Table of Contents

Error Messages Overview

Quick Reference Table

Mission Planner Output	Log File	Error Type
sensor disconnected, tube: #	Sensor Disconnected (sensor fail)	Sensor failure/disconnected
Error when selecting tube #	Failed to switch channel (multiplexer fail)	Multiplexer failure/disconnected

Explination

When using the script, if there is an error with the sensor array, you will see an error message in the Mission Planner output, and in the error column in the data from the sensor in the BIN file.

There are two main errors you will see, a sensor failure error, and a multiplexer failure error.

A sensor failure is when one, or more of the pressure sensors is not responding. This could mean that they are disconnected from the multiplexer, or there could be an issue with that specific channel on the multiplexer.

If there is a sensor error, you would see a message similar to the one below in Mission Planner.

```
6/11/2024 11:02:42 AM : sensor disconnected, tube: 2
```

In the log file you would see the message `"Sensor Disconnected (sensor fail)"`.

A multiplexer failure is when the script cannot communicate with the multiplexer to switch the selected channel. This usually means the multiplexer is either disconnected or simply not working.

If there is a multiplexer error, you would see a similar message in Mission Planner.

```
6/11/2024 11:02:42 AM : Error when selecting tube 2
```

It is important to note that this error can also occur when channel number that is passed into the function is not within the valid range of channels on the multiplexer. So if you experience this error and have modified the script, make sure that the channel number you are passing in is a valid channel.

In the log file, if you are having a multiplexer failure, you will see the message `"Failed to switch channel (multiplexer fail)"` in the error column of the specific channel

Code Overview

Global Values and Initialization

```
-- list for the log data from the sensors
local log_data_list = {}

-- list for errors when reading channels of multiplexer
local error_list = {}
```

The `log_data_list` table is a table to hold the data from each of the pressure sensors connected to the multiplexer.

The `error_list` table is table to store whether a channel on the multiplexer, (which corresponds to one of the sensors) is not responding.

Both of these tables will be the length of the amount of sensors we have connected, which normally will be 5.

```
-- init i2c bus
-- Get interface at bus 0 (first I2C bus) and set device address to 0x0
local I2C_BUS = i2c:get_device(0, 0)

-- set the number of retries to 10
I2C_BUS:set_retries(10)
gcs:send_text(7, "i2c_tca Script Started!")
```

We start by checking the I^2C lines for an available bus, and store our interface with this I^2C in the `i2c_bus` variable. We set an amount of retries in the off-chance that the autopilot does not detected it immediately.

Once we get a connection to the I²C bus, we then send a debug message to Mission Planner.

```
-- shared address of the sensors
local SENSOR_ADDR = 0x28

-- 0x70 is default, to change, set or reset A0, A1, A2 on the
    multiplexer
local TCA_ADDRESS = 0x70

-- table of which channels on the multiplexer are being used
local CHANNEL_NUMBERS = {#, #, #, #, #}

-- error type table
local ERROR_LIST = {
  "Sensor Disconnected (sensor fail)",            -- 1
  "Failed to switch channel (multiplexer fail)" -- 2
}
```

We store the shared sensor address in the `SENSOR_ADDR`, this prevent us from having to use a unnamed constant when referring to the sensors address later in the script. This also reduces the amount of changes needed to change to a different sensor with a different address.

The `TCA_ADDRESSES` variable stores the address of the multiplexer (TCA). Again, this prevents us from having to refer to the multiplexer's address with an unnamed constant

The `CHANNEL_NUMBERS` table stores the channels on the multiplexer that we want to read data from. These can range from 0-7, corresponding to the channels on the multiplexer. These can be set in any order if one wishes to have specific sensors log to specific columns in the BIN files. In the code snippet above, the channel numbers are replaces with "#" as a placeholder.

The `ERROR_LIST` table hold the errors that can occur. This is mainly used when calling the `log_channel_error()` function to specify the type of error message we would like to log to the log file.

TCA Channel Selecting

```
-- set the current channel on the TCA
local function tcaselect(channel)
  -- set multiplexer address
  I2C_BUS:set_address(TCA_ADDRESS)

  -- make sure channel value passed through is between 0-7
  if (channel > 7) or (channel < 0) then
    return false
  end

  -- set/open the correct channel
```

```
  return (I2C_BUS:write_register(TCA_ADDRESS, 1 << channel))
end
```

The `tcaselect()` function is responsible for telling the multiplexer what channel it should be listening to.

We start by setting what I²C device address we are going to read and write from to the address of the multiplexer.

After that we check the channel number that was passed in and make sure it is withing the range of channels on the multiplexer, which is 0-7, if not we return false.

To select the channel, we write data to a register on the multiplexer. We use the `write_register()` method with the multiplexer address and the number one, bitwise left shifted by the number of the channel we select.

By left shifting the number one by the channel number, we send a binary number with only one bit set to one. The position of that bit specifies which channel we would like to listen to.

```
          7654 3210
1 << 0 = 0000 0001 <- channel 0
1 << 1 = 0000 0010 <- channel 1
1 << 2 = 0000 0100 <- channel 2
1 << 3 = 0000 1000 <- channel 3
1 << 4 = 0001 0000 <- channel 4
1 << 5 = 0010 0000 <- channel 5
1 << 6 = 0100 0000 <- channel 6
1 << 7 = 1000 0000 <- channel 7
```

Above we can see a chart of what each operation looks like to select each channel.

We then take the return value of `write_register()`, which is a boolean, and return it for the caller to handle.

Logging

Logging Data To Bin

```
local function log_data()
  logger:write('PRBE','tube1,tube2,tube3,tube4,tube5,err1,err2,err3,
     err4,err5','NNNNNNNNNN',
               log_data_list[1],
               log_data_list[2],
               log_data_list[3],
               log_data_list[4],
               log_data_list[5],
               error_list[1],
```

```
                error_list[2],
                error_list[3],
                error_list[4],
                error_list[5])
end
```

This function takes the data that takes the date we have collected from the pressure sensors, and any errors that we detected while collecting this data, and logs it to the BIN file of the autopilot.

The `logger:write()` method take several arguments to define the various parameters that go into the log file.

The first argument, `'PRBE'`, is the section name for the data we are going to log in the file. This name has to be at most 4 characters, and cannot be the same as any other section name that ArduPilot logs. The second argument, `'tube1,tube2,tube3,tube4,tube5,err1,err2,err3,err4,err5'`, specifies the name of each piece of data logged. These labels are stored under the section name in the log file, in total these names cannot exceed 64 characters.

The third argument, '`NNNNNNNNNN`', specifies the type of each label. In this case '`N`', specifies a `char[16]`, which is a string of a maximum of 16 characters.

Once we specify the parameters for the data that is going to be logged, we then pass in the data we would like to log in the file. In this case, we use the 5 elements in the `log_data_list` table for the channel data, and the 5 elements in `error_list` for the errors for each channel. The pressure data is the data that is reported from the sensor, and is normalized to [-2, 2] in H_2O. The errors simply log "`NORMAL`" or "`ERROR`" depending on the state of the channel at the time the data is recording.

Logging Errors

```
-- write an error to the channel that is experience an error
local function log_channel_error(channel_index, error_type)
  log_data_list[channel_index] = "0"
  error_list[channel_index] = error_type
end
```

This function logs an error for the channel index that is specified. It simply sets the data value to zero and places the `error_type` string, which should be from the `ERROR_LIST` table into the error list to be logged.

This function is called whenever there is an issue with specific channel on the multiplexer, primarily if there is a connection issue where no data is read from the sensor.

Update

```lua
function update()
  for key, value in pairs(CHANNEL_NUMBERS) do

    -- select channel i on TCA
    if not (tcaselect(value)) then
      gcs:send_text(0, "Error when selecting tube " .. tostring(key))
      log_channel_error(key)
    else
```

For the main loop in the script, we start by iterating through the list of channels in `CHANNEL_NUMBERS`. We tell the TCA to switch to channel `i` with the `tcaselect()` function. If `tcaselect()` returns false meaning we called a channel that does not exist on the multiplexer, or that we failed to switch the channel on the multiplexer, we then send an error message to the Mission Planner output, specifying which channel is invalid, and call the `log_channel_error()` function. We then skip the rest of the loop and start on the next iteration

```lua
-- open the address of the sensor
I2C_BUS:set_address(SENSOR_ADDR)

-- read_registers(begin at register, number of bytes to read)
local returnTable = I2C_BUS:read_registers(0, 2)

-- if there is no i2c device connected (or no data is read in general) log it as an error
if (returnTable == nil) then
  gcs:send_text(0, "sensor disconnected, " .. " tube: " .. tostring(key))
  log_channel_error(key)
```

If we successfully switch the channel on the multiplexer, we can continue to read data from the sensors. We set the sensor address we are going to read from, since `tcaselect()` sets that to the TCA's address to select the channel.

We then read two bytes from the I^{2}C bus with the `read_registers()` method.

The two arguments in `read_registers()` define the offset (in our case `0`), and how many bytes we would like to read (which is 2 in our case).

`read_registers()` returns a table with the bytes we read from the I^{2}C bus. We store this table in the `returnTable` variable.

We first check if `returnTable` is empty or `nil`, if it is empty, this means that `read_registers()` did not receive any data from the I^{2}C bus. This is most likely caused by the sensor on that channel being disconnected, or the data and clock lines of the I^{2}C bus are experiencing a lot of noise.

If this is the case, we send an error message to Mission Planner saying that the sensor on channel `i` is disconnected. We then log an error and skip the rest of the loop and start on the next iteration.

```lua
    else
      -- format data to remove first 2 bits
      local msg = (returnTable[1] << 8 | returnTable[2]) & 0x3FFF

      -- normalize data to [-2 2] in inH2O and make the datatype string
      -- math is ((range*data)/max(data) - 2)
      local normalized_data = tostring((4.0 * msg) / 0x3FFF - 2)
      -- add the data to the list
      log_data_list[key] = normalized_data
      error_list[key] = "NORMAL"
    end
  end
end
```

If we get data from the I^{2}C bus, we then can process it. In the table below we can see that the pressure data is stored in bits 29-16. Since this is 14 bits in total, we need to read two bytes from the bus, which is 16 bits.

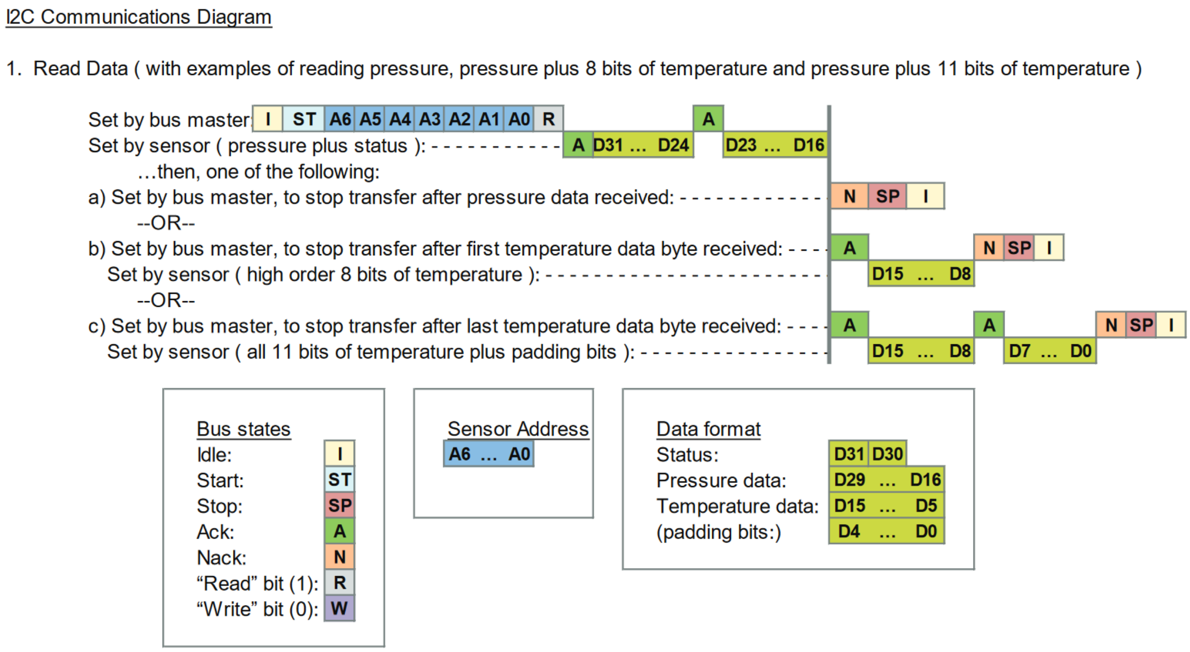

Figure 1: Diagram of the data sent by the pressure sensors

After we get the two bytes from the bus, we need to take the bytes in `returnTable` and reconstruct the whole number from them. We do this by performing a bitwise left-shift 8 times. Doing this gives us room to place the second byte of the data at the end by performing an OR operation. Below is an example of what is happening. (Note this data is random and not representative of what data is sent by the sensors)

```
1110 1101 << 8 = 1110 1101 0000 0000
1110 1101 0000 0000 | 0011 0110 = 1110 1101 0011 0110
```

The above operations essentially take the two bytes stored and place them in the correct order into a singular number.

Since we do not need the first two bits of the data from the I^2C bus we can perform a bitwise operation on the data. In our case we will AND the data with the hexadecimal value `0x3FFF`.

For example, we have the below data (note this data is random and not representative of what data is sent by the sensors).

```
1110 1101 0011 0110
```

Since we want to remove the first two bits of the data, we will AND it with `0x3FFF`, which is represented in binary below.

```
0011 1111 1111 1111
```

Once we perform the AND operation with `0x3FFF`, as can be seen below, we preserve the pressure data but remove the unnecessary data that we do not want to interpret.

```
  1110 1101 0011 0110
& 0011 1111 1111 1111
---------------------
= 0010 1101 0011 0110
```

Once we have formatted our data, we can now normalize the data. According to the sensors data sheet, the range of the sensors is [-2, 2] in H_2O.

The formula for this normalization can be seen below

$$\frac{range \cdot data}{\max(data) - 2}$$

In our case the maximum of our data is `0x3FFF`, which is a number where all 14 bits are set to one.

After we have normalized our data we then convert it to a string to be stored in our `log_data_list` table. Here since we have not hit any errors up until this point, we will also set the error for channel `i` to "`NORMAL`", since there are no errors to log

Once we have gone through each channel and logged their data (or their errors if they have any), we get out of the for loop and get to the following code snippet.

```
  log_data()

```

```
  -- send_text(priority level (7 is Debug), text is formed dynamically
     from the function)
  gcs:send_text(7, form_message())

  -- reset everything for the next loop
  I2C_BUS:set_address(0x00)
  log_data_list = {}
  error_list = {}
  return update, 50 -- reschedules the loop every 50ms (20hz)
end
```

First we call the `log_data()` function, which takes the data we have placed into the `log_data_list` and `error_list` tables and logs their data to the BIN file.

Then we can send the data we have collected to the Mission Planner output. This is optional but is helpful to verify the sensors are sending logical data. The message that we send to Mission Planner comes from the `form_message()` function.

```
-- dynamically create the message that gets reported to mission planner
-- prevents us from having to manually change the message form every
   time we add
-- or remove sensors or decide to change the format of the message
local function form_message()
  local message = ""
  for key, value in pairs(CHANNEL_NUMBERS) do
    message = message .. string.format(key) .. string.format(": %.2f ",
        log_data_list[key])
  end
  return message
end
```

Here we

Then we can send the data we have collected to the Mission Planner output. This is optional but is helpful to verify the sensors are sending logical data. The above message assumes that there are 5 sensors connected, but this can be modified for other configurations.

We then set the address of the I^{2}C device we are reading to zero to prepare for the next iteration of the `update()` function. We then return the function, and schedule the `update()` function to run again in 50 milliseconds.