INTEGRATION WITH A KENSHO CONTROLLER

1. Overview

Kensho offers a range of diesel engine controllers designed for off-road stationary applications such as irrigation, dewatering, power generation, and firefighting. These controllers are engineered to enhance engine performance, reduce operational costs, and provide robust protection and remote monitoring capabilities

By integrating a Senquip device with a Kensho control panel, users gain real-time visibility and control of engine operation, enabling proactive maintenance, faster response to faults, and improved operational efficiency.

Figure 1 – Kensho K27 Controller

This application note describes how to interface a Senquip telemetry device with a Kensho K27 controller via Modbus over RS485. By connecting to the RS485 network, the Senquip device enables both remote monitoring of engine parameters and remote start/stop control. This integration allows operators to engines in real time, improving asset utilisation, enabling proactive maintenance, and reducing the need for site visits, particularly in remote or unmanned locations. The concepts applied in this application note are also applicable to the K21, and K37 controller. The interface and available data may however vary.

The following sections provide guidance on wiring, configuration, and scripts required for the Senquip device.

***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 Kensho K27 Controller

In this application note, we will use the RS485 serial port on a Senquip ORB, wired to the telemetry port on connector C on the rear of the Kensho panel. RS485 is stated as the default for the telemetry port, however it may have been changed to RS232. If this is the case, it will need to be changed back, and the panel reset.

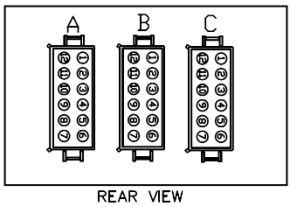


Figure 2 – Connector C on the Rear of the K27

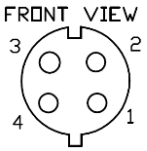


Figure 3 - Telemetry Connector Supplied with Wiring Loom

Some Kensho controllers may be supplied with a wiring loom that contains a Telemetry Ready Connector. The specifications of this connector are:

Brand	**Bulgin**
Series	Buccaneer
Contacts	4
Mounting	Cable mount
IP Rating	IP68
Connector on K27	PX07 47/S (socket)
Mating connector	PX07 48/P (plug)

Figure 4 - Bulgin Buccaneer Plug

The following connections are required:

Connection	**Senquip QUAD**	**K27 on Controller**	**K27 on Telemetry Connector**
RS485 A	Pin 7, A / TX	C7, COMMS RX (A)	3
RS485 B	Pin 6, B / RX	C8, COMMS Tx (B)	2
GND	Pin 2, GND	C9, BAT-	4
Switched PWR	Pin 1, PWR +	C1, SW BAT+	1

If a screened wire is available, it should be connected to either the Senquip or the Kensho controller ground but not both. Connecting to both can create a ground loop which will be susceptible to magnetic fields.

The K27 does not include the RS485 120 ohm termination resistor. If the cable length between the controller and Senquip device is long, or communications are unstable, one should be fitted. The termination resistor on the Senquip device is selected as a setting and will be turned on.

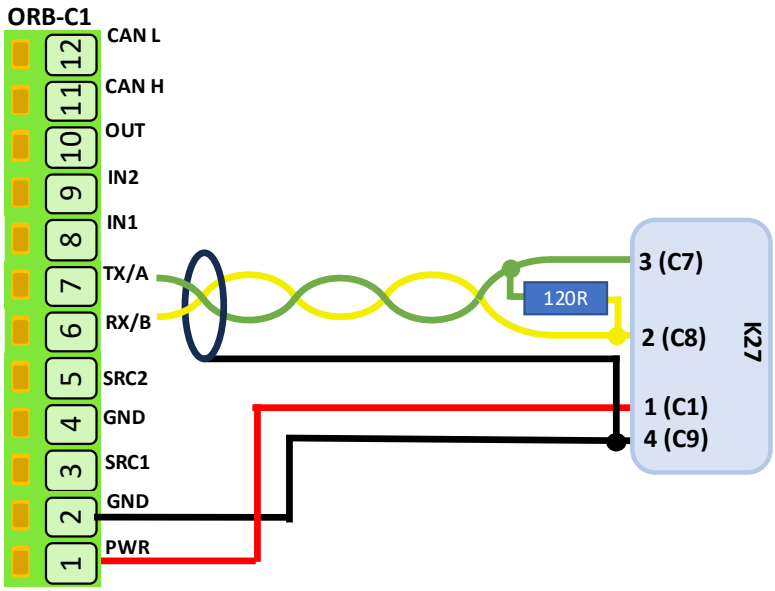


Figure 5 - Senquip ORB to LC40 Wiring

3. Senquip Device Configuration

We get the communications specification for the K270 controller from the K27 Engine Control Operation Manual. The manual contains panel setup information, and a Modbus register map.

The default serial port settings are found to be:

Parameter	Value
Interface	RS485
Bit rate	9600bps
Data bits	8
Stop bits	1
Parity	None
Protocol	Modbus RTU
Address	10

Note that if multiple Kensho controllers are on the same RS485 network, they will each need a different Modbus Address.

The serial port on the Senquip device is configured to mirror the K27 default settings:

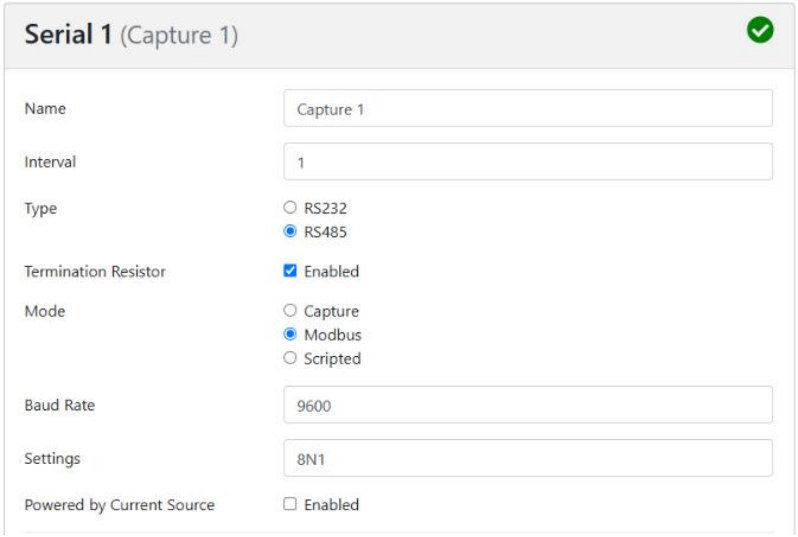


Figure 6 - Senquip Serial Port Settings

4. Reading Values from the Controller

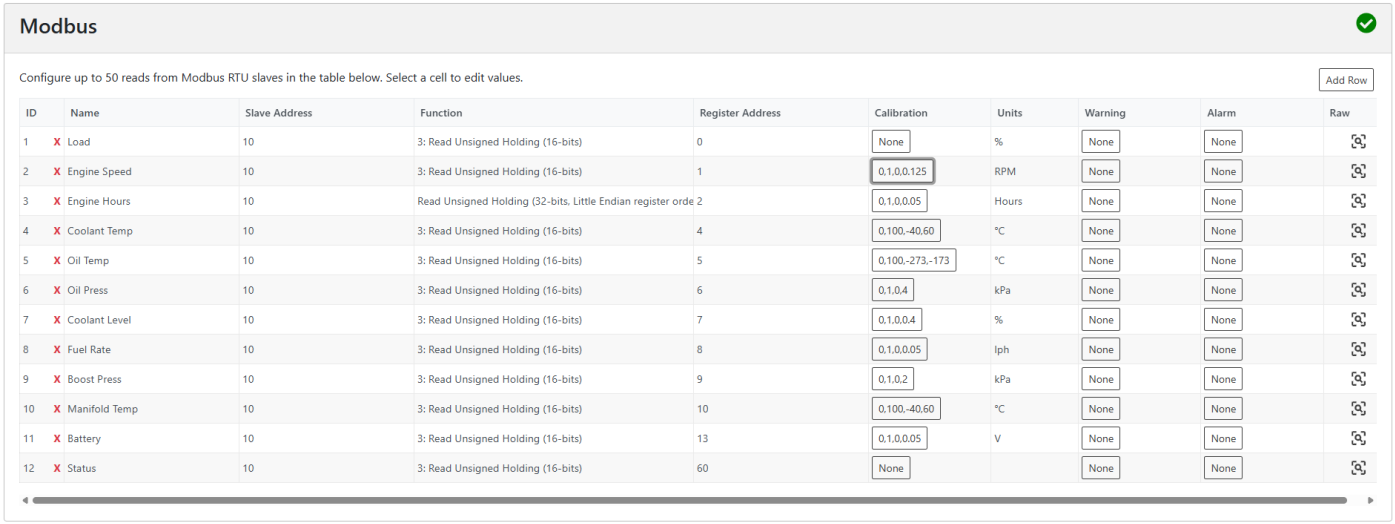
The K27 control panel communicates over Modbus RTU. A Modbus register map is available. All data is read as a 16 bit unsigned word. The Modbus map uses reference (Modicon) addressing where the leading "4" tells us that the registers are holding registers. The first register, 40001, will have address 0.

Modicon Address (K27)	Absolute Addressing (Senquip)
40001	0 (holding register)
40002	1 (holding register)

We will read the following 16-bit holding registers:

MODBUS	Source	Description	Resolution
40001	J1939	Percent Load	1%/bit
40002	J1939	Engine RPM	0.125/bit
40003	J1939	Total Engine Hours LSB	0.05Hrs/bit
40004	J1939	Total Engine Hours MSB	
40005	J1939	Engine Coolant Temp	1°C/bit (-40°C offset)
40006	J1939	Engine Oil Temp	0.03125°C/bit (-273°C offset)
40007	J1939	Engine Oil Pressure	4kPa/bit, 0.58015psi/bit
40008	J1939	Coolant Level	0.4%/bit
40009	J1939	Fuel Rate	0.05L/H/bit
40010	J1939	Boost Pressure	2kPa/bit, 0.29007psi/bit
40011	J1939	Intake Manifold Temp	1°C/bit (-40°C offset)
40013	J1939	Battery Potential	0.05V/bit
40061		Status / Fault Code	See Appendix A – K27 Fault Codes

The Senquip device Modbus map is configured accordingly:



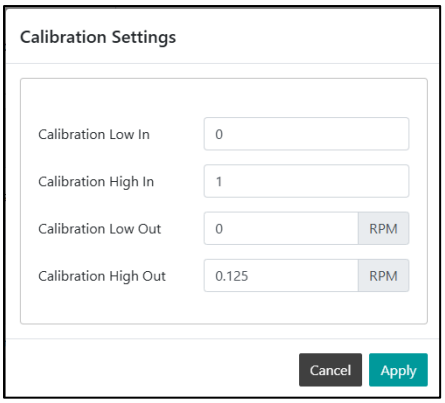
Modbus

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	Raw
1	X	Load	10	3: Read Unsigned Holding (16-bits)	0	None	%	None	None	
2	X	Engine Speed	10	3: Read Unsigned Holding (16-bits)	1	0,1,0,0.125	RPM	None	None	
3	X	Engine Hours	10	Read Unsigned Holding (32-bits, Little Endian register orde	2	0,1,0,0.05	Hours	None	None	
4	X	Coolant Temp	10	3: Read Unsigned Holding (16-bits)	4	0,100,-40,60	°C	None	None	
5	X	Oil Temp	10	3: Read Unsigned Holding (16-bits)	5	0,100,-273,-173	°C	None	None	
6	X	Oil Press	10	3: Read Unsigned Holding (16-bits)	6	0,1,0.4	kPa	None	None	
7	X	Coolant Level	10	3: Read Unsigned Holding (16-bits)	7	0,1,0,0.4	%	None	None	
8	X	Fuel Rate	10	3: Read Unsigned Holding (16-bits)	8	0,1,0,0.05	lph	None	None	
9	X	Boost Press	10	3: Read Unsigned Holding (16-bits)	9	0,1,0.2	kPa	None	None	
10	X	Manifold Temp	10	3: Read Unsigned Holding (16-bits)	10	0,100,-40,60	°C	None	None	
11	X	Battery	10	3: Read Unsigned Holding (16-bits)	13	0,1,0,0.05	V	None	None	
12	X	Status	10	3: Read Unsigned Holding (16-bits)	60	None		None	None	

Notice how calibration has been applied directly to the register read, where required. For example, Engine RPM is given as 0.125 per bit and is implemented as below. In this example, 0 bit will give 0 RPM and 1 bit will give 0.125 RPM.



Calibration Settings

Calibration Low In: 0

Calibration High In: 1

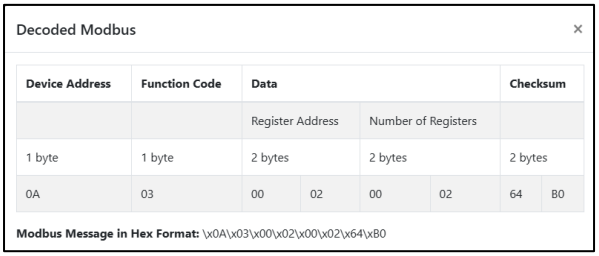
Calibration Low Out: 0 RPM

Calibration High Out: 0.125 RPM

Cancel Apply

Figure 7 - Engine Speed Calibration

The engine hours are read as a single 32 bit number using little Endian format where 40003, the lower word will arrive first and 40004, the higher word will arrive last.



Decoded Modbus

Device Address	Function Code	Data				Checksum	
		Register Address		Number of Registers			
1 byte	1 byte	2 bytes		2 bytes		2 bytes	
0A	03	00	02	00	02	64	B0

Modbus Message in Hex Format: \x0A\x03\x00\x02\x00\x02\x64\xB0

Figure 8 - Engine Hours Read as Two 16 Bit Words

Instead of displaying the controller status code, we will display the text description provided in Appendix A – K27 Fault Codes.

5. Remote Control

To ensure write commands are accepted by the controller, you must first change or increment the value in Configuration Register 40023. This mechanism prevents accidental or repeated actions.

For example, to issue a Start or Stop command via Modbus:

1. Set *Start Type* to Momentary in the controller menu:
   3. User Settings → 14. Start Types → 1. Set Start Types
2. Change or increment Register 40023.
   a. Provide enough time for a response
3. Write to Register 40020:
   a. 0xAA to start
   b. 0x55 to stop
4. Before issuing a new command, change or increment Register 40023 again.

We will write a script to perform start and stop functions. 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.

> Warning: Because of new features used in this script, please update your firmware version to SFW003-6.0.0 or later.

6. The Device Script

First, we load the required libraries and create a global variable that will be incremented and sent to the Configuration Register before issuing a command.

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

let configReg = 0; // value that will be incremented before each write
```

Displaying the status in a meaningful way requires that we interpret the Modbus status code in accordance with the text descriptions described in Appendix A – K27 Fault Codes. A function is created, which when called with the status code, returns the text description of that code. To do this, we use an object with the status code as key, and the descriptor as data. A shortened version of the function is shown below.

```
function statusLookup(i) {
  let enum_lookup = {
    '0': 'Normal Operation',
    '1': 'Low Oil Pressure',
    '2': 'High Engine Temp.',
    '3': 'Auxiliary 3',
    '4': 'Loss of Flow Sw.',
  };
  return enum_lookup[i] || 'Default';
}
```

The function is called once every base interval.

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

  if (typeof obj.mod12 === "number"){
      SQ.dispatch(1, statusLookup(obj.mod12)); // status as text
  }
}, null);
```

To perform engine starting and stopping, we need to perform Modbus writes. A function, writeMod, is created to write an unsigned 16-bit value to a Modbus holding register. The function accepts an object containing the slave address (sadr), register address (radr), and value (val) to be written.

It constructs a Modbus write message (function code 6) by encoding the slave address as a 1-byte hexadecimal string and both the register and value as 2-byte hexadecimal strings using the SQ.encode() function. These components are concatenated to form the base Modbus message.

A CRC checksum is then computed using the SQ.crc() function and encoded with byte order reversed (using -SQ.U16) to meet Modbus MSB-first requirements. This completes the full Modbus packet, which is transmitted via SERIAL.write() on serial port 1.

```
function writeMod(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
}
```

Adding Control Buttons

We will add two trigger buttons for start and stop. The start button will have an associated confirmation message that must be acknowledged “Safety Check: Area must be clear before remote start.”

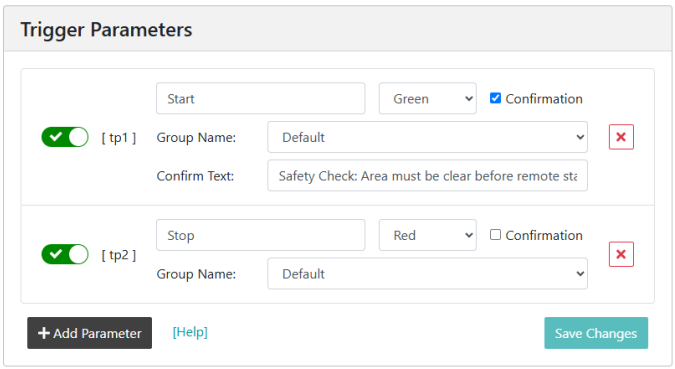


Figure 9 - Configuring Start and Stop Buttons

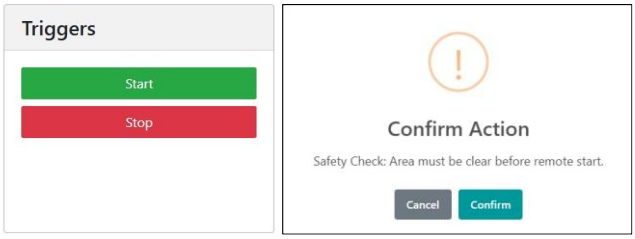


Figure 10 - Start and Stop Buttons on the Device Page and Confirm Message when Start is Pressed

When a trigger button is pressed, the next time the Senquip device contacts the Senquip Portal, the trigger will be retrieved, and the *trigger handler* will run. In the *trigger handler*, the trigger number is checked, and if active, the associated script will run on the Senquip device.

The *trigger handler* handles two trigger types:

- Trigger 1 (Start Command):
  Increments the configuration register. Logs an informational event ("Engine Starting") to the Senquip Portal. Writes the updated config value to Modbus register 22. After a one-second delay, it sends a start command (0xAA) to register 19 to initiate engine start.
- Trigger 2 (Stop Command):
  Similarly increments configReg and logs "Engine Stopping". It writes the updated value to register 22 and, after a one-second delay, sends a stop command (0x55) to register 19 to stop the engine.

Both sequences use writeMod to format and send Modbus messages, and Timer.set to insert a one-second delay between the configuration and command writes.

Warning: If the Senquip device goes to sleep before the 1 second timeout, the start or stop write will not execute. A low base interval like 5 second or using the Always On function will ensure the Senquip device stays

```
SQ.set_trigger_handler(function(tp) {
  if (tp === 1) { // start
    configReg++; if (configReg > 65535){configReg = 0;}
    SQ.dispatch_event(1,SQ.INFO,"Engine Starting");
    writeMod({sadr:10, radr:22, val:configReg}); // call the send Modbus routine
    Timer.set(1000, 0, function() { // After one second, send the 2nd serial string
      writeMod({sadr:10, radr:19, val:0xAA}); // send the start command
    }, null);
  }

  if (tp === 2) { // stop
    configReg++; if (configReg > 65535){configReg = 0;}
    SQ.dispatch_event(1,SQ.INFO,"Engine Stopping");
    writeMod({sadr:10, radr:22, val:configReg}); // call the send Modbus routine
    Timer.set(1000, 0, function() { // After one second, send the 2nd serial string
      writeMod({sadr:10, radr:19, val:0x55}); // send the stop command
    }, null);
  }
}, null);
```

7. Conclusions

The Senquip scripting language makes it simple to interface to a Kensho K27 pump controller. Most Kensho controllers use RS232 or RS485 and a similar Modbus register map, and so the application note is applicable to most other models of controller.

In addition to data received from the BBA 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.

Remote control is easily achieved with the implementation of a short script.

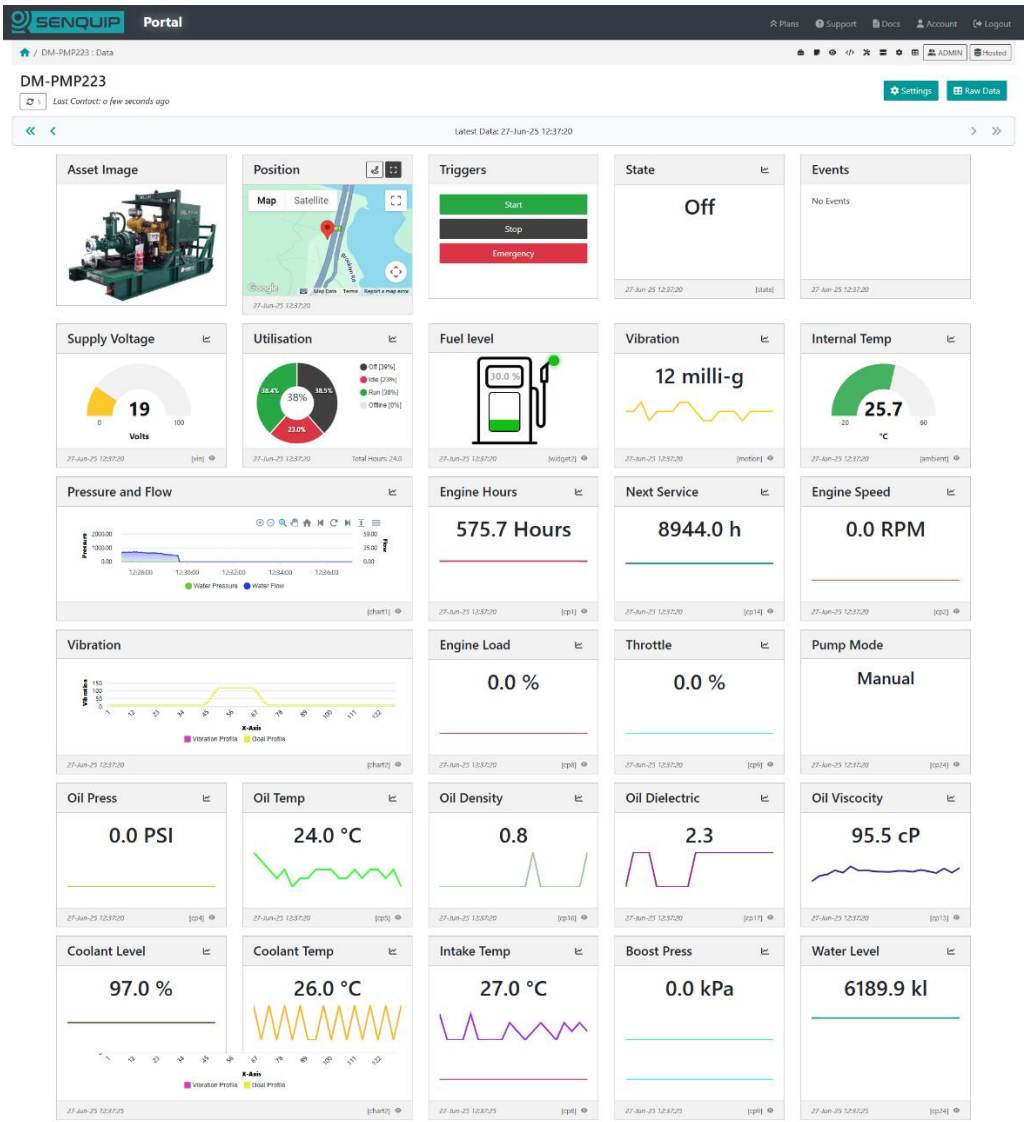


Figure 11 - Typical Pump Display on the Senquip Portal

8. Appendix A – K27 Fault Codes

Code	Meaning
0	Normal Operation
1	Low Oil Pressure
2	High Engine Temp.
3	Auxiliary 3
4	Loss of Flow Sw.
5	Alt Failure
6	Coolant Level Low
7	Overspeed
8	Underspeed
9	Bad or NO RPM
10	Failed Crank Attempts
11	Aux. Input 1
12	Aux. Input 2
13	Aux. Input 3
14	Low Fuel Level
15	Low Pump Press #2
16	Max Pump Press #2
17	Low Pump Pressure
18	Max Pump Pressure
19	CAN BUS Failure
20	Pump Temperature
21	Internal Protection
22	Suction Pressure
23	Check ECU Codes
24	Timer Complete
25	Normal Shutdown
26	Dam Level Sensor Error
29	Low Flow
30	High Flow
31	Stagnant Timer
32	Normal Shutdown
33	Pressure Stagnant
34	Gear box Temp

9. Appendix B – Full Application Script

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

let configReg = 0; // value that will be incremented before each write

function statusLookup(i) {
  let enum_lookup = {
    '0': 'Normal Operation',
    '1': 'Low Oil Pressure',
    '2': 'High Engine Temp.',
    '3': 'Auxiliary 3',
    '4': 'Loss of Flow Sw.',
    '5': 'Alt Failure',
    '6': 'Coolant Level Low',
    '7': 'Overspeed',
    '8': 'Underspeed',
    '9': 'Bad or NO RPM',
    '10': 'Failed Crank Attempts',
    '11': 'Aux. Input 1',
    '12': 'Aux. Input 2',
    '13': 'Aux. Input 3',
    '14': 'Low Fuel Level',
    '15': 'Low Pump Press #2',
    '16': 'Max Pump Press #2',
    '17': 'Low Pump Pressure',
    '18': 'Max Pump Pressure',
    '19': 'CAN BUS Failure',
    '20': 'Pump Temperature',
    '21': 'Internal Protection',
    '22': 'Suction Pressure',
    '23': 'Check ECU Codes',
    '24': 'Timer Complete',
    '25': 'Normal Shutdown',
    '26': 'Dam Level Sensor Error',
    '29': 'Low Flow',
    '30': 'High Flow',
    '31': 'Stagnant Timer',
    '32': 'Normal Shutdown',
    '33': 'Pressure Stagnant',
    '34': 'Gear box Temp'
  };
  return enum_lookup[i] || 'Default';
}

function writeMod(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
}


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

  if (typeof obj.mod12 === "number"){
      SQ.dispatch(1, statusLookup(obj.mod12)); // status as text
  }
```

```
}, null);

SQ.set_trigger_handler(function(tp) {
  if (tp === 1) { // start
    configReg++; if (configReg > 65535){configReg = 0;}
    SQ.dispatch_event(1,SQ.INFO,"Engine Starting");
    writeMod({sadr:10, radr:22, val:configReg}); // call the send Modbus routine
    Timer.set(1000, 0, function() { // After one second, send the 2nd serial string
      writeMod({sadr:10, radr:19, val:0xAA}); // send the start command
    }, null);
  }

  if (tp === 2) { // stop
    configReg++; if (configReg > 65535){configReg = 0;}
    SQ.dispatch_event(1,SQ.INFO,"Engine Stopping");
    writeMod({sadr:10, radr:22, val:configReg}); // call the send Modbus routine
    Timer.set(1000, 0, function() { // After one second, send the 2nd serial string
      writeMod({sadr:10, radr:19, val:0x55}); // send the stop command
    }, null);
  }
}, null);
```