CANOPEN INTEGRATION WITH MICROCONTROL TEMPERATURE ACQUISITION DEVICE

1. Introduction

MicroControl's µCAN.4.ti-BOX is a compact and rugged I/O module designed for precision temperature sensor measurement in demanding industrial environments. Capable of acquiring high-resolution temperature data from up to four thermocouples, the device is ideal for applications in process control, environmental monitoring, and machinery diagnostics where accurate thermal insight is critical.

By integrating a Senquip telemetry device with the MicroControl module over a CANopen interface, users gain real-time remote access to temperature data without the need for complex gateways or expensive control systems. This direct communication enables continuous monitoring of thermal conditions, early fault detection, and the ability to trend performance over time—all from anywhere with connectivity to the Senquip Portal.

Figure 1 – MicroControl µCAN.4.ti-BOX

This application-note outlines how to interface a Senquip device with the µCAN.4.ti-BOX using CANopen. Specifically, it covers how to issue a Start Node command and periodically send SYNC messages to trigger Process Data Object (PDO) transmissions. Once configured, the Senquip device decodes and publishes temperature values to the Portal, allowing remote visibility of all four thermocouple channels.

If you need more information on CANopen, CSS Electronics provides a comprehensive guide and video introduction. Further information is also available in the MicroControl manual.

The following sections detail wiring requirements, CAN bus setup, and scripting required to establish a reliable CANopen link for remote temperature monitoring.

***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. Introduction to CANopen

CANopen is a communication protocol built on the Controller Area Network (CAN) physical layer. It is widely used in industrial automation, mobile machinery, and process control systems due to its reliability, modular structure, and well-defined message types.

In a CANopen network, each device (or node) is assigned a Node ID from 1 to 127. Communication is structured around standard message types, which serve distinct roles in synchronising devices, exchanging data, and configuring behaviour.

Some of the most important message types are:

Process Data Objects (PDOs)

PDOs are used for real-time data exchange. These messages are short (up to 8 bytes) and transmitted without confirmation. A device might use a Transmit PDO to broadcast sensor readings (like thermocouple temperatures), while a Receive PDO is used to receive control commands. PDOs are typically linked to a SYNC message to keep data timing consistent across the network.

Service Data Objects (SDOs)

SDOs provide a way to read and write configuration parameters from a device's object dictionary. They are used less frequently and are not real-time—think of them as setup and diagnostics channels. SDO communication is more complex and often not needed for simple monitoring tasks.

Network Management (NMT)

NMT messages are used to control the state of nodes, such as starting, stopping, or resetting them. These messages are sent by a CANopen master (in our case, the Senquip device).

SYNC

The SYNC message is a broadcast that tells all listening nodes to send their PDOs. This creates coordinated data updates across multiple devices.

2.1. CANopen Message Identifiers

The following table lists common CANopen messages and their hex identifiers:

Function	Identifier (Hex)	Notes
NMT (Network Management)	0x000	Sent by master to control node states
SYNC	0x080	Broadcast to trigger PDO transmission
Time Stamp	0x100	(Optional) Synchronizes time across devices
PDO1 (Transmit)	0x180 + Node ID	Device sends data in response to SYNC
PDO1 (Receive)	0x200 + Node ID	Device receives data (not used in this example)
SDO (Transmit)	0x580 + Node ID	Device sends configuration/status
SDO (Receive)	0x600 + Node ID	Device receives configuration
Heartbeat/Node Guard	0x700 + Node ID	Used to monitor device availability

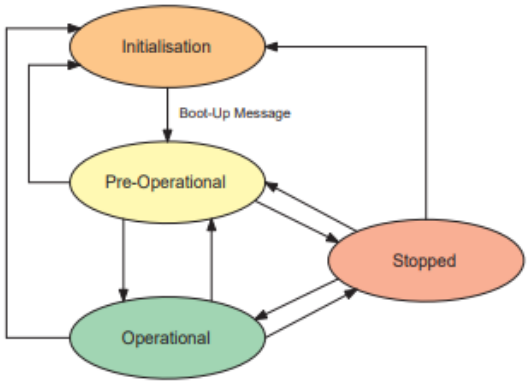
For example, if a device has a Node ID of 1:

- It will transmit its first PDO at 0x181
- It will respond to SDO requests at 0x581

For this application, the interface is elegantly simple. The Senquip device only needs to perform three actions:

1 - Issue Start Node Command

We send a Start Node Command (NMT function) to transition the MicroControl module from pre-operational to operational state.



Start Node Command:

COB ID	Data Length	Byte 0	Byte 1
0x00	2	0x01	Node

In this case, the Node will be 1, or 0 for all.

2 - Issue Sync Command

In this application-note, we assume that the MicroControl unit has already been configured to transmit PDOs for four K-type thermocouples upon receipt of a SYNC message. The Senquip device will parse and publish these PDO values, allowing remote monitoring of temperature in real time.

We issue periodic SYNC messages to prompt the device to transmit its configured Process Data Objects (PDOs), which contain the thermocouple readings.

SYNC Command:

COB ID	Data Length
0x80	0

The sync command has no data bytes.

3 – Parse the PDO response

The MicroControl module will respond to a SYNC Command by sending PDOs.

PDO Response:

COB ID	Data Length	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x181	8	Thermocouple 1		Thermocouple 2		Thermocouple 3		Thermocouple 4	
		LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

Since data is sent in little endian format, bytes will need to be flipped when the CAN message is parsed by the Senquip device.

3. Wiring the Senquip Device to MicroControl Temperature Acquisition Unit

In this application note, we will use the CAN 1 port on a Senquip QUAD, wired to the CAN port of the MicroCOntrol unit.

Figure 2 shows the top view of the µCAN.4.ti-BOX PCB. Use the figure to identify the terminal blocks, LEDs and DIP switches.

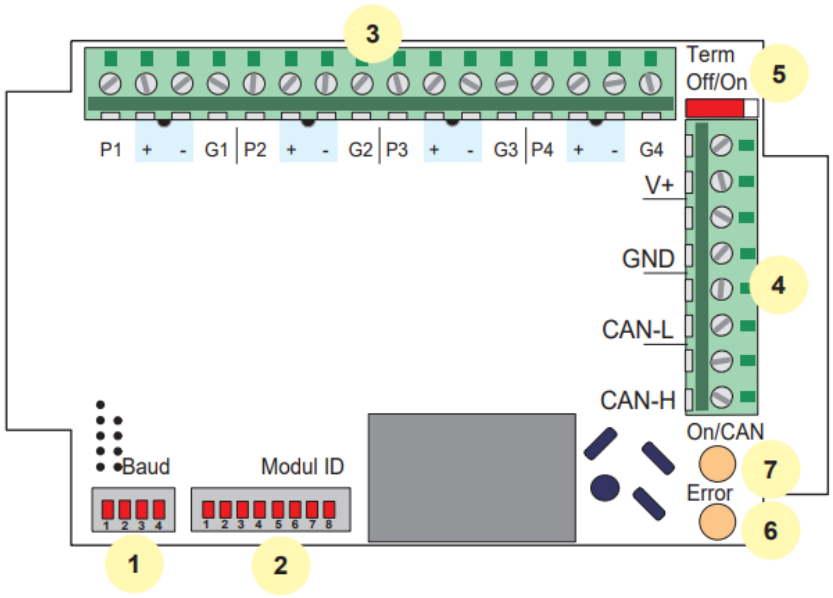


Figure 2 - uCAN.4.ti-BOX PCB Layout

1. Switch to configure baud rate
2. Switch to configure node ID
3. Terminal block for temperature sensors
4. Terminal block for voltage supply and CAN
5. Switch for CAN bus termination
6. Bi-colour LED for device status
7. Bi-colour LED for network status

From the MicroControl manual, we find that a Baud DIP switch position of 0110 sets a baud rate of 250kbps. The Senquip device baud rate will be set to match.

The module id is set to 1 using the Modul ID DIP switch.

The following connections are required:

Connection	Senquip QUAD	µCAN.4.ti-BOX
CAN L	Pin 12, CAN L	CAN L
CAN H	Pin 11, CAN H	CAN H
GND	Pin 2, GND	V+
Switched PWR	Pin 1, PWR +	GND

If a screened wire is available, it should be connected to ground on the Senquip device. Connecting the shield to the Senquip and MicroControl unit can create a ground loop which will be susceptible to magnetic fields.

Is it suggested that the 120ohm CAN termination resistor be turned on using the Term switch, on the MicroControl unit. If the cable length between the Senquip and the MicroControl device is long, or communications are unstable, one should be fitted to the Senquip device as well.

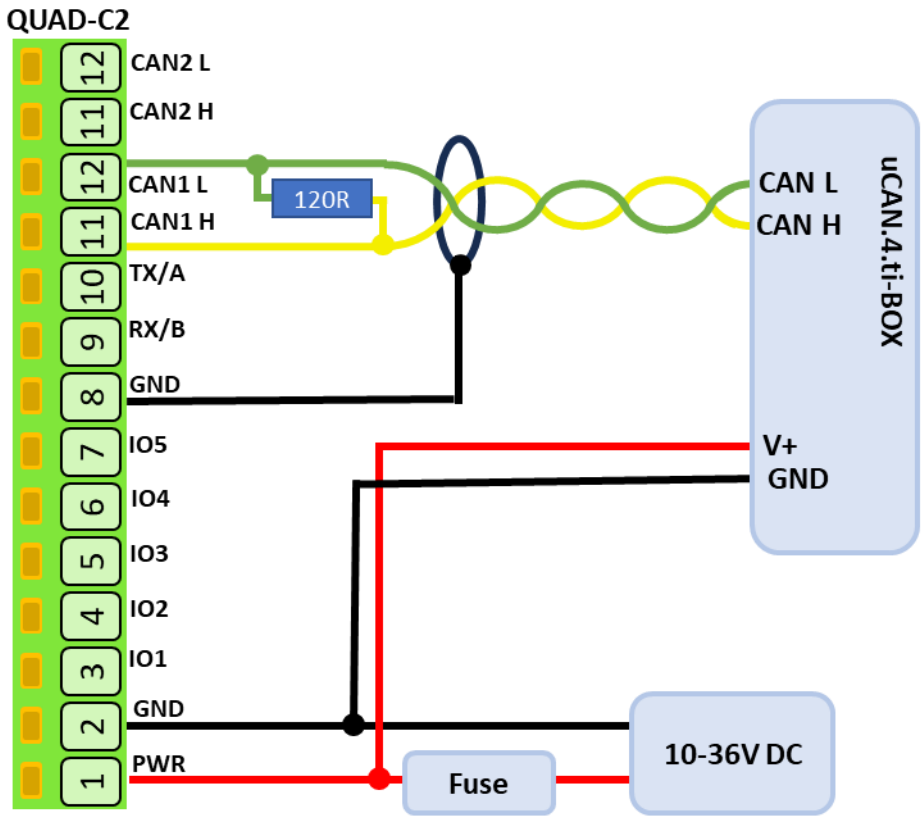


Figure 3 - Senquip QUAD to MicroControl uCAN.4.ti-BOX Wiring

4. Senquip Device Configuration

The Senquip device is setup with a *base interval* of 5 seconds. The CAN port is configured for a bit rate of 250kbps and is set to scan for incoming messages for the full base interval of 5 seconds. The CAN peripheral will need to be able to transmit on to the bus to issue START and SYNC commands. No filters are used as very few messages are expected from the MicroControl device. Raw CAN messages are set to transmit to the Senquip Portal for debug purposes, this can be turned off later.

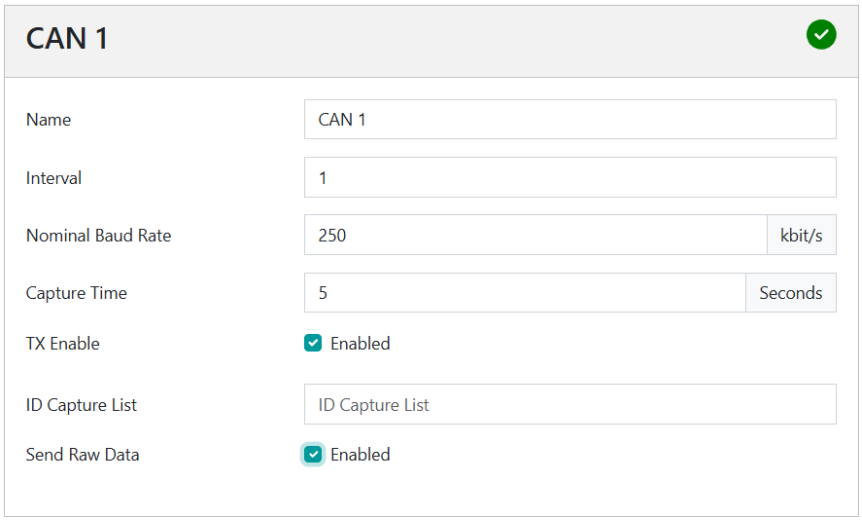


Figure 4 - Senquip Device CAN Configuration

5. The Device Script

This script enables a Senquip device to communicate with a CANopen thermocouple measurement module, retrieve four temperature values, and dispatch them for remote monitoring via the Senquip Portal or other endpoints.

Firs, we load the required libraries and create some constants related to:

- NODE_ID: CANopen node address of the MicroControl temperature acquisition device.
- NMT_COB_ID: COB-ID for Network Management (always 0x00).
- SYNC_COB_ID: COB-ID for SYNC messages (always 0x80).
- PDO_COB_ID: Expected COB-ID of the Process Data Object (PDO) from the device. Calculated as 0x180 + NODE_ID.
- NMT_START: Start command for NMT (value 0x01).

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

let NODE_ID   = 0x01; // MicroControl device address
let NMT_COB_ID = 0x00;
let SYNC_COB_ID = 0x80;
let PDO_COB_ID  = 0x180 + NODE_ID;
let NMT_START = 0x01;
```

We then issue 2 CAN send commands. The first one sends the Start Node command to put the MicroControl device into operational mode where it can handle PDO-communication. The second is a repeating send that sends the SYNC command every 2.5 seconds. The MicroControl device should then respond every 2.5 seconds with a PDO message containing the thermocouple data. Note the use of the SQ.encode function to convert a number into an 8 bit string.

```
CAN.tx(1, NMT_COB_ID, SQ.encode(NMT_START,SQ.U8) + SQ.encode(NODE_ID,SQ.U8), 2, CAN.STD); // Start
CANopen node 1
CAN.tx(1, SYNC_COB_ID, "", 0, CAN.STD + CAN.TX_SLOT(0),2500); // Send the SYNC message every 2.5 sec
```

SQ.set_data_handler registers an inline function that will be called after each base interval. This function is passed all measured data including CAN messages.

In the handler function, if CAN data exists, each message ID is checked to see if it is a PDO message from the MicroControl device. If it is, the 4 thermocouple values are extracted and are dispatched to the Senquip Portal.

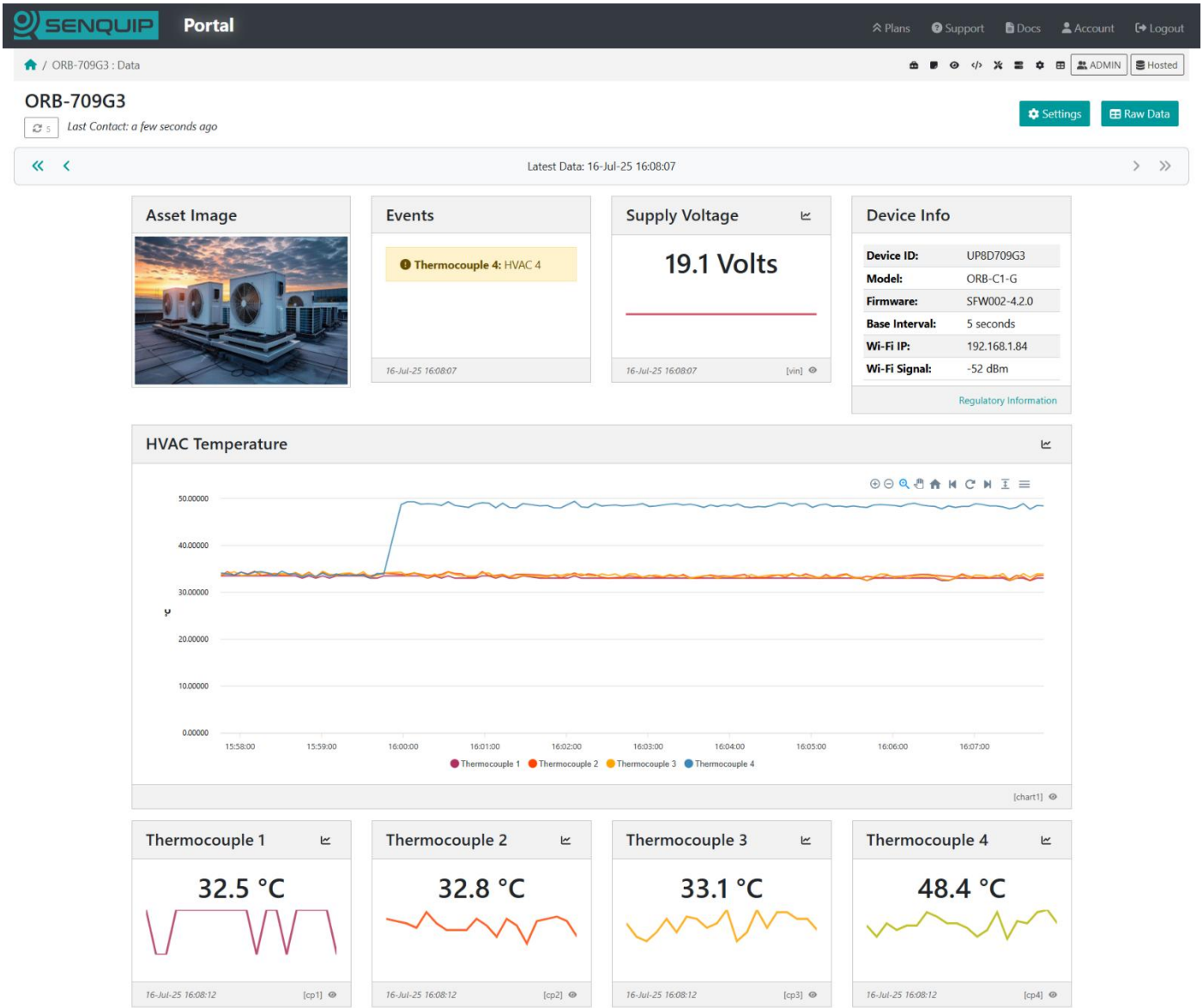
The CAN data arrives as a text string. The SQ.parse function extracts bytes from the string and interprets them as numbers. For instance, looking at tc1, we start at byte 0 in the string and extract 4 characters (for example “abcd”). The function interprets the text as a hex number 0xabcd and reverses the order because it is little endian to become 0xcdab and then interprets it as a signed number. Finally, the extracted value is divided by 10 to reveal the temperature in °C.

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

  if (typeof obj.can1 !== "undefined") {
    for (let i = 0; i < obj.can1.length; i++) {
      if (obj.can1[i].id === PDO_COB_ID) {

        let tc1 = SQ.parse(obj.can1[i].data, 0,  4, -16, SQ.S16) / 10;
        let tc2 = SQ.parse(obj.can1[i].data, 4,  4, -16, SQ.S16) / 10;
        let tc3 = SQ.parse(obj.can1[i].data, 8,  4, -16, SQ.S16) / 10;
        let tc4 = SQ.parse(obj.can1[i].data, 12, 4, -16, SQ.S16) / 10;

        SQ.dispatch(1, tc1);
        SQ.dispatch(2, tc2);
        SQ.dispatch(3, tc3);
        SQ.dispatch(4, tc4);
      }
    }
  }
}, null);
```

Finally, four custom parameters are defined with a descriptive name and unit:

6. Conclusions

This script demonstrates how a Senquip device can be used to interface with a CANopen temperature module, initiating communication, synchronising data flow, and decoding multiple temperature values in real-time. By leveraging the built-in Senquip scripting capabilities, users can monitor critical thermal parameters from remote locations with minimal configuration. The solution is flexible, efficient, and ideal for industrial environments where reliable temperature data is essential for performance and safety.



7. Appendix A – Full Application Script

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

let NODE_ID = 0x7F; // MicroControl device address
let NMT_COB_ID = 0x00;
let SYNC_COB_ID = 0x80;
let PDO_COB_ID = 0x180 + NODE_ID;
let NMT_START = 0x01;

CAN.tx(1, NMT_COB_ID, SQ.encode(NMT_START,SQ.U8) + SQ.encode(NODE_ID,SQ.U8), 2, CAN.STD); // Start CANopen
node 1
CAN.tx(1, SYNC_COB_ID, "", 0, CAN.STD + CAN.TX_SLOT(0),2500); // Send the SYNC message every 2.5 sec

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);
  if (typeof obj.can1 !== "undefined") {
    for (let i = 0; i < obj.can1.length; i++) {

      if (obj.can1[i].id === PDO_COB_ID) {
        let tc1 = SQ.parse(obj.can1[i].data, 0, 4, -16, SQ.S16) / 10;
        let tc2 = SQ.parse(obj.can1[i].data, 4, 4, -16, SQ.S16) / 10;
        let tc3 = SQ.parse(obj.can1[i].data, 8, 4, -16, SQ.S16) / 10;
        let tc4 = SQ.parse(obj.can1[i].data, 12, 4,-16, SQ.S16) / 10;

        // -437 is a fault
        if (tc1 > -436){SQ.dispatch(1, tc1);} else {SQ.dispatch(1, "fault");
SQ.dispatch_event(1,SQ.WARNING,"check sensor");}
        if (tc2 > -436){SQ.dispatch(2, tc2);} else {SQ.dispatch(2, "fault");
SQ.dispatch_event(2,SQ.WARNING,"check sensor");}
        if (tc3 > -436){SQ.dispatch(3, tc3);} else {SQ.dispatch(3, "fault");
SQ.dispatch_event(3,SQ.WARNING,"check sensor");}
        if (tc4 > -436){SQ.dispatch(4, tc4);} else {SQ.dispatch(4, "fault");
SQ.dispatch_event(4,SQ.WARNING,"check sensor");}
      }
    }
  }
}, null);
```