Table of Contents

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

Overview

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
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 I²C lines for an available bus, and store our interface with this I²C 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.

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

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

-- 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

```
-- 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
```

Logging

Logging Data To Bin

```
function log_data()
  -- care must be taken when selecting a name, must be less than four characters and not clash with an existing log type
  -- format characters specify the type of variable to be logged, see AP_Logger/README.md
  -- https://github.com/ArduPilot/ardupilot/tree/master/libraries/AP_Logger
  -- not all format types are supported by scripting only: i, L, e, f, n, M, B, I, E, and N
  -- Data MUST be integer|number|uint32_t_ud|string , type to match format string
  -- lua automatically adds a timestamp in micro seconds
  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

```
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

```
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 Misson Planner output, specifiying 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

```
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 I²C bus with the `read_registers()` method, which returns a table with the bytes we read from the I²C 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 I²C bus. This is most likely caused by the sensor on that channel being disconnected, or the data and clock lines of the I²C 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.

```
    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
```

```
  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])
  )

  i2c_bus:set_address(0x00)
  return update, 50 -- reschedules the loop every 50ms (20hz)
end
```

Important Links