Datasheet

SC803A

100/200 MHz Reference Source

Rev 1.0

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 SC803A is part of the nanoSynth® family from our nanoCircuits® line of products. It is a SMT device designed for PCB applications that required a reference source or frequency synthesizer with low phase noise and OCXO grade stability. Its internal OCXO maintains frequency stability of better than 20 ppb over the operational temperature range. The typical measured phase noise at 10 kHz offset from a 200 MHz carrier is less than -160 dBc/Hz. Communicating to the device is flexible as it has 3 built-in communications interfaces: USB, SPI, and UART (RS232).

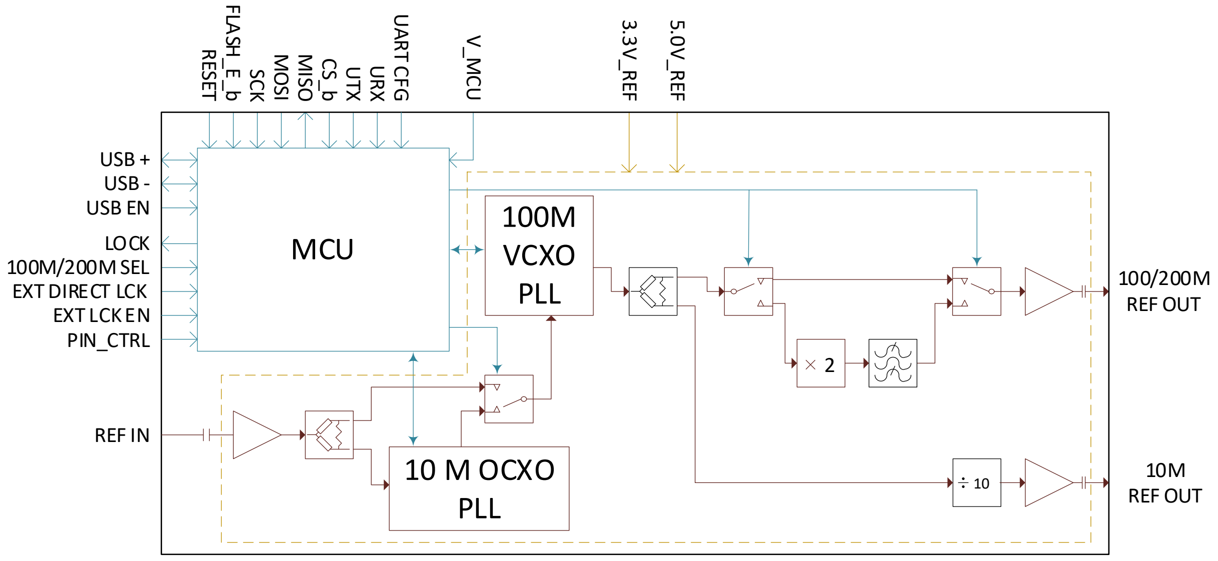


Figure 1. SC803A Functional Block Diagram

Product Features

- Selectable 100 MHz or 200 MHz out
- Locks to external 5/10/20/50/100 MHz
- OCXO disciplined
- Phase Noise < -162 dBc/Hz at 10 kHz offset at 200 MHz
- Rugged and miniature 2.00” x 1.25” x 0.35” SMT package
- USB, UART and SPI

Applications

- Test and measurement equipment
- Wireless communication equipment
- PLO replacement
- Reference signal clean-up
- Quantum computing
- Network equipment

3 Specifications

3.1 Spectral Specifications

Reference Output			10/100/200 MHz
Stability			
	Ambient (0 ^{O}C to +70 ^{O}C)		± 20 ppb
	Aging		
		Daily	± 3 ppb
		Yearly	± 0.6 ppm
		10 years	± 3 ppm

Phase Noise (dBc/Hz) typ. Add 3 dB for max.

Offset	10 MHz	100 MHz	200 MHz
10 Hz	-100	-80	-74
100 Hz	-136	-120	-114
1 kHz	-158	-147	-141
10 kHz	-160	-166	-160
100 kHz	-160	-170	-164
1 MHz	-160	-172	-166

Harmonics & Spurs		
	2nd Harmonics	< -15 dBc typical
	Sub Harmonics (200 MHz)	< -70 dBc typical
	Spurs	< -70 dBc typical
Reference Input		5/10/20/50/100 MHz

3.2 Amplitude Specifications

Reference Output			
	10 MHz		+5 dBm typical
	100 MHz		+5 dBm typical
	200 MHz		+5 dBm typical
Reference Input			
	Amplitude		
		Min	0 dBm
		Typical	+3 dBm
		Max	+7 dBm
	Frequency accuracy		< ± 2 ppm

3.3 Electrical Specifications

Voltage and Current

	Parameter	Minimum	Typical	Maximum	Unit
	V_REF3.3	3.15	3.3	3.5	V
	V_MCU	3.2	3.3	3.4	V
	V_REF5.0	4.95	5.0	5.3	V
	I_REF3.3	-	-	320	mA
	I_MCU	-	-	60	mA
	I_REF5.0	-	-	180	mA
	Power dissipation	1.8	1.9	2.2	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

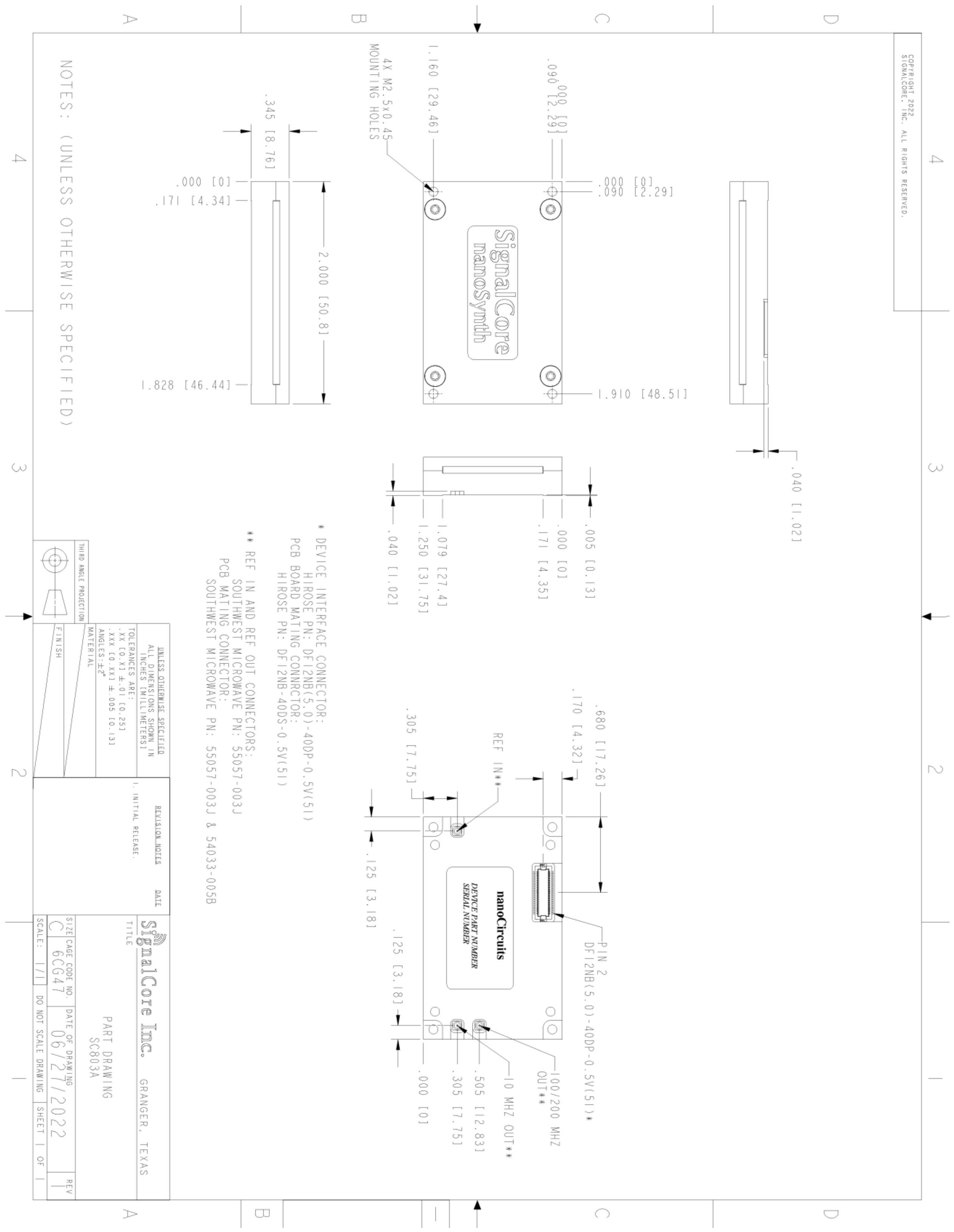
Continuous Power Dissipation	2.5 W
Storage Temperature	-20 to 90 ^{0}C
Operating Temperature	0 to +75 ^{0}C

3.4 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.
2,4,20,22	NC	Not connected
8,10,12,14,16	5V_REF	Supply for reference circuit
24,26,28,30,32	3.3V_REF	Supply for reference circuit
36,38	V_MCU	Supply for the microprocessor and digital interface
1	USB_N	USB negative line
3	USB_P	USB positive line
5	USB_EN	Pull high to enable USB
9	UTX	UART Transmit
11	URX	UART Receive
13	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 re-flashing with firmware.
17	$\overline{\text{RESET}}$	Hardware reset
19	200M_SEL	Pull high to select 200 MHz output.[1]
21	LCK_STATUS	Phase-lock Status
23	LOCK_ENABLE	Enable locking to external source[1]
25	DIRECT_LOCK	Locks VCXO directly to external reference, bypass internal OCXO. Only if reference is 10 MHz.[1]
29	REF_IN_SEL	Selects input reference for 10 MHz or 100MHz.[1]
31	SERIAL_CTRL	Selects the UART baud rate.
33	$\overline{\text{CS}}$	SPI device/chip select
35	MOSI	SPI receive
37	MISO	SPI transmit
39	SCK	SPI clock

[1] Pins determine the powerup state. Pins can be overridden via software and powerup status is retrieved from stored default values.

3.5 Mechanical Data



3.6 Product Evaluation

A full development board is available from SignalCore to evaluate the SC803A together with either the SC801A or SC802A. Following this link for more information:
https://www.signalcore.com/nano_hb2.html#eval

3.7 Ordering Information

SC803A nanoSynth SMT Reference Module Kit	7100050-01
SC801A nanoSynth-HB Evaluation Development Kit	7100152-01
SC802A nanoSynth-HB Evaluation Development Kit	7100153-01

3.7.1 SC803A Module Kit Contents

• SC803A Module	1
• Software in USB drive	1

4 Theory and Operation

The SC803A is a very small and high performing surface mount reference synthesizer with easy to program register-level control. Being small and fully integrated, this source is the ideal solution for board-level designs that require a frequency stable reference signal with very low phase noise. *Figure 2* shows the block diagram of the device, and the following sub-sections provide details of its operation.

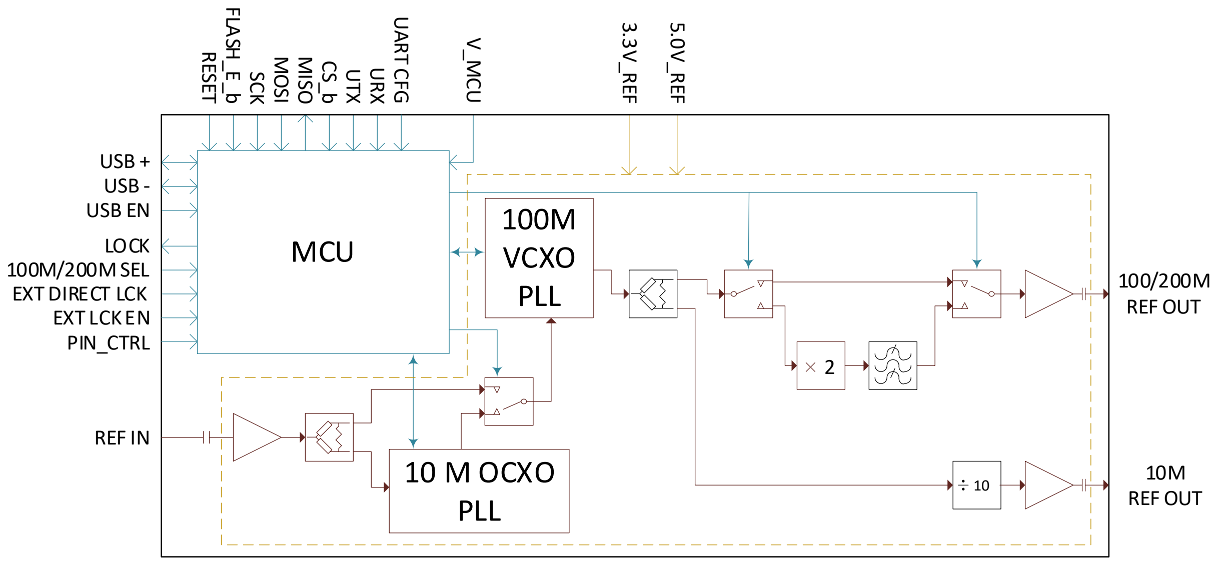


Figure 2. Block Diagram of the SC803A

4.1 Reference Signal Generation

The SC803A synthesizes 10MHz, 100MHz and 200 MHz output signals from its base reference, a 10 MHz OCXO. The primary oscillator is an extremely low phase noise 100 MHz VCXO whose signal is frequency doubled and filtered to produce a 200 MHz MHz signal. A switch selects either the 100 MHz or the 200 MHz signal as the output. The VCXO is also frequency divided to obtain a 10 MHz signal as a secondary reference output.

To achieve better stability, the VCXO is continuously phase-locked to the internal OCXO. The OCXO temperature stability is better than the VCXO by a magnitude order of 1000. In applications where a system reference clock is present externally, the internal OCXO can lock to it if external locking is enabled. While the VCXO is generally locked to the internal OCXO, it can also directly lock to the external reference clock provided that the clock frequency is 10 MHz. The choice to phase-lock the VCXO directly to an external reference would depend on the quality of the phase noise of the clock. If the phase noise of the external system clock is higher than the OCXO, especially in the regions less than 100 Hz offset, it is not recommended to directly lock.

Upon powerup of the device, the input reference source frequency is determined by either pin 29, or if this pin is overridden by software, it can be set to the stored default value. If pin 29 is set low the device assumes a clock of 10 MHz, and a high for 100 MHz. If this pin is overridden, then the device can be programmed to the frequencies list in the following table.

Index	Frequency
0	10 MHz
1	100 MHz
2	5 MHz
3	20 MHz
4	50 MHz

Table 1 Reference Input indices and corresponding frequencies

4.2 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 or header pin even though it may not be used.

4.2.1 SPI Interface

PINS 33, 35, 37, and 39 are configured as an SPI interface that corresponds to $\overline{\mathrm{CS}}$, MOSI, MISO, and SCK respectively. Detailed SPI read and write operations are discussed in detail in section 6.

4.2.2 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. Detailed UART read and write operations are discussed in detail in the *Universal Asynchronous Receive-Transmit (UART) Interface* section.

4.2.3 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 SC803A 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 0x10 REFOUT_FREQUENCY

Bits	Type	Name	Width	Description
[0]	WO	Frequency select	1	0 = 100 MHz, 1 = 200 MHz
[55:1]	WO	Frequency word	55	Set to zero

5.3 Register 0x11 REFIN_FREQUENCY (4 Bytes)

Bits	Type	Name	Width	Description
[1:0]	WO	Frequency select	3	0 = 10 MHz, 1 = 100 MHz, 2 = 5 MHz, 3 = 20 MHz, 4 = 50 MHz
[56:3]	WO	SIGN BIT	53	Set to zero

5.4 Register 0x12 CONFIG

Bits	Type	Name	Width	Description
[0]	WO	External Lock	1	Enable/Disable external lock
[1]	WO	Direct Lock	1	Enable/Disable direct locking to external reference. Direct locking is only possible if the input reference frequency is set to 10 MHz.

Bits	Type	Name	Width	Description
[2]	WO	Override HW pins	1	1 overrides hardware pin setting and uses the current stored default setting on powerup. Pins that are overridden are: Ext Lock Enable Direct Lock Ref In Select
[56:3]	WO	Not used	53	Set to zeros

5.5 Register 0x13 ADJUST_OCXO

Bits	Type	Name	Width	Description
[10:0]	WO	11-bit word	11	Adjust accuracy of the OCXO by this increment on each call to the register
[11]	WO	Sign bit	1	1 = negative word value
[55:13]	WO	Unused	43	Set all bits to 0

5.6 Register 0x14 SERIAL_CONFIG

Bits	Type	Name	Width	Description
[3:0]	WO	Baud rate index	4	Values set here are only valid if PIN 31 (SERIAL_CNTRL) is pull high. 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

5.7 Register 0x15 SET_DEFAULT

Bits	Type	Name	Width	Description
[55:0]	WO	Value	56	Set all to 0. Calling this register will save the current state of the device as default. This register needs to be called if either or all the parameters for registers 0x10 to 0x13 are to be set as default.

5.8 Register 0x20 GET_DEVICE_PARAM

Bits	Type	Name	Width	Description
[1:0]	WO	Get Device Parameters	2	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 = ref input and output frequency indices 2 = OCXO adjust value
[55:2]	WO	Reserved	54	Zero
[63:0]	RO	Return Data	64	Valid data for the requested parameter: 0 Device status [0] lock enable [1] direct lock [2] ext. ref detected [3] ocxo locked [4] vcxo locked [5] hw pin override [6] uart pin state [11:8] uart baudrate [13:12] reserved 1 Ref frequency indices [3:0] ref in index [7:4] ref out index 2 Last OCXO adjust value

Bits	Type	Name	Width	Description
				[11:0] value [12] sign

5.9 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. [39:32] Model option. Custom options. [47:40] Model number. 0 = SC803A [55:48] Interface code. Default 7. USB, RS232, SPI available.
[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	Date of manufacture.

Bits	Type	Name	Width	Description
				[7:0] day [15:8] month [23:16] year (add 2000)

5.10 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

5.11 Register 0x23 OCXO DAC VALUE

Bits	Type	Name	Width	Description
[55:0]	WO		64	Set all bits to 0
[63:0]	RO	Data in	63	[15:0] = OCXO DAC value. Value controls the accuracy of the OCXO. [63:16] invalid

5.12 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 0x23 registers.

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.

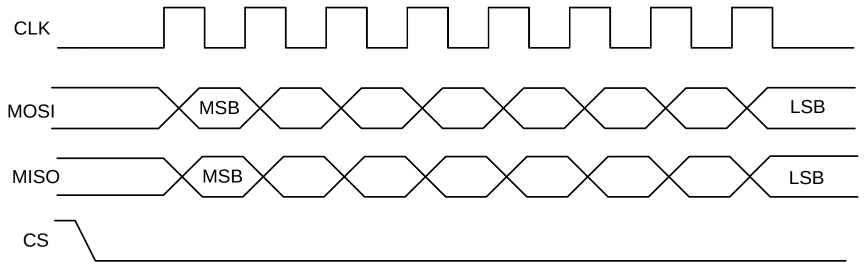


Figure 3. Clock Phase

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.

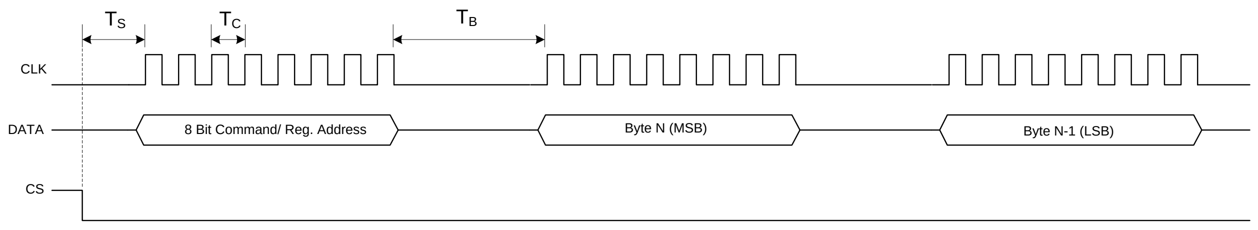


Figure 4. SPI Timing

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 reference output frequency (register 0x10) to 200 MHz. The value written is 0x01 so the register is written with 0x1000000000000001. The *Device Registers* section provides information on the number of data bytes and their contents for an associated register.

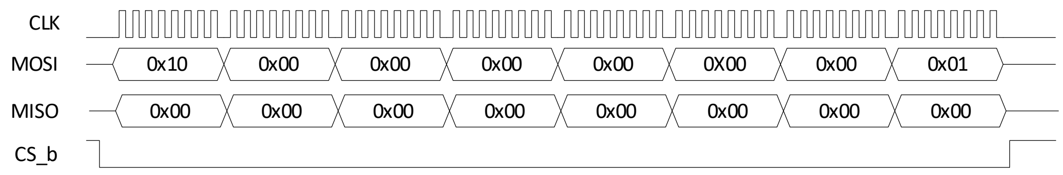


Figure 5. Single transfer buffer to change the frequency.

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 Register 0x25 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 Register 0x25 SERIAL_OUT_BUFFER (0x2500000000000000) register, is shown in *Figure 6*.

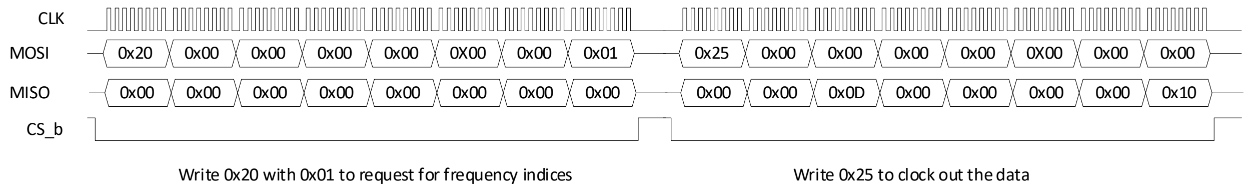


Figure 6. Reading queried data

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_CFG pin is pulled low. However, when the line is pulled high it will select the rate that was set programmatically. See Register 0x14 SERIAL_CONFIG 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_CFG pin is pulled high is 115200.

Table 2. UART Baud Rate

UART_CFG PIN	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 0x23. 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 reference output frequency to 200 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] [0x00] [0x00] [0x00] [0x00] [0x01].

2. Second, poll to read the acknowledge byte [0x01] with a timeout period of 1 second at most. Reading this byte clears out the receive buffer to avoid errors when querying for data.

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] [0x00] [0x00] [0x00] [0x00] [0x10]. 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 few 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 Front Panel.
3. Write a custom application using the software API.

8 USB Interface

The SC803A 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
- 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 0x23. 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 USB Software API

A software API is provided to control the device via USB in Windows and Linux, see next section.

9 Software API

The SC803A application programming interface (API) software for both USB and RS232 (requires a 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
nanowb_SearchDevices	Finds all the SC803A Devices connected to the host
nanowb_SearchDevices_LV	Same as previous, but interfaces well with LabView and C#.
nanowb_OpenDevice	Opens a session for the device and returns the handle
nanowb_CloseDevice	Closes a session for the device and frees the handle
nanowb_RegWrite	Write directly to the device configuration registers
nanowb_RegRead	Read directly from the device query registers
nanowb_InitDevice	Initialize the device to power up state
nanowb_SetRefOutFreq	Sets the reference output frequency
nanowb_SetRefInFreq	Sets the reference input frequency
nanowb_SetRefConfig	Set up the reference behavior
nanowb_SetSerialConfig	Sets the UART baud rate
nanowb_SetRefDacAdjust	Configures the sweep/list behavior
nanowb_SetAsDefault	Stores the current configuration as default on reset or power-up
nanowb_GetDeviceStatus	Gets the device status, such as lock status
nanowb_GetDeviceInfo	Gets the device information
nanowb_GetRfParams	Reads the current frequency, sweep/list frequency parameters
nanowb_GetTemperature	Gets the device operating temperature
nanowb_GetRefDacValue	Reads the list points from list buffer in RAM

9.1 API Description

The API functions are contained in the **nanowb.dll** for Windows™ operating systems, or libnanowb.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, nanowb.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:	**nanowb_SearchDevices**	
Definition:	int nanowb_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:	nanowb_SearchDevices() searches for SignalCore SC803A 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 nanowb_OpenDevice function on how to open a device.	

Function:	**nanowb_SearchDevices_LV**	
Definition:	int nanowb_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:	nanowb_SearchDevices_LV() searches for SignalCore SC803A 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 nanowb_OpenDevice function on how to open a device.	

Function:	**nanowb_OpenDevice**	
Definition:	int nanowb_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:	PHANDLE deviceHandle	Device handle
Description:	nanowb_OpenDevice() opens the device and returns a handle for access.	

Function:	**nanowb_CloseDevice**
Definition:	int nanowb_CloseDevice(HANDLE deviceHandle)

Input:	HANDLE deviceHandle	Handle to the device
Output:		
Description:	nanowb_CloseDevice() closes the device associated with the device handle.	

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

```
// Includes
  #include "nanowb.h"
// Declaring
  #define MAXDEVICES 50
  HANDLE devHandle; //device handle, HANDLE is defined as void*
  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 = nanowb_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 NANOWB 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 = nanowb_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 = nanowb_CloseDevice(devHandle); // Close the device
```

Function:	**nanowb_RegWrite**	
Definition:	int nanowb_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:	nanowb_RegWrite() writes data to the register address.	

Function:	**nanowb_RegRead**	
Definition:	int nanowb_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:	nanowb_RegRead() writes data to the register address and then receives back.	

Function:	**nanowb_InitDevice**	
Definition:	int nanowb_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:	nanowb_InitDevice(); 0 makes the device reprogram its components to the current state; 1 resets the device to powerup state.	

Function:	**nanowb_SetRefOutFreq**	
Definition:	int nanowb_SetFrequency(HANDLE deviceHandle, uint32_t refOutFreqIndex)	
Input:	HANDLE deviceHandle	Handle to the device
	uint32_t refOutFreqIndex	0 = 100 MHz, 1 = 200 MHz
Output:		
Description:	nanowb_SetRefOutFreq() the output reference frequency to either 100 MHz or 200 MHz.	

Function:	**nanowb_SetRefInFreq**

Definition:	int nanowb_SetRefInFreq(HANDLE deviceHandle, uint32_t refInFreqIndex)	
Input:	HANDLE deviceHandle	Handle to the device
	uint32_t refInFreqIndex	0 = 10 MHz, 1 = 100 MHz, 2 = 5 MHz, 3 = 20 MHz, 4 = 100MHz
Output:		
Description:	nanowb_SetRefInFreq() set the input reference frequency if hardware pin override is enabled in software.	

Function:	**nanowb_SetSerialConfig**	
Definition:	int nanowb_SetSerialConfig(HANDLE deviceHandle, uint8_t reserved, uint8_t uartBaudrateIndex)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t uartBaudrateIndex	Indexes have corresponding rate, see register 0x14
Output:		
Description:	nanowb_SetSerialConfig() sets up the UART baud rate, when the SERIAL_CONF hardware pin 31 is pulled high.	

Function:	**nanowb_SetRefDacAdjust**	
Definition:	int nanowb_SetRefDacAdjust(HANDLE deviceHandle, int16_t adcAdjustValue)	
Input:	HANDLE deviceHandle	Handle to the device
	uint8_t adcAdjustValue	Signed value to be added to the DAC value
Output:		
Description:	nanowb_SetRefDacAdjust() will adjust the current DAC value by the amount written to the device via this function.	

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

Function:	**nanowb_GetDeviceStatus**	
Definition:	int nanowb_GetDeviceStatus(HANDLE deviceHandle, deviceStatus_t *deviceStatus)	
Input:	HANDLE deviceHandle	Handle to the device
Output:	deviceStatus_t *deviceStatus	Status of the device

Description:	nanowb_GetDeviceStatus() gets the current device status such as the PLL lock status, output frequency, input reference frequency, lock configuration, etc.	

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

Function:	**nanowb_GetRFParameters**	
Definition:	int nanowb_GetFreqParams(HANDLE deviceHandle, rfParams_t *freqParams)	
Input:	HANDLE deviceHandle	Handle to the device
Output:	rfParams_t *freqParams	Reference in and out indices
Description:	nanowb_GetFreqParams() gets the device frequency parameters.	

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

Function:	**nanowb_GetRefDacValue**	
Definition:	int nanowb_GetRefDacValue(HANDLE deviceHandle, uint16_t dacValue)	
Input:	HANDLE deviceHandle	Handle to the device
Output:	uint16_t dacValue	Current OCXO DAC control value
Description:	nanowb_GetRefDacValue () obtains the current control value of the DAC.	

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 nanowb.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	7/24/2022	Initial Release

SignalCore™
PRESERVING SIGNAL INTEGRITY