MODBUS INTEGRATION WITH A DEEP SEA CONTROLLER

1. Introduction

Deep Sea Electronics is one of the world's top manufacturers of generator controllers, auto transfer switch controllers, battery chargers, and vehicle & off-highway controllers. The DSE8610 MKII represents the latest in complex load sharing & synchronising control technology and is designed to handle the most complex grid type generator applications.

The DSE8610 offers a Mobus RTU interface over RS485 through which the operation of the controller can be monitored and controlled by a Senquip telemetry device. This application note describes the process of connecting a Senquip ORB or QUAD as a Modbus master to the Deep Sea DSE8610 auto start control module as a slave.

In this application note, a single Deep Sea controller has been connected to a Senquip ORB. However, RS485 supports up to 32 connected devices, so a single Senquip device could be connected to multiple controllers.

In this application note, we will show how to connect to a DSE8610 controller and will then write a script to implement a remote start, stop function.

Figure 1 – DSE8610 Engine Controller

Disclaimer: The information provided in this application note is intended for informational purposes only. Users of the remote machine control system described herein should exercise caution and adhere to all relevant safety guidelines and regulations. By utilising the information provided in this application note, users acknowledge their understanding and acceptance of the associated risks. The authors and contributors disclaim any warranties, expressed or implied, regarding the accuracy or completeness of the information presented.

2. Wiring the Senquip Device to the Deep Sea Controller

In this application note, we will use an ORB-C1-G wired to RS485 Port 1 on the Deep Sea controller.

The following connections are required:

Connection	Senquip ORB	Deep Sea DSE8610
RS485 B	Pin 6, B	Pin 72, B
RS485 A	Pin 7, A	Pin 73, A
GND	Pin 4, GND	Pin 71, SCR

If the Senquip device and Deep Sea controller 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.

Since the Senquip device and Deep Sea controller share a common power supply ground, the ground connection between pins 4 and 71 is not required. If a screened wire is available, it should be connected to either the Senquip or Deep Sea controller ground but not both. Connecting to both can create a ground loop which will be susceptible to magnetic fields.

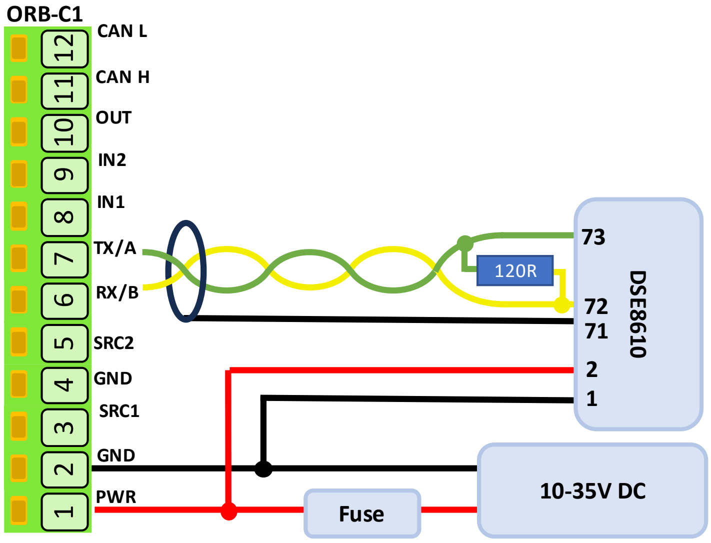


Figure 2 - Senquip ORB to DSE8610 Wiring

3. Senquip Device Configuration

We get the communications specification for the Deep Sea controller from the Deep Sea GenComm Communications Protocol manual. GenComm was devised by Deep Sea Electronics Plc to provide a uniform standard for communicating with any generating set control equipment. It allows all telemetry information relevant to a generating set to be read from the control equipment, regardless of manufacturer or specification.

Parameter	Value
Baud rate	115200
Data bits	8
Parity	None
Stop bits	1

The *Master inactivity timeout* on the Deep Sea controller should be set to at least twice the value of the Senquip base interval. For example, if the Senquip device reads from the controller every 5 seconds, the timeout should be set to at least 10 seconds.

The Senquip ORB serial port is set to match the Deep Sea controller requirements:

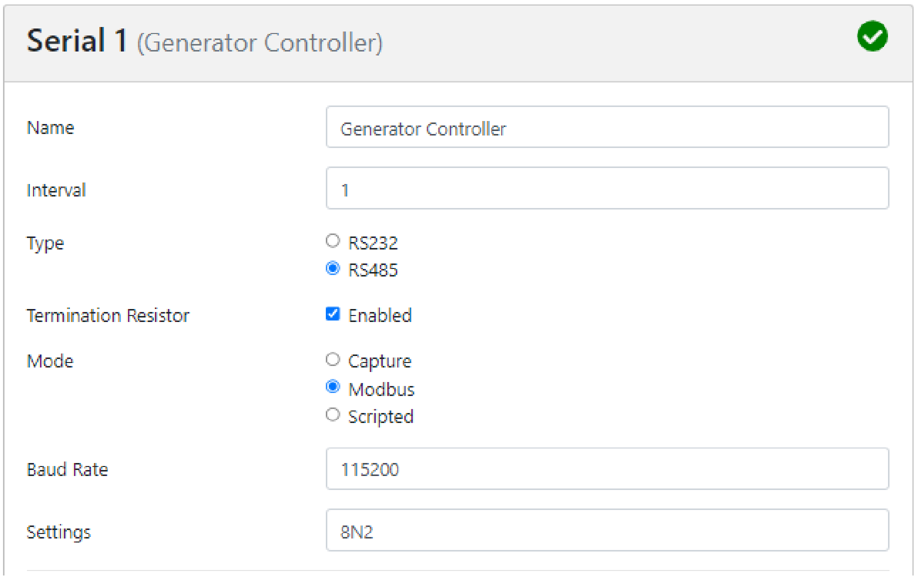


Figure 3 - Senquip ORB serial Port Settings

The default Modbus slave id for the controller is 10. This can be changed, for instance when multiple controllers are to be connected to a single Senquip device.

Registers are divided into 256 pages each containing up to 256 registers, the actual register address is obtained from the formula: register_address =page_number*256+register_offset. Available register pages are given in Appendix A.

Note that the register addresses can change per model.

We will read the following registers from the Deep Sea controller:

Register	Page	Offset	Address	Scaling	Unit	Type
Oil pressure	4	0	1024	1	Kpa	16 bit unsigned
Coolant temperature	4	1	1025	1	°C	16 bit signed
Battery voltage	4	5	1029	0.1	V	16 bit unsigned
Stop led	190	14	48654	1		16 bit unsigned
Gen available led	190	21	48661	1		16 bit unsigned
Gen breaker led	190	19	48659	1		16 bit unsigned
Control mode	3	4	772	1		16 bit unsigned

The meaning of the allowable values for control more are given below:

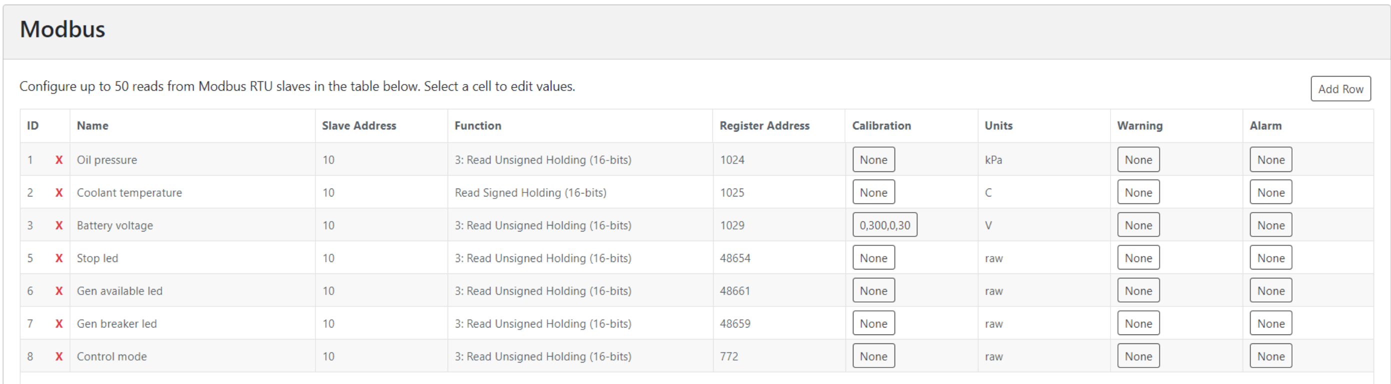
Mode	Description
0	Stop mode
1	Auto mode
2	Manual mode
3	Test on load mode
4	Auto with manual restore mode/Prohibit Return
5	User configuration mode
6	Test off load mode
7	Off Mode
8-65534	Reserved
65535	Unimplemented

Stop mode means stop the engine (generator) and in the case of 'automatic mains failure units' transfer the load to the mains if possible.

Auto mode means automatically start the engine (generator) in the event of a remote start signal or a mains-failure, and in the case of 'automatic mains failure units' transfer the load to the generator when available. When the remote start signal is removed or the mains returns, stop the engine (generator) and in the case of 'automatic mains failure units' transfer the load back to the mains.

Manual mode means start the engine (generator). With some control units it will also be necessary to press the start button before such a manual start is initiated. In the case of 'automatic mains failure units' do not transfer the load to the generator unless the mains fails.

Modbus reads are configured on the Senquip device and are shown in Figure 4. Note the calibration applied to the battery voltage, 0-300 in gives 0-30 out, or 0.1 as specified in the Deep Sea register specification.



Modbus

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

Add Row

ID		Name	Slave Address	Function	Register Address	Calibration	Units	Warning	Alarm
1	X	Oil pressure	10	3: Read Unsigned Holding (16-bits)	1024	None	kPa	None	None
2	X	Coolant temperature	10	Read Signed Holding (16-bits)	1025	None	C	None	None
3	X	Battery voltage	10	3: Read Unsigned Holding (16-bits)	1029	0,300,0,30	V	None	None
5	X	Stop led	10	3: Read Unsigned Holding (16-bits)	48654	None	raw	None	None
6	X	Gen available led	10	3: Read Unsigned Holding (16-bits)	48661	None	raw	None	None
7	X	Gen breaker led	10	3: Read Unsigned Holding (16-bits)	48659	None	raw	None	None
8	X	Control mode	10	3: Read Unsigned Holding (16-bits)	772	None	raw	None	None

Figure 4 - Senquip Device Modbus Settings

4. Implementing Start, Stop Control in a Script

We will now write a script to enable remote starting and stopping of a genset using the DSE8610 controller. The full script is available in Appendix B. It is assumed that the reader has scripting access, and that they have a fair knowledge of the Senquip scripting language. Further details on the Senquip scripting language can be found in the Senquip Scripting Guide.

We will use trigger buttons on the Senquip Portal to implement functions to put the generator in manual mode, start the generator, synchronise, and stop the generator. We create 4 *Trigger Parameters* on the device scripting page. We have named the triggers Manual, Start, Synchronise, and Stop, and have made them yellow, green, blue, and red.

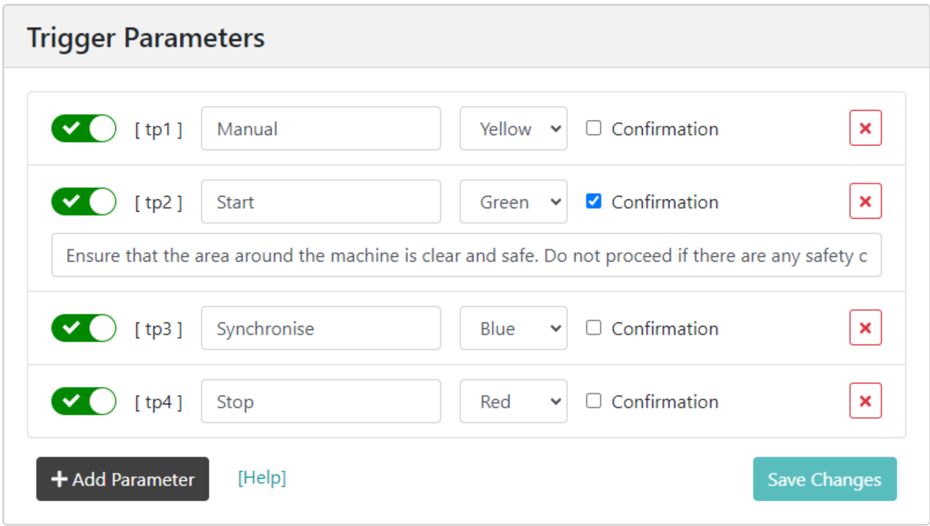


Figure 5 - Creating the Start and Stop Trigger Buttons

Note the confirmation message that will appear when a user activates that start button.

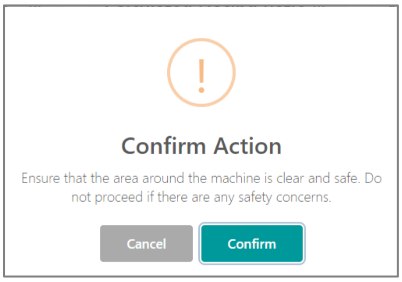


Figure 6 - Example Trigger Button Confirmation Message

The order of operations implemented in the script will be as shown in Figure 7.

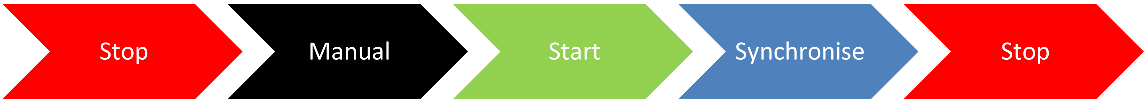


Figure 7 - Order of Operations

A common requirement for monitoring applications is knowing what ‘state’ a machine is in. The concept of machine states is built into Senquip devices. From the current measurement data, a script can work out what state the machine is in and record it accordingly using a single function call. From state information, the Senquip device can automatically calculate utilisation.

We will configure the following states on the device scripting page.

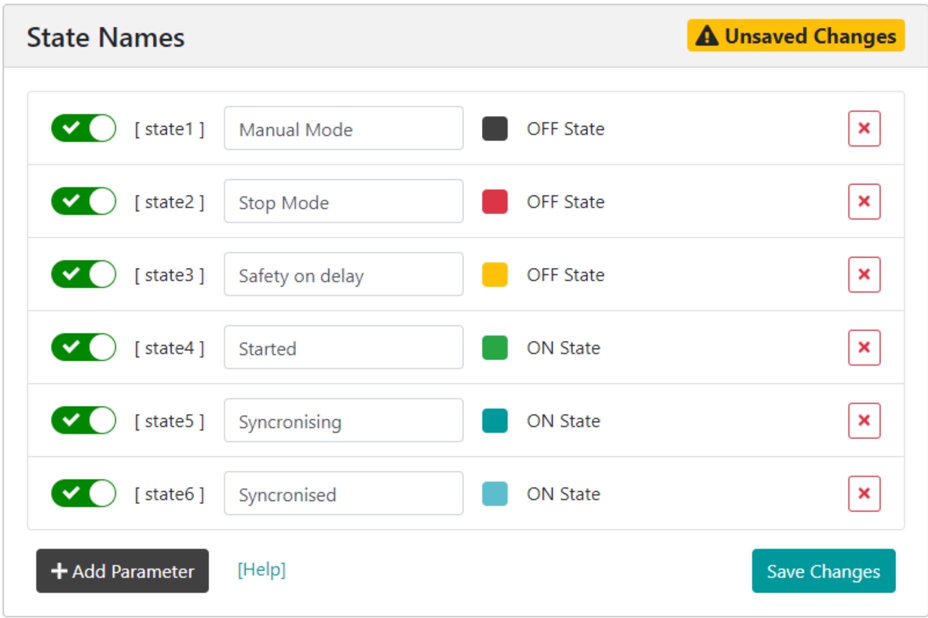


Figure 8 - Generator States

First, we load the required libraries and create some global variables to store the state of the machine. Variables declared outside of a function have scope in all functions and are used to store state between measurement cycles. The following global variables are used:

Variable	Function
state	Holds current state of the machine (manual, stop, safety, started, synchronising, syncronised)
Gen_available	Indicates if the generator is available to supply power i.e. it is syncronised
Gen_breaker	Indicates if the generator is currently supplying power or not
Control_mode	Indicates the current generator mode (stop, auto, or manual)

We also declare constant manual, stop, safety, started, synchronising, and syncronised states to make the code easier to read.

```
1 load('senquip.js');
2 load('api_serial.js');
3 load('api_math.js');
4 load('api_config.js');
5 load('api_timer.js');
6
7 let state = 0; // variable to indicate the current state of the machine
8 let Gen_available = 0; // modbus 6, is the generator available to provide power indicator
9 let Gen_breaker = 0; // modbus 7, breaker state, open or closed
10 let Control_mode = 0; // modbus 8, control mode --- 0  is stop --- 1 is auto --- 2 is manual
11
12 let MANUAL = 1;
13 let STOP = 2;
14 let SAFETY = 3;
15 let STARTED = 4;
16 let SYNC = 5;
17 let SYNCD = 6;
```

The main data handler is called after the Senquip device completes all measurement tasks. We check if the current state and the latest set of Modbus reads are valid before attempting to use them further in the script.

```
19 SQ.set_data_handler(function(data) {
20   let obj = JSON.parse(data);
21
22   if (typeof obj.state === "number"){state = obj.state;} // read the current state
23   if ((typeof obj.mod6 === "number") && (typeof obj.mod7 === "number") && (typeof obj.mod8 === "number")) { // clean modbus read
24     Gen_available = obj.mod6;
25     Gen_breaker = obj.mod7;
26     Control_mode = obj.mod8;
```

In line 20, we first parse the JSON data file that is passed to the data handler to create a structure that contains all the data measured in the last measurement cycle. Based on the latest set of Modbus reads, we update the current state variable to reflect the status of the generator. In line 36, we update the Senquip device state so that utilisation can be calculated automatically by the device. The following table describes the state allocation. Note that there are other states not considered in this application that must be handled in a final solution.

Control_Mode	Gen_available	Gen_Breaker	Current State	New State
STOP	X	X	X	STOP
MANUAL	No	No	X	MANUAL
MANUAL	Yes	No	X	STARTED
MANUAL	Yes	Yes	X	SYNCD
MANUAL	Yes	No	SAFETY	STARTED
MANUAL	Yes	Yes	SYNC	SYNCD

```
    if(Control_mode === 0) {state = STOP;} // if the generator is in stop mode, set state to stop mode
    else if(Control_mode === 2 && Gen_available === 0 && Gen_breaker === 0) {state = MANUAL;} // if in manual mode, unavailable, and the breaker is off, set to manual state
    else if(Control_mode === 2 && Gen_available === 1 && Gen_breaker === 0) {state = STARTED;} // if in manual mode, available, and the breaker is off, set to started state
    else if(Control_mode === 2 && Gen_available === 1 && Gen_breaker === 1) {state = SYNCD;} // if in manual mode, available, and the breaker is on, set to synchronised state

    if (obj.state === SAFETY && Gen_available === 1 && Gen_breaker === 0 && Control_mode === 2) {state = STARTED;} // if in safety delay and available, set to started state
    if (obj.state === SYNC && Gen_available  === 1 && Gen_breaker === 1 && Control_mode === 2) {state = SYNCD;} // if synchronising and breaker closed, set to synced state

    SQ.set_state(state); // set the new state
```

Once the current state reflects that of the generator, we dispatch an info event to the Senquip Portal to keep the user informed. Note the else in line 46 which is dispatched as a warning if all the Modbus registers were not correctly read. In a final solution, consideration should be given to what other actions to take. This ends the data handler function.

```
38      if (state === 0){SQ.dispatch_event(1,SQ.INFO,"Controller in undefined state");}
39      else if (state === MANUAL){SQ.dispatch_event(1,SQ.INFO,"Controller in manual mode");}
40      else if (state === STOP){SQ.dispatch_event(1,SQ.INFO,"Controller in stop mode");}
41      else if (state === SAFETY){SQ.dispatch_event(1,SQ.INFO,"Controller in safety delay");}
42      else if (state === STARTED){SQ.dispatch_event(1,SQ.INFO,"Controller in start state");}
43      else if (state === SYNC){SQ.dispatch_event(1,SQ.INFO,"Controller sychronising");}
44      else if (state === SYNCD){SQ.dispatch_event(1,SQ.INFO,"Controller synchronised");}
45    }
46    else {SQ.dispatch_event(1,SQ.WARNING,"Controller Modbus read Fault");} // we didn't get a clean Modbus read
47
48  }, null);
```

We will now look at the trigger functions that are used to change the state of the Deep Sea controller.

The first trigger function requests manual mode if the controller is currently in stop mode. To change the controller mode, we send a write Modbus command to register 4104 = page 16, register 8. Register 8 must be written with a specific code “System Control Key” to change the controller state.

A summary of system control keys is given below. Further keys can be found in the GenComm standard. Function codes 0 to 31 perform exactly the same function as pressing the equivalent button on the control unit.

Function code	System Control Function	System Control Key
0	Select Stop mode	35700
1	Select Auto mode	35701
2	Select Manual mode	35702
3	Select Test on load mode	35703
4	Select Auto with manual restore mode	35704
5	Start engine if in manual or test modes	35705
6	Mute alarm	35706
7	Reset alarms	35707
8	Transfer to generator	35708

A safety mechanism is built into the Deep Sea controller to protect from inadvertant writes that may change state. To execute a write, one of the system control keys must be written into register 8 and its ones-compliment value written into register 9 with a single write function. Writing any other value or using a function that is not available will return extended exception code 7 (Illegal value written to register) and have no affect. A production script should read and take action based on the return.

To write to multiple Modbus registers, function 16 (write multiple registers) is used.

We will look at an example write to set the control mode to request manual mode. The System Control Key to request manual mode is:

- Decimal: 35702,
- Hexadecimal: 0x8B76 hex,
- Binary: 0b1000 1011 0111 0110.

The ones complement value is:

- Binary: 0b0111 0100 1000 1001,
- Hexadecimal: 0x7498.

The table below describes the required Modbus write message.

Byte	Value	Meaning
1	0x0A	The Deep Sea Controller slave address
2	0x10	Function code 16, write multiple registers
3	0x10	MSB of register address 4104
4	0x08	LSB of register address 4104 (0x1008 = 4104)
5	0x00	MSB of number of registers to write
6	0x02	LSB of number of registers to write (0x0002 = 2 x 16-bit registers)
7	0x04	Number of bytes to follow 2 x 16-bit registers equals 4 bytes
8	0x8b	MSB of System Function Code to write into register 4104
9	0x76	LSB of System Function Code to write into register 4104 (0x8b76 = 35702)
10	0x74	MSB of the ones compliment System Function Code
11	0x89	LSB of the ones compliment System Function Code
12	TBD	MSB of Modbus checksum as calculated by the Senquip SQ.crc function
13	TBD	LSB of Modbus checksum as calculated by the Senquip SQ.crc function

The trigger function to request manual mode is given below. The SQ.crc function is used to create the modbus crc. A SQ.encode function encodes the crc number into hexadecimal ASCII, with the MSB encoded first. The SQ.write function is used to send the serial message to port 1. Also coded are some responses if manual mode is requested, and the generator is already in manual mode, is started, or is already synchronised.

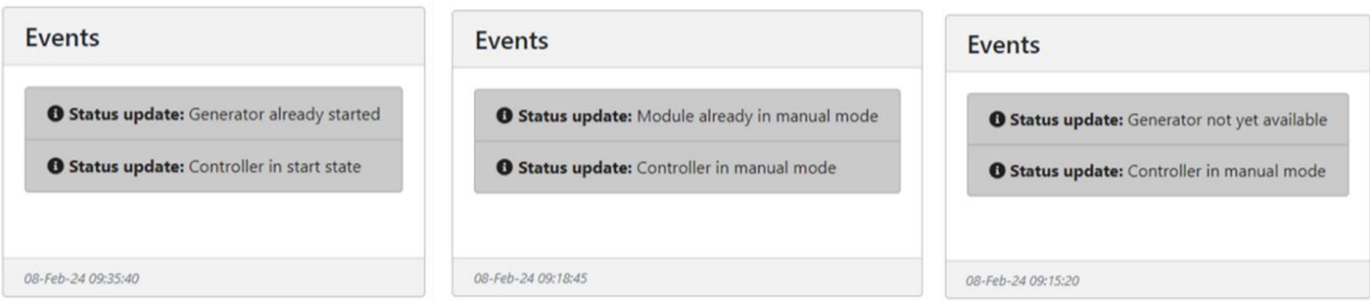


Figure 9 - Example Trigger Button Responses

```
SQ.set_trigger_handler(function(tp) {
  if (tp === 1 ) { // Request manual mode
    if(Control_mode === 0 && Gen_available === 0 && Gen_breaker === 0){ // if stopped and unavailable to supply power and the breaker is open
      // Create a Modbus command
      // \x0A = Slave address 10
      // \x10 = Modbus Function 16 (Write Multiple Holding Register)
      // \x10\x08 = Register Address 4104
      // \x00\x02 = Number of registers to write
      // \x04 = Number of bytes to follow
      // \x8b\x76 = Value to write to register 1 = 35702
      // \x74\x89 = Value to write to register 2 = 1's compliment of above
      let cmd_str = "\x0A\x10\x10\x08\x00\x02\x04\x8B\x76\x74\x89";
      let crc = SQ.crc(cmd_str);
      let crc_str = SQ.encode(crc, -SQ.U16);
      let modbus_str = cmd_str + crc_str;
      SERIAL.write(1, modbus_str, modbus_str.length);
    }
    else {
      if(Control_mode === 2 && Gen_available === 0 && Gen_breaker === 0) {SQ.dispatch_event(1,SQ.INFO,"Module already in manual mode");}
      else if(Control_mode === 2 && Gen_available === 1 && Gen_breaker === 0) {SQ.dispatch_event(1,SQ.INFO,"Generator already started");}
      else if(Control_mode === 2 && Gen_available === 1 && Gen_breaker === 1) {SQ.dispatch_event(1,SQ.INFO,"Generator already synchronised");}
    }
  }
```

Trigger functions to request start mode, synchronise, and stop mode are also provided. They are similar to the manual mode request and are not discussed further.

5. Conclusions

The Senquip scripting language makes it simple to interface into Deep Sea generator controllers like the DSE8610. Most Deep Sea controllers use the GenComm standard for Modbus communication and so the application note is applicable to many other models of controller.

In addition to data received from the Deep Sea controller, additional parameters such as location, battery voltage, pitch, roll, and vibration can be added using sensors integrated into the Senquip device. Other sensors can be added to measure oil quality, tamper and more.

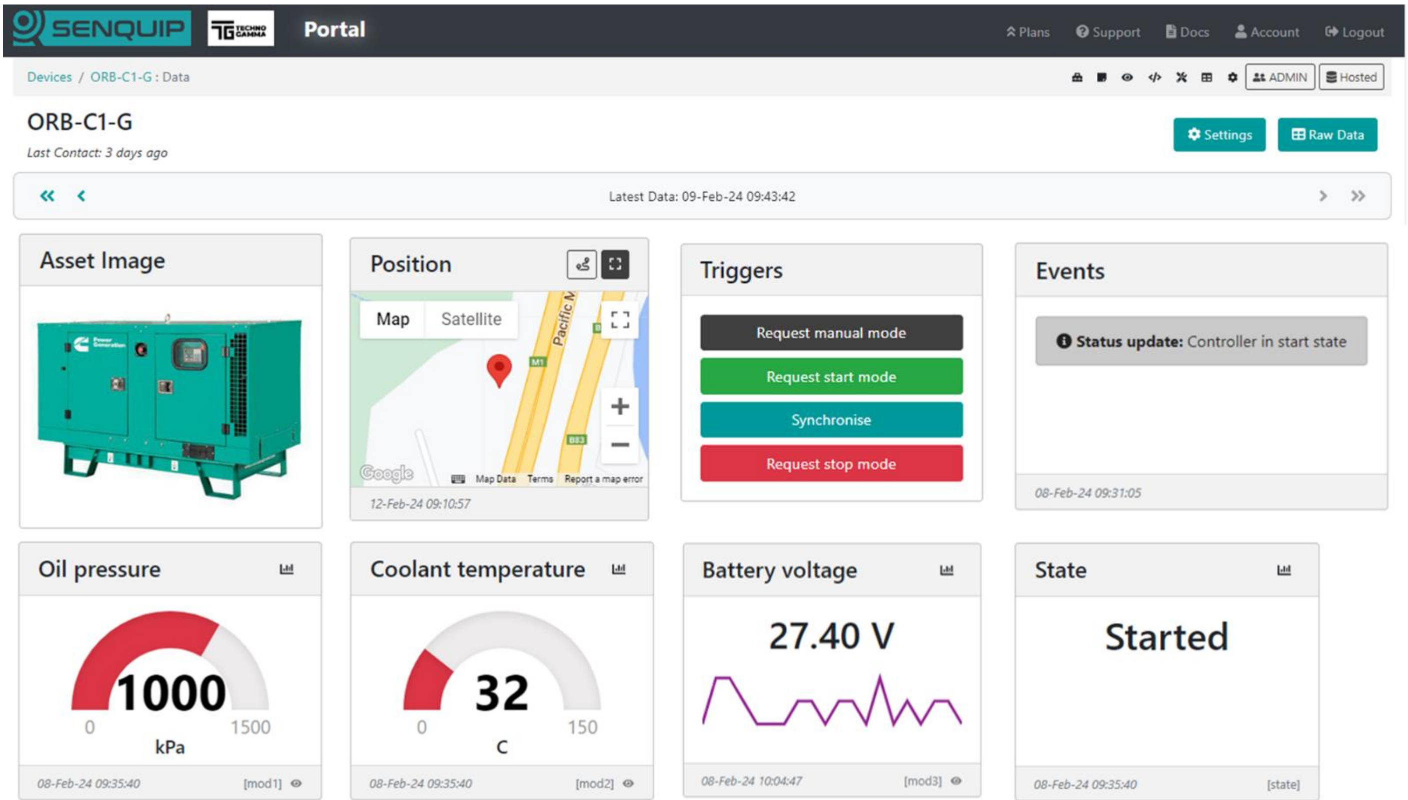


Figure 10 - Typical Portal Display with Minimal Parameters Shown

6. Appendix A – System Function Keys

Page number	Description	Read/write
0	Communications status information	Read only
1	Communications configuration	Read/write and write only
2	Modem configuration	Read/write
3	Generating set status information	Read only
4	Basic instrumentation	Read only
5	Extended instrumentation	Read only
6	Derived Instrumentation	Read only
7	Accumulated Instrumentation	Read/write
8	Alarm conditions	Read only
9	Total Harmonic Distortion information	Read only
10	Reserved	
11	Diagnostic – general	Read only
12	Diagnostic – digital inputs	Read only
13	Diagnostic – digital outputs	Read only and read write
14	Diagnostic – LEDs	Read only and read write
15	Diagnostic – Reserved	
16	Control registers	Read only and write only
17	J1939 active diagnostic trouble codes in decoded format	Read only
18	J1939 active diagnostic trouble codes in raw format	Read only
19	Reserved	
20	Various strings	Read only
24	Identity strings	Read/write
26	State machine name strings	Read only
28	State machine state strings	Read only
29-31	Reserved	
32-95	Alarm strings (Old alarm system)	Read only
32-36	2131 Expansion module name strings	Read only
37-40	2133 Expansion module name strings	Read only
41-43	2152 Expansion module name strings	Read only
44-48	2131 Expansion module digital alarm strings	Read only
49-58	2131 Expansion module analogue alarm strings	Read only
59-66	2133 Expansion module analogue alarm strings	Read only
142	ECU Trouble Codes	Read only
143-149	ECU Trouble Code short description string	Read only
152	User calibration of expansion module analogue inputs	Read/write
153	Unnamed alarm conditions	Read only
154	Named Alarm Conditions	Read only
156	Expansion module enable status	Read only
158	Expansion module communications status	Read only
160	Unnamed input function	Read only
166-169	User configurable pages	Read only
170	Unnamed input status	Read only

171	Unnamed input status continued	Read only
180	Unnamed output sources & polarities	Read only
181	Unnamed output sources & polarities continued	Read only
182	Virtual output sources & polarities	Read only
183	Configurable output sources & polarities	Read only
184	Analogue output sources, types and values	Read only
190	Unnamed output status	Read only
191	Virtual output status	Read only
192	Configurable output status	Read only
193	Remote control sources	Read/write
200-239	Unnamed alarm strings	Read only
240-246	Analogue Input Name Strings	Read only
250	Misc strings	Read only
251-255	Reserved	

7. Appendix B – Full Application Script

```
load("senquip.js");
load("api_serial.js");
load("api_math.js");
load("api_config.js");
load("api_timer.js");

let state = 0; // variable to indicate the current state of the machine
let Gen_available = 0; // modbus 6, is the generator available to provide power indicator
let Gen_breaker = 0; // modbus 7, breaker state, open or closed
let Control_mode = 0; // modbus 8, control mode --- 0  is stop --- 1 is auto --- 2 is manual

let MANUAL = 1;
let STOP = 2;
let SAFETY = 3;
let STARTED = 4;
let SYNC = 5;
let SYNCD = 6;

SQ.set_data_handler(function (data) {
  let obj = JSON.parse(data);

  if (typeof obj.state === "number") {
    state = obj.state;
  } // read the current state
  if (
    typeof obj.mod6 === "number" &&
    typeof obj.mod7 === "number" &&
    typeof obj.mod8 === "number"
  ) {
    // clean modbus read
    Gen_available = obj.mod6;
    Gen_breaker = obj.mod7;
    Control_mode = obj.mod8;

    if (Control_mode === 0) {
      state = STOP;
    } // if the generator is in stop mode, set state to stop mode
    else if (Control_mode === 2 && Gen_available === 0 && Gen_breaker === 0) {
      state = MANUAL;
    } // if in manual mode, unavailable, and the breaker is off, set to manual state
    else if (Control_mode === 2 && Gen_available === 1 && Gen_breaker === 0) {
      state = STARTED;
    } // if in manual mode, available, and the breaker is off, set to started state
    else if (Control_mode === 2 && Gen_available === 1 && Gen_breaker === 1) {
      state = SYNCD;
    } // if in manual mode, available, and the breaker is on, set to synchronised state

    if (
      obj.state === SAFETY &&
      Gen_available === 1 &&
      Gen_breaker === 0 &&
      Control_mode === 2
    ) {
      state = STARTED;
    } // if in safety delay and available, set to started state
    if (
      obj.state === SYNC &&
      Gen_available === 1 &&
      Gen_breaker === 1 &&
      Control_mode === 2
    ) {
      state = SYNCD;
```

```
    } // if synchronising and breaker closed, set to synced state

    SQ.set_state(state); // set the new state

    if (state === 0) {
      SQ.dispatch_event(1, SQ.INFO, "Controller in undefined state");
    } else if (state === MANUAL) {
      SQ.dispatch_event(1, SQ.INFO, "Controller in manual mode");
    } else if (state === STOP) {
      SQ.dispatch_event(1, SQ.INFO, "Controller in stop mode");
    } else if (state === SAFETY) {
      SQ.dispatch_event(1, SQ.INFO, "Controller in safety delay");
    } else if (state === STARTED) {
      SQ.dispatch_event(1, SQ.INFO, "Controller in start state");
    } else if (state === SYNC) {
      SQ.dispatch_event(1, SQ.INFO, "Controller sychronising");
    } else if (state === SYNCD) {
      SQ.dispatch_event(1, SQ.INFO, "Controller synchronised");
    }
  } else {
    SQ.dispatch_event(1, SQ.WARNING, "Controller Modbus read Fault");
  } // we didn't get a clean Modbus read
}, null);

SQ.set_trigger_handler(function (tp) {
  if (tp === 1) {
    // Request manual mode
    if (Control_mode === 0 && Gen_available === 0 && Gen_breaker === 0) {
      // if stopped and unavailable to supply power and the breaker is open
      // Create a Modbus command
      // \x0A = Slave address 10
      // \x10 = Modbus Function 16 (Write Multiple Holding Register)
      // \x10\x08 = Register Address 4104
      // \x00\x02 = Number of registers to write
      // \x04 = Number of bytes to follow
      // \x8b\x76 = Value to write to register 1 = 35702
      // \x74\x89 = Value to write to register 2 = 1's compliment of above
      let cmd_str = "\x0A\x10\x10\x08\x00\x02\x04\x8B\x76\x74\x89";
      let crc = SQ.crc(cmd_str);
      let crc_str = SQ.encode(crc, -SQ.U16);
      let modbus_str = cmd_str + crc_str;
      SERIAL.write(1, modbus_str, modbus_str.length);
    } else {
      if (Control_mode === 2 && Gen_available === 0 && Gen_breaker === 0) {
        SQ.dispatch_event(1, SQ.INFO, "Module already in manual mode");
      } else if (
        Control_mode === 2 &&
        Gen_available === 1 &&
        Gen_breaker === 0
      ) {
        SQ.dispatch_event(1, SQ.INFO, "Generator already started");
      } else if (
        Control_mode === 2 &&
        Gen_available === 1 &&
        Gen_breaker === 1
      ) {
        SQ.dispatch_event(1, SQ.INFO, "Generator already synchronised");
      }
    }
  }

  if (tp === 2) {
    // Request start mode
    if (Control_mode === 2 && Gen_available === 0 && Gen_breaker === 0) {
      // if in manual and unavailable to supply power and the breaker is open
      let cmd_str = "\x0A\x10\x10\x08\x00\x02\x04\x8B\x79\x74\x86"; // system key code 35705
```

```
      let crc = SQ.crc(cmd_str);
      let crc_str = SQ.encode(crc, -SQ.U16);
      let modbus_str = cmd_str + crc_str;
      SERIAL.write(1, modbus_str, modbus_str.length);
      state = SAFETY;
    } else {
      if (Control_mode === 0) {
        SQ.dispatch_event(1, SQ.INFO, "Module not in manual mode");
      } else if (
        Control_mode === 2 &&
        Gen_available === 1 &&
        Gen_breaker === 0
      ) {
        SQ.dispatch_event(1, SQ.INFO, "Generator already started");
      } else if (
        Control_mode === 2 &&
        Gen_available === 1 &&
        Gen_breaker === 1
      ) {
        SQ.dispatch_event(1, SQ.INFO, "Generator already synchronised");
      }
    }
  }

  if (tp === 3) {
    // Request syncronise
    if (Control_mode === 2 && Gen_available === 1 && Gen_breaker === 0) {
      // if in manual and available to supply power and the breaker is open
      let cmd_str = "\x0A\x10\x10\x08\x00\x02\x04\x8B\x7C\x74\x83"; // system key code 35708
      let crc = SQ.crc(cmd_str);
      let crc_str = SQ.encode(crc, -SQ.U16);
      let modbus_str = cmd_str + crc_str;
      SERIAL.write(1, modbus_str, modbus_str.length);
      state = 5;
    } else {
      if (Control_mode === 0) {
        SQ.dispatch_event(1, SQ.INFO, "Module not in manual mode");
      } else if (Control_mode === 2 && Gen_available === 0) {
        SQ.dispatch_event(1, SQ.INFO, "Generator not yet available");
      } else if (
        Control_mode === 2 &&
        Gen_available === 1 &&
        Gen_breaker === 1
      ) {
        SQ.dispatch_event(1, SQ.INFO, "Generator already synchronised");
      }
    }
  }

  if (tp === 4) {
    // Request stop mode
    if (Control_mode === 2) {
      let cmd_str = "\x0A\x10\x10\x08\x00\x02\x04\x8B\x74\x74\x8B"; // system key code 35700
      let crc = SQ.crc(cmd_str);
      let crc_str = SQ.encode(crc, -SQ.U16);
      let modbus_str = cmd_str + crc_str;
      SERIAL.write(1, modbus_str, modbus_str.length);
      SQ.set_state(2);
    } else {
      if (Control_mode === 0) {
        SQ.dispatch_event(1, SQ.INFO, "Module already in stop mode");
      }
    }
  }
}, null);
```