CONNECTING TO A CONTROLS INC. CONTROLLER

1. Introduction

Controls Inc. engine and pump controllers provide a Modbus RTU interface that can be used to read operating data and issue remote control commands. A Senquip device can connect directly to the controller over RS485, collect engine and pump measurements, and expose selected controls through Senquip Portal triggers.

Figure 1 - RX Series Controller

This application note describes a tested implementation using a Senquip ORB connected to a Controls Inc. controller. The implementation uses scripted Modbus rather than individual Modbus channels so that registers can be read efficiently in blocks to respect the controller polling-rate limitation.

The example demonstrates:

- **Reading engine and pump data using Modbus function 03 (Read Holding Registers).**
- **Combining many measurements into a small number of block reads.**
- **Decoding 16-bit and 32-bit values and applying engineering-unit scaling.**
- **Using Portal triggers for AUTO, MANUAL OFF, START and STOP commands.**
- **Scheduling all Modbus reads and writes so that no more than one command is sent per second.**

Note: The register map and control functions shown are specific to the Controls Inc. controller used for this application. Confirm the register map and enabled Modbus options for the target controller before deployment.

2. Modbus Interface

2.1. Serial Configuration

The Senquip device is connected to the Controls Inc. controller using RS485 on Serial 1. Because the script sends and receives Modbus frames directly, Serial 1 is configured in Scripted mode rather than the built-in Modbus mode.

Setting	Value
Interface	RS485
Senquip Serial channel	Serial 1
Controller slave address	1
Baud rate	19200
Protocol	Modbus RTU
Senquip serial mode	Scripted

2.2. Controller Timing Limitation

The Controls Inc. Modbus documentation specifies a maximum polling frequency of one command per second. This limit applies to the Modbus bus, not to each individual register. Reading 20 or more values as separate reads would therefore make a fast telemetry update impractical.

The solution is to read contiguous ranges of holding registers with function 03 and extract the required values in the script. Standard Modbus function 03 permits up to 125 holding registers to be read in a single request. On the Controls Inc. controller used for this application, 50-register reads were tested successfully while larger reads were not reliable. The implementation therefore limits each read to a maximum of 50 registers.

The one-command-per-second limit also applies when control commands are added. A read and a write cannot be scheduled independently without potentially exceeding this limit. For this reason, the implementation treats every Modbus transaction - whether a telemetry read or a control write - as a command on the same bus schedule. The block-read arrangement is described first, followed by the control commands and the common scheduler that coordinates both.

3. Block Read Strategy

The required data is spread across two areas of the Controls Inc. holding-register map. Engine data occupies registers 40001 to 40090 and is read as two blocks of 50 and 40 registers. Pump data occupies 40801 to 40806, while autostart data begins at 40820. These are read as separate blocks rather than crossing the undefined gap between 40806 and 40820.

Block	Registers References	Protocol Start	Quantity	Purpose
1A	**40001-40050**	**0**	**50**	**Engine data**
1B	**40051-40090**	**50**	**40**	**Additional engine data**
2A	**40801-40806**	**800**	**6**	**Pump measurements**
2B	**40820-40824**	**819**	**5**	**Pump/autostart status**

3.1. Modbus Register Addressing

Modbus documentation commonly identifies registers using a reference number, such as 40001 or 40820. The leading digit identifies the Modbus data type. In the conventional Modbus reference notation:

0xxxx — Coils

1xxxx — Discrete Inputs

3xxxx — Input Registers

4xxxx — Holding Registers

The Controls Inc. registers used in this application are Holding Registers, hence the leading 4. The 4 is not part of the address transmitted over Modbus; it identifies the register type.

Modbus messages address registers from zero. For Holding Registers, reference 40001 corresponds to protocol address 0, 40002 to address 1, and so on. The protocol address can therefore be calculated by subtracting 40001 from the documented register number.

For example, register 40820 has a protocol address of:

40820 − 40001 = 819 = 0x0333

A Function 03 request to read register 40820 therefore contains the starting address 819, not 40820. When constructing the Modbus message as bytes, the decimal address 819 is represented as 0x0333, giving address bytes 0x03 0x33.

This distinction between the register reference number and the protocol address is important when constructing Modbus requests directly in a script.

3.2. Reading Holding Registers

Holding registers are read using Modbus Function 03 (Read Holding Registers**)**. A read request identifies the slave device, the starting protocol address, and the number of consecutive 16-bit registers to return.

For example, Block 2B reads holding registers 40820 to 40824. As described above, register reference 40820 corresponds to protocol address 819, and the block contains 5 registers.

The Function 03 request, before the CRC is added, is therefore:

01 03 03 33 00 05

where:

Bytes	Description	Value
01	**Slave address**	**1**
03	**Function**	**Read Holding Registers**
03 33	**Starting address**	**819**
00 05	**Quantity**	**5 registers**

The starting address and quantity are each transmitted as two-byte values, most-significant byte first. In this example, decimal 819 is 0x0333, giving the bytes 03 33, while a quantity of 5 is 0x0005, giving 00 05.

The Modbus RTU message also requires a two-byte CRC. Rather than calculating and entering the CRC manually, the script constructs the six-byte request and uses the Senquip SQ.crc() function to calculate and append it:

```
let READ_BANK2B = '\x01\x03\x03\x33\x00\x05';

function modbusSend(cmd) {
  let crc = SQ.crc(cmd);
  let msg = cmd + SQ.encode(crc, -SQ.U16);
  SERIAL.write(SERIAL_CH, msg, msg.length, SERIAL.LOOPBACK);
}

modbusSend(READ_BANK2B);
```

The same approach is used for all four read blocks:

```
let READ_BANK1A = '\x01\x03\x00\x00\x00\x32';  // 40001 - 40050
let READ_BANK1B = '\x01\x03\x00\x32\x00\x28';  // 40051 - 40090
let READ_BANK2A = '\x01\x03\x03\x20\x00\x06';  // 40801 - 40806
let READ_BANK2B = '\x01\x03\x03\x33\x00\x05';  // 40820 - 40824
```

Each request asks the controller for a contiguous block of registers. Reading blocks rather than requesting every measurement individually greatly reduces the number of Modbus commands required.

3.3. Processing the Modbus Response

Sending a Function 03 request causes the controller to return the contents of the requested holding registers as a sequence of bytes. The script must associate the response with the block that was requested, locate each required register within that block, interpret the returned bytes using the correct data type, and apply any required scaling or offset.

For example, a response to the Block 1A request contains 50 registers, or 100 bytes of register data, corresponding to register references 40001 through 40050. Engine Speed is register 40004, so its value is found three registers, or six bytes, from the beginning of the returned data.

Rather than writing separate parsing code for every measurement, the implementation uses a table to describe the required registers. Each entry specifies the Custom Parameter (CP) to which the result will be dispatched, the register reference, the data type, scaling and any offset:

```
let BANK1 = [
  { cp: 1, reg: 40002, name: 'Percent Load', type: 'U16', scale: 1, offset: 0 },
  { cp: 2, reg: 40004, name: 'Engine Speed', type: 'U16', scale: 1, offset: 0 },
  { cp: 3, reg: 40009, name: 'Engine Hours', type: 'U32_LM', scale: 0.1, offset: 0 },
  { cp: 4, reg: 40013, name: 'Total Fuel Used', type: 'U32_LM', scale: 0.1, offset: 0 }
];
```

This separates the register map from the code used to process it. Measurements can therefore be added, removed or moved to different Custom Parameters by changing the table rather than changing the Modbus parsing logic.

4. Remote Control Commands

The same RS485 connection used to read telemetry can also be used to issue control commands to the Controls Inc. controller. In this application, four remote functions are exposed through Senquip Portal triggers:

AUTO

MANUAL OFF

START

STOP

Two different Modbus functions are required. AUTO and MANUAL OFF are written to a holding register using Function 06 (Write Single Register), while START and STOP use Function 05 (Write Single Coil).

4.1. AUTO and MANUAL OFF

The controller uses holding register reference 40538 for panel requests. The lower three bits select the requested operating state:

1 = Manual Off

2 = Auto

3 = Manual On

Register reference 40538 corresponds to protocol address:

40538 - 40001 = 537

Decimal 537 is 0x0219, so the Function 06 command for AUTO, before the CRC is added, is:

01 06 02 19 00 02

where:

Bytes	Description	Value
01	Slave address	1
06	Function	Write Single Register
02 19	Register address	537
00 02	Value	Auto

MANUAL OFF uses the same register with a value of 1:

01 06 02 19 00 01

The commands can therefore be defined in the script as:

```
let CMD_AUTO       = '\x01\x06\x02\x19\x00\x02';
let CMD_MANUAL_OFF = '\x01\x06\x02\x19\x00\x01';
```

The same modbusSend() function used for read requests can be used because it appends the CRC and transmits the completed Modbus RTU frame.

4.2. START and STOP

START and STOP are implemented using Function 05 (Write Single Coil). The controller defines Coil 2 as the AutoStart control.

For Function 05, the standard Modbus values are:

ON = 0xFF00

OFF = 0x0000

Coil 2 has protocol address 1 because coil addressing is zero-based. The START command is therefore:

01 05 00 01 FF 00

and the STOP command is:

01 05 00 01 00 00

These are defined in the script as:

```
let CMD_START = '\x01\x05\x00\x01\xFF\x00';
let CMD_STOP  = '\x01\x05\x00\x01\x00\x00';
```

Again, the CRC is calculated and appended by modbusSend() before transmission.

4.3. Portal Triggers

The four commands are associated with Senquip Portal triggers. The trigger handler does not immediately transmit the Modbus command. Instead, it places the requested command into a pending-command variable so that it can be sent by the common Modbus scheduler.

```
let pendingCommand = null;

SQ.set_trigger_handler(function(tp) {
  if (tp === 1) { pendingCommand = CMD_AUTO; }
  else if (tp === 2) { pendingCommand = CMD_MANUAL_OFF; }
  else if (tp === 3) { pendingCommand = CMD_START; }
  else if (tp === 4) { pendingCommand = CMD_STOP; }
}, null);
```

This is important because a trigger may arrive at any time. Sending the command directly from the trigger handler could cause it to occur too close to a telemetry read and violate the controller's one-command-per-second limit.

5. Modbus Scheduling

The four block reads are performed sequentially. The first block is requested by the Senquip data handler:

```
SQ.set_data_handler(function() {

if (pendingCommand !== null) {
    let cmd = pendingCommand;
    pendingCommand = null;
    modbusSend(cmd);
    Timer.set(NEXT_READ_DELAY_MS, 0, function() { modbusSend(READ_BANK1A); }, null);
    return;
  }

  modbusSend(READ_BANK1A);
}, null);
```

When a valid response is received, the response handler processes the returned registers and schedules the next block:

```
if (modbus.func === 3 && modbus.reg_addr === 0) {
  if (modbus.qty !== 50 || modbus.len !== 100) { return; }
  parseBank(modbus.data, 40001, 40050, BANK1);
  nextCommand(READ_BANK1B);
  return;
}
```

The *nextCommand()* function waits 1050ms (*NEXT_READ_DELAY_MS*) before allowing the next Modbus transaction. At the end of this delay it first checks whether a control command is pending:

```
function nextCommand(readCmd) {
  Timer.set(NEXT_READ_DELAY_MS, 0, function(readCmd) {
    if (pendingCommand !== null) {
      let cmd = pendingCommand;
      pendingCommand = null;
      modbusSend(cmd);
      Timer.set(NEXT_READ_DELAY_MS, 0, function(readCmd) { modbusSend(readCmd); }, readCmd);
      return;
    }

    modbusSend(readCmd);
  }, readCmd);
}
```

If no control command is waiting, the next block is read immediately. If a command is waiting, the control command occupies that Modbus opportunity and the postponed read is sent 1050 ms later.

Thus a normal acquisition cycle is:

Read 1A → Read 1B → Read 2A → Read 2B

whereas a STOP command arriving between Blocks 1B and 2A results in:

Read 1A → Read 1B → STOP AND EXECUTE TRIGGER → Read 2A → Read 2B

The delay is started after the preceding Modbus response has been received, so consecutive requests are separated by at least 1050 ms.

6. Register Mapping and Data Conversion

The block reads return all registers within each requested range, but only a subset of those registers is required by the application. A table in the script defines which registers are used, the Custom Parameter (CP) to which each value is dispatched, the data type, and any scaling or offset required to convert the raw Modbus value into engineering units.

For example:

```
let BANK1 = [
  { cp: 1, reg: 40002, name: 'Percent Load',    type: 'U16',    scale: 1,   offset: 0 },
  { cp: 2, reg: 40004, name: 'Engine Speed',    type: 'U16',    scale: 1,   offset: 0 },
  { cp: 3, reg: 40009, name: 'Engine Hours',    type: 'U32_LM', scale: 0.1, offset: 0 },
  { cp: 4, reg: 40013, name: 'Total Fuel Used', type: 'U32_LM', scale: 0.1, offset: 0 }
];
```

This table-driven approach keeps the register map separate from the Modbus communication code. Measurements can be added, removed, assigned to different Custom Parameters, or given different scaling without changing the block-read or scheduling logic.

6.1. Locating a Register Within a Block

A Modbus holding register contains 16 bits, or two bytes. The byte position of a register within a returned block is therefore:

```
(reg - first_register) × 2
```

For example, Engine Speed is register reference 40004 and Block 1A begins at 40001:

(40004 - 40001) × 2 = 6

Engine Speed therefore begins at byte offset 6 in the data returned for Block 1A.

The script performs this calculation in getRegister():

let index = (reg - first_register) * 2;

6.2. 16-bit and 32-bit Values

Most measurements occupy a single 16-bit holding register and are decoded as either unsigned (U16) or signed (S16) values:

```
if (type === 'S16') {
  return SQ.parse(s, index, 2, 0, SQ.S16);
}

return SQ.parse(s, index, 2, 0, SQ.U16);
```

Some values, such as Engine Hours and Total Fuel Used, occupy two consecutive registers and form a 32-bit value. On this controller, the least-significant 16-bit word is stored first, followed by the most-significant word:

```
if (type === 'U32_LM') {
  let lsw = SQ.parse(s, index, 2, 0, SQ.U16);
  let msw = SQ.parse(s, index + 2, 2, 0, SQ.U16);
  return lsw + msw * 65536;
}
```

6.3. Scaling and Offsets

The raw register value is converted to engineering units using the scale and offset specified in the table:

value = raw * p.scale + p.offset;

For example, a Fuel Rate register with a raw value of 15 and a scale of 0.1 produces:

15 × 0.1 = 1.5 gal/hr

Suction Pressure additionally requires an offset:

{ cp: 16, reg: 40803, name: 'Suction Pressure', type: 'U16', scale: 0.1, offset: -14.7 }

The resulting value is then dispatched to the nominated Senquip Custom Parameter:

```
SQ.dispatch(p.cp, value);
```

6.4. Enumerated Values

Not all registers represent numerical measurements. Several Controls Inc. registers contain an integer that represents an operating mode or state. For these registers, sending the raw number to the Senquip Portal would make the data difficult to interpret, so the script converts the value to a descriptive string before dispatching it.

The data table identifies these registers using types such as *KEY*, *MANAUTO*, *THROTTLE* and *AUTOSTART*:

```
{ cp: 14, reg: 40084, name: 'Key Position',        type: 'KEY',       scale: 1, offset: 0 },
{ cp: 20, reg: 40820, name: 'Manual Auto Status', type: 'MANAUTO',    scale: 1, offset: 0 },
{ cp: 21, reg: 40821, name: 'Throttle Mode',      type: 'THROTTLE',   scale: 1, offset: 0 },
{ cp: 22, reg: 40822, name: 'Autostart State',    type: 'AUTOSTART', scale: 1, offset: 0 }
```

For example, Key Position is converted using:

```
function keyPosition(v) {
  if (v === 0) { return 'Manual'; }
  if (v === 1) { return 'Auto'; }
  if (v === 2) { return 'Off'; }
  return 'Unknown (' + v + ')';
}
```

Similarly, the throttle mode register is converted to the operating mode defined by the controller:

```
function throttleMode(v) {
  if (v === 0) { return 'None'; }
  if (v === 1) { return 'Single Speed, Target'; }
  if (v === 2) { return 'Single Speed'; }
  if (v === 3) { return 'Linear'; }
  if (v === 4) { return 'Target'; }
  if (v === 5) { return 'Booster'; }
  return 'Unknown (' + v + ')';
}
```

The Autostart State register provides particularly useful diagnostic information because it identifies the current stage of the controller's start and stop sequence. The script converts the numerical state to descriptions such as Start delay, Crank, Warmup, Rampup, Operate and Rampdown.

The appropriate conversion is selected by *parseBank():*

```
if (p.type === 'KEY') { value = keyPosition(raw); }
else if (p.type === 'MANAUTO') { value = manAuto(raw); }
else if (p.type === 'THROTTLE') { value = throttleMode(raw); }
else if (p.type === 'AUTOSTART') { value = autostartState(raw); }
else { value = raw * p.scale + p.offset; }
```

The converted string is then dispatched to the nominated Custom Parameter in the same way as a numerical measurement:

```
SQ.dispatch(p.cp, value);
```

This allows the Senquip Portal to display meaningful values such as **Auto**, **Single Speed** and **Operate** rather than the corresponding raw register values.

7. Custom Parameter Configuration

The example script dispatches the decoded Modbus values to Senquip Custom Parameters CP1 to CP22. The Custom Parameters should be configured in the Senquip Portal with names and units that correspond to the register mapping used by the script.

CP	Name	Units
CP1	Percent Load	%
CP2	Engine Speed	rpm
CP3	Engine Hours	hours
CP4	Total Fuel Used	gal
CP5	Coolant Temperature	°F
CP6	Oil Temperature	°F
CP7	Oil Pressure	psi
CP8	Coolant Level	%
CP9	Fuel Rate	gal/hr
CP10	Alternator Potential	V
CP11	Electrical Potential	V
CP12	Keyswitch Potential	V
CP13	Fuel Level	%
CP14	Key Position	—
CP15	Pump Level	in
CP16	Suction Pressure	psi
CP17	Discharge Pressure	psi
CP18	Pump Flow Rate	gal/min
CP19	Pump Temperature	°F
CP20	Manual/Auto Status	—
CP21	Throttle Mode	—
CP22	Autostart State	—

The CP number is defined in the register table at the beginning of the script. For example:

```
{ cp: 2, reg: 40004, name: 'Engine Speed', type: 'U16', scale: 1, offset: 0 }
```

dispatches Engine Speed from register reference 40004 to CP2.

The mapping can be changed to suit a particular installation by editing the `cp` value in the table. Similarly, registers can be added or removed without changing the Modbus block-read mechanism, provided the required register lies within one of the ranges being read.

Note: The script defines how a value is decoded and dispatched, while the Custom Parameter configuration in the Senquip Portal defines how that value is named and displayed.

8. Commissioning and Testing

Before testing the Modbus interface, confirm that Serial 1 is configured for RS485 operation at 19200 baud and that the Controls Inc. controller is configured as Modbus slave address 1. The serial port must be set to **Scripted** mode because the Modbus requests are generated by the script.

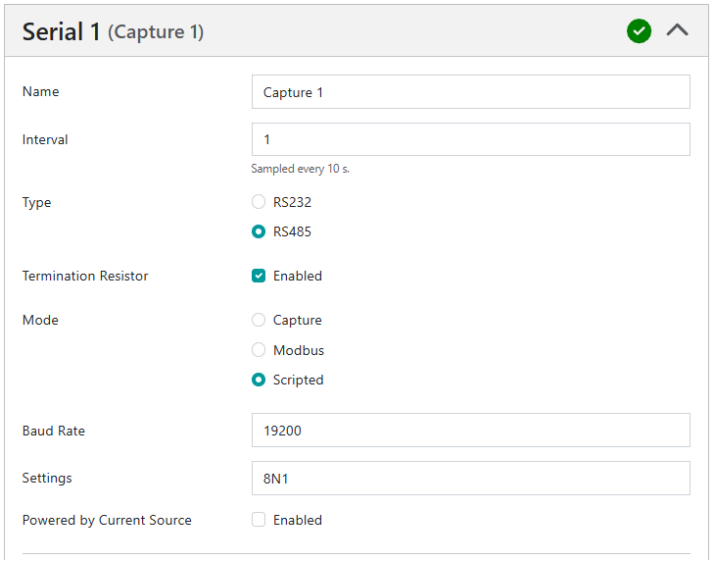

Figure 2 - Senquip ORB Serial Settings

8.1. Verify Communications

With the script running, confirm that data is being received from each of the four register blocks:

Block	Register References	Expected Registers
1A	40001–40050	50
1B	40051–40090	40
2A	40801–40806	6
2B	40820–40824	5

Check that the expected Custom Parameters are being updated in the Senquip Portal.

8.2. Verify Measurements

Compare several reported values with those displayed locally by the controller. Engine Speed, Engine Hours, battery voltage, fuel level and pump measurements provide useful checks that the register addressing, data type and scaling are correct.

Where a sensor or data source is unavailable, the controller may return a value such as 0xFFFF. During commissioning, retaining these values can be useful because it distinguishes an unavailable measurement from a Modbus communications failure.

8.3. Verify Operating States

Check that the enumerated Custom Parameters correspond with the controller:

- **Key Position** – Manual, Auto or Off
- **Manual/Auto Status** – Manual or Auto
- **Throttle Mode** – current throttle operating mode
- **Autostart State** – current stage of the autostart sequence

The Autostart State is particularly useful during testing because it allows the start and stop sequence to be followed through states such as Start delay, Crank, Warmup, Rampup, Operate and Rampdown.

Figure 3 - Sequence of Events Following a Stop Command

8.4. Test Remote Control

Test each Portal trigger individually and confirm the resulting controller behaviour:

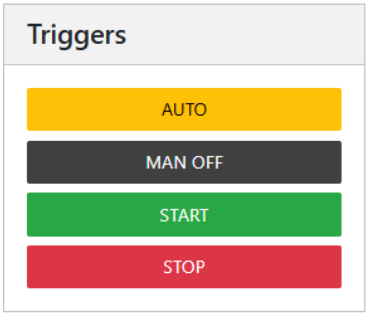

Figure 4 - Triggers 1 through 4 as Shown on the Senquip Portal

Allow sufficient time between commands for the controller state to change and for the updated state to be returned in the next acquisition cycle.

When testing AUTO, note that selecting Auto mode does not itself necessarily request an engine start. If the controller's automatic start conditions are already satisfied, however, changing to Auto may cause the controller to begin its normal autostart sequence.

The START and STOP triggers operate the controller's AutoStart coil and are separate from the Auto/Manual mode selection.

8.5. Verify Command Timing

Finally, confirm that read and control commands share the same Modbus schedule. No two Modbus commands should be transmitted less than 1050 ms apart.

A control command received during an acquisition cycle should be inserted between block reads, with the interrupted read sequence continuing after the control command:

Read 1A → Read 1B → STOP → Read 2A → Read 2B

This confirms both remote control operation and compliance with the controller's maximum polling frequency of one command per second.

9. Conclusion

This application note demonstrates how a Senquip device can use scripted Modbus RTU communications to monitor and control a Controls Inc. engine and pump controller.

Reading groups of contiguous holding registers significantly reduces the number of Modbus transactions required, while the command scheduling ensures that both data acquisition and remote control comply with the controller's maximum polling frequency.

The table-driven register mapping allows measurements, scaling and Custom Parameter assignments to be readily adapted for different applications, while Senquip Portal triggers provide remote AUTO, MANUAL OFF, START and STOP control using the same RS485 connection.

The result is a single interface for remote monitoring, diagnostics and control of the controller through the Senquip Portal.

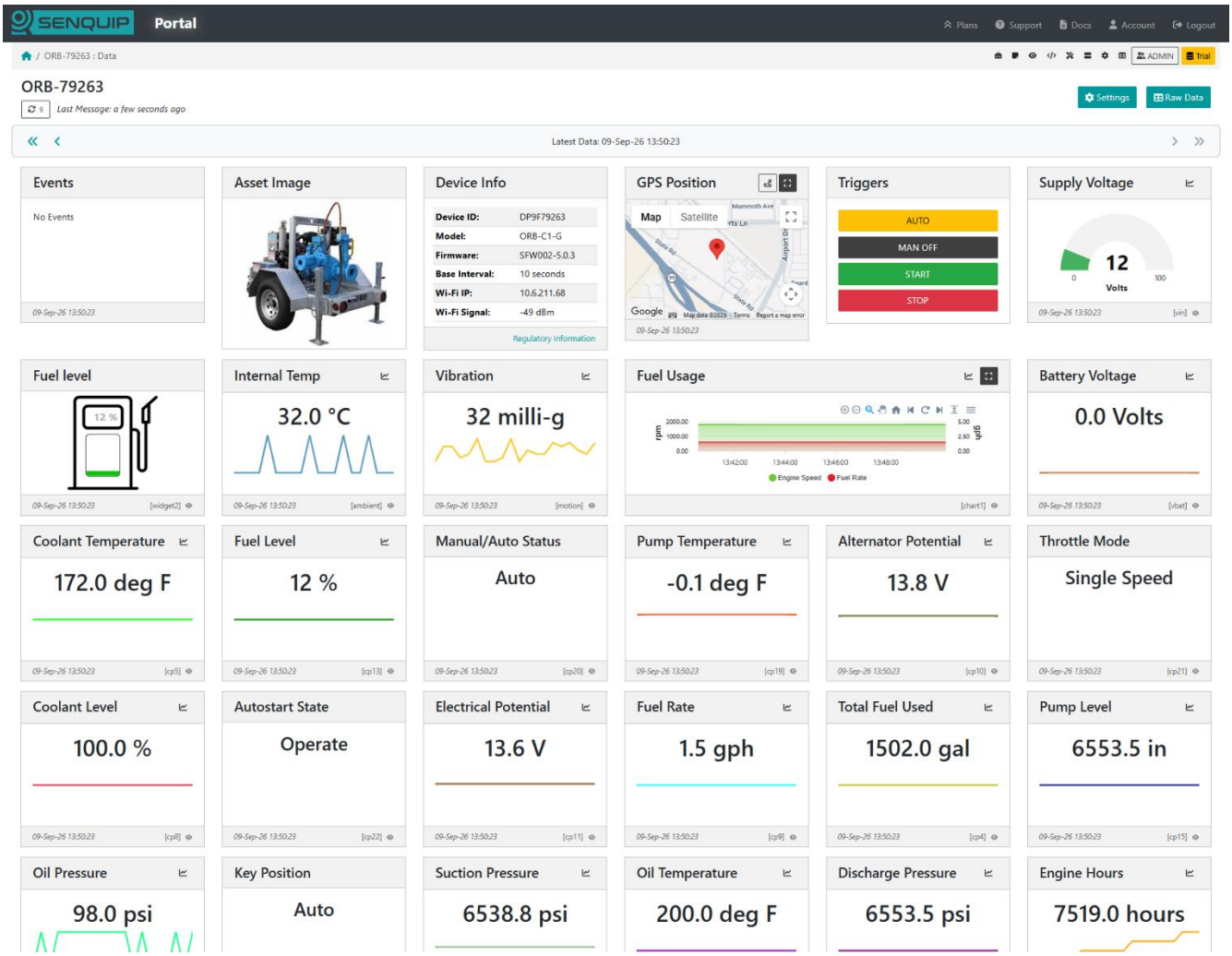

Figure 5 - Typical Device Dashboard for a Pump

10. Disclaimer

The information provided in this application note is intended for informational purposes only. Remote starting and stopping of engines and machinery can create hazardous conditions. The machine control system must retain all required interlocks, emergency stops and safety functions, and remote control should only be enabled where the installation has been assessed as safe and compliant with applicable requirements.

Register maps and controller behaviour can vary by model and configuration. Verify all commands and status values against the documentation for the specific Controls Inc. controller before deployment.

Appendix A: Complete Script

```
/*
 * Controls Inc. Modbus Interface
 *
 * Reads:
 *   40001 - 40050
 *   40051 - 40090
 *   40801 - 40806
 *   40820 - 40824
 *
 * Triggers:
 *   1 = Auto
 *   2 = Manual Off
 *   3 = Start
 *   4 = Stop
 *
 * Controls specifies a maximum polling rate of one command per second.
 */

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

let SLAVE_ADDR = 1;
let SERIAL_CH = 1;
let NEXT_READ_DELAY_MS = 1050;
let pendingCommand = null;

let BANK1 = [
  { cp: 1,  reg: 40002, name: 'Percent Load',          type: 'U16',      scale: 1,    offset: 0 },
  { cp: 2,  reg: 40004, name: 'Engine Speed',          type: 'U16',      scale: 1,    offset: 0 },
  { cp: 3,  reg: 40009, name: 'Engine Hours',          type: 'U32_LM',   scale: 0.1,  offset: 0 },
  { cp: 4,  reg: 40013, name: 'Total Fuel Used',       type: 'U32_LM',   scale: 0.1,  offset: 0 },
  { cp: 5,  reg: 40015, name: 'Coolant Temperature',   type: 'S16',      scale: 1,    offset: 0 },
  { cp: 6,  reg: 40017, name: 'Oil Temperature',       type: 'S16',      scale: 1,    offset: 0 },
  { cp: 7,  reg: 40021, name: 'Oil Pressure',          type: 'U16',      scale: 1,    offset: 0 },
  { cp: 8,  reg: 40023, name: 'Coolant Level',         type: 'U16',      scale: 0.01, offset: 0 },
  { cp: 9,  reg: 40025, name: 'Fuel Rate',             type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 10, reg: 40034, name: 'Alternator Potential',  type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 11, reg: 40035, name: 'Electrical Potential',  type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 12, reg: 40036, name: 'Keyswitch Potential',   type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 13, reg: 40069, name: 'Fuel Level',            type: 'U16',      scale: 1,    offset: 0 },
  { cp: 14, reg: 40084, name: 'Key Position',          type: 'KEY',      scale: 1,    offset: 0 }
];

let BANK2 = [
  { cp: 15, reg: 40801, name: 'Pump Level',            type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 16, reg: 40803, name: 'Suction Pressure',      type: 'U16',      scale: 0.1,  offset: -14.7 },
  { cp: 17, reg: 40804, name: 'Discharge Pressure',    type: 'U16',      scale: 0.1,  offset: 0 },
  { cp: 18, reg: 40805, name: 'Pump Flow Rate',        type: 'U16',      scale: 1,    offset: 0 },
  { cp: 19, reg: 40806, name: 'Pump Temperature',      type: 'S16',      scale: 0.1,  offset: 0 },
  { cp: 20, reg: 40820, name: 'Manual Auto Status',    type: 'MANAUTO',  scale: 1,    offset: 0 },
  { cp: 21, reg: 40821, name: 'Throttle Mode',         type: 'THROTTLE', scale: 1,    offset: 0 },
  { cp: 22, reg: 40822, name: 'Autostart State',       type: 'AUTOSTART',scale: 1,    offset: 0 }
];

let READ_BANK1A = '\x01\x03\x00\x00\x00\x32';  // 40001 - 40050
let READ_BANK1B = '\x01\x03\x00\x32\x00\x28';  // 40051 - 40090
let READ_BANK2A = '\x01\x03\x03\x20\x00\x06';  // 40801 - 40806
let READ_BANK2B = '\x01\x03\x03\x33\x00\x05';  // 40820 - 40824

let CMD_AUTO       = '\x01\x06\x02\x19\x00\x02';  // 40538 = Auto
```

```
let CMD_MANUAL_OFF = '\x01\x06\x02\x19\x00\x01';  // 40538 = Manual Off
let CMD_START      = '\x01\x05\x00\x01\xFF\x00';  // Coil 2 ON
let CMD_STOP       = '\x01\x05\x00\x01\x00\x00';  // Coil 2 OFF

function modbusSend(cmd) {
  let crc = SQ.crc(cmd);
  let msg = cmd + SQ.encode(crc, -SQ.U16);
  SERIAL.write(SERIAL_CH, msg, msg.length, SERIAL.LOOPBACK);
}

function nextCommand(readCmd) {
  Timer.set(NEXT_READ_DELAY_MS, 0, function(readCmd) {
    if (pendingCommand !== null) {
      let cmd = pendingCommand;
      pendingCommand = null;
      modbusSend(cmd);
      Timer.set(NEXT_READ_DELAY_MS, 0, function(readCmd) { modbusSend(readCmd); }, readCmd);
      return;
    }

    modbusSend(readCmd);
  }, readCmd);
}

function getRegister(s, first_register, reg, type) {
  let index = (reg - first_register) * 2;

  if (type === 'S16') {
    return SQ.parse(s, index, 2, 0, SQ.S16);
  }

  if (type === 'U32_LM') {
    let lsw = SQ.parse(s, index, 2, 0, SQ.U16);
    let msw = SQ.parse(s, index + 2, 2, 0, SQ.U16);
    return lsw + msw * 65536;
  }

  return SQ.parse(s, index, 2, 0, SQ.U16);
}

function keyPosition(v) {
  if (v === 0) { return 'Manual'; }
  if (v === 1) { return 'Auto'; }
  if (v === 2) { return 'Off'; }
  return 'Unknown (' + v + ')';
}

function manAuto(v) {
  if (v === 0) { return 'Manual'; }
  if (v === 1) { return 'Auto'; }
  return 'Unknown (' + v + ')';
}

function throttleMode(v) {
  if (v === 0) { return 'None'; }
  if (v === 1) { return 'Single Speed, Target'; }
  if (v === 2) { return 'Single Speed'; }
  if (v === 3) { return 'Linear'; }
  if (v === 4) { return 'Target'; }
  if (v === 5) { return 'Booster'; }
  return 'Unknown (' + v + ')';
}

function autostartState(v) {
```

```
  let states = ['Disabled','Idle, fuel on','Idle, fuel off','Launch','Start delay','Begin
preheat','Preheat','Begin crank','Crank','Reset','Begin warmup','Warmup','Begin prime','Prime
rampup','Prime','Begin rampup','Rampup','Begin operate','Operate','Begin prime rampdown','Prime
rampdown','Begin rampdown','Rampdown','Begin cooldown','Cooldown','Shutdown','Fault','Manual crank','Fault,
disabled','Reserved','Manual idle','Manual stop','Manual preheat','Manual preheat complete','Manual
operate','Begin forced warmup','Forced warmup','Manual crank release delay','Manual crank release'];

  if (v >= 0 && v < states.length) { return states[v]; }
  return 'Unknown (' + v + ')';
}

function parseBank(s, first_register, last_register, table) {
  for (let i = 0; i < table.length; i++) {
    let p = table[i];

    if (p.reg < first_register || p.reg > last_register) { continue; }

    let raw = getRegister(s, first_register, p.reg, p.type);
    let value;

    if (p.type === 'KEY') { value = keyPosition(raw); }
    else if (p.type === 'MANAUTO') { value = manAuto(raw); }
    else if (p.type === 'THROTTLE') { value = throttleMode(raw); }
    else if (p.type === 'AUTOSTART') { value = autostartState(raw); }
    else { value = raw * p.scale + p.offset; }

    SQ.dispatch(p.cp, value);
  }
}

SERIAL.set_modparse_handler(SERIAL_CH, 3, function(evdata) {
  let modbus = SERIAL.getModbusResult(evdata);

  if (modbus.slave_addr !== SLAVE_ADDR) { return; }

  if (modbus.func === 3 && modbus.reg_addr === 0) {
    if (modbus.qty !== 50 || modbus.len !== 100) { return; }
    parseBank(modbus.data, 40001, 40050, BANK1);
    nextCommand(READ_BANK1B);
    return;
  }

  if (modbus.func === 3 && modbus.reg_addr === 50) {
    if (modbus.qty !== 40 || modbus.len !== 80) { return; }
    parseBank(modbus.data, 40051, 40090, BANK1);
    nextCommand(READ BANK2A);
    return;
  }

  if (modbus.func === 3 && modbus.reg_addr === 800) {
    if (modbus.qty !== 6 || modbus.len !== 12) { return; }
    parseBank(modbus.data, 40801, 40806, BANK2);
    nextCommand(READ_BANK2B);
    return;
  }

  if (modbus.func === 3 && modbus.reg_addr === 819) {
    if (modbus.qty !== 5 || modbus.len !== 10) { return; }
    parseBank(modbus.data, 40820, 40824, BANK2);
  }
}, 2000, null);

SQ.set_trigger_handler(function(tp) {
  if (tp === 1) { pendingCommand = CMD_AUTO; }
```

```
  else if (tp === 2) { pendingCommand = CMD_MANUAL_OFF; }
  else if (tp === 3) { pendingCommand = CMD_START; }
  else if (tp === 4) { pendingCommand = CMD_STOP; }
}, null);

SQ.set_data_handler(function() {
  if (pendingCommand !== null) {
    let cmd = pendingCommand;
    pendingCommand = null;
    modbusSend(cmd);
    Timer.set(NEXT_READ_DELAY_MS, 0, function() { modbusSend(READ_BANK1A); }, null);
    return;
  }

  modbusSend(READ_BANK1A);
}, null);
```