# SERIAL DEVICE INTERFACE TECHNIQUES

## 1. Introduction

In some cases, in can be convenient to power external serial sensors from a Senquip device. This can be done by connecting the serial device to the IO pins on a Senquip QUAD, or the Source pins on a Senquip ORB. Sampling of serial data can become complicated where the external sensor has a significant boot time, or where it sends data immediately after being powered. Where the sensor has a significant boot time, the Senquip device may have try to sample serial data before the external device is ready. In the case of a sensor that sends data instantly when powered, the serial data may have been sent before the Senquip device is ready to receive it.

This application note will look at five externally connected serial devices and will discuss how to control the power and serial reads in each case.

1. Senquip device always on, powering external Modbus device.
2. Senquip device requests serial data from external serial device.
3. Senquip device waking periodically and powering external Modbus device with boot time.
4. Senquip device waking periodically and powering external Modbus device that requires Modbus write.
5. Senquip device waking periodically and taking multiple samples from a serial sensor.

The first two examples are simple, can be implemented using simple settings only and are shown only as an introduction. The third example uses delayed serial reads but can also be implemented with settings only. The last 2 examples are more complex and will require a script. Where the Senquip device is waking periodically, it will be assumed that the application is low power with the Senquip device running from AA batteries or Solar.

This application note assumes that the Senquip devices are running the following or newer firmware versions:

- Senquip ORB: SFW002-3.0.0
- Senquip QUAD: SFW003-4.0.0

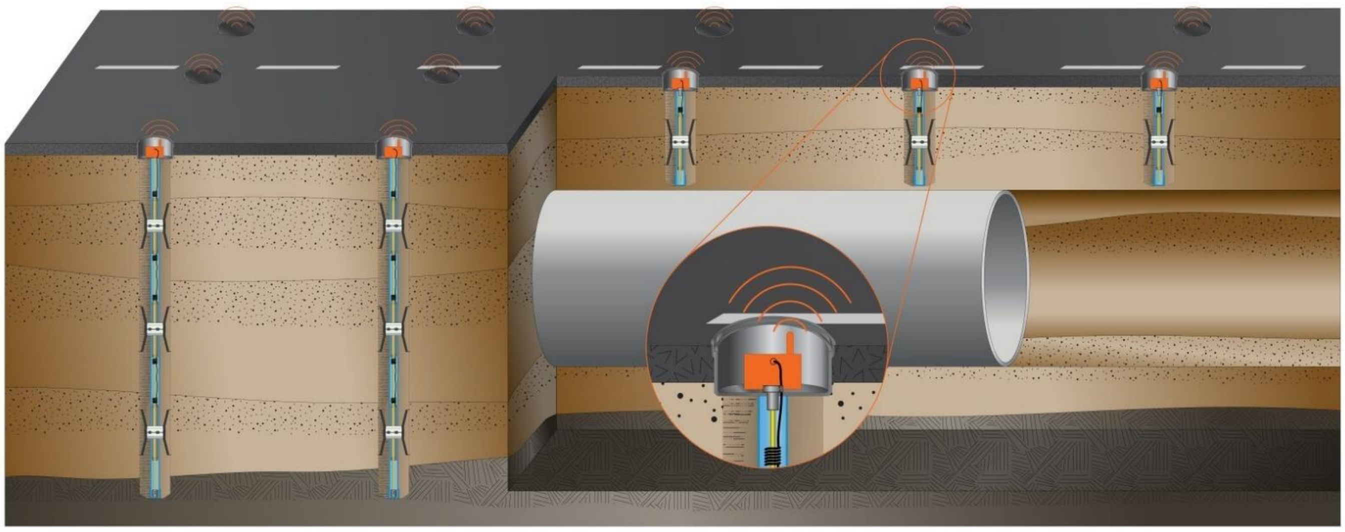


*Figure 1 - Extensometers from Osprey Measurement Systems are an Example of Serial Sensors with a Boot Time*

## 2. Connecting Serial Sensors to the Senquip Device

In this application note, we will assume that the external sensors are RS485, and that they are powered by Current Loop 1 on the Senquip ORB and IO1 on the Senquip QUAD.

The following pins will be used on the Senquip ORB and QUAD devices.

| Connection | Senquip ORB | Senquip QUAD |
|---|---|---|
| RS485 B | Pin 6, B | Pin 9, B |
| RS485 A | Pin 7, A | Pin 10, A |
| Power | Pin 3, SRC1 | Pin 3, IO1 |
| GND | Pin 4, GND | Pin 8, GND |

If the Senquip device and sensor are at the end of the line on the RS485 bus, then a 120ohm termination resistor must be placed at each end of the line. The 120 ohm resistor on the Senquip device can be enabled as a setting.

### 2.1. Senquip ORB

With a Senquip ORB, the external serial device is powered by Current Loop 1. The current loops are powered by the internal LiPo backup battery so that they can provide power when the external power supply is removed, for instance at night in a solar application. An internal DCDC converter boosts the 3.7V LiPo voltage to 12V to power the external sensor. The current loop voltage on the Senquip ORB is fixed at 12V.

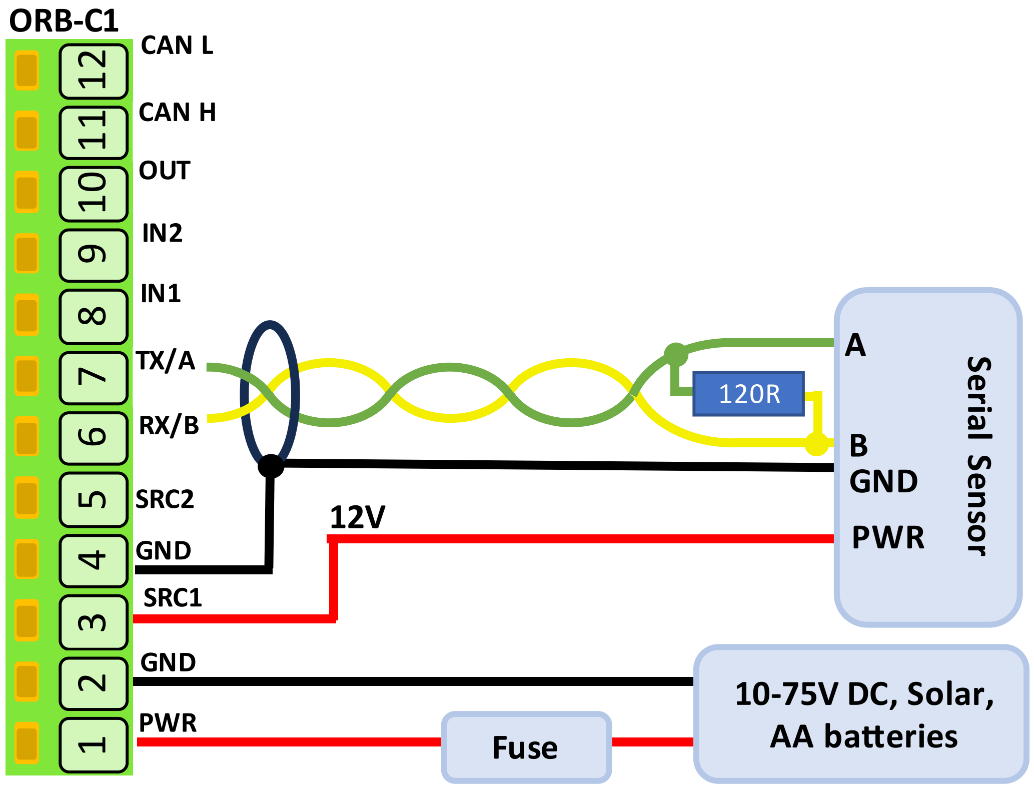
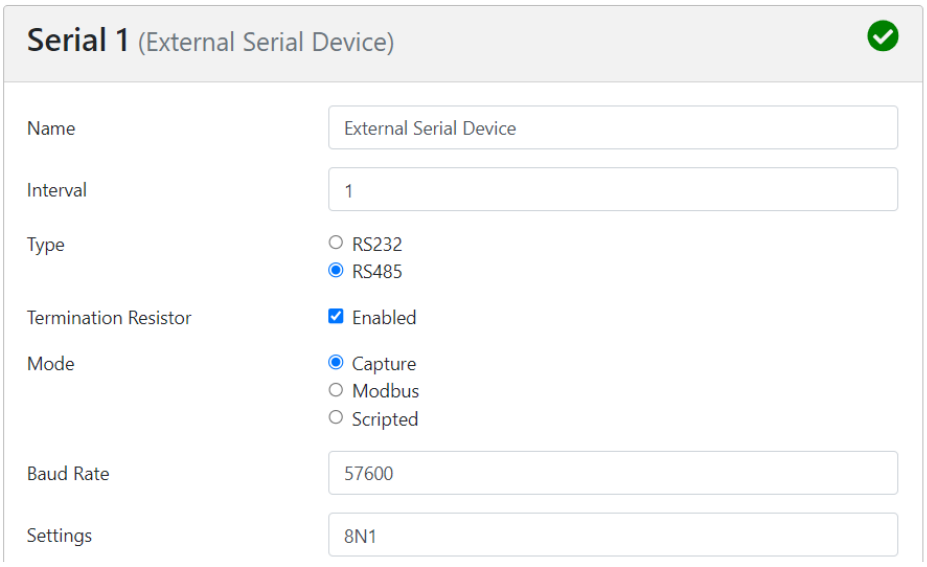


*Figure 2 - Senquip ORB Powering External Serial Sensor*

The basic settings for the serial device will remain the same for all examples:

| Settings | Value | Comment |
|---|---|---|
| Interval | 1 | The serial port will be sampled on every base interval |
| Serial Type | RS485 | |
| Termination Resistor | Enabled | Must be enabled at the ends of the twisted pair cable |
| Baud Rate | 57600 | |
| Settings | 8N1 | 8 data bits, no parity, 1 stop bit |

The *Mode* will change depending on the application.


Serial 1 (External Serial Device)

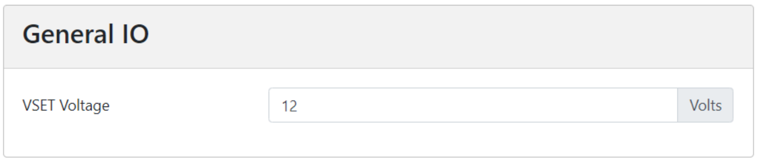
| | |
|---|---|
| Name | External Serial Device |
| Interval | 1 |
| Type | RS232 [✓ RS485] |
| Termination Resistor | [✓ Enabled] |
| Mode | [✓ Capture] Modbus Scripted |
| Baud Rate | 57600 |
| Settings | 8N1 |

*Figure 3 - Senquip ORB Serial Port Settings*

## 2.2. Senquip QUAD

With a Senquip QUAD, external devices can be powered from any of the 5 IO. The IO can be configured to supply the externally connected devices with the supply voltage (Vin), or and internally generated voltage that is boosted from the internal LiPo backup battery (Vset). If Vset is chosen, then the external device can be powered when supply to the Senquip QUAD is lost, for instance at night in a solar application. Vset can be configured between 5V and 25V.

In our application, we will use Vset and will configure it as 12V. The rest of the IO settings will depend on the application.


General IO

VSET Voltage: 12 Volts

*Figure 4 - Vset Configured to be 12V*

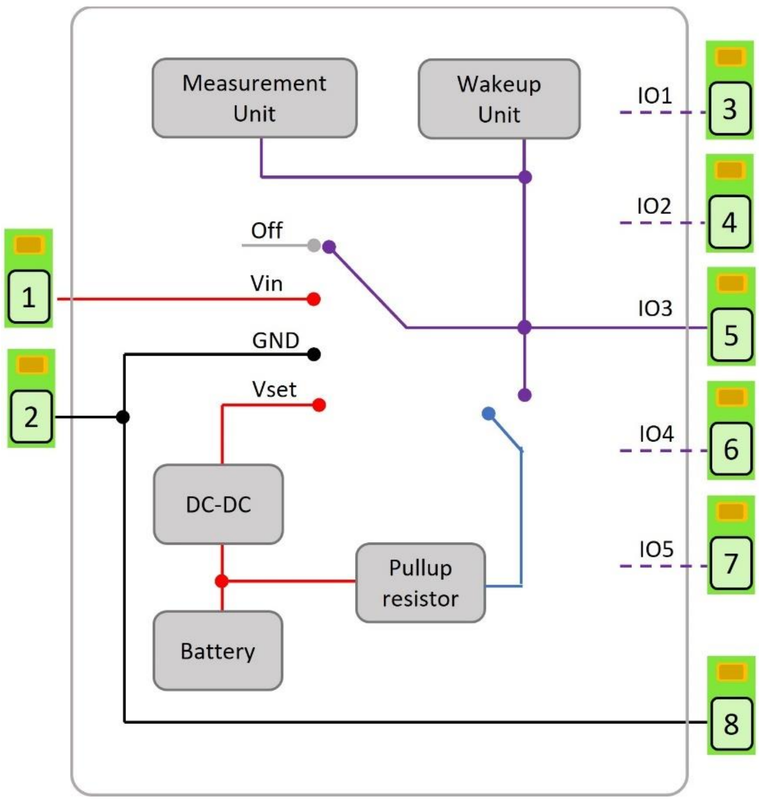


*Figure 5 - An Externally Connected Device can be Powered by Vin or Vset*

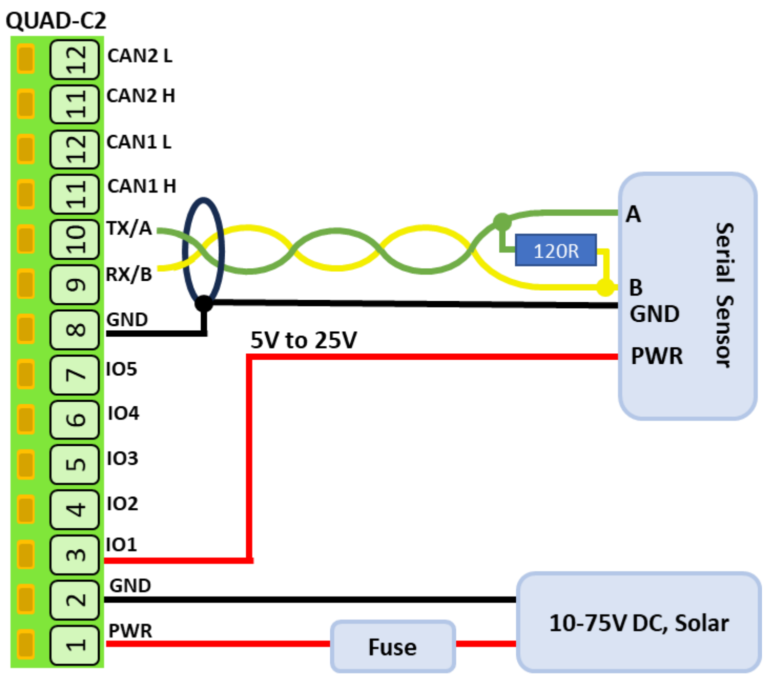
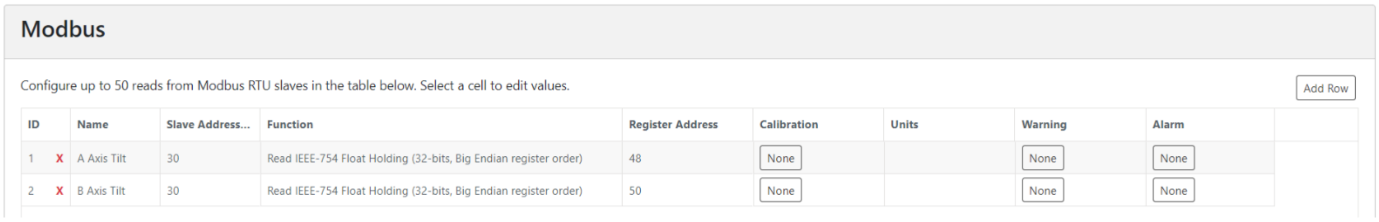


*Figure 6 - External Serial Device Powered by IO1*

## 3. Example 1: Senquip Device Always On, Powering External Modbus device.

This is the simplest of the examples as the external device is always powered, and so will be ready when the Senquip device reads Modbus data.

In this example, the serial *Mode* is set as Modbus, and the following two floating point Modbus reads are defined:



Modbus

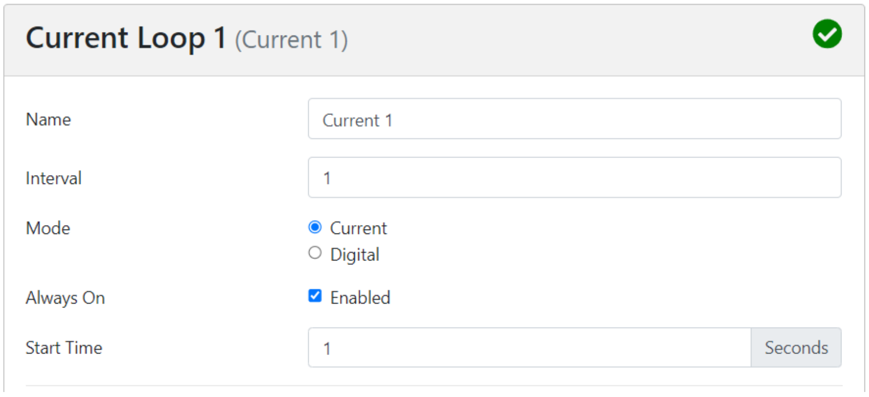
Configure up to 50 reads from Modbus RTU slaves in the table below. Select a cell to edit values.

| ID | | Name | Slave Address... | Function | Register Address | Calibration | Units | Warning | Alarm |
|---|---|---|---|---|---|---|---|---|---|
| 1 | X | A Axis Tilt | 30 | Read IEEE-754 Float Holding (32-bits, Big Endian register order) | 48 | None | | None | None |
| 2 | X | B Axis Tilt | 30 | Read IEEE-754 Float Holding (32-bits, Big Endian register order) | 50 | None | | None | None |

*Figure 7 - Modbus Settings for ORB and QUAD*

In the serial settings, we set the *Slave Timeout* as 400msec. This is the time that the Senquip device will wait before moving onto the next measurement. We set the *Delay Between Reads* as 15 msec. This is the time that the Senquip device will wait after having received a response from the slave before it sends the next request. Although the Modbus standard sets the minimum as 3.5 character periods, we find that many Modbus sensors need a longer time between reads.

For the ORB, Current Loop 1 is configured for current measurement, and is set as always on.

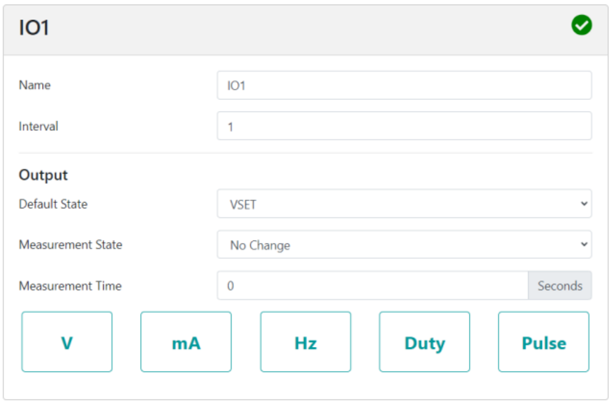


Current Loop 1 (Current 1)

| | |
|---|---|
| Name | Current 1 |
| Interval | 1 |
| Mode | [✓ Current] Digital |
| Always On | [✓ Enabled] |
| Start Time | 1 Seconds |

*Figure 8 - Senquip ORB Current Loop 1 Settings*

For the Senquip QUAD, IO1 is configured to supply Vset as default with no change during a measurement cycle. No measurements are set, although in a real application, voltage can be used to confirm the supply to the serial device is ok, and the current drawn by the serial device may provide a useful diagnostic.



IO1

| | |
|---|---|
| Name | IO1 |
| Interval | 1 |

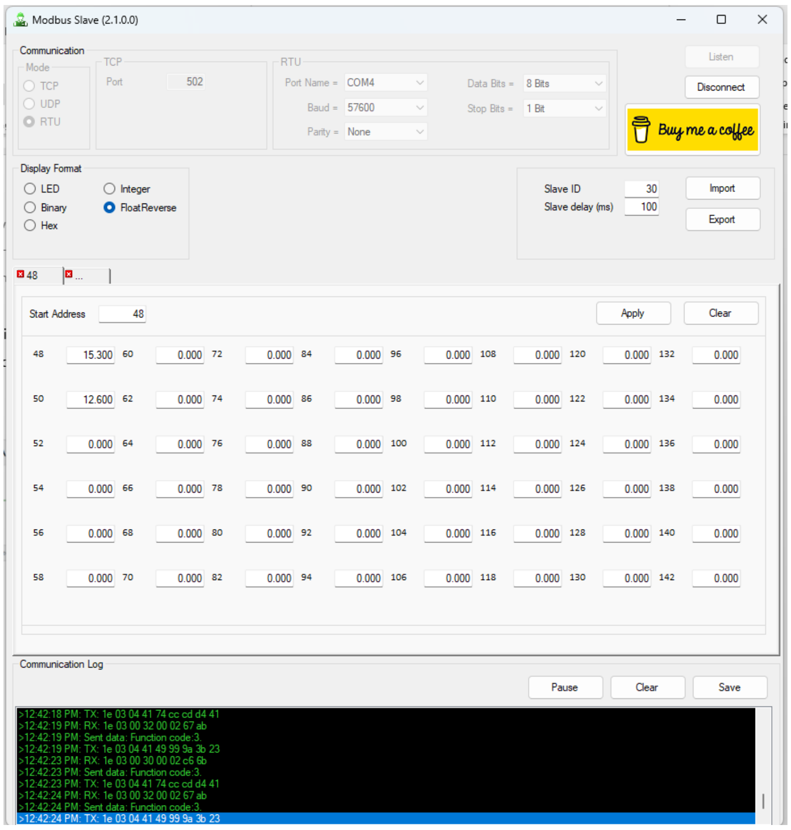
**Output**

| | |
|---|---|
| Default State | VSET |
| Measurement State | No Change |
| Measurement Time | 0 Seconds |

*Figure 9 - Senquip QUAD IO settings*

A base interval of 5 seconds is selected for both the Senquip ORB and QUAD.

A Modbus slave simulator from [ModbusTool](ModbusTool) is used to simulate an externally attached inclinometer. In Figure 10, we can see that the serial settings have been set to match those of the Senquip device, and that a slave with address 30 and two floating point registers at addresses 48 and 50 have been created. The software is set to delay 100msec before responding to a Modbus read request. This is significantly smaller than the 400 sec *Slave Timeout* set on the Senquip device.



Modbus Slave (2.1.0.0)

Communication

Mode: TCP / UDP / [✓ RTU]

TCP: Port 502

RTU: Port Name = COM4; Baud = 57600; Parity = None; Data Bits = 8 Bits; Stop Bits = 1 Bit

Display Format: LED / Binary / Hex / Integer / [✓ FloatReverse]

Slave ID 30

Slave delay (ms) 100

Start Address 48

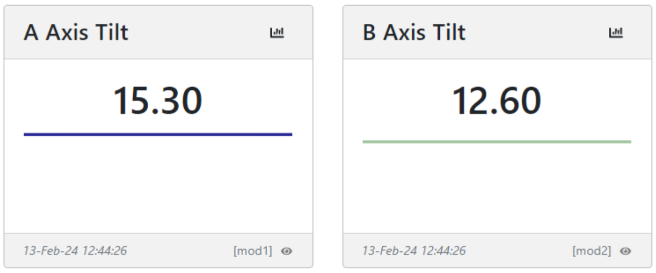
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 48 | 15.300 | 60 | 0.000 | 72 | 0.000 | 84 | 0.000 | 96 | 0.000 | 108 | 0.000 | 120 | 0.000 | 132 | 0.000 |
| 50 | 12.600 | 62 | 0.000 | 74 | 0.000 | 86 | 0.000 | 98 | 0.000 | 110 | 0.000 | 122 | 0.000 | 134 | 0.000 |
| 52 | 0.000 | 64 | 0.000 | 76 | 0.000 | 88 | 0.000 | 100 | 0.000 | 112 | 0.000 | 124 | 0.000 | 136 | 0.000 |
| 54 | 0.000 | 66 | 0.000 | 78 | 0.000 | 90 | 0.000 | 102 | 0.000 | 114 | 0.000 | 126 | 0.000 | 138 | 0.000 |
| 56 | 0.000 | 68 | 0.000 | 80 | 0.000 | 92 | 0.000 | 104 | 0.000 | 116 | 0.000 | 128 | 0.000 | 140 | 0.000 |
| 58 | 0.000 | 70 | 0.000 | 82 | 0.000 | 94 | 0.000 | 106 | 0.000 | 118 | 0.000 | 130 | 0.000 | 142 | 0.000 |

Communication Log

```
>12:42:18 PM: TX: 1e 03 04 41 74 cc cd d4 41
>12:42:19 PM: RX: 1e 03 00 32 00 02 67 ab
>12:42:19 PM: Sent data: Function code:3.
>12:42:19 PM: TX: 1e 03 04 41 49 99 9a 3b 23
>12:42:23 PM: RX: 1e 03 00 30 00 02 c6 6b
>12:42:23 PM: Sent data: Function code:3.
>12:42:23 PM: TX: 1e 03 04 41 74 cc cd d4 41
>12:42:24 PM: RX: 1e 03 00 32 00 02 67 ab
>12:42:24 PM: Sent data: Function code:3.
>12:42:24 PM: TX: 1e 03 04 41 49 99 9a 3b 23
```

*Figure 10 - Modbus Slave Simulator Settings*

The Modbus data is being correctly displayed on the Senquip Portal.



Looking at a trace of the Modbus comms on the Senquip QUAD, we can see the following:

1. The first Modbus read of register 48,
2. The first response from the external sensor approx. 150msec later. We found that the 100msec delay in the simulator software was not very accurate and varied between reads,
3. The second Modbus read of register 15 msec after the response, as set in *Delay Between Reads*,
4. The second Modbus response from the sensor,
5. The always on 12V from IO1.

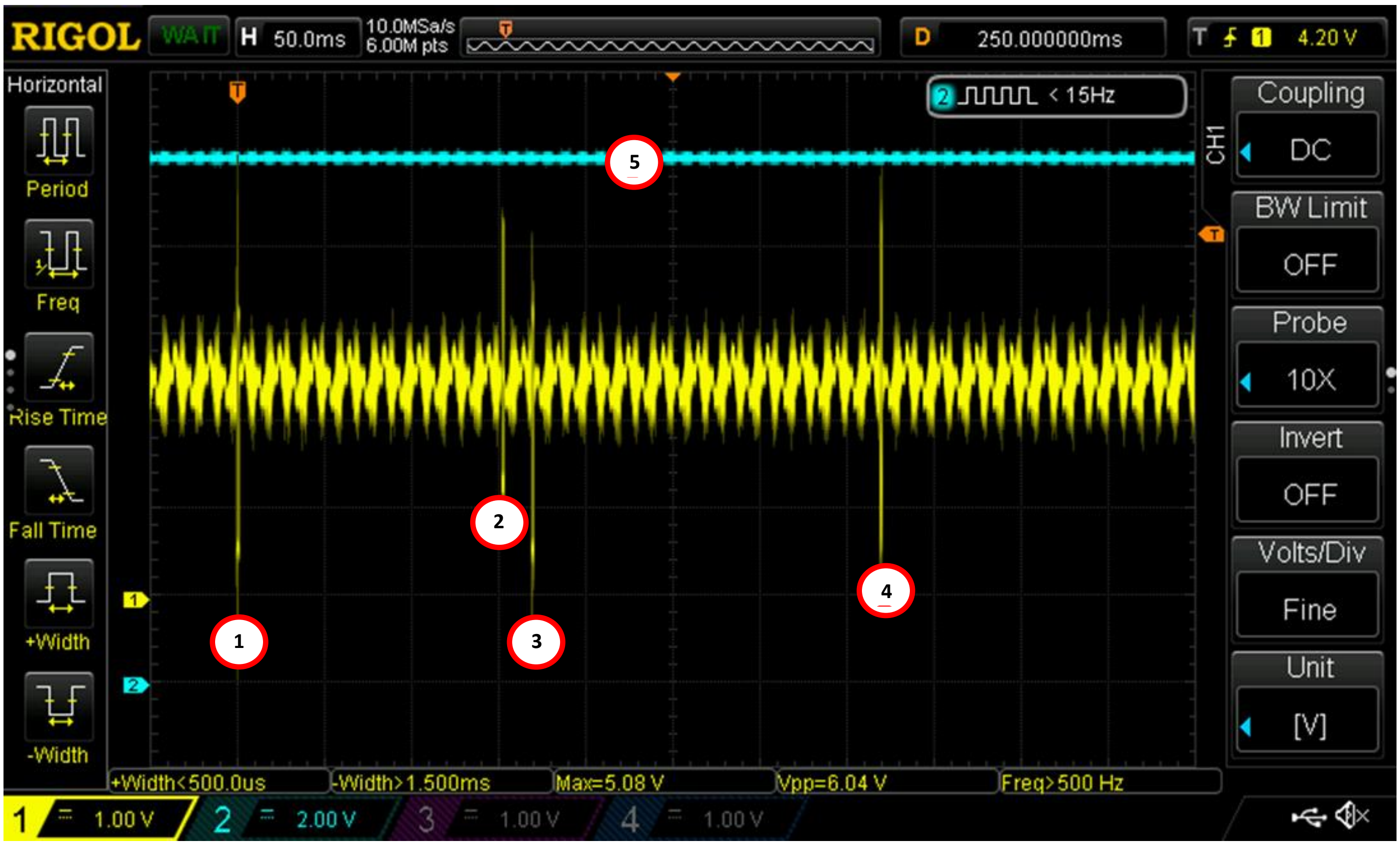


*Figure 11 - Oscilloscope Trace Showing Modbus Read Timing*

The RS485 data is noisy between reads as the line is high impedance and is only loosely referenced to ground. The same noise will be present on the A and B line and will cancel when the 2 signals are subtracted at the RS485 driver

Figure 12 shows Figure 12 RS485 data zoomed in to show the first few bytes of the first Modbus read. The figure shows:

1. Noisy RS485 A signal,
2. Noisy RS485 B signal,
3. Cleaned up A-B signal,
4. The decoded Rs485 signal showing 0x1E and 0x03, which represent the address of the device to be read (30 decimal) and the function code (3) for read a holding register.

For more information on Modbus, see APN0020 - Writing to Modbus Devices.

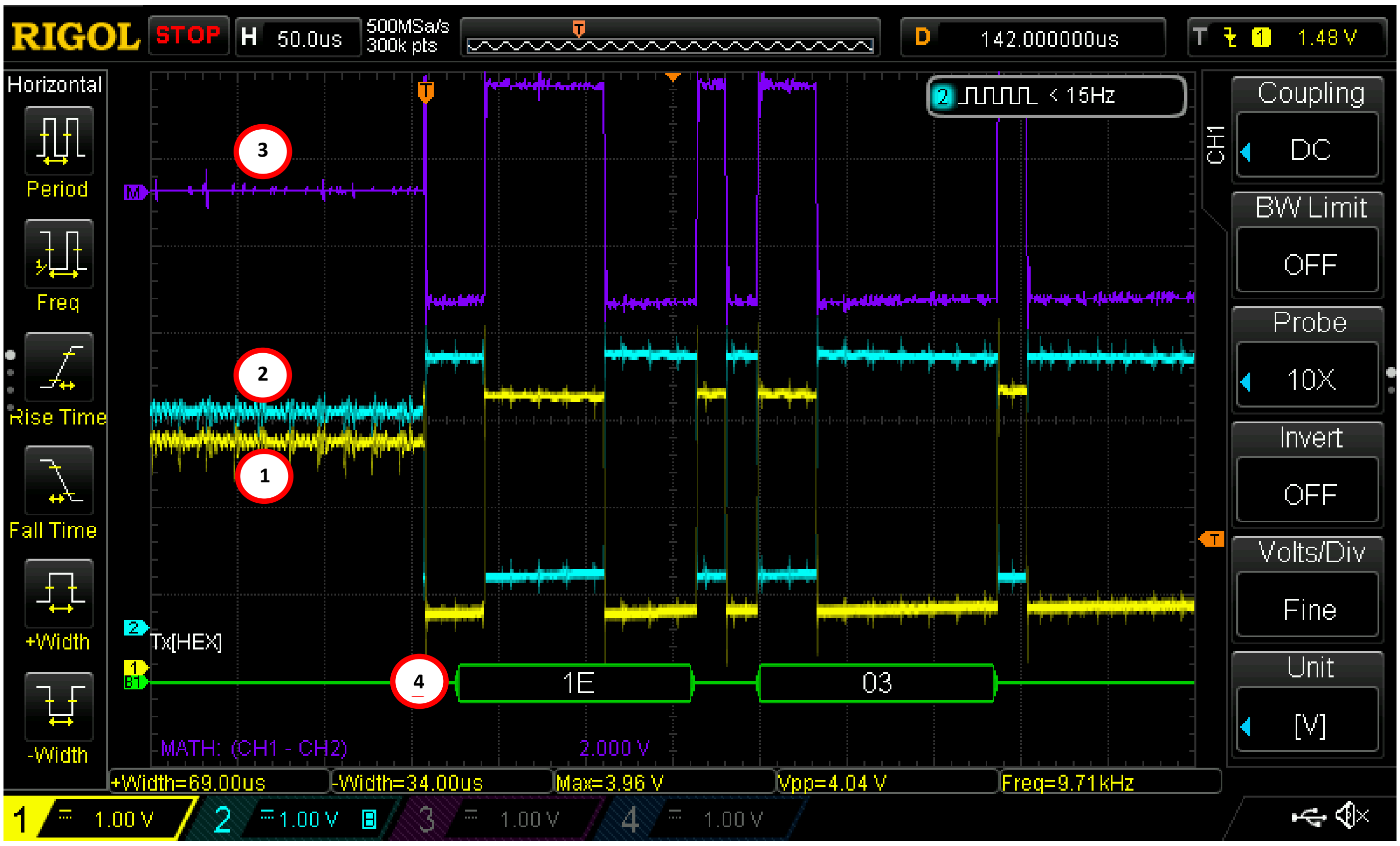


*Figure 12 - RS485 Signals Showing A, B and Decoded Data*

We have demonstrated how a Senquip device can power an external sensor and read Modbus data from that sensor in a powered, “always on” application.

## 4. Example 2: Senquip Device Requests Serial Data from External Serial Device.

In this example, we will assume that an engine controller used in a lighting tower is connected to a Senquip device. The engine controller sends serial data in response to a request.

- Request: “CST:”
- Response: “CST:2024-02-12-MON-04:08,M,RUN,1783,9124.8,43,13.7,ON,No Action in Man\x0A”

The serial string response contains the following information separated by commas. The data could easily be parsed using a simple script.

| Parameter | Value |
|---|---|
| Start of Message | CST: |
| Time and Date | 2024-02-12-MON-04:08 |
| Engine Controller Mode | Manual (M) |
| Engine State | Run |
| Engine Speed | 1783 RPM |
| Fuel Level | 43% |
| Battery Voltage | 13.7V |
| Light Status | On |
| End of Message | Line Feed ( 0x0A) |

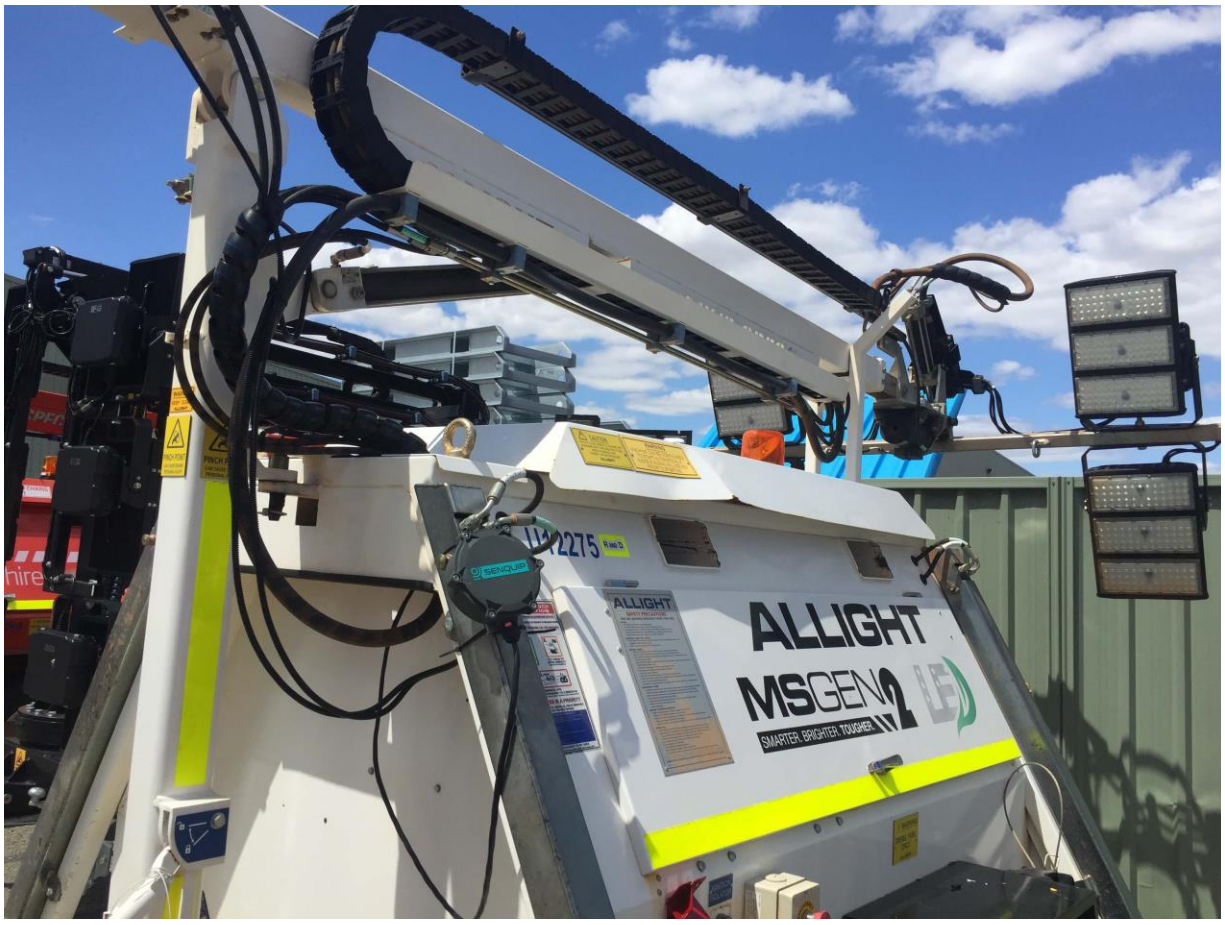
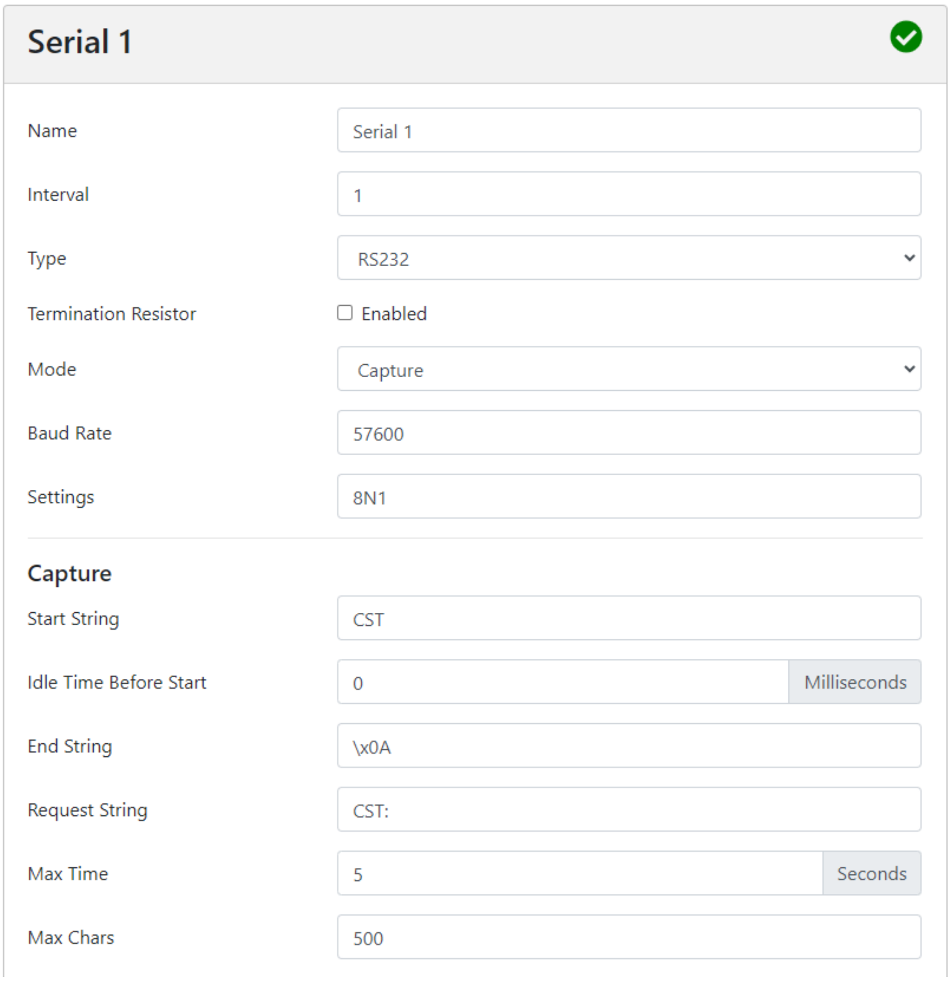


*Figure 13 - Typical Lighting Tower with Serial Communications*

In this example, the serial port mode is changed to RS232 to match that of the lighting tower engine controller. The same IO1 and Current Loop 1 pin configuration as example 1 are used.

This time, the serial port is configured as RS232 *Capture*. In capture mode, the Senquip device serial port will capture everything that arrives on the serial port. To make the data meaningful, a *Start String* can be specified. Data will only be captured after the start string arrives. Start strings can be useful in correctly aligning data as it arrives. In this example, the string starts with "CST" and so this has been used as the *Start* String. A *Stop String* is also specified. Data capture will end when the stop string is detected. In this example, the message ends with a carriage return and so 0x0A (ASCII for LF) has been used as the *Stop String*.

A *Request String* can be used where the externally connected serial device requires a prompt to return data. The lighting tower engine controller responds to "CST:" and so this has been used as the *Request String*.



Serial 1

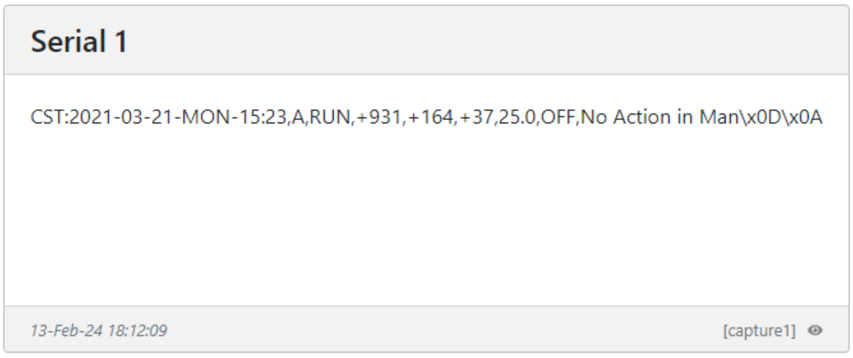
| | |
|---|---|
| Name | Serial 1 |
| Interval | 1 |
| Type | RS232 |
| Termination Resistor | ☐ Enabled |
| Mode | Capture |
| Baud Rate | 57600 |
| Settings | 8N1 |

Capture

| | |
|---|---|
| Start String | CST |
| Idle Time Before Start | 0 Milliseconds |
| End String | \x0A |
| Request String | CST: |
| Max Time | 5 Seconds |
| Max Chars | 500 |

*Figure 14 - Serial Port Settings*

The serial data is arriving correctly on the Senquip Portal.


Serial 1

CST:2021-03-21-MON-15:23,A,RUN,+931,+164,+37,25.0,OFF,No Action in Man\x0D\x0A

13-Feb-24 18:12:09 [capture1]

*Figure 15 - Serial Data Arriving on the Senquip Portal*

Looking at the serial data, we see the following:

1. The serial request string is sent by the Senquip device,
2. The response from the lighting tower controller follows shortly.

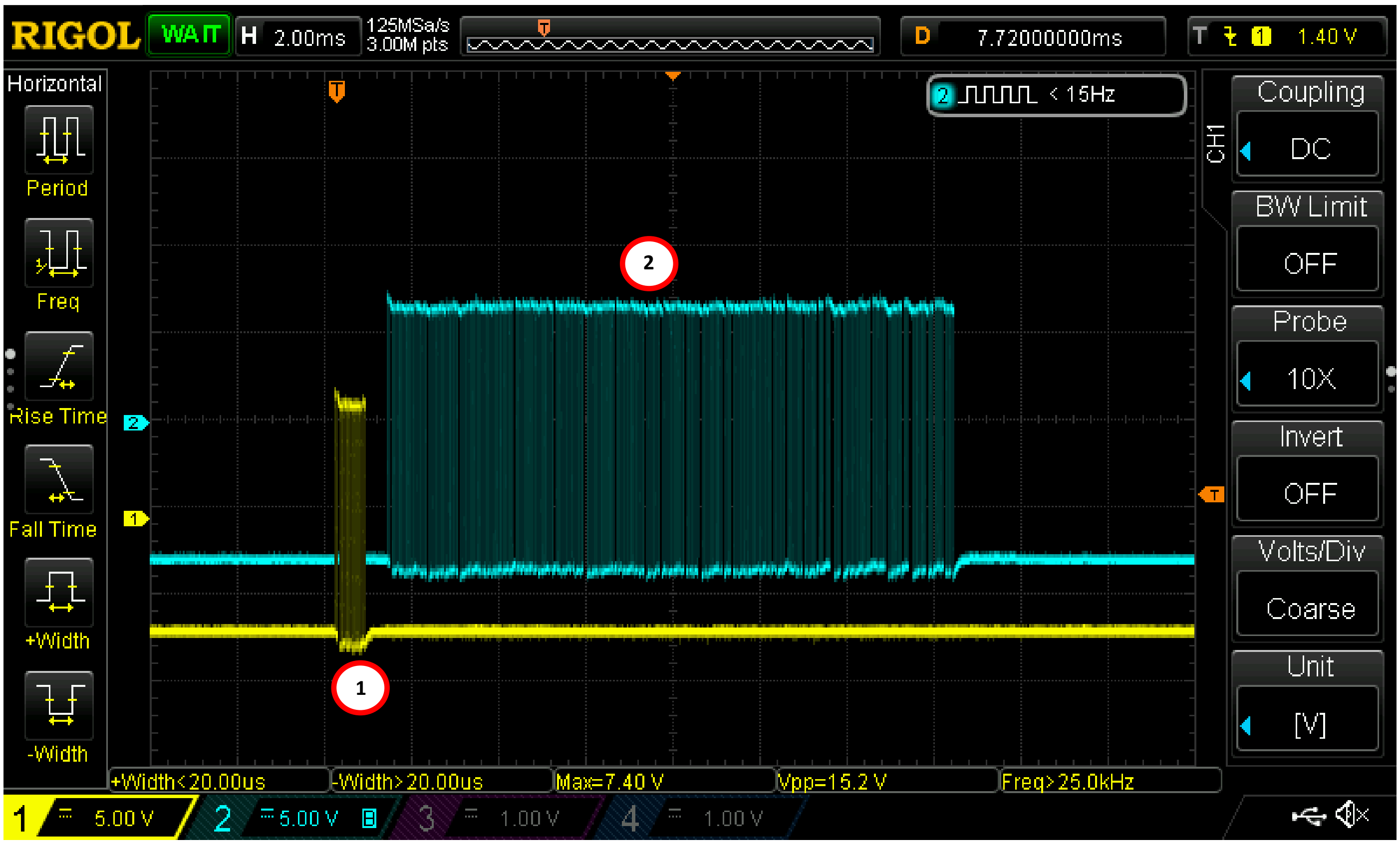


*Figure 16 - Request String and Response*

Zooming in on the data transmitted by the Senquip device, we see the request string “CST:”

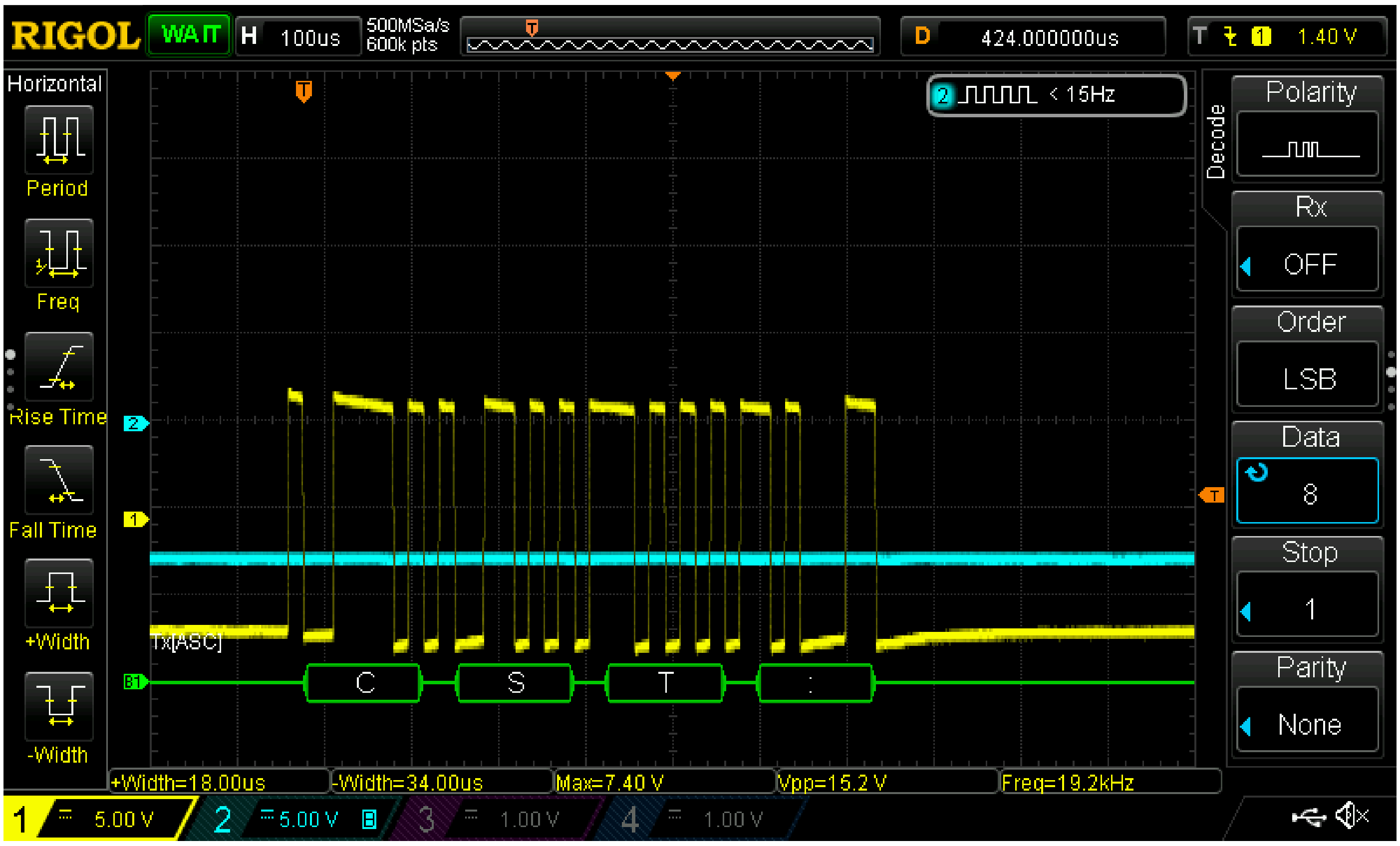


*Figure 17 - Request String Decoded*

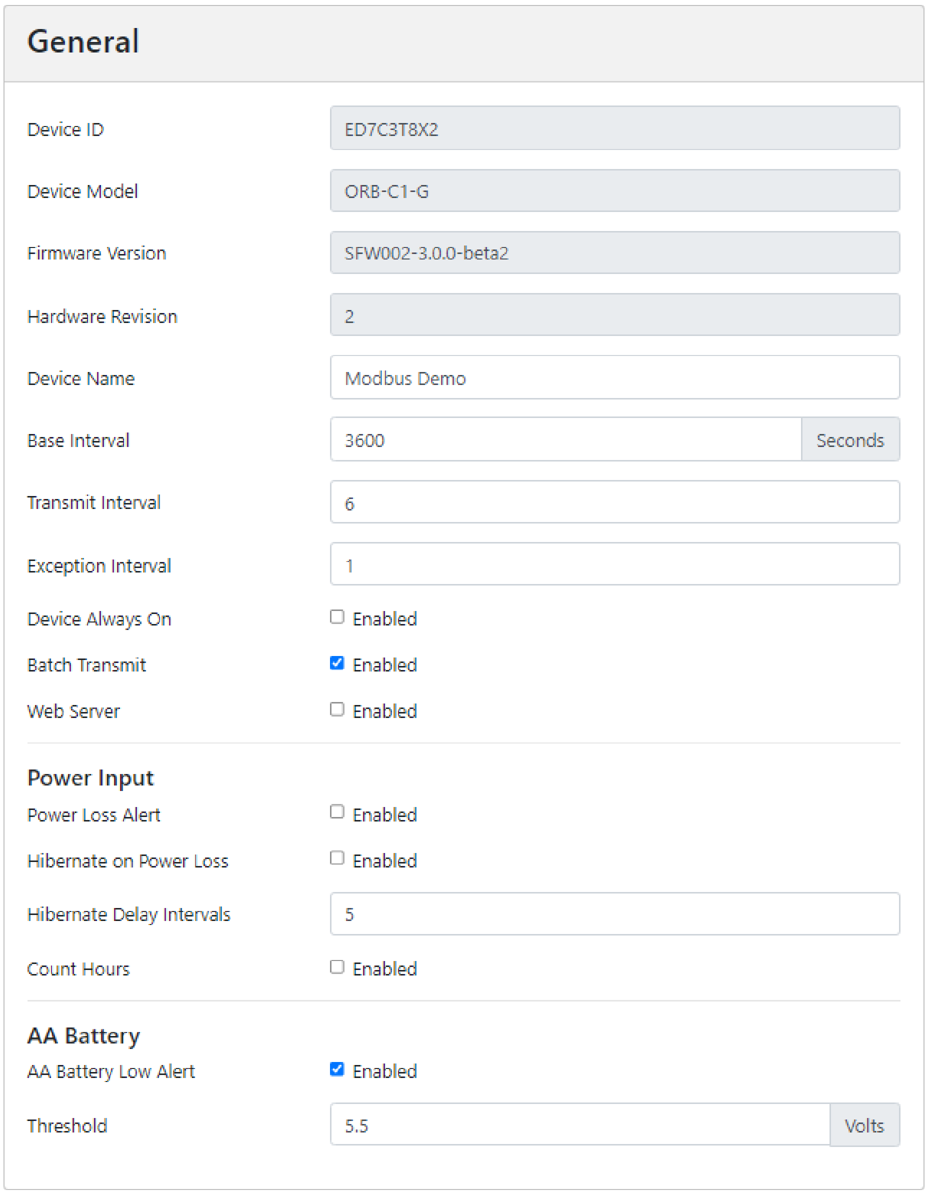
We have demonstrated how a Senquip device can wake periodically and poll an external device to request serial data.

## 5. Example 3: Waking Periodically and Powering External Modbus Device with Boot Time.

In this example, we assume that the Senquip device is running on AA batteries or solar and so we will configure it for low power. The Senquip device will be off for most of the time, and will wake, power the connected Modbus slave, read data and then return to sleep. The connected sensor requires 5 seconds to boot and so the Modbus reads must be delayed for at least 5 seconds.

The *General* settings of the Senquip device need to be configured for low power, and for operation with AA or Solar. Firstly, the *Base Interval* is set to 1 hour and the transmit interval to 6 so that measurements are taken hourly and sent every 6 hours. *Batch Transmit* is enabled so that all 6 measurements are sent on transmit, rather than just the latest. *Hibernate* is turned off as there is no external power and we do not want the device entering hibernate mode, which is a higher power mode than sleep. For the Senquip ORB, an AA battery alert is enabled at 5.5V.

Other changes made are to turn off GPS, and all other unused peripherals. Also, avoid the use of HTTP endpoints that take longer to complete transmission than MQTT endpoints.



General

| | |
|---|---|
| Device ID | ED7C3T8X2 |
| Device Model | ORB-C1-G |
| Firmware Version | SFW002-3.0.0-beta2 |
| Hardware Revision | 2 |
| Device Name | Modbus Demo |
| Base Interval | 3600 Seconds |
| Transmit Interval | 6 |
| Exception Interval | 1 |
| Device Always On | ☐ Enabled |
| Batch Transmit | ☑ Enabled |
| Web Server | ☐ Enabled |

Power Input

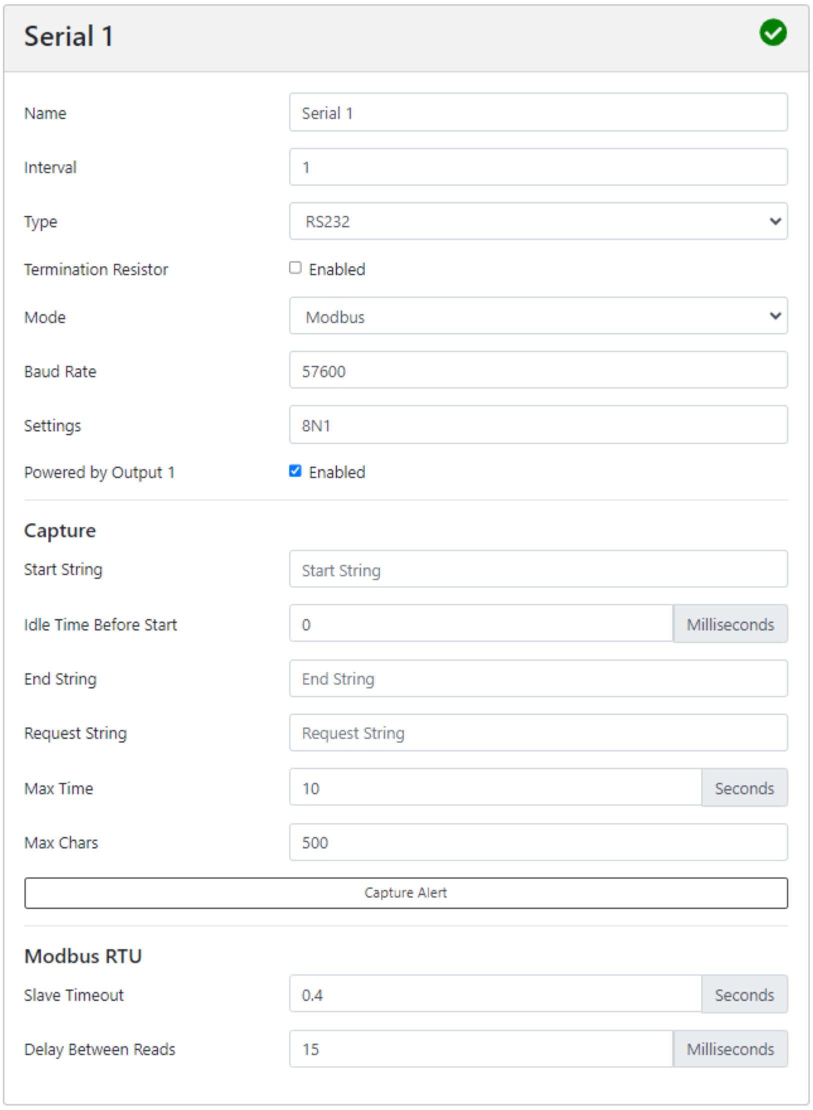
| | |
|---|---|
| Power Loss Alert | ☐ Enabled |
| Hibernate on Power Loss | ☐ Enabled |
| Hibernate Delay Intervals | 5 |
| Count Hours | ☐ Enabled |

AA Battery

| | |
|---|---|
| AA Battery Low Alert | ☑ Enabled |
| Threshold | 5.5 Volts |

*Figure 18 - General Settings for Low Power*

The serial is reconfigured for Modbus and the same 2 reads are configured as in Example 1. A new setting, *Powered by Output 1*, available on SFW002-3.0.0 and SFW003-4.0.0 and later variants allows the read of serial data to be delayed by the measurement time associated with IO1 on the Senquip QUAD or Current Loop 1 on the Senquip ORB. The sensor connected to IO1 or Current Loop 1 will be powered during this time. This allows for external serial sensors to boot before being read.



**Serial 1**

| | |
|---|---|
| Name | Serial 1 |
| Interval | 1 |
| Type | RS232 |
| Termination Resistor | ☐ Enabled |
| Mode | Modbus |
| Baud Rate | 57600 |
| Settings | 8N1 |
| Powered by Output 1 | ☑ Enabled |

**Capture**

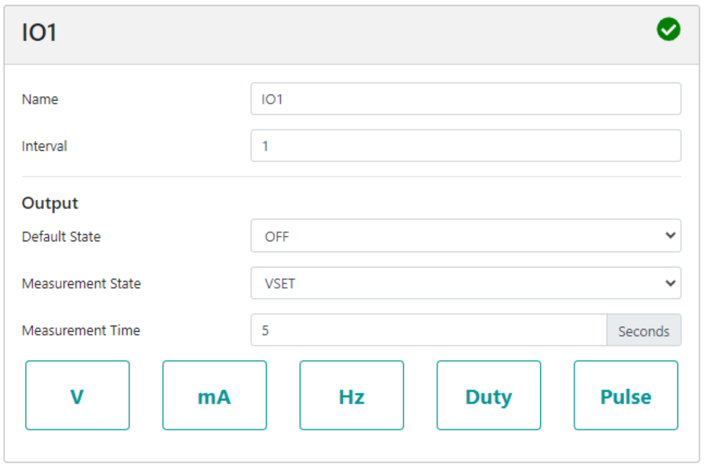
| | |
|---|---|
| Start String | Start String |
| Idle Time Before Start | 0 Milliseconds |
| End String | End String |
| Request String | Request String |
| Max Time | 10 Seconds |
| Max Chars | 500 |

**Modbus RTU**

| | |
|---|---|
| Slave Timeout | 0.4 Seconds |
| Delay Between Reads | 15 Milliseconds |

*Figure 19 - Serial Port Settings Including New Link to IO Function*

IO1 on a Senquip QUAD is configured as normally off. During a measurement cycle, it boosts the LiPo backup battery voltage to 12V (as configured by *VSET Voltage*), and after measurement, returns to the default of off. Our sensor has a boot time of 5 seconds and so we will set the IO1 Measurement Time to 5 seconds. Because the serial port is now linked to IO1, the Modbus reads will be delayed by 5 seconds.



IO1

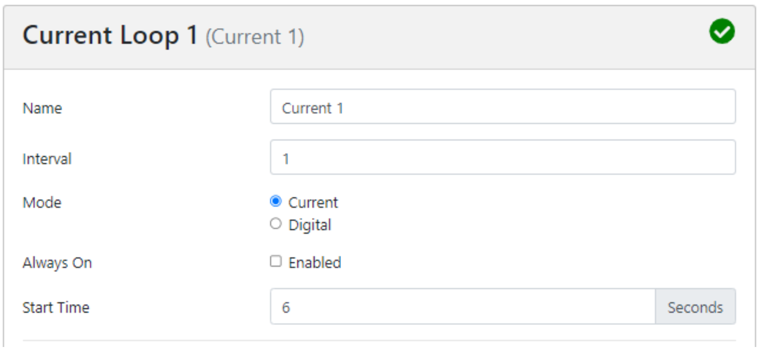
| | |
|---|---|
| Name | IO1 |
| Interval | 1 |

**Output**

| | |
|---|---|
| Default State | OFF |
| Measurement State | VSET |
| Measurement Time | 5 Seconds |

*Figure 20 - Senquip QUAD IO1 Settings*

If using a Senquip ORB, configure Current Loop 1 with a 5 second *Start Time.*



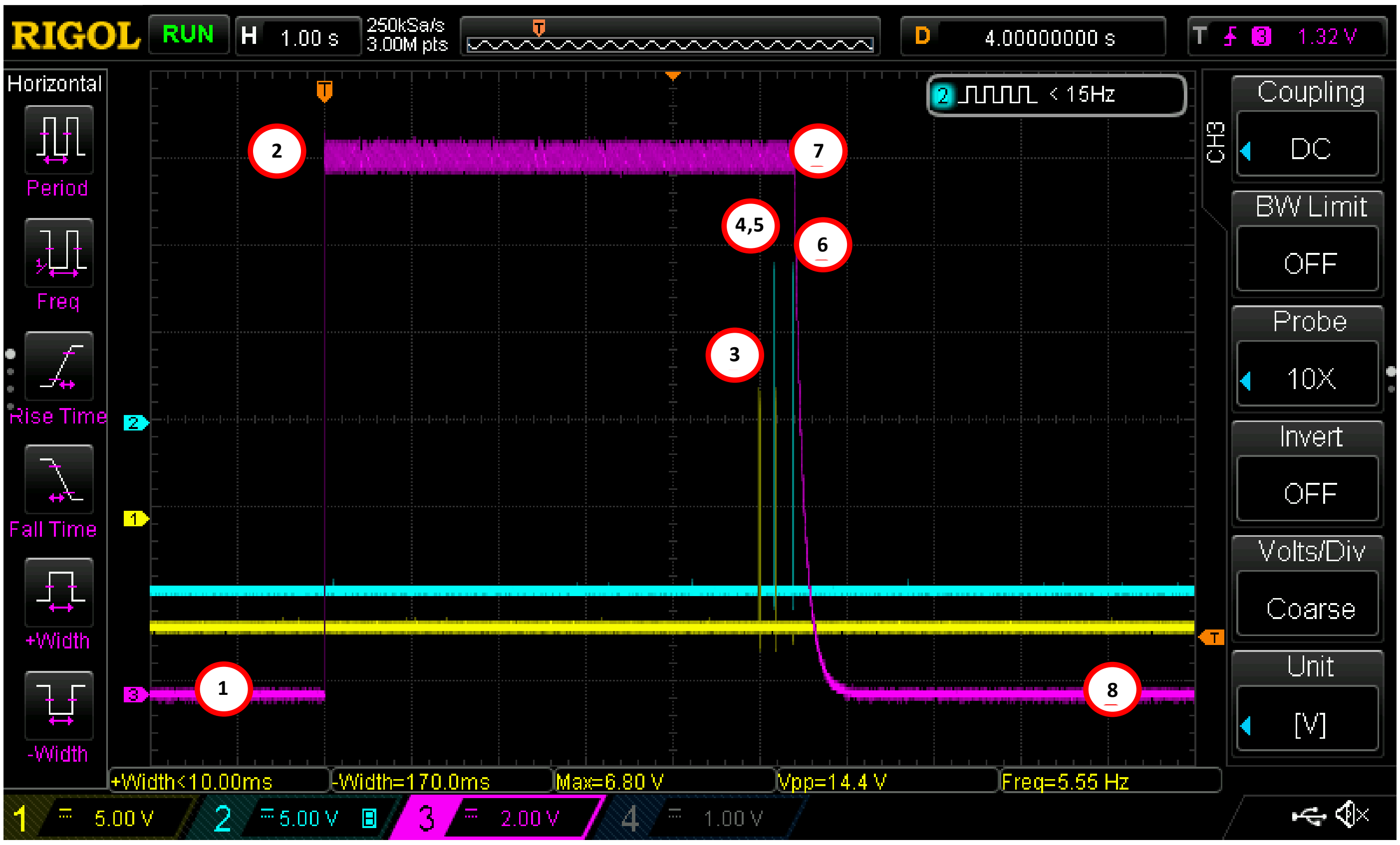
Current Loop 1 (Current 1)

| | |
|---|---|
| Name | Current 1 |
| Interval | 1 |
| Mode | [✓ Current] Digital |
| Always On | Enabled |
| Start Time | 6 Seconds |

*Figure 21 - Senquip ORB Current Loop 1 Settings*

In Figure 22, which shows the timing of a delayed Modbus read, we see the following:

1. Before a measurement cycle, there is no power on IO1,
2. At the start of a measurement cycle, 12V is applied to IO1,
3. The first Modbus read is executed exactly 5 seconds after power is applied to the sensor,
4. The sensor replies to the read command,
5. The Senquip device performs the second read 15 msec after receiving the first reply,
6. The sensor responds to the second read,
7. Power is removed from IO1 only when Modbus measurements are complete,
8. Power remains off till the next measurement cycle.


RIGOL RUN H 1.00 s 250kSa/s 3.00M pts D 4.00000000 s T 3 1.32 V

Coupling DC, BW Limit OFF, Probe 10X, Invert OFF, Volts/Div Coarse, Unit [V]

+Width<10.00ms -Width=170.0ms Max=6.80 V Vpp=14.4 V Freq=5.55 Hz

1 5.00 V 2 5.00 V 3 2.00 V 4 1.00 V

*Figure 22 - Delayed Modbus Read*

We have demonstrated that a Senquip device can wake periodically, power an external Modbus slave, and delay measurement until the slave device has booted.

## 6. Example 4: Waking Periodically and Powering Device that Requires Modbus Write.

In this application, we assume that the externally connected Modbus slave requires a 3 second boot time after which the Senquip device must perform a Modbus write to the slave to start a measurement. One second after sending the Modbus write command, we can perform our Modbus reads. We will again assume this is a low power implementation where the Senquip device will wake from sleep, take a sample, and return to sleep to save power.

For more information on writing to Modbus slave devices, see APN0020, Writing to Modbus Devices.

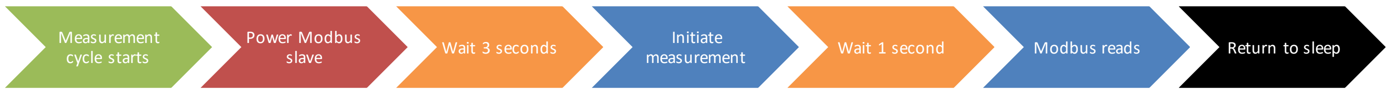


*Figure 23 - Sensor Read Timing*

In this application, we will use the device settings to apply power to the Modbus slave, and perform the Modbus reads. We will use a script to wait the boot period and then send the Modbus write command. Figure 24 shows how the script starts running at approximately the same time that the device boots and applies power to the sensor. The script is then set to execute a Modbus write after 3 seconds, allowing a further second before the Modbus reads are initiated by the Senquip device.

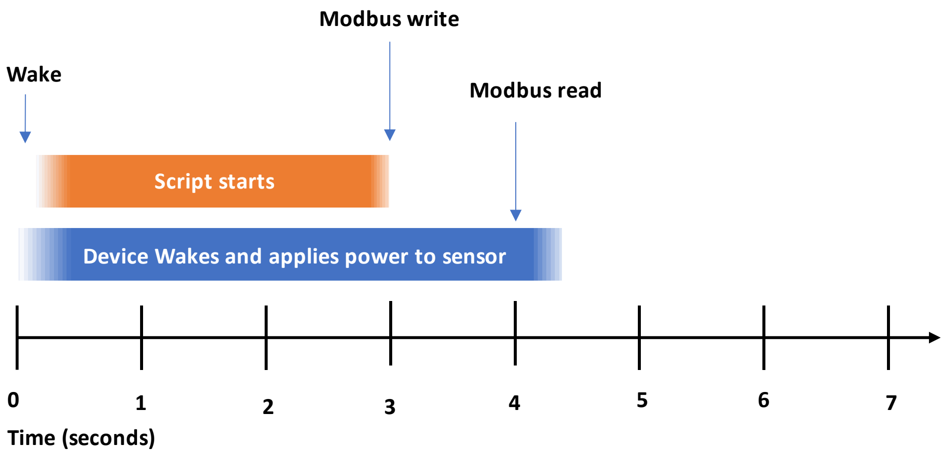
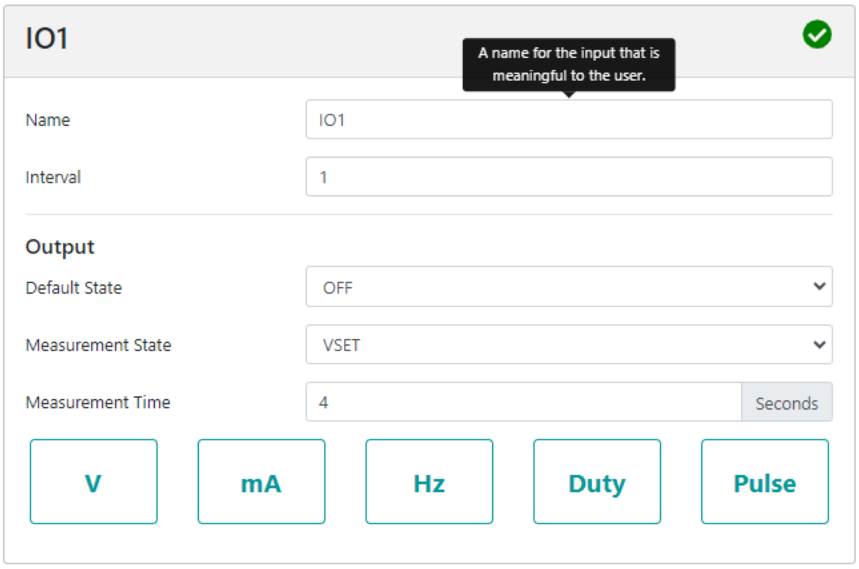


*Figure 24 - Relative Timing of Device and Script*

We will use the same general and serial port settings as Example 3, including the use of the new *Powered by Output 1* setting, except that we will only define a single Modbus read. We will set the *Measurement Time* on IO 1 to 4 seconds to allow the 3 seconds for boot and an extra second between writing to the Modbus and reading the response.


IO1

A name for the input that is meaningful to the user.

| | |
|---|---|
| Name | IO1 |
| Interval | 1 |

**Output**

| | |
|---|---|
| Default State | OFF |
| Measurement State | VSET |
| Measurement Time | 4 Seconds |

*Figure 25 - IO1 Settings with Extended Measurement Time*

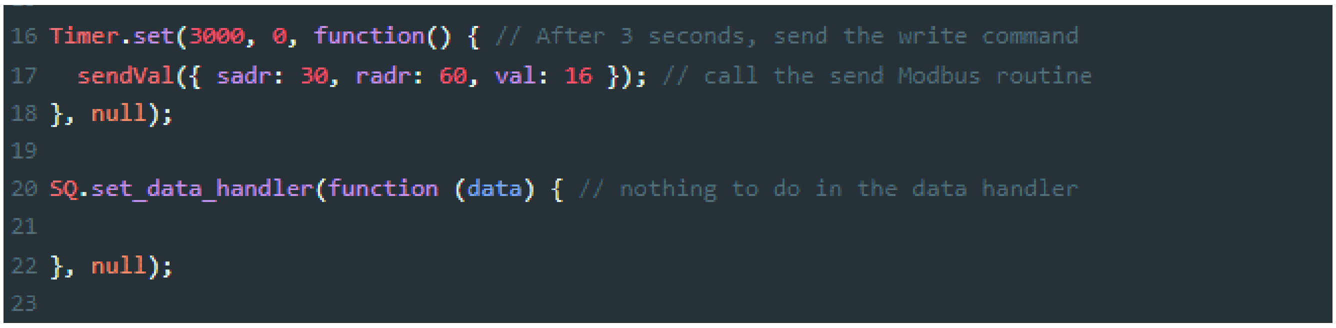
We will now write a script to send a Modbus write at the 3 second mark. The external sensor has address 30, and requires a write of value 16 to register number 60.

The script starts by loading the required library files. It then defines a function that receives address, register and value, and executes a Modbus write. The details of this function are described in APN0020, Writing to Modbus Devices.

```
load("senquip.js");
load("api_serial.js");
load('api_timer.js');

function sendVal(sendObj) {
  let s = SQ.encode(sendObj.sadr, SQ.U8); // encode dec address into hex
  let r = SQ.encode(sendObj.radr, SQ.U16); // encode dec register number into hex
  let v = SQ.encode(sendObj.val, SQ.U16); // encode dec data into hex
  let a = s + "\x06" + r + v; // 6 is the MODBUS write unsigned 16 function code
  let c = SQ.crc(a); // use the Senquip CRC function to calculate the Modbus CRC
  c = SQ.encode(c, -SQ.U16); // encode the CRC function in hex + flip byte order
  let t = a + c; // create the final Modbus write message
  SERIAL.write(1, t, t.length); // send the message to serial port 1
}
```

A timer is started that, when it expires after 3 seconds, calls the sendVal function to perform a Modbus write with the required address, register, and value.



```
Timer.set(3000, 0, function() { // After 3 seconds, send the write command
  sendVal({ sadr: 30, radr: 60, val: 16 }); // call the send Modbus routine
}, null);

SQ.set_data_handler(function (data) { // nothing to do in the data handler

}, null);

```

Figure 26 shows the timing of the Modbus write and read. In the figure, we can see that:

1. Before a measurement cycle, there is no power on IO1,
2. At the start of a measurement cycle, 12V is applied to IO1,
3. A Modbus write is executed approximately 3 seconds after power is applied,
4. The sensor responds to the write,
5. One second later, the Modbus read is executed,
6. The sensor replies to the read command,
7. Power is removed from IO1 only when Modbus measurements are complete,
8. Power remains off till the next measurement cycle.

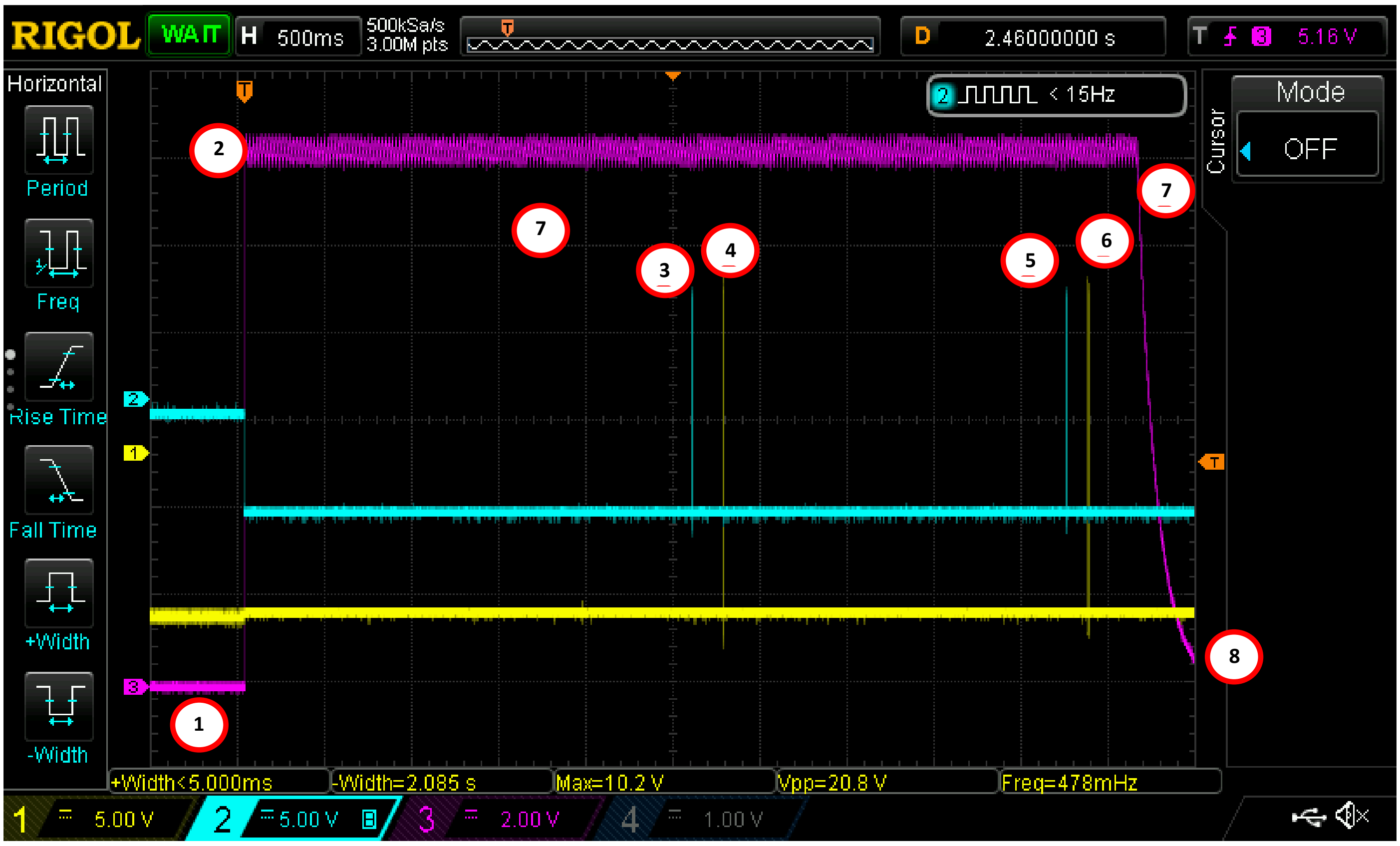


*Figure 26 - Timing of Modbus Write and Read*

### 6.1. Writing to Multiple Sensors

An extension to this example would be a string of Modbus sensors that each require a write to start a measurement. An example would be the Osprey IPX in-place extensometer where each of the Modbus elements require a write of value 55 to register 99 to initiate a measurement. The writes need to be spaced by at least 200msec to allow each sensor to respond.

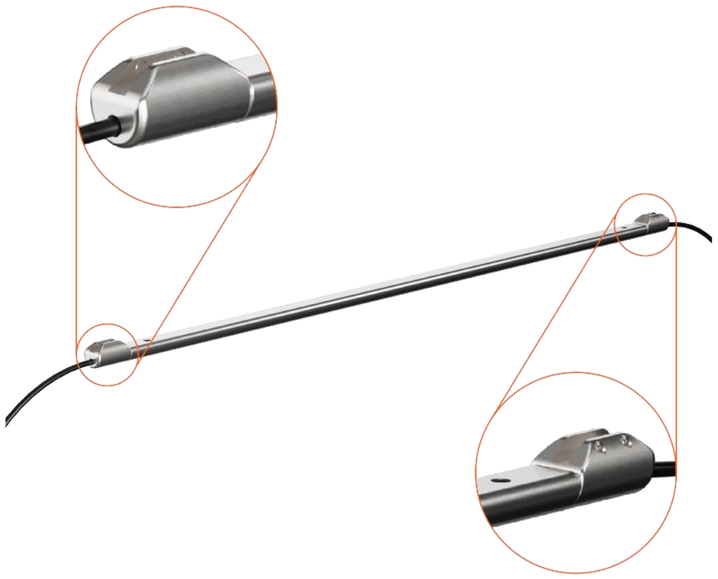


*Figure 27 - Osprey IPX Extensometer*

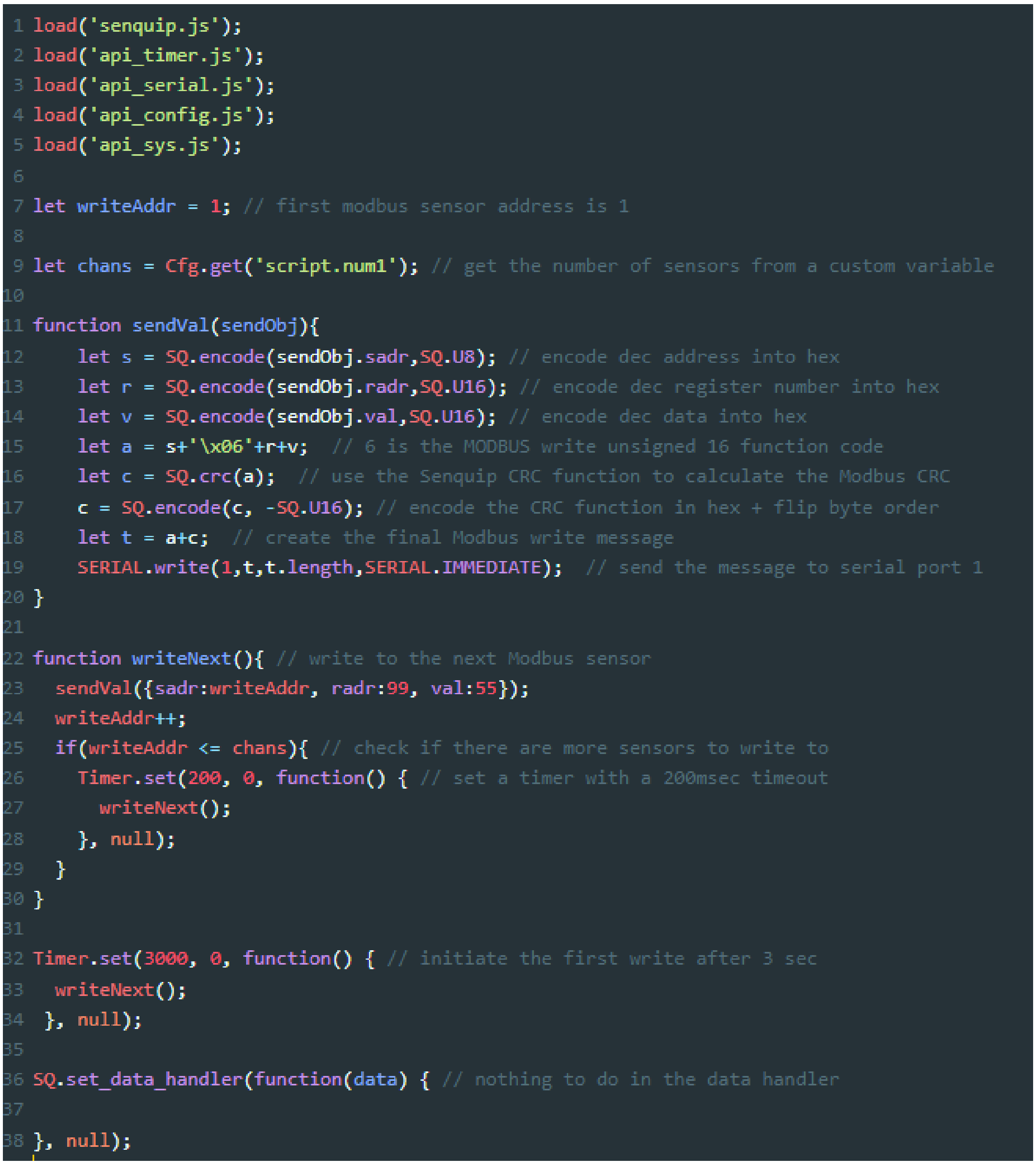
In the script below, the same Modbus write function is used, but this time it is called multiple times, staring with Modbus address 1 and ending when all the slave devices have been written. The number of slave devices is read from a custom variable to allow simple setup of strings of devices with different numbers of sensors.

The script starts by initialising a variable, writeAddr, to the fist Modbus address, 1 in this case, and loading the chans variable with the number of elements in the string. We then declare a function writeNext that executes a Modbus write, increments the address counter and if there are still more sensors to write to, sets a 200msec timer which when it expires, calls itself. The process continues until all the sensors have been written to.

To initiate the process, and to allow the string of sensors time to boot, a timer with a 3 second expiration is set to call the first instance of writeNext.

The corresponding Modbus reads are configured in the Modbus settings.

The script is available in Appendix A.



```
load('senquip.js');
load('api_timer.js');
load('api_serial.js');
load('api_config.js');
load('api_sys.js');

let writeAddr = 1; // first modbus sensor address is 1

let chans = Cfg.get('script.num1'); // get the number of sensors from a custom variable

function sendVal(sendObj){
    let s = SQ.encode(sendObj.sadr,SQ.U8); // encode dec address into hex
    let r = SQ.encode(sendObj.radr,SQ.U16); // encode dec register number into hex
    let v = SQ.encode(sendObj.val,SQ.U16); // encode dec data into hex
    let a = s+'\x06'+r+v;  // 6 is the MODBUS write unsigned 16 function code
    let c = SQ.crc(a);  // use the Senquip CRC function to calculate the Modbus CRC
    c = SQ.encode(c, -SQ.U16); // encode the CRC function in hex + flip byte order
    let t = a+c;  // create the final Modbus write message
    SERIAL.write(1,t,t.length,SERIAL.IMMEDIATE);  // send the message to serial port 1
}

function writeNext(){ // write to the next Modbus sensor
  sendVal({sadr:writeAddr, radr:99, val:55});
  writeAddr++;
  if(writeAddr <= chans){ // check if there are more sensors to write to
    Timer.set(200, 0, function() { // set a timer with a 200msec timeout
      writeNext();
    }, null);
  }
}

Timer.set(3000, 0, function() { // initiate the first write after 3 sec
  writeNext();
 }, null);

SQ.set_data_handler(function(data) { // nothing to do in the data handler

}, null);
```

*Figure 28 - Modbus Writes of Multiple Modbus Sensors*

We have demonstrated that a simple script in conjunction with settlings can allow a delayed write and subsequent read of a single and multiple Modbus slave sensors.

## 7. Example 5: Senquip Device Waking Periodically and Taking Multiple Samples

In this application, an external RS232 sensor sends a single measurement every time it is powered. We would like to wake periodically and take 3 readings from the sensor to form an average, before returning to sleep. If the sensor were a Modbus device, we would simply schedule 3 Modbus reads from the same register. This sensor, however, only sends ASCII data when powered, and only sends data once before returning to sleep.

A script will be written to control the power to the sensor, and the serial port will be set to capture the serial data as it arrives. The script is available in Appendix B.

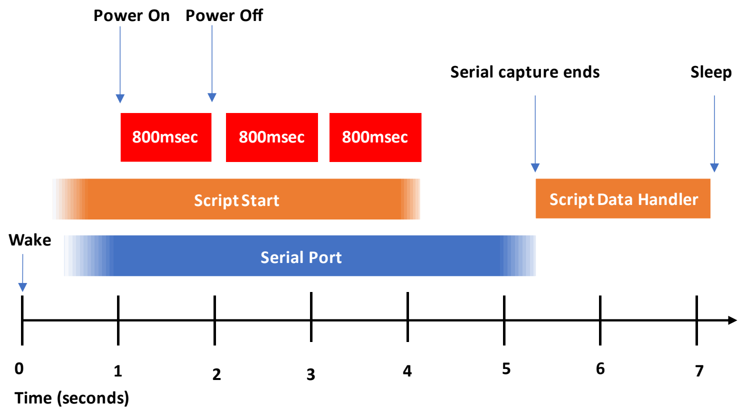
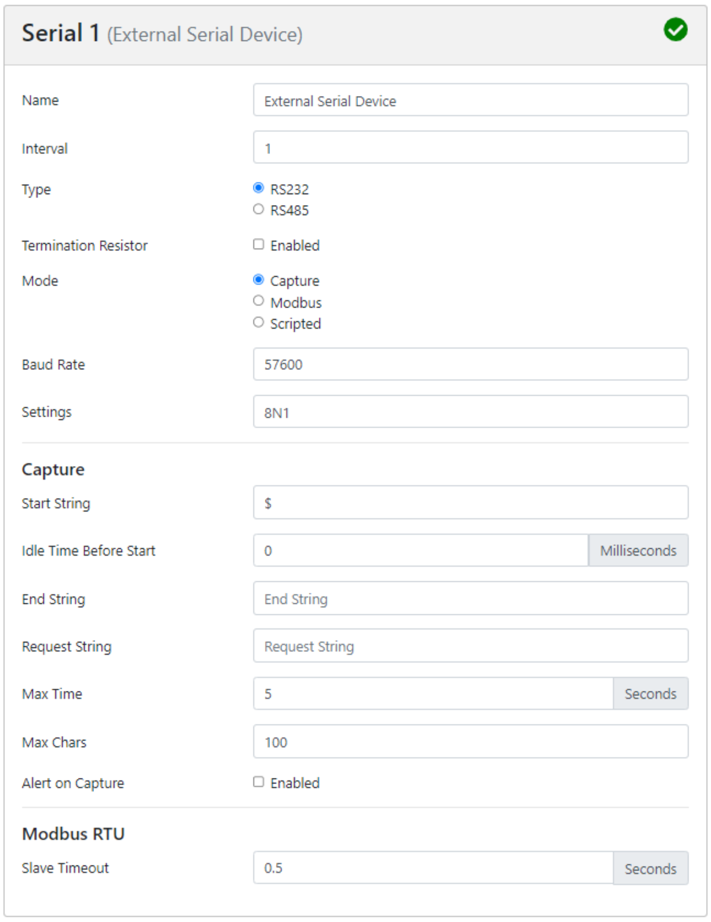


*Figure 29 - Sensor Power Timing*

The serial port on an ORB is configured in capture mode, with a *Max Time* of 5 seconds. The capture time servers the dual function of listening for serial data for 5 seconds to capture all three sensor transmissions, and also to hold the Senquip device on while the script is scheduling timers that otherwise might timeout after the device has returned to sleep. Since we know that the serial data starts with "$", we have set a *Start String* to match.

Current Loop 1 is disabled in the settings as it will be under script control.



**Serial 1** (External Serial Device)

| | |
|---|---|
| Name | External Serial Device |
| Interval | 1 |
| Type | (●) RS232 ( ) RS485 |
| Termination Resistor | ☐ Enabled |
| Mode | (●) Capture ( ) Modbus ( ) Scripted |
| Baud Rate | 57600 |
| Settings | 8N1 |

**Capture**

| | |
|---|---|
| Start String | $ |
| Idle Time Before Start | 0 Milliseconds |
| End String | End String |
| Request String | Request String |
| Max Time | 5 Seconds |
| Max Chars | 100 |
| Alert on Capture | ☐ Enabled |

**Modbus RTU**

| | |
|---|---|
| Slave Timeout | 0.5 Seconds |

*Figure s - Senquip ORB Serial Settings*

The script will create 3 power pulses to force the serial sensor to send three measurements. A delay between the start of the script and the power cycles will ensure that the serial port is ready to capture data.

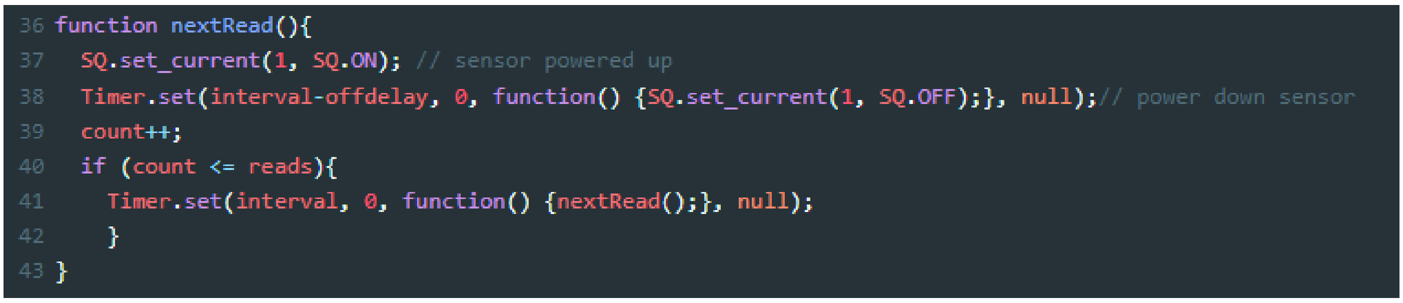
The script starts by including the required libraries. It then declares the interval between power cycles, the length for which the sensor will be powered down, the number of reads to be executed, and a temporary variable to hold the current power cycle number.

```
load('senquip.js');
load('api_timer.js');

let interval = 1000; // msec - time between reads
let offdelay = 200; // msec - power down time
let reads = 3; // number of reads to perform
let count = 1; // current read no
```

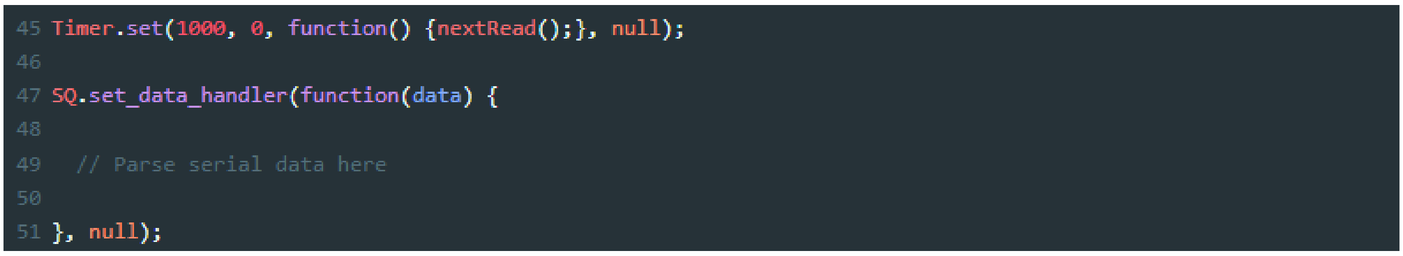
A function, nextRead, is then declared, that when called, will turn Current Loop 1 on and will set a timer for 800 msec later to turn the current loop off. If there are still more pulses to create, the function calls itself again. In simple terms, the function sets up a number of timers that set the on and off edges of the pulses to be generated.

It should be noted that the actual timeout of timers generated in scripts may slightly longer than that requested if the processor is busy with a high priority task when the timer expires.



```
function nextRead(){
  SQ.set_current(1, SQ.ON); // sensor powered up
  Timer.set(interval-offdelay, 0, function() {SQ.set_current(1, SQ.OFF);}, null);// power down sensor
  count++;
  if (count <= reads){
    Timer.set(interval, 0, function() {nextRead();}, null);
    }
}
```

The function nextRead is called after a delay of 1 second to allow time for the serial port to be initialised. The data handler in this case does nothing, but in a real application would be where the serial message is parsed, and the average generated.



```
Timer.set(1000, 0, function() {nextRead();}, null);

SQ.set_data_handler(function(data) {

  // Parse serial data here

}, null);
```

Figure 30 shows the pulse and serial timing. Note that the off times are slightly different from each other. This will be because the timer interrupts would not have been able to be serviced immediately when the timers expired as the process was busy performing high priority tasks. In the figure, we can see:

1. A small change in the serial level; this may be the Senquip device initialising the serial port,
2. The start of pulse 1, which is approximately 800msec wide,
3. Serial message 1 arriving,
4. The start of pulse 2,
5. Serial message 2 arriving,
6. The start of pulse 3,
7. Serial message 3 arriving,
8. End of power cycles.

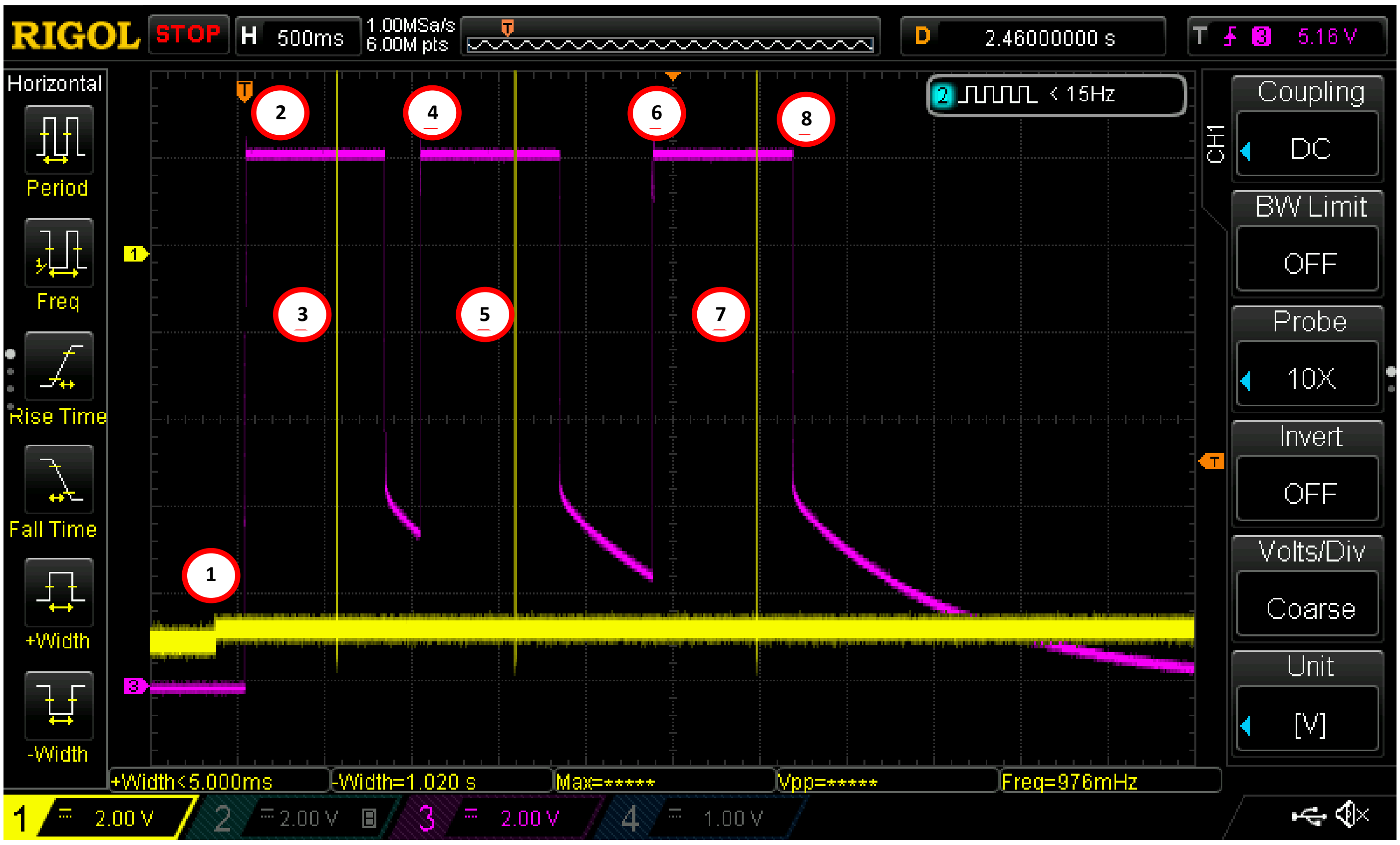
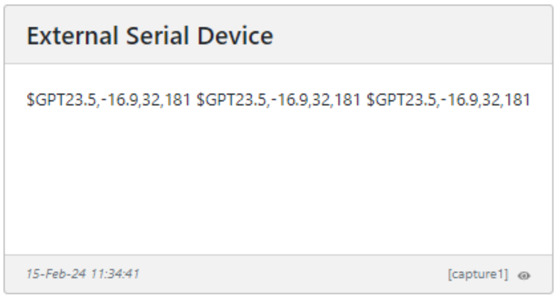


*Figure 30 - Pulse and Serial Timing*

Figure 31 shows that the Senquip ORB has captured three serial messages that will be simple to parse in the script, using the built in parse function.



External Serial Device

```
$GPT23.5,-16.9,32,181 $GPT23.5,-16.9,32,181 $GPT23.5,-16.9,32,181
```

15-Feb-24 11:34:41 [capture1]

*Figure 31 - Serial Data Captured by Senquip ORB*

We have demonstrated that a simple script in conjunction with settlings can allow a connected sensor to powered in a complex way.

## 8. Conclusions

Serial devices with diverse power and timing needs can be accommodated using a combination of settings and scripts.

The user needs to be aware that timer timeouts are not always precise, and expiration can be delayed if the processor is completing high priority tasks.

An oscilloscope will be of immense help when writing and debugging sensors with unusual power and timing requirements.

## 9. Appendix A – Multiple Delayed Modbus Write

```
load('senquip.js');
load('api_timer.js');
load('api_serial.js');
load('api_config.js');
load('api_sys.js');

let writeAddr = 1; // first modbus sensor address is 1

let chans = Cfg.get('script.num1'); // get the number of sensors from a custom variable

function sendVal(sendObj){
    let s = SQ.encode(sendObj.sadr,SQ.U8); // encode dec address into hex
    let r = SQ.encode(sendObj.radr,SQ.U16); // encode dec register number into hex
    let v = SQ.encode(sendObj.val,SQ.U16); // encode dec data into hex
    let a = s+'\x06'+r+v;  // 6 is the MODBUS write unsigned 16 function code
    let c = SQ.crc(a);  // use the Senquip CRC function to calculate the Modbus CRC
    c = SQ.encode(c, -SQ.U16); // encode the CRC function in hex + flip byte order
    let t = a+c;  // create the final Modbus write message
    SERIAL.write(1,t,t.length,SERIAL.IMMEDIATE);  // send the message to serial port 1
}

function writeNext(){ // write to the next Modbus sensor
  sendVal({sadr:writeAddr, radr:99, val:55});
  writeAddr++;
  if(writeAddr <= chans){ // check if there are more sensors to write to
    Timer.set(200, 0, function() { // set a timer with a 200msec timeout
      writeNext();
    }, null);
  }
}

Timer.set(3000, 0, function() { // initiate the first write after 3 sec
  writeNext();
 }, null);

SQ.set_data_handler(function(data) { // nothing to do in the data handler

}, null);
```

## 10. Appendix B - Senquip Device Waking Periodically and Taking Multiple Samples

```
load('senquip.js');
load('api_timer.js');

let interval = 1000; // msec - time between reads
let offdelay = 200; // msec - power down time
let reads = 3; // number of reads to perform
let count = 1; // current read no

function nextRead(){
  SQ.set_current(1, SQ.ON); // sensor powered up
  Timer.set(interval-offdelay, 0, function() {SQ.set_current(1, SQ.OFF);}, null);//
power down sensor
  count++;
  if (count <= reads){
    Timer.set(interval, 0, function() {nextRead();}, null);
    }
}

Timer.set(1000, 0, function() {nextRead();}, null);

SQ.set_data_handler(function(data) {

  // Parse serial data here

}, null);
```