Table of Contents

- Code Overview
  - Global Values and Initialization
  - TCA Channel Selecting
  - Logging
  - Update

Code Overview

Global Values and Initialization

```lua
-- 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.

```lua
-- init i2c bus
local i2c_bus = i2c:get_device(0, 0)
i2c_bus:set_retries(10)
gcs:send_text(7, "i2c_tca Script Started!")
```

We start by checking the I^{2}C lines for an available bus, and store our interface with this I^{2}C 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 I^{2}C bus, we then send a debug message to Mission Planner.

```lua
-- var for address of the sensors
local SENSOR_ADDR = 0x28

-- var for list of which channels on the multiplexer are connected
local CHANNEL_NUMBERS = {#, #, #, #, #}
```

```lua

-- for each TCA9548A, add an entry with its address
-- 0x70 is default, to add more set or reset A0, A1, A2
TCA_ADDRESSES = {0x70}
```

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 `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 `TCA_ADDRESSES` table stores the address of the mulitplexer (TCA). Currently this has only one TCA address, as we only plan to connect one mulitplexer to each autopilot. You can add more addresses to the table, and read data from multiple multiplexers (along with minor modification to the script).

TCA Channel Selecting

```lua
-- opens the channel to the designated TCA module
function tcaselect(tca, channel)
  -- verify that tca index passed through is valid
  if (tca > #TCA_ADDRESSES) or (tca < 0) then
    return false
  end

  -- choose multiplexer from array
  i2c_bus:set_address(TCA_ADDRESSES[tca])

  -- 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
  -- i2c_bus:write_register(0x70, 1 << channel)
  i2c_bus:write_register(TCA_ADDRESSES[tca], 1 << channel)
  return true
end
```

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

We start by setting what I^{2}C device address we are going to read and write from to the address of the multiplexer. We first check that the TCA index then we set the address, if it is not a valid address, we return false for the caller to handle.

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.

Once we select the channel on the TCA, we then return true for the caller to handle.

Logging

Logging Data To Bin

```
function log_data()
  logger:write('SENS','s1,s2,s3,s4,s5,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])
```

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, `'SENS'`, 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, `'s1,s2,s3,s4,s5,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.

Here we have to major sections of data, the actual data collected and processed by the sensor, and if there were any errors collecting the data from that channel on the multiplexer. 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

```lua
function log_channel_error(channel_index)
  log_data_list[channel_index] = "0"
  error_list[channel_index] = "ERROR"
end
```

This function logs an error for the channel index that is specified. It simply sets the data value to zero and places the word "`ERROR`" 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 TCA module 1, and channel i
    if not (tcaselect(1, value)) then
      gcs:send_text(0, "Called TCA channel " .. tostring(value) .. ",
          which does not exist")
      log_channel_error(key)
```

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, 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
else
  -- once open use the address of the sensor
  i2c_bus:set_address(SENSOR_ADDR)
  -- read_registers(begin at register, number of bytes to read)
  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, "returnTable val nil," .. " disconn sensor," .. "
        channel: " .. string.format("%d", value))
    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
      -- output data to MP Messages
      -- format data to remove first 2 bits
      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)
      normalized_data = tostring((4.0 * msg) / 0x3FFF - 2)
      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.

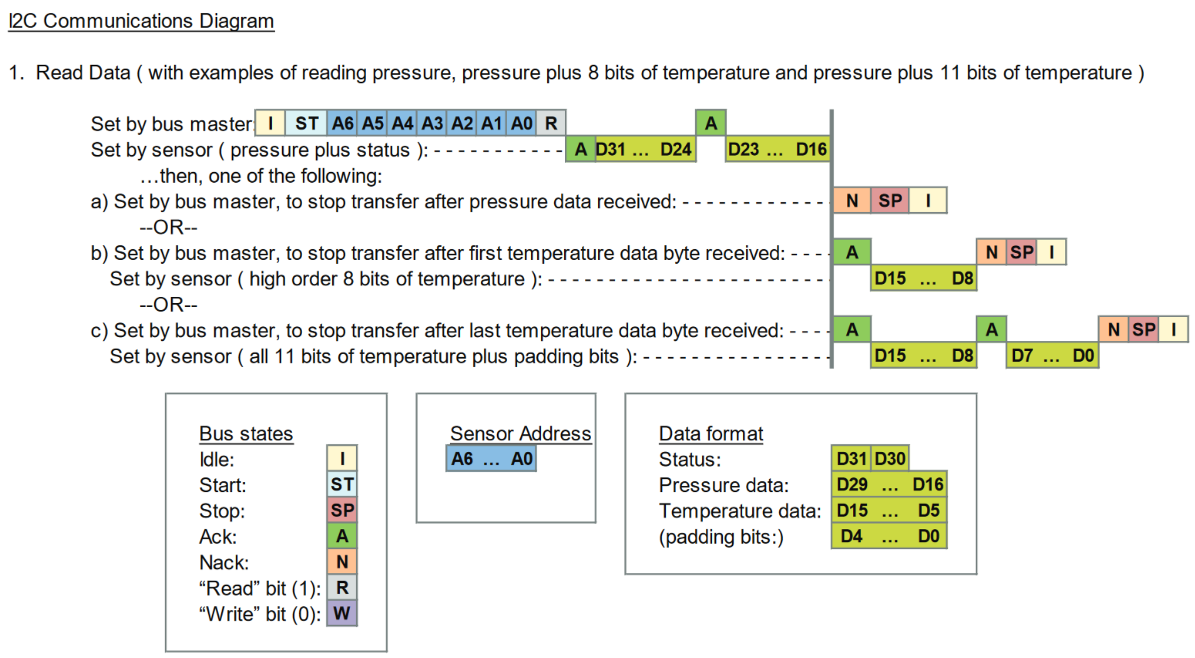
I2C Communications Diagram
1. Read Data (with examples of reading pressure, pressure plus 8 bits of temperature and pressure plus 11 bits of temperature)
Set by bus master
I ST A6 A5 A4 A3 A2 A1 A0 R
A
Set by sensor (pressure plus status):
A D31 … D24
D23 … D16
…then, one of the following:
a) Set by bus master, to stop transfer after pressure data received:
N SP I
--OR--
b) Set by bus master, to stop transfer after first temperature data byte received:
A
N SP I
Set by sensor (high order 8 bits of temperature):
D15 … D8
--OR--
c) Set by bus master, to stop transfer after last temperature data byte received:
A
A
N SP I
Set by sensor (all 11 bits of temperature plus padding bits):
D15 … D8
D7 … D0
Bus states
Idle: I
Start: ST
Stop: SP
Ack: A
Nack: N
"Read" bit (1): R
"Write" bit (0): W
Sensor Address
A6 … A0
Data format
Status: D31 D30
Pressure data: D29 … D16
Temperature data: D15 … D5
(padding bits:) D4 … D0

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^{2}C 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 as a string formatted
     to float)
  -- report data to misson planner output
  gcs:send_text(7, "chan " .. string.format("%d: %.3f | ",
     CHANNEL_NUMBERS[1], log_data_list[1]) ..
                 "chan " .. string.format("%d: %.3f | ",
                    CHANNEL_NUMBERS[2], log_data_list[2]) ..
                 "chan " .. string.format("%d: %.3f | ",
                    CHANNEL_NUMBERS[3], log_data_list[3]) ..
                 "chan " .. string.format("%d: %.3f | ",
                    CHANNEL_NUMBERS[4], log_data_list[4]) ..
                 "chan " .. string.format("%d: %.3f ",
                    CHANNEL_NUMBERS[5], log_data_list[5])

  )

  i2c_bus:set_address(0x00)
  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 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.