to RF or DC ground. It is important to place as many ground vias as possible in or around these ground pads to improve signal performance as well as thermal conduction from the device to the board.
2,4	V_SYN25	Supply for V_C of 5-10 GHz or 10-20 GHz VCO
8,10,12	V_SYNB5	Supply for the fine and summing synthesizer
14,16	V_SYNA5	Supply for the coarse synthesizer
20,22	-V_SYN5	Supply for all synthesizers
24,26,28	V_SYNB3.3	Supply for the fine and summing synthesizer
30,32	V_SYNA3.3	Supply for the coarse synthesizer
36,38	V_MCU	Supply for the microprocessor and digital interface
1	USB_N	USB negative line
3	USB_P	USP positive line
5	USB_EN	Pull high to enable USB
9	UTX	UART Transmit
11	URX	UART Receive
13	$\overline{\text{FLASH ERASE}}$	This pin must always be pulled low on power up for the device to operate. If this pin is pulled high and reset (pin 17) is toggled low, the device flash memory will be erased. When memory is erased, it will require reflashing with firmware.
17	$\overline{\text{RESET}}$	Hardware reset
19	UART_CFG	Serial (RS232) config pin 0
21	LCK_STATUS	Phase-lock Status
23	OVER_TEMP	Over temperature indicator
25	$\overline{\text{TRIG\_IN}}$	Hardware input trigger for list/sweep mode
29	TRIG_OUT	Trigger signal out on step or cycle
31	SERIAL_RDY	Indicate high if device is ready for next register write
33	$\overline{\text{CS}}$	SPI device select
35	MOSI	SPI receive
37	MISO	SPI transmit
39	SCK	SPI clock

3.7 Mechanical Data



3.8 Product Evaluation

A full development board is available from SignalCore to evaluate either the SC801A or SC802A. Following this link for more information: https://www.signalcore.com/nano_hb2.html#eval

3.9 Ordering Information

SC801A nanoSynth 10 GHz SMT Synthesizer Module Kit	7100048-01
SC802A nanoSynth 20 GHz SMT Synthesizer Module Kit	7100049-01
SC801A nanoSynth- HB Evaluation Development Kit	7100152-01
SC802A nanoSynth- HB Evaluation Development Kit	7100153-01
nanoSynth HB2 Heatsink	7100154-01

3.9.1 Module Kit Contents

- SC801A or SC802A Module 1
- HB2 Heatsink 1
- Software in USB drive 1

3.9.2 Evaluation Development Kit Contents

- HB2 EVK card 1
- SC801A or SC802A 1
- HB2 Heatsink 1
- SC803A reference source module 1
- Power Supply 12V, 4A 1
- USB cables 2
- Software in USB drive 1

4 Theory and Operation

The SC801A/SC802A is a very small and high performing surface mount synthesizer with easy to program register-level control. It functions as a standard synthesized source with the added capability of a sweep/list mode that makes it ideal for applications ranging from automated test systems to telecommunication equipment to scientific research labs. Being small and fully integrated, this source is the ideal solution for board-level designs that require a high-performance RF source(s) without demanding a large investment and engineering effort. *Figure 2* shows the block diagram of the device, and the following sub-sections provide details of its operation.

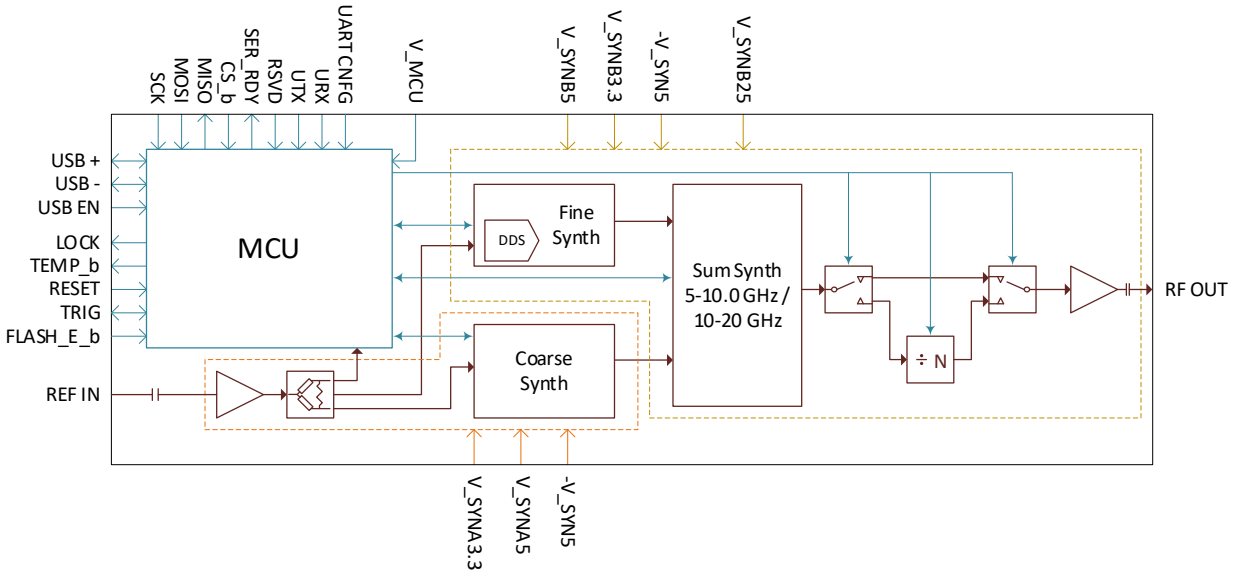


Figure 2. Block Diagram of the SC801A/SC802A

4.1 RF Generation

The SC801A/SC802A is a low phase noise and low spur synthesizer, using a hybrid architecture of multiple phase lock loops (PLL), a harmonic multiplier, and a direct digital synthesizer (DDS). The synthesizer is sub-divided into three separate synthesizers: the fine synthesizer providing fine 1 mHz tuning, a coarse synthesizer that steps at about 25 MHz, and the sum synthesizer that combines the coarse and the fine synthesizers. Using a hybrid architecture with well-shielded cavities improves the overall phase noise performance and reduces the spurious signal content of this synthesizer. The sum synthesizer frequency range, depending on whether the device is an SC801A or SC802A, is either 5 GHz to 10 GHz or 10 GHz to 20 GHz respectively. Following the sum synthesizer is a switch to route the signal for further frequency division of 2, 4, or 8, bringing the signal down as low as 625 MHz or 1.25 GHz. Finally, the signal is buffered out via an RF amplifier.

Signals are synthesized from an external 200 MHz reference clock. It is important to have a very low phase noise and spuriously clean reference source so that the synthesized signal conforms to the purity specifications of the product. Table 1 provides the typical phase noise specification for an external reference source whose values are normalized to 100 MHz. A SignalCore SC803A reference source was used for the qualification of this product.

Table 1. Reference Phase Noise Requirement

Offset Frequency	PN (Normalized to 100 MHz)	PN at 200 MHz
100	-120 dBc/Hz	-114 dBc/Hz
1000	-150 dBc/Hz	-144 dBc/Hz
10,000	-168 dBc/Hz	-162 dBc/Hz
100,000	-170 dBc/Hz	-164 dBc/Hz
1,000,000	-170 dBc/Hz	-164 dBc/Hz

4.2 Modes of Generation

The SC801A/SC802A has both single fixed tone and list mode operation. In single fixed tone mode, it operates as a normal synthesizer where the user writes the frequency (RF_FREQUENCY) register to change the frequency. In list mode, the device is triggered to automatically run through a set of frequency points that are either entered directly by the user or pre-computed by the device based on user parameters. Configuration of the device for list mode operation is accomplished by setting up the LIST_MODE_CONFIG register.

4.2.1 Sweep Function

When frequency points are generated based on the start/stop/step set of frequencies, this is (in the context of this product) known as putting the device into sweep. When the sweep function is enabled, the frequency points are incrementally stepped with a constant step size either in a linearly increasing or decreasing fashion.

4.2.2 List Function

The list function requires that the frequency points are read in from a list provided by the user. The user will need to load the frequency points into the list buffer via the LIST_BUFFER_WRITE register, or have the device read into it the frequency points from the EEPROM.

4.2.3 Sweep Direction

The sweep can be chosen to start at the beginning of a list and incrementally step to the end of the list or vice versa.

4.2.4 Sweep Waveform

The list of frequency points may be swept in a saw-tooth or triangular manner. If sawtooth is selected, the device will return to the starting point upon reaching the last frequency point. Plotting frequency versus time reveals a sawtooth pattern. If triangular is selected, the device will sweep linearly from the starting point, then reverse its direction after the last (highest or lowest) frequency and sweep backwards toward the start point, mapping out a triangular waveform on a frequency versus time graph.

4.2.5 Dwell Time

The dwell time at each frequency in sweep mode is determined by writing to the LIST_DWELL_TIME register. The dwell time step increment is 500 µs. However, the recommended minimum dwell time is 1 ms to allow sufficient time for the signal to settle before a measurement is made. When the user list is used, each list frequency has a corresponding dwell time, allowing variable dwell times. Due to the size limitation of the onboard RAM, it is not possible to have a pre-calculated configuration parameters list that could be used to program the various functions of the device, which would decrease the setup time of the device for frequency changes. As a result, for each frequency change, the configuration parameters are dynamically computed. This overhead computational time to handle the mathematics, triggers, timers, and interrupts may increase the effective settling time close to or slightly exceeding 500 µs.

4.2.6 List Cycles

The number of repeat cycles for a sweep or list is set by writing the LIST_CYCLE_COUNT register. Writing the value 0 to the register will cause the device to repeat the sweep/list forever until a trigger is sent or the RF mode is changed to single fixed tone mode via the SYNTH_MODE register. Upon completion of a cycle, the frequency may be set to end on the last frequency point or return back the starting point. This cycle ending behavior is configured with bit [3] of the LIST_MODE_CONFIG register.

4.2.7 Trigger Sources

The device may be set up for software or hardware triggering. This is defined in bit [4] of the LIST_MODE_CONFIG register. If software trigger is selected, writing the LIST_SOFT_TRIGGER register will trigger the device to perform the sweep/list function defined in the LIST_MODE_CONFIG register. The device may also be triggered via pin 25, the hardware trigger pin ($\overline{\text{TRIG\_IN}}$). Hardware triggering occurs on a high to low transition state of this pin.

4.2.8 Trigger In Modes

The device may be triggered to start a sweep or list then uses the next trigger to stop it. In triggered start/stop mode, alternating triggers will start and stop the sweep/list. In this mode, start triggering will always return the frequency point to the beginning of the sweep/list. It does not continue from where it left off from a stop trigger. The device may also be triggered to step to the next frequency with each start trigger. This is known as the triggered step mode and bit [5] of the LIST_MODE_CONFIG register configures it. When step triggering has started, changing the RF mode to single fixed tone will take the device out of step trigger state before the cycles are completed.

4.2.9 Trigger Out Enable

The device can be set to send out a low to high transition signal when the configuration of a frequency by the device is completed; that is, it has completed all necessary computations, and has successfully written data to the appropriate components. This trigger pulse is sent on the completion of every step frequency. This trigger signal is present on pin 29 (TRIG_OUT).

4.3 Communication Interfaces

The device has 3 communication interfaces, USB, UART, and SPI and are all enabled by default. The USB can be disabled by pulling the USN_EN pin low. It is also used for firmware update, so it is strongly recommended to wire its interface pins to a connector even though it may not be used.

4.3.1 SPI Interface

PINS 33, 35, 37, and 39 are configured as an SPI interface that corresponds to $\overline{CS}$, MOSI, MISO, and SCK respectively. When the device receives data, it takes time to execute the command instruction associated with the register, such as setting a new frequency. While the device is busy, the SERIAL_RDY pin (#31) will assert low and returns high upon execution completion. Detailed SPI read and write operations are discussed in detail in the section.

4.3.2 UART Interface

Pins 9 and 11 are configured as a 2 wire UART serial interface and they correspond to UTXD and URXD respectively, which are the transmit and receive lines. When a register is written, the device takes time to execute the command instruction associated with it, such as setting a new frequency. While the device is busy, the SERIAL_RDY pin (#31) will assert low and returns high upon execution completion. Detailed UART read and write operations are discussed in detail in the *Universal Asynchronous Receive-Transmit (UART) Interface* section.

4.3.3 USB Interface

The device has a built-in USB controller configured in client mode. The two wires USB- and USB+ can be routed directly to a USB connector or an embedded host port. The transfer types supported by the device are control and bulk. The USB port can be turned off by grounding or pulling low pin 5. More information on the use of the USB interface is provided in the

USB Interface section.

5 Device Registers

Communication to the SC801A/SC802A 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, while the query registers, register 0x20 to 0x24, request the device to perform an operation and return its results. The tables below list the device registers and their operation. All registers are 8 bytes long. The register address is the first byte, followed by 7 bytes of data.

5.1 Register 0x01 INITIALIZE

Bits	Type	Name	Width	Description
[0]	WO	Initialize	1	0 = Device reprograms all components to the current state 1 = Device resets to power on state
[55:1]	WO		55	Zeros

5.2 Register 0x05 LIST_MODE_CONFIG

Bits	Type	Name	Width	Description
[0]	WO	Use Buffer List	1	0 = The device computes the frequency points using the start, stop, and step frequencies. 1 = Device gets its frequency points from the list buffer uploaded via the LIST_BUFFER_WRITE register (0x0B).
[1]	WO	Sweep Direction	1	0 = Forward. In the forward direction, the sweep starts from the lowest start frequency or the beginning of the list buffer. 1 = Reverse. In the reverse direction, the sweep starts with the stop frequency and steps down toward the start frequency or starts at the end and steps toward the beginning of the buffer.
[2]	WO	Sweep Pattern	1	0 = Sawtooth waveform. Frequency returns to the beginning frequency upon reaching the end of a sweep cycle. 1 = Triangular waveform. Frequency reverses direction at the end of the list and steps back towards the beginning to complete a cycle.
[3]	WO	Return to Start	1	0 = Stop at end of sweep/list. The frequency will stop at the last point of the sweep/list. 1 = Return to start. The frequency will return and stop at the beginning point of the sweep/list after a cycle.

Bits	Type	Name	Width	Description
[4]	WO	Soft/Hardware Trigger	1	0 = Software trigger. Software trigger can only be used to start and stop a sweep/list cycle. It does not work for step-on-trigger mode. 1 = Hardware trigger. A high-to-low transition on the TRIG pin will trigger the device. It can be used for both start/stop or step-on-trigger functions.
[5]	WO	Trigger Mode	1	0 = Start/Stop behavior. The sweep starts and continues to step through the list for the number of cycles set, dwelling at each step frequency for a period set by the LIST_DWELL_TIME register. The sweep/list will end on a consecutive trigger. 1 = Step-on-trigger. The device will step to the next frequency on a trigger. Upon completion of the number of cycles (set by the LIST_CYCLE_COUNT register), the device will exit from the stepping state and stop. Further triggering will set the device back into the stepping state. To exit the stepping state and stop before reaching the end of a cycle, a change in the SYNTH_MODE register bit 0 to high is needed. See register (0x14) for more information.
[6]	WO	Trigger Out Enable	1	Enable the trigger output pin to send out a 50us pulse on every frequency change.
[55:7]	WO	Reserved	51	Set all bits to 0.

5.3 Register 0x06 LIST_START_FREQ

Bits	Type	Name	Width	Description
[55:0]	WO	List Start Frequency	56	Sets the start frequency for a sweep in mHz. The start frequency should always be lower than the stop frequency. The Sweep Direction bit [1] of register 0x05 should be used to determine where the sweep should begin.

5.4 Register 0x07 LIST_STOP_FREQ

Bits	Type	Name	Width	Description
[55:0]	WO	List Stop Frequency	56	Sets the stop frequency for a sweep in mHz. The stop frequency should always be greater than the start frequency. The Sweep Direction bit [1] of

Bits	Type	Name	Width	Description
				register 0x05 should be used to determine where the sweep should begin.

5.5 Register 0x08 LIST_STEP_FREQ

Bits	Type	Name	Width	Description
[55:0]	WO	List Step Frequency	56	Sets the step frequency for a sweep in mHz. The step size should not exceed the difference between the start and stop frequencies.

5.6 Register 0x09 LIST_DWELL_TIME

Bits	Type	Name	Width	Description
[31:0]	WO	List Dwell Time	32	Sets the dwell time at each step frequency. The dwell time is incremented in 500µs increments. For example, to produce a 10ms dwell time the value written to this register is 20d.
[55:32]	WO		24	Zero

5.7 Register 0x0A LIST_CYCLE_COUNT

Bits	Type	Name	Width	Description
[31:0]	WO	List Cycle Count	32	0 = Cycle indefinitely. This will set the device to cycle forever. None zero number will set the number of cycles the device will sweep or step through the list, then stop. This applies for both start-stop and step trigger modes.
[55:32]	WO		24	Zero

5.8 Register 0x0B LIST_BUFFER_WRITE

Bits	Type	Name	Width	Description
[54:0]	WO	Buffer Frequency	55	Writing this register stores the frequency and dwell time pair of points into the list buffer held in RAM. Writing 0x00 to this buffer resets the pointer to buffer location [0] and enables storing data. Consecutive non-zero writes to this register will increase the buffer counter up to 2047.

Bits	Type	Name	Width	Description
				Further writes beyond this point are not recognized. Writing 0xFFFFFFFFFFFFFF to this register at any time will terminate the write process and stops the pointer increment. The value at which the pointer stops is the new count of list frequency points.
[55]	WO	Select	1	If 0, [54:0] is frequency data in mHz If 1, [31:0] is dwell time in 500µs

5.9 Register 0x0C LIST_BUF_MEM_TRNSFER

Bits	Type	Name	Width	Description
[0]	WO	Transfer Direction	1	0 = Transfers the contents of the list buffer into EEPROM memory. The size of the transfer is set by the list frequency points. 1 = Transfers the contents from EEPROM memory to the list buffer (in RAM)
[55:1]	WO	Reserved	55	Set all bits to 0

5.10 Register 0x0F LIST_SOFT_TRIGGER

Bits	Type	Name	Width	Description
[55:0]	WO	Reserved	56	Set all bits to 0. Calling this register provides a soft trigger to the device.

5.11 Register 0x10 RF_FREQUENCY

Bits	Type	Name	Width	Description
[55:0]	WO	Frequency Word	56	Sets the single fixed tone frequency in mHz

5.12 Register 0x11 RF_PHASE (4 Bytes)

Bits	Type	Name	Width	Description
[14:0]	WO	Phase Word	14	Change the phase of the signal in degrees 0 – 360 in 1 deg step. If a phase change of 270° is needed, increment to 90°, 180°, then 270° (3 steps) instead of a single increment from 0° to 270°.

Bits	Type	Name	Width	Description
[15]	WO	SIGN BIT	1	0 = positive, 1 = negative

5.13 Register 0x12 STORE_DEFAULT_STATE

Bits	Type	Name	Width	Description
[55:0]	WO		56	Set all bits to 0. Calling this register will store the current configuration into memory. On reset or power up these values are read from memory and set as the default values. These values are: • RF Frequency • List Mode Configuration • Synth Mode • Start/Stop/Step Frequency • Dwell Time • Sweep/List Cycles • List Buffer from EEPROM

5.14 Register 0x13 SYNTH_SELF_CAL

Bits	Type	Name	Width	Description
[0]	WO	Device Standby	1	0 = Calibrates the coarse synthesizer 1 = Calibrates the sum synthesizer
[55:1]	WO	Reserved	55	Set all bits to 0

5.15 Register 0x14 SYNTH_MODE

Bits	Type	Name	Width	Description
[0]	WO	RF Mode	1	0 = Single fixed tone mode. This mode must be set to change the frequency value via register 0x10. 1 = Sweep/list mode. In this mode, writing to register 0x10 will be unresponsive. This register must be called first for sweep/list triggering to function.
[1]	WO	Synth Loop Gain	1	0 = Normal 1 = Low
[2]	WO	Lock Mode	1	0 = Harmonic Lock1 = Int-N Lock

Bits	Type	Name	Width	Description
[3]	WO	Freq Resolution	1	0 = 1 Hz step (default) 1 = 0.001 Hz step
[4]	WO	Auto Spur Suppress	1	Only in harmonic lock mode. Enable it to have device bounce between lock modes to minimize spurs.

5.16 Register 0x15 SERIAL_CONFIG

Bits	Type	Name	Width	Description
[3:0]	WO	Baud rate index	4	0 = 57600 1 = 115200 2 = 19200 3 = 38400 4 = 230400 5 = 460800 6 = 921600 7 = 1843200
[55:1]	WO	Reserved	55	Set all bits to 0

5.17 Register 0x20 GET_DEVICE_PARAM

Bits	Type	Name	Width	Description
[3:0]	WO	Get Device Parameters	4	Writing this register will place the requested contents into the output buffer. In the case of USB and RS232, all 8 bytes are sent back and need to be read to clear host buffers. In the case of SPI, a second query to the SERIAL_OUT_BUFFER (0x25) register is required to transfer its contents and clear the output buffer. 0 = Device status 1 = Current fix tone frequency 2 = Current phase in degrees 3 = Sweep start frequency in 4 = Sweep stop frequency in 5 = Sweep step frequency in 6 = Sweep dwell time 7 = Sweep cycles

Bits	Type	Name	Width	Description
				8 = Sweep Points 9 = List Buffer Points 10 = List Mode Config
[55:4]	WO	Reserved	52	Zero
[63:0]	RO	Return Data for parameters values 1 – 9	64	Valid data for the requested parameter: 1 Fix tone freq – 64-bit unsigned (mHz) 2 Phase – unsigned 32-bit (degrees) 3 Start freq – unsigned 64-bit (mHz) 4 Stop freq – unsigned 64-bit (mHz) 5 Step freq – unsigned 64-bit (mHz) 6 Dwell time – unsigned 32-bit 7 Sweep Cycles – unsigned 32-bit 8 Sweep Points – unsigned 32-bit 9 List Buffer Points – unsigned 32-bit
[63:0]	RO	Return Data for parameter value 0	64	Valid data for the device status: [0] Sum PLL status [1] Coarse harmonic PLL status [2] Fine PLL status [3] Coarse fracN PLL status [4] Coarse Ref PLL status [5] MCU PLL status [8] RF mode (fixed or sweep) [9] Loop gain [10] Lock Mode [11] Freq resolution mode [12] Spur suppress mode [13] List mode running status – is it running? [63: 14] Invalid
[63:0]	RO	Return Data for parameter value 10	64	List mode configuration: [0] Use buffer list? [1] Sweep direction [2] Sweep pattern [3] Return to start [4] HW trigger?

Bits	Type	Name	Width	Description
				[5] Trigger Mode [6] Trigger Out [63:7] Invalid

5.18 Register 0x21 DEVICE_INFO

Bits	Type	Name	Width	Description
[1:0]	WO	Device Info	2	Writing this register will place the requested contents into the output buffer. Contents are immediately available for USB read. The contents effectively occupy 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 = Obtain the product serial number, model option, and model number 1 = Obtain the hardware and firmware version 2 = Obtain the manufacture date
[55:2]	WO	Reserved	6	
[63:0]	RO	Data for Parameter 0	64	Data format for the serial number, model option, and model number: [31:0] Product serial number. Convert to string of its hexadecimal presentation. [47:32] Model option. Custom options. [63:48] Model number. 0 = SC801A, 1 = SC802A
[63:0]	RO	Data for Parameter 1	64	Data format for the firmware and hardware versions: [7:0] FW fix [15:8] FW minor [23:16] FW major [39:32] HW fix [47:40] HW minor [55:48] HW major
[63:0]	RO	Data for parameter 2	64	[7:0] day [15:8] month

Bits	Type	Name	Width	Description
				[23:16] year (add 2000)

5.19 Register 0x22 DEVICE_TEMPERATURE

Bits	Type	Name	Width	Description
[55:0]	WO		64	Set all bits to 0

[63:0]	RO	Data in	63	[15:0] = data
				Calculate the temperature as follows:
				if (data & 0x2000)
				temp = (float)(((data&0x1FFF)-0x1FFF)/32.0);
				else
				temp = (float)((data&0x1FFF)/32.0);
				[63:16] invalid

5.20 Register 0x23 LIST_BUFFER_READ

Bits	Type	Name	Width	Description
[15:0]	WO	Buffer Address	16	The data point (0 – 2047) to read
[16]	WO	Freq or dwell time	1	0 = Freq 1 = Dwell time
[55:17]	WO			
[63:0]	RO	Data in	63	If Freq: [63:0] frequency in mHz If Dwell time: [31:0] time in 0.5 ms

5.21 Register 0x24 CAL_EEPROM_READ

Bits	Type	Name	Width	Description
[15:0]	WO	Start EEPROM Address	16	
[55:16]				

Bits	Type	Name	Width	Description
[63:0]	RO	Data in	64	8 bytes of data from the starting EEPROM address

5.22 Register 0x25 SERIAL_OUT_BUFFER

Bits	Type	Name	Width	Description
[55:0]	WO	Serial Out Buffer	56	Set all bits to 0. Use of this register is only available for the SPI interface.
[63:0]	RO	Request Data	64	The data clocked back are the contents requested by the 0x20 to 0x24 registers.

Registers 0x20 to 0x24 are query registers. With the SPI interface, the write-only portion of the register must be written first, followed by writing register 0x25 to clock back the requested data. With USB and RS232 interfaces, the data is available to be read into the register following the write to a query register, so this register is not required.

6 Serial Peripheral Interface

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 the SRDY pin is re-asserted high. It is important that the host either monitors the SRDY pin or waits for 500 µs between register writes.

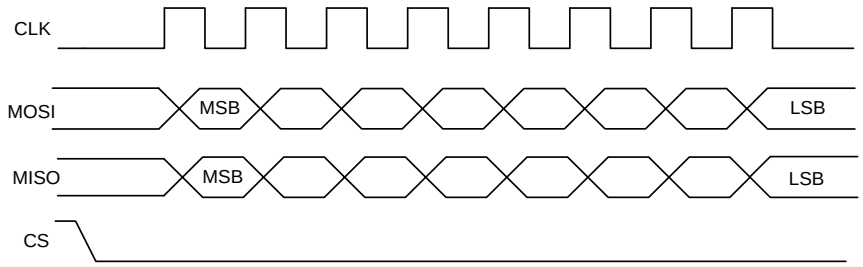


Figure 3. Clock Phase

Register writes are accomplished in a single write operation; their lengths are 8 bytes with the first byte being the register address, followed by the data associated with that register. All data transferred to and from the device is clocked on the falling edge of the clock as shown in *Figure 3*. 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 minimum clock recommended clock rate is 100 kHz and maximum 5.0 MHz (T_c = 0.2 µs). However, if the external SPI signals do not have sufficient integrity due to trace issues, then the rate should be lowered.

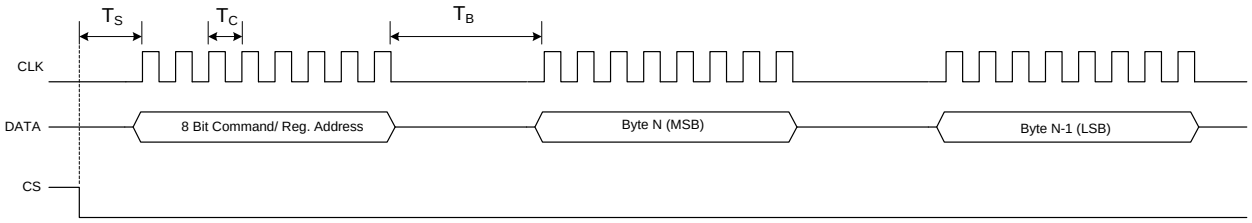


Figure 4. SPI Timing

As mentioned above, the SPI architecture limits the byte rate since 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 bidirectionally 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 2 µs. It is important that all 8 bytes are transferred, because once the first byte (MSB) containing the device

register is received, the device will wait for the rest of the 7 bytes; If an insufficient number of bytes are clocked in for the register, it could cause the device to hang. To clear the hung condition, the device will need an external hard reset. The time required to process a register is dependent on the command itself. Measured times for command completions are between 40 µs to 300 µs after reception.

6.1 Writing to Configure via SPI

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 MOSI line, data present on its SPI output buffer is simultaneously transferred back, MSB first, via the MISO line. The data return is invalid for most transfers except for those registers querying for data from the device. See the *Reading via SPI* section below for more information on retrieving data from the device. *Figure 5* shows the contents of setting the device frequency (register 0x10) to 15 GHz. The number written in mHz is 15E12 or in hexadecimal is 0xDA475ABF000 so the register is written with 0x10000DA475ABF000. The *Device Registers* section provides information on the number of data bytes and their contents for an associated register.

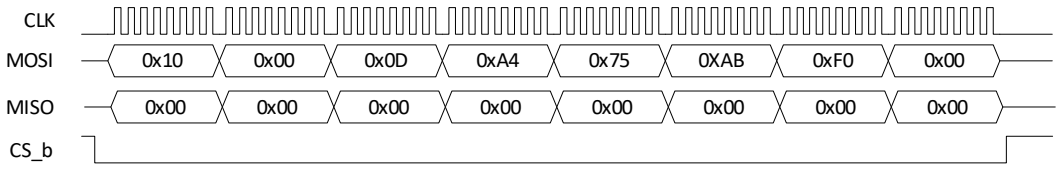


Figure 5. Single transfer buffer to change the frequency.

6.2 Reading via SPI

Data is simultaneously read back during an 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 MISO pin. To obtain valid requested data requires querying the SERIAL_OUT_BUFFER, which requires 8 bytes of clock cycles: 1 byte for the device register (0x25) and 7 empty bytes (MOSI) to clock out the returned data (MISO). An example of reading the device frequency back by first writing the GET_DEVICE_PARAM register with 0x01 (0x2000000000000001) and then followed by writing the SERIAL_OUT_BUFFER (0x2500000000000000), is shown in *Figure 6*.

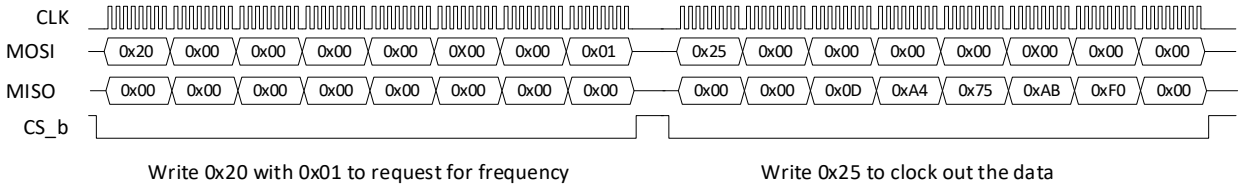


Figure 6. Reading queried data

7 Universal Asynchronous Receive-Transmit (UART) Interface

The UART is configured as a 2-wire serial whose logic level is 3.3V CMOS, which allows it to interface directly to a micro-controller or a minicomputer with UART/USART built in. An RS232 transceiver chip, like the MAX3232, is required to properly shift the CMOS levels for RS232 (COM) interfaces such as those of a PC. However, many of these transceivers have baud rates less than 1 MHz, and that may limit the UART maximum baud rate.

The UART baud rate is set to a default of 57600 when the UART_CNFG line is pulled low. However, when the line is pulled high it will select the rate that was set programmatically. See register 0x15 for the available rates. The device must be reset or restarted for the newly programmed baud rate to take effect. The factory set default rate when the UART_CNFG line is pulled high is 115200.

Table 2. UART Baud Rate

UART_CNFG	BAUD RATE
L	57600
H	Software settable, factory default is 115200

Table 3. UART Data Format

Property	Value
Baud Rate	*Table 2*
Data bits	8
Parity	None
Stop Bits	1
Flow Control	None

7.1 UART Data Transfer

Writing data to the device consists of 8 bytes; however, only writing query registers will return data in 8 bytes. The query registers are 0x20 to 0x24. The other registers (configuration registers) will return 1 byte, the acknowledge byte, which has a value of 0x01 to indicate the configuration was carried out successfully. This value must be read to clear the buffer for the next returned data. The data format for each register is provided in the *Device Registers* section. As an example, setting the device frequency to 15 GHz (15,000,000,000,000 mHz) involves the following steps:

1. First, send the value in 8 bytes of data with the first byte being the RF_FREQUENCY register byte, [0x10] [0x00] [0x0D] [0xA4] [0x75] [0xAB] [0xF0] [0x00].
2. Second, poll to read the acknowledge byte [0x01] with a timeout period of 1 second at most.

To query back information, such as the current rf frequency, first write the GET_DEVICE_PARAM register with the value 0x01, which requests for the frequency, followed by polling to read back 8 bytes of data containing the frequency value. The steps are shown here:

1. Write the 8 bytes [0x20] [0x00] [0x00] [0x00] [0x00] [0x00] [0x00] [0x01].
2. Poll to read 8 bytes back that contain the value, which may look like [0x00] [0x00] [0x0D] [0xA4] [0x75] [0xAB] [0xF0] [0x00]. The format of the returned data is detailed in the register description.

If an RS232 transceiver is connected to the device, it can be controlled via a host PC COM port. There are a couple of ways to perform the communication:

1. Use a HyperTerminal like Realterm, one that can transfer in hexadecimal instead of ASCII characters.
2. Use the provided software API and its Software Front Panel. Or write a custom application using the API.

8 USB Interface

The SC801A/SC802A has a full speed USB interface that works in parallel with the SPI/UART interface. Both interfaces are active at the same time if the USB interface is available for the device and the USB_EN pin is pulled high. Although the USB interface may not be used as a communications interface, it is recommended to be wired up as a port used for firmware updates.

8.1 USB Configuration

The device 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_BULK        0x81
#define SCI_ENDPOINT_OUT_BULK       0x02

// 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 16 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 supported by SignalCore.

8.2 Writing the Device Registers

Device registers are 8 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, the higher bits are transmitted first. To ensure that a register instruction has been fully executed by the device, read all 8 bytes back from the device. Data read back for configuration registers are invalid.

8.3 Reading the Device Registers Directly

Valid data is only available to be read back after writing one of the query registers such as 0x20, 0x21, …, and 0x24. As soon as one of these registers is written, data is available on the device to be read back. When reading the device, the MSB is returned as the first byte for a total of eight bytes. In many cases not all eight bytes carry valid data, however, all eight bytes must be read in since valid data begins at the LSB. The format of the returned data is detailed in the register description.

8.4 Software API

A software API is provided to control the device via USB in Windows or Linux.

9 Software API

The SC801A/SC802A application programming interface (API) software for both USB and RS232 (requires an RS232 transceiver) provided by SignalCore is available for Windows™. The USB portion of the API is also available for the Linux™ operating system. Source code for both platforms is available upon request by emailing support@signalcore.com. Programming platforms such as C/C++, C#, and LabView are supported. Python assistance is also available for those who want to port the API to that platform. The API functions are summarized in the table below and their function descriptions are provided in the API Description section.

Function	Description
nanohb_SearchDevices	Finds all the SC801A/SC802A Devices connected to the host
nanohb_SearchDevices_LV	Same as previous, but interfaces well with LabView and C#.
nanohb_OpenDevice	Opens a session for the device and returns the handle
nanohb_CloseDevice	Closes a session for the device and frees the handle
nanohb_RegWrite	Write directly to the device configuration registers
nanohb_RegRead	Read directly from the device query registers
nanohb_InitDevice	Initialize the device to power up state
nanohb_SetFrequency	Sets the device frequency for single fixed tone mode
nanohb_SetPhase	Sets the phase of the RF signal
nanohb_SetAsDefault	Stores the current configuration as default on reset or power-up
nanohb_SetRfMode	Sets the RF output to fixed or sweep/list mode
nanohb_SetSynthSelfCal	The device performs synthesizer self-calibration
nanohb_SetSynthMode	Sets synthesizer to use either harmonic or frac-N lock modes
Nanohb_SetSerialConfig	Sets the UART baud rate
nanohb_SweepConfig	Configures the sweep/list behavior
nanohb_SweepStartFreq	Sets the start frequency of a list for a sweep
nanohb_SweepStopFreq	Sets the stop frequency of a list for sweep
nanohb_SweepStepFreq	Sets the step frequency to generate a list for sweep
nanohb_SweepDwellTime	Sets the dwell time at each list frequency point
nanohb_SweepCycleCount	Sets the number of cycles to repeat the sweep
nanohb_ListBufferWrite	Writes a preconfigured list of frequencies to the list buffer in RAM
nanohb_ListBufferTransfer	Transfers the list buffer between RAM and EEPROM
nanohb_ListSoftTrigger	Software trigger
nanohb_GetDeviceStatus	Gets the device status, such as lock status
nanohb_GetDeviceInfo	Gets the device information
nanohb_GetRfParameters	Reads the current frequency, sweep/list frequency parameters
nanohb_GetTemperature	Gets the device operating temperature
nanohb_ListBufferRead	Reads the list points from list buffer in RAM

9.1 API Description

The API functions are contained in the **nanohb.dll** for Windows™ operating systems, or libnanohb.so.1.0 for Linux™ operating systems. For other operating systems or embedded systems, source code is available and can be requested by emailing support@signalcore.com. Information provided below represents the contents of the C/C++ header file, nanohb.h, but are expanded here, and listed for convenience. The integer returned for all functions holds the error identity, which is defined in the sci_errors.h file.

Function: nanohb_SearchDevices

Definition: int nanohb_SearchDevices(int interface, char **serial_NumberList, int numberDevices)

Input:	int interface	0 = USB, 1 = RS232
Output:	char **serialNumberList	2-D array pointer list
	int *numberDevices	The number of devices found

Description: nanohb_SearchDevices() searches for SignalCore SC801A/SC802A devices that are connected to the host computer and returns the number of devices found. It also populates the 2D char array with their serial numbers. The user can use this information to open a specific device(s) based on its unique serial number. See the nanohb_OpenDevice function on how to open a device.

Function: nanohb_SearchDevices_LV

Definition: int nanohb_SearchDevices_LV(int interface, char *serial_NumberList, int numberDevices)

Input:	int interface	0 = USB, 1 = RS232
Output:	char **serialNumberList	1D array list of concatenated serial numbers, 8 char ea.
	int *numberDevices	The number of devices found

Description: nanohb_SearchDevices_LV() searches for SignalCore SC801A/SC802A devices that are connected to the host computer and returns the number of devices found. It also populates the 1D character (8 bit/char) array with their serial numbers concatenated. Split the array up in 8 characters (8 bytes). The user can use this information to open a specific device(s) based on its unique serial number. See the nanohb_OpenDevice function on how to open a device.

Function: nanohb_OpenDevice

Definition: int nanohb_OpenDevice(int interface, char *devSerialNum, uint8_t baudRateIndex, PHANDLE deviceHandle)

Input:	int interface	0 = USB, 1 = RS232
	char *devSerialNum	The serial number string of 8 characters

	uint8_t baudRateIndex	RS232: 0 = 57600, 1 = 115200 Set to 0 for USB interface.
Output:	char **serialNumberList	1D array list of concatenated serial numbers, 8 char ea.
	int *numberDevices	The number of devices found
Description:	nanohb_OpenDevice() opens the device and returns a handle for access.	

Function:	**nanohb_CloseDevice**
Definition:	int nanohb_CloseDevice(HANDLE deviceHandle)
Input:	HANDLE deviceHandle — Handle to the device
Output:	
Description:	nanohb_CloseDevice() closes the device associated with the device handle.

Example Code: Exercise the functions that open and close the device.

```
// Declaring
  #define MAXDEVICES 50
  HANDLE 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

  status = nanohb_SearchDevices(0, deviceList, &numOfDevices); //searches for SCI for
  USB device type

  if (numOfDevices == 0)
  {
    printf("No available signal core devices found or cannot obtain serial numbers\n");
    for(i = 0; i<MAXDEVICES;i++) free(deviceList[i]);
    free(deviceList);
    return 1;
  }
  printf("\n There are %d SignalCore %s NANOHB 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++;
}

  // Open first device found, deviceList[0], with USB interface and baud rate = 0;
  Status = nanohb_OpenDevice(0, deviceList[0], 0, &devHandle);
  // Free memory
```

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

Function:	**nanohb_RegWrite**	
Definition:	int nanohb_RegWrite(HANDLE deviceHandle, uint8_t regByte, uint64_t instructWord)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t regByte	Register address
	uint64_t instructWord	Data associated with the register
Output:		
Description:	nanohb_RegWrite() writes data to the register address.	

Function:	**nanohb_RegRead**	
Definition:	int nanohb_RegRead(HANDLE deviceHandle, uint8_t regByte, uint64_t instructWord, uint64_t *receivedWord)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t regByte	Register address
	uint64_t instructWord	Instruct data associated with the register
Output:	uint64_t receivedWord	Received data associated with the register
Description:	nanohb_RegRead() writes data to the register address and then receives back.	

Function:	**nanohb_InitDevice**	
Definition:	int nanohb_InitDevice(HANDLE deviceHandle, uint8_t mode)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t mode	Current (0) or start up state (1)
Output:		
Description:	nanohb_InitDevice(); 0 makes the device reprogram its components to the current state; 1 resets the device to powerup state.	

Function:	**nanohb_SetFrequency**	
Definition:	int nanohb_SetFrequency(HANDLE deviceHandle, double frequency)	
Input:	HANDLE deviceHandle	Handle to the device

	double frequency	Frequency in Hz
Output:		
Description:	nanohb_SetFrequency() sets the single fixed tone RF frequency.	

Function:	**nanohb_SetPhase**	
Definition:	int nanohb_SetPhase(HANDLE deviceHandle, float phase)	
Input:	HANDLE deviceHandle	Handle to the device
	float phase	Phase in degrees
Output:		
Description:	nanohb_SetPhase() adjusts the phase of the signal.	

Function:	**nanohb_SetAsDefault**	
Definition:	int nanohb_SetAsDefault(HANDLE deviceHandle)	
Input:	HANDLE deviceHandle	Handle to the device
Output:		
Description:	nanohb_SetAsDefault() stores the current configuration into EEPROM memory and is used as the default state upon reset or power up.	

Function:	**nanohb_SetSynthSelfCal**	
Definition:	int nanohb_SetSynthSelfCal(HANDLE deviceHandle, uint8_t source)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t source	0 = coarse synth, 1 = sum synth
Output:		
Description:	nanohb_SetSynthSelfCal() starts the synthesizer calibration for the VCO in either the coarse synth loop (0), or the sum synth loop (1).	

Function:	**nanohb_SetSynthMode**	
Definition:	int nanohb_SetSynthMode(HANDLE deviceHandle, synthMode_t *synthMode)	
Input:	HANDLE deviceHandle	Handle to the device
	synthMode_t *synthMode	synth mode struct (see nanohb_defs.h file for info)
Output:		
Description:	nanohb_SetSynthMode() sets up the rf mode, loop gain, lock mode, freq resolution, and automatic spur suppression.	

Function:	**nanohb_SetSerialConfig**
Definition:	int nanohb_SetSerialConfig(HANDLE deviceHandle, uint8_t uartBaudrateIndex)

Input:	HANDLE deviceHandle	Handle to the device
	Uint8_t **uartBaudrateIndex**	**Indexes have corresponding rate, see register 0x15**
Output:		
Description:	nanohb_SetSerialConfig() sets up the UART baud rate, when the UART_CONF hardware pin is pulled high.	

Function:	**nanohb_SweepConfig**
Definition:	int nanohb_ListModeConfig(HANDLE deviceHandle, sweepParam_t *sweepParam)

Input:	HANDLE deviceHandle	Handle to the device
	sweepParam_t *sweepParam	sweepParam_t struct type
Output:		
Description:	nanohb_SweepConfig() configures the list mode behavior. See the nanohb_defs.h header file for more information on the sweepParam_t structure.	

Function:	**nanohb_SweepStartFrequency**
Definition:	int nanohb_SweepStartFrequency(HANDLE deviceHandle, double frequency)

Input:	HANDLE deviceHandle	Handle to the device
	double frequency	Frequency in Hz
Output:		
Description:	nanohb_SweepStartFrequency() sets the start frequency of the list to sweep.	

Function:	**nanohb_SweepStopFrequency**
Definition:	int nanohb_SweepStopFrequency(HANDLE deviceHandle, double frequency)

Input:	HANDLE deviceHandle	Handle to the device
	double frequency	Frequency in Hz
Output:		
Description:	nanohb_SweepStopFrequency() sets the stop frequency of the list to sweep.	

Function:	**nanohb_SweepStepFrequency**
Definition:	int nanohb_SweepStepFrequency(HANDLE deviceHandle, double frequency)

Input:	HANDLE deviceHandle	Handle to the device
	double frequency	Frequency in Hz
Output:		
Description:	nanohb_SweepStepFrequency() sets the step frequency of the list to sweep.	

Function:	**nanohb_SweepDwellTime**	
Definition:	int nanohb_SweepDwellTime(HANDLE deviceHandle, float dwellTime)	
Input:	HANDLE deviceHandle	Handle to the device
	float dwellTime	In milli-seconds or 0.5ms increment
Output:		
Description:	nanohb_SweepDwellTime() sets the sweep/list dwell time at each frequency point.	

Function:	**nanohb_SweepCycleCount**	
Definition:	int nanohb_SweepCycleCount(HANDLE deviceHandle, uint32_t cycleCount)	
Input:	HANDLE deviceHandle	Handle to the device
	uint32_t cycleCount	Number of sweep cycles
Output:		
Description:	nanohb_SweepCycleCount() sets the number of sweep cycles to perform before stopping. To repeat the sweep continuously, set the value to 0.	

Function:	**nanohb_ListBufferWrite**	
Definition:	int nanohb_ListBufferWrite(HANDLE deviceHandle, double* frequencies, float*dwellTimes, uint16_t len)	
Input:	HANDLE deviceHandle	Handle to the device
	double* frequencies	Array of frequencies in Hz
	float* dwellTimes	Array of corresponding dwell times in 500 µs
	uint16_t len	Length of the array to write
Output:		
Description:	nanohb_ListBufferWrite() writes preconfigured arrays of frequency and dwell time data to the list buffer which is used to perform the sweep.	

Function:	**nanohb_ListBufferTransfer**	
Definition:	int nanohb_ListBufferTransfer(HANDLE deviceHandle, uint8_t transferMode)	
Input:	HANDLE deviceHandle	Handle to the device
	unsignedchar transferMode	0 or 1
Output:		
Description:	nanohb_ListBufferTransfer() transfers the frequency list buffer from RAM to EEPROM (0) or vice versa (1). Transfer to EEPROM will keep the data that can be retrieved on power-up or reset.	

Function:	**nanohb_ListSoftTrigger**

Definition:	int nanohb_ListSoftTrigger(HANDLE deviceHandle)
Input:	HANDLE deviceHandle — Handle to the device
Output:	
Description:	nanohb_ListSoftTrigger() triggers the device when it is configured for list mode and soft trigger is selected as the trigger source.

Function:	**nanohb_GetDeviceStatus**
Definition:	int nanohb_GetDeviceStatus(HANDLE deviceHandle, deviceStatus_t *deviceStatus)
Input:	HANDLE deviceHandle — Handle to the device
Output:	deviceStatus_t *deviceStatus Status of the device
Description:	nanohb_GetDeviceStatus() gets the current device status such as the PLL lock status, RF mode, sweep mode, sweep status, as well as the list mode configuration.

Function:	**nanohb_GetDeviceInfo**
Definition:	int nanohb_GetDeviceInfo(HANDLE deviceHandle, deviceInfo_t *deviceInfo)
Input:	HANDLE deviceHandle — Handle to the device
Output:	deviceInfo_t *deviceInfo — Device info
Description:	nanohb_GetDeviceInfo() gets the device information such as firmware version, hardware version, model option, model number, and serial number.

Function:	**nanohb_GetRFParameters**
Definition:	int nanohb_GetRFParameters(HANDLE deviceHandle, rfParams_t *rfParams)
Input:	HANDLE deviceHandle — Handle to the device
Output:	rfParams_t *rfParams — RF parameters struct
Description:	nanohb_GetRfParameters() gets the device RF parameters such as frequency, phase, sweep frequencies, and sweep configuration parameters.

Function:	**nanohb_GetTemperature**
Definition:	int nanohb_GetTemperature(HANDLE deviceHandle, float *temperature)
Input:	HANDLE deviceHandle — Handle to the device
Output:	float *temperature — Device temperature
Description:	nanohb_GetTemperature() gets the device temperature.

Function:	**nanohb_GetListBuffer**
Definition:	int nanohb_GetListBuffer(HANDLE deviceHandle, double* frequencies,

		float *dwellTimes, uint16_t len)
Input:	HANDLE deviceHandle	Handle to the device
Output:	double *frequencies	Array of frequencies of length len
	float *dwellTimes	Array of frequencies of length len
	uint16_t len	Length of array to get
Description:	nanohb_GetListBuffer() gets frequencies and dwell times of the length (len) from the device's list buffer.	

9.2 Code Examples

Code examples in C/C++, and C# are provided to illustrate programming the device with the API. Precompiled 32-bit and 64-bit executables are provided with the software package.

9.3 LabVIEW Support

A LabVIEW library is provided for development on that platform. The function VIs are wrappers that call on the nanohb.dll C/C++ API. An executable software front panel (GUI) developed in LabVIEW, along with its source code, is also included in the software package. VI function description can be found by pressing keys [Ctrl] and [H] on the keyboard.

10 Revision Table

Revision	Revision Date	Description
1.0	1/19/2022	Initial Release
1.1	3/27/2022	Corrected pinouts, updated specifications, updated function calls