Daughter Board Documentation

Introduction

Goal

Designing a Daughter Board responsible for gathering data from multiple sensors and sending it to a Payload GCS.
This board is to be integrated on different type of platforms (eg MAV, VTOL, etc.).
This is why the board needs to be agnostic to the controller board (eg Pixhawk flight controller on a MAV).
The board needs to have its own way of communication with the payload GCS. The Payload GCS could have been integrated on an existing GCS (eg Mission Planner, ArduiPilot) but the main difficulty lies in the source code modification.
A dedicated GCS called Payload GCS has been developped for data visualization and recording.
This paper aims to explain the technical choices that have been made and provides all necessary information and tools to go on developping and adapting the project.
Eventually, a simple procedure is described to easily integrate new sensors to the Daughter Board.

I-Specifications

II-Daughter Board

The Daughter Board is a Rapsberry Pi 3 model B running the linux-based OS Raspbian. Below is a flow chart representing the Daughter Board:

II-1-Communication with the Payload GCS

The Payload GCS will need to send configuration messages to the Daughter Board. The Daughter Board will need to send messages containing data from the sesors to the Payload GCS.
As a matter of fact, we need duplex communication.

II-2-Sensors Overview and interface with Daughter Board

III-Payload GCS

IV-Procedure to add a Sensor

IV-1 On the Daughter Board side

To add a new sensor an executable responsible for needs to be produced
Create a folder for your sensor.
Create 2 files in it:
* main_YOUR_SENSOR.cpp
* YOUR_SENSOR.h
*

IV-2 On the GCS side

TUTORIAL: Add a new sensor

I- Prerequisites

Installing MAVLink
Follow this turorial to install MAVLink:

MAVLink Installation

II- DaughterBoard side

II-1- Writing the source code

In the “Sources” folder, create a new folder named YOUR_SENSOR (replace YOUR_SENSOR by the actual name of the sensor).
In this folder create 2 files: main_YOUR_SENSOR.cpp and YOUR_SENSOR.h.
The file YOUR_SENSOR.h consists in the YOUR_SENSOR class implementation.

Whatever the type of interface used for your sensor, the file main_YOUR_SENSOR.cpp will always have this skeleton:

```
#include "YOUR_SENSOR.h"

int main( int argc, char **argv ){
    YOUR_SENSOR sensor;      // Object of type YOUR_SENSOR described in YOUR_SENSOR.h header file.

    sensor.waitForConfig();  // Configuration coming from the payload GCS
    if( !sensor.isAdded() )  // Check if YOUR_SENSOR is part of the current configuration
        return EXIT_SUCCESS; // If it is not, finish the program
    sensor.init();           // If it is, initialize YOUR_SENSOR
    usleep( 1000000 );       // Wait for the YOUR_SENSOR interface to be ready

    while( 1 ){
        sensor.measure();                               // Get data from YOUR_SENSOR
        sensor.msgQueue.setYOUR_DATA( sensor.getYOUR_DATA() );  // Initialize the msgQueue with m
easured data
        sensor.msgQueue.sendData();                     // Write data to the msgQueue
    }

    return EXIT_SUCCESS;     // Never reached
}
```

NB: For example let’s assume that **YOUR_SENSOR** is a **Lidar**. You would in this case need to replace all occurence of **YOUR_SENSOR** by **Lidar** and all occurence of **YOUR_DATA** by **distance**.

Now a more difficult step is to implement **YOUR_SENSOR** class in the *YOUR_SENSOR.h* file.

Depending on the type of interface your sensor is using, the class design may be different.
There are three different types of interfaces already used: serial, i2c and analog. Let’s see how to implement each type of sensor.

II-1-a I2C sensor

If your sensor uses an i2c interface, copy-paste in YOUR_SENSOR folder the file named I2cSensor.h which you can find in the “Add_New_Sensor” folder.

This file defines an abstract class named **I2cSensor**. This class contains the methods used for initializing an i2c device, and for reading and writing to/from the i2c bus.

```
class I2CSensor {
public:
    const char* I2C_FILENAME = "/dev/i2c-1";
    I2CSensor();
    ~I2CSensor();
    int init( char i2cAddr );
    virtual void waitForConfig() = 0;
    virtual void measure() = 0;
    bool isAdded();
protected:
    int fd;  // file descriptor for /dev/i2c-1
    void i2cWrite( char* buf, int nbBytes );
    void i2cRead ( char* buf, int nbBytes );
    bool added;
};
```

Obviously, **YOUR_SENSOR** class needs to derivate from **I2cSensor** class to be able to use *i2cWrite()* and *i2cRead()* methods.

```
class YOUR_SENSOR : public I2cSensor{  // public derivation of I2cSensor class

public:
  const char* I2C_FILENAME = "/dev/i2c-1";

    //-------------------------------------------------
    // The register's address used to configure
    // and sense data. You will find them in the
    // datasheet of your sensor.  The below ones are
    // for the Lidar Garmin v3 sensor and you can find
    // the datasheet on this link: https://static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual
_and_Technical_Specifications.pdf
    //-------------------------------------------------
    const char LIDAR_I2C_ADDR      = 0x62;
    const char ACQ_COMMAND_REG     = 0x00;
    const char BIAS_CORREC_VAL     = 0x04;
    const char STATUS_REG          = 0x01;
    const char FULL_DELAY_HIGH_REG = 0x0F;
    const char FULL_DELAY_LOW_REG  = 0x10;
    //-------------------------------------------------
    YOUR_SENSOR();
    ~YOUR_SENSOR();
    void waitForConfig();
    void measure();
    Type getYOUR_DATA();
    YOUR_SENSORMsgQueue msgQueue;

private:
    Type YOUR_DATA;
};
```

In **YOUR_SENSOR** class you still have to implement the *waitForConfig()* and *measure()* methods. **YOUR_SENSOR** class also needs to contain one or more private variables for your data (*Type YOUR_DATA*) and the associated accessor(s) (*Type getYOUR_DATA()*).

Eventually, it will contain an instance of the **YOUR_SENSORMsgQueue** class which allows the communication with the **Handler** (described later).

Let's detail the methods.

The accessor(s) should only do one thing: return your data:

```
Type YOUR_SENSOR::getYOUR_DATA() {
    return YOUR_DATA;
}
```

The *waitForConfig()* method should always be like this:

```
void YOUR_SENSOR::waitForConfig() {
    while( msgQueue.receiveConfig() != RCV_SUCCESS );
    added = msgQueue.getSensorStatus();
}
```

And the *measure()* method is too dependant of your sensor to have a generic definition of it. However, it should use the *i2cRead()* and *i2cWrite()* methods of the **I2cSensor** base class. Below is the *measure()* definition for the **Lidar** class:

```
void Lidar::measure() {
    // Take distance measurement with receiver bias correction
    char buf[]={ ACQ_COMMAND_REG, BIAS_CORREC_VAL };
    i2cWrite( buf, 2 );

    // Wait until device is ready for new command
    char buf2[] = { STATUS_REG };
    i2cWrite( buf2, 1 );
    char tmp[] = { 1 };
    while( ( tmp[ 0 ] & 1 ) != 0 ) {
        i2cRead( tmp, 1 );
        usleep( 50000 );
    }

    // Read distance measurement high byte
    char hiVal[] = { FULL_DELAY_HIGH_REG };
    i2cWrite( hiVal, 1 );
    i2cRead ( hiVal, 1 );

    // Read distance measurement low byte
    char loVal[] = { FULL_DELAY_LOW_REG };
    i2cWrite( loVal, 1 );
    i2cRead ( loVal, 1 );

    // Build distance word with high and low bytes
    distance = hiVal[ 0 ];
    distance <<= 8;
    distance += loVal[ 0 ];
}
```

You can inspire from this to write your own *measure()* method.

You now have to create a class for communicating with the **Handler**. This class needs to

implement everything that is necessary to use Linux Inter-Processus Communication (IPC) message queues.
In YOUR_SENSOR folder, create a file named **YOUR_SENSORMsgQueue.h**.
Copy-paste this content in it:

```
#include <sys/msg.h>

class YOUR_SENSORMsgQueue {

public:
    const long DATA_MSG_TYPE   = 1;
    const long CONFIG_MSG_TYPE = 2;

    YOUR_SENSORMsgQueue();
    ~YOUR_SENSORMsgQueue();
    int sendConfig();
    int receiveConfig();
    int sendData();
    int receiveData();
    void setSensorStatus( uint8_t _sensorStatus );
    void setYOUR_DATA( Type _YOUR_DATA );
    uint8_t getSensorStatus();
    Type getYOUR_DATA();

private:
    key_t key;
    int msgid;
    struct _dataMessage {
        long type;
        Type YOUR_DATA;
    } dataMessage;
    int dataMsgLength;

    struct _configMessage {
        long type;
        uint8_t sensorStatus;
    } configMessage;

    int configMsgLength;
};


YOUR_SENSORMsgQueue::YOUR_SENSORMsgQueue() {
    key = 0x9999000X;
    if( ( msgid = msgget( key, 0600 | IPC_CREAT ) ) == -1 ) {
        perror( "msgget" );
    }
    dataMessage.type = DATA_MSG_TYPE;
    dataMessage.YOUR_DATA = 0;
    dataMsgLength = sizeof( dataMessage ) - sizeof( long );
    configMessage.type  = CONFIG_MSG_TYPE;
    configMsgLength = sizeof( configMessage ) - sizeof( long );
}

YOUR_SENSORMsgQueue::~YOUR_SENSORMsgQueue() {
    msgctl(msgid, IPC_RMID, NULL);   // Clear Msg Queue
}
```

```
int YOUR_SENSORMsgQueue::sendConfig() {
    if( msgsnd( msgid, &configMessage, configMsgLength, IPC_NOWAIT ) == -1 )
        return SND_FAILURE;
    else
        return SND_SUCCESS;
}

int YOUR_SENSORMsgQueue::receiveConfig() {
    if( msgrcv( msgid, &configMessage, configMsgLength, CONFIG_MSG_TYPE, IPC_NOWAIT ) == -1 ) {
        return RCV_FAILURE;
    }else {
        return RCV_SUCCESS;
    }
}

int YOUR_SENSORMsgQueue::sendData() {
    if( msgsnd( msgid, &dataMessage, dataMsgLength, IPC_NOWAIT ) == -1 )
        return SND_FAILURE;
    else
        return SND_SUCCESS;
}

int YOUR_SENSORMsgQueue::receiveData() {
    if( msgrcv( msgid, &dataMessage, dataMsgLength, DATA_MSG_TYPE, IPC_NOWAIT ) == -1 ) {
        return RCV_FAILURE;
    }else  {
        return RCV_SUCCESS;
    }
}

void YOUR_SENSORMsgQueue::setSensorStatus( uint8_t _sensorStatus ) {
    configMessage.sensorStatus = _sensorStatus;
}

void YOUR_SENSORMsgQueue::setYOUR_DATA( Type _YOUR_DATA ) {
    dataMessage.YOUR_DATA = _YOUR_DATA;
}

uint8_t YOUR_SENSORMsgQueue::getSensorStatus() {
    return configMessage.sensorStatus;
}

Type YOUR_SENSORMsgQueue::getYOUR_DATA() {
    return dataMessage.YOUR_DATA;
}
```

Replace every occurences of YOUR_SENSOR by the actual name of your sensor as well as YOUR_DATA by the name of the data your sensor is measuring.

In the constructor of the class there is this line:

```
        key = 0x9999000X;
```

Replace the “X” by the total number of sensors that are used in your configuration.
This key is used to get a unique id for the message queue used to communicate with the **Handler**. It needs to be different from the other already existing message queues.
If you are not sure you can check the other SENSORMsgQueue.h files and choose a different value.

II-1-b Serial sensor

If your sensor uses a Serial interface, copy-paste in YOUR_SENSOR folder the files named *SerialSensor.h* and *serial_port.h* which you can find in the "Add_New_Sensor" folder.

This file defines an abstract class named **SerialSensor**. This class contains the methods used for initializing a serial interface and contains an instance of the **Serial_Port** class. The **Serial_Port** class contains the methods for reading and writing from and to a serial interface.

```
class SerialSensor {

public:
    SerialSensor();
    ~SerialSensor();
    void init();
    bool isAdded();
    virtual void waitForConfig() = 0;
    virtual void measure()       = 0;

protected:
    virtual void extractData();
    Serial_Port* serial;
    string port;
    int baudRate;
    bool added;
};
```

Obviously, **YOUR_SENSOR** class needs to derivate from **SerialSensor** class to be able to use the *init()* method and the *read_port()* and *write_port()* methods (from the **Serial_Port** class).

```
class YOUR_SENSOR : public SerialSensor {

public:
    YOUR_SENSOR();
    ~YOUR_SENSOR();
    void waitForConfig();
    void measure();
    float getYOUR_DATA();
    YOUR_SENSORMsgQueue msgQueue;

private:
    void extractData();
    Type YOUR_DATA;
    string buf;
    char byte;
};
```

In **YOUR_SENSOR** class you still have to implement the *waitForConfig()*, *measure()* and *extractData()*methods.
YOUR_SENSOR class also needs to contain one or more private variables for your data (*Type YOUR_DATA*) and the associated accessor(s) (*Type getYOUR_DATA()*).
It will contain an instance of the **YOUR_SENSORMsgQueue** class which allows the communication with the **Handler** (described later).
Eventually, it will contain a buffer (*string buf*) and a byte (*char byte*). These two last variables are necessary to extract data from the serial interface.

Let’s detail the methods.

The accessor(s) should only do one thing: return your data:

```
Type YOUR_SENSOR::getYOUR_DATA() {
    return YOUR_DATA;
}
```

The *waitForConfig()* method should always be like this:

```
void Pyranometer::waitForConfig() {
    while( msgQueue.receiveConfig() != EXIT_SUCCESS );
    port = "/dev/ttyUSB" + to_string( msgQueue.getSerialPortNum() );
    baudRate = XXXXX;
    added = msgQueue.getSensorStatus();
}
```

Replace the “XXXXX” by the baudrate value that your sensor is using.

If the output for your sensor is composed of lines ended by the “\r\n” escape sequence the *measure()* method should look like this:

```
void YOUR_SENSOR::measure() {
    byte = '\0';
    while( byte != '\r' ) {
        if( serial->read_port( &byte ) != 1 ) {![](/home/ground/Pictures/FT205out.png)
            cout << "Error reading from YOUR_SENSOR port " << port << endl;
        }
        buf += byte;
    }
    byte = '\0';
    buf = buf.substr( 1 );
    extractData();
}
```

You can inspire from this to write your own *measure()* method if your escape sequence is different.

You can see that the *extractData()* method is called at the end of the *measure()* method. Its purpose is to parse the “buf” buffer in order to retrieve the relevant data. It is too sensor specific to have a generic definition of it.
However, you will find below the *extractData()* method implemented for the FT205 wind sensor.
And here is a typical string that this sensor outputs:

$WI,WVP=000.1,313,0*70

In red this is the string corresponding to the wind speed value and in blue the wind angle value.

```
void WindSensor::extractData() {
    stringstream ss(buf);
    int count = 0;
```

```
    while( ss.good() )
    {
        count++;
        string substr;
        getline( ss, substr, ',' );
        if( count == 2 ) {
            try {
                windSpeed = stof( substr.substr( 4 ) );
            }
            catch ( const std::invalid_argument& ia ) {
                std::cerr << "Invalid argument: " << ia.what() << '\n';
            }
        }
        if( count == 3 ){
            try {
                angle = (uint16_t)stoi( substr );
            }
            catch ( const std::invalid_argument& ia ) {
                std::cerr << "Invalid argument: " << ia.what() << '\n';
            }
        }
    buf.clear();
}
```

NB: You will need to include the header to use *stringstream* type.
The *getline()* function is very useful to parse a string seperated by a specific character (',' in this case).
The *try ... catch* block protects the program from being killed by an "invalid_argument" thrown by *stoi()/stof()* functions if they receive an inappropriate argument (which can happen if a transmission error occurs on the serial interface).

Here is another example for the *measure()* implementation if the sensor output string is separated by spaces:

```
void YOUR_SENSOR::extractData() {
    string NotAData;
        ss >> DATA1 >> DATA2 >> NotAData >> DATA3;
}
```

You now have to create a class for communicating with the **Handler**. This class needs to implement everything that is necessary to use Linux Inter-Processus Communication (IPC) message queues.
In YOUR_SENSOR folder, create a file named **YOUR_SENSORMsgQueue.h**.
Copy-paste this content in it:

```
#include <sys/msg.h>

class YOUR_SENSORMsgQueue {

public:
    const long DATA_MSG_TYPE   = 1;
    const long CONFIG_MSG_TYPE = 2;

    YOUR_SENSORMsgQueue();
    ~YOUR_SENSORMsgQueue();
```

```
    int sendConfig();
    int receiveConfig();
    int sendData();
    int receiveData();
    void setSensorStatus( uint8_t _sensorStatus );
    void setSerialPortNum( uint8_t _num );
    void setYOUR_DATA( Type _YOUR_DATA );
    uint8_t getSensorStatus();
    int getSerialPortNum();
    Type getYOUR_DATA();

private:
    uint8_t serialPortNum;

    key_t key;
    int msgid;
    struct _dataMessage {
        long type;
        Type YOUR_DATA;
    } dataMessage;
    int dataMsgLength;

    struct _configMessage {
        long type;
        uint8_t sensorStatus;
        uint8_t serialPortNum;
    } configMessage;

    int configMsgLength;
};


YOUR_SENSORMsgQueue::YOUR_SENSORMsgQueue() {
    key = 0x9999000X;
    if( ( msgid = msgget( key, 0600 | IPC_CREAT ) ) == -1 ) {
        perror( "msgget" );
    }
    dataMessage.type = DATA_MSG_TYPE;
    dataMessage.YOUR_DATA = 0;
    dataMsgLength = sizeof( dataMessage ) - sizeof( long );
    configMessage.type  = CONFIG_MSG_TYPE;
    configMsgLength = sizeof( configMessage ) - sizeof( long );
}

YOUR_SENSORMsgQueue::~YOUR_SENSORMsgQueue() {
    msgctl(msgid, IPC_RMID, NULL);   // Clear Msg Queue
}

int YOUR_SENSORMsgQueue::sendConfig() {
    if( msgsnd( msgid, &configMessage, configMsgLength, IPC_NOWAIT ) == -1 )
        return SND_FAILURE;
    else
        return SND_SUCCESS;
}

int YOUR_SENSORMsgQueue::receiveConfig() {
    if( msgrcv( msgid, &configMessage, configMsgLength, CONFIG_MSG_TYPE, IPC_NOWAIT ) == -1 ) {
        return RCV_FAILURE;
    }else {
        return RCV_SUCCESS;
```

```
    }
}

int YOUR_SENSORMsgQueue::sendData() {
    if( msgsnd( msgid, &dataMessage, dataMsgLength, IPC_NOWAIT ) == -1 )
        return SND_FAILURE;
    else
        return SND_SUCCESS;
}

int YOUR_SENSORMsgQueue::receiveData() {
    if( msgrcv( msgid, &dataMessage, dataMsgLength, DATA_MSG_TYPE, IPC_NOWAIT ) == -1 ) {
        return RCV_FAILURE;
    }else  {
        return RCV_SUCCESS;
    }
}

void YOUR_SENSORMsgQueue::setSensorStatus( uint8_t _sensorStatus ) {
    configMessage.sensorStatus = _sensorStatus;
}

void YOUR_SENSORMsgQueue::setSerialPortNum( uint8_t _num ) {
    configMessage.serialPortNum = _num;
}


void YOUR_SENSORMsgQueue::setYOUR_DATA( Type _YOUR_DATA ) {
    dataMessage.YOUR_DATA = _YOUR_DATA;
}

uint8_t YOUR_SENSORMsgQueue::getSensorStatus() {
    return configMessage.sensorStatus;
}

int YOUR_SENSORMsgQueue::getSerialPortNum() {
    return configMessage.serialPortNum;
}

Type YOUR_SENSORMsgQueue::getYOUR_DATA() {
    return dataMessage.YOUR_DATA;
}
```

Replace every occurences of YOUR_SENSOR by the actual name of your sensor as well as YOUR_DATA by the name of the data your sensor is measuring.

In the constructor of the class there is this line:

```
        key = 0x9999000X;
```

Replace the “X” by the total number of sensors that are used in your configuration.
This key is used to get a unique id for the message queue used to communicate with the **Handler**. It needs to be different from the other already existing message queues.
If you are not sure you can check the other SENSORMsgQueue.h files and choose a different value.

II-2 Compiling the source code

You have now to create a *Makefile* for compiling your new sensor sources and create an executable.

In YOUR_SENSOR folder create a file named *Makefile* and copy-paste this in it:

```
CXX=g++
CXXFLAGS=-W -Wall -ansi -pedantic -std=c++11
LDFLAGS=
EXEC=main_YOUR_SENSOR
SRC= main_YOUR_SENSOR.c
OBJ= $(SRC:.c=.o)

all: $(EXEC)

main_YOUR_SENSOR: $(OBJ)
    $(CXX) -o $@ $^ $(LDFLAGS)

main.o: main_YOUR_SENSOR.h

%.o: %.c
    $(CXX) -o $@ -c $< $(CXXFLAGS)

.PHONY: clean mrproper

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

Replace every occurence of YOUR_SENSOR by the actual name of your sensor.

You can try to compile your sources.
Open a terminal in YOUR_SENSOR folder and type:

```
$ make
```

Then, as the executable needs to be executed on another platform (not your computer but the DaughterBoard), run this command to delete the executabes:

```
$ make mrproper
```

Finally, you have to modify the **Handler** to take into account your new sensor.

In **Handler** folder open *Handler.h* file and add every lines with the “HERE” comment .

```
#include <string>
#include <iostream>
#include <sstream>
#include <list>
#include <sys/time.h>
#include <time.h>
#include "MAVLinkSerial_port.h"
#include "LidarMsgQueue.h"
#include "WindMsgQueue.h"
```

```
#include "PyranometerMsgQueue.h"
#include "OPLSMsgQueue.h"
//-------------------------------------------------------------
#include "YOUR_SENSORMsgQueue.h" // HERE
//-------------------------------------------------------------


class Handler{

 public:
  const char* XBeePort = "/dev/ttyUSB0";
  const int baudRate   = 38400;
  const int DAUGHTER_BOARD_SYSID = 32;

  Handler();
  ~Handler();
  void init();
  void handle();
  void waitForConfig();
  void heartBeat();

 private:
  LidarMsgQueue lidarMsgQueue;
  WindMsgQueue windMsgQueue;
  PyranometerMsgQueue pyranometerMsgQueue;
  OPLSMsgQueue oplsMsgQueue;
  //-------------------------------------------------------------
  YOUR_SENSORMsgQueue yOUR_SENSORMsgQueue; // HERE
  //-------------------------------------------------------------

  void receiveMsgQueue();
  void buildMAVMsg();
  void sendMAVMsg();
  void sendConfigToSensors();

  struct timeval tv;
  int32_t sec, usec;
  void setTime();

  MAVLinkSerial_Port* MAVSerial;
  //-------------------------------------------------------------
  mavlink_message_t heartBeatMsg, configMsg, windMsg, lidarMsg, pyranometerMsg, oplsMsg, YOUR_SEN
SORMsg; // HERE
  //-------------------------------------------------------------
  mavlink_config_t cfg;
  //-------------------------------------------------------------
  bool windBuild, lidarBuild, pyranometerBuild, oplsBuild, YOUR_SENSORBuild, windSend, lidarSend,
 pyranometerSend, oplsSend, YOUR_SENSORSend; // HERE
  //-------------------------------------------------------------
  uint16_t angle;
  float windSpeed;
  float temperature;
  uint16_t distance;
  float solarIrradiance;
  oplsData oplsdata;
  uint8_t status;
  //-------------------------------------------------------------
  Type YOUR_DATA; // HERE
  //-------------------------------------------------------------
};
```

```
Handler::Handler() {
}

Handler::~Handler() {
  MAVSerial->close_serial();
  delete MAVSerial;
}

void Handler::init() {
  MAVSerial = new MAVLinkSerial_Port( XBeePort, baudRate, 0 );
  MAVSerial->start();
  windBuild        = false;
  lidarBuild       = false;
  pyranometerBuild = false;
  oplsBuild        = false;
  //-----------------------------------------------------------
  YOUR_SENSORBuild = false; // HERE
  //-----------------------------------------------------------
  windSend         = false;
  lidarSend        = false;
  pyranometerSend  = false;
  oplsSend         = false;
  //-----------------------------------------------------------
  YOUR_SENSORSend  = false; // HERE
  //-----------------------------------------------------------
}


void Handler::waitForConfig() {
  while( 1 ) {
    if( MAVSerial->read_message( &configMsg ) ) {
      if( configMsg.msgid == MAVLINK_MSG_ID_CONFIG ) {
    mavlink_msg_config_decode( &configMsg, &cfg );
    sendConfigToSensors();
    return;
      }
    }
    std::cout << "no config received" << std::endl;
  }
}

void Handler::handle() {
  receiveMsgQueue();
  //-----------------------------------------------------------
  if( lidarBuild || windBuild || pyranometerBuild || oplsBuild || YOUR_SENSORBuild ) { // HERE
  //-----------------------------------------------------------
    buildMAVMsg();
  }
  //-----------------------------------------------------------
  if( lidarSend || windSend || pyranometerSend || oplsSend || YOUR_SENSORSend ) { // HERE
  //-----------------------------------------------------------
    sendMAVMsg();
  }
}


void Handler::buildMAVMsg() {
  // Wind Sensor
```

```
  if( windBuild ) {
    mavlink_msg_wind_sensor_pack( DAUGHTER_BOARD_SYSID, 1, &windMsg, sec, usec, angle, windSpeed,
 temperature, 0 );
    windSend = true;
    windBuild = false;
  }else {
    windSend = false;
  }
  // Lidar
  if( lidarBuild ) {
    mavlink_msg_lidar_pack( DAUGHTER_BOARD_SYSID, 1, &lidarMsg, sec, usec, distance, 0 );
    lidarSend = true;
    lidarBuild = false;
  }else {
    lidarSend = false;
  }
  // Pyranometer
  if( pyranometerBuild ) {
    mavlink_msg_pyranometer_pack( DAUGHTER_BOARD_SYSID, 1, &pyranometerMsg, sec, usec, solarIrrad
iance, 0 );
    pyranometerSend = true;
    pyranometerBuild = false;
  }else {
    pyranometerSend = false;
  }
  // OPLS
  if( oplsBuild ) {
    mavlink_msg_opls_pack( DAUGHTER_BOARD_SYSID, 1, &oplsMsg, sec, usec, oplsdata.time_, oplsdata
.ch4,
                oplsdata.et , oplsdata.h2o, oplsdata.p, oplsdata.t, oplsdata.rf,
                oplsdata.lon, oplsdata.lat, oplsdata.lsr );
    oplsSend  = true;
    oplsBuild = false;
  }else {
    oplsSend = false;
  }
  //------------------------------------------------------------
  // HERE
  //------------------------------------------------------------
  // YOUR_SENSOR
  if( YOUR_SENSORBuild ) {
    mavlink_msg_YOUR_SENSOR_pack( DAUGHTER_BOARD_SYSID, 1, &YOUR_SENSORMsg, sec, usec, YOUR_DATA,
0 );
    YOUR_SENSORSend = true;
    YOUR_SENSORBuild = false;
  }else {
    YOUR_SENSORSend = false;
  }
  //------------------------------------------------------------
}

void Handler::sendMAVMsg() {
  // Wind sensor
  if( windSend ) {
    MAVSerial->write_message( &windMsg );
    windSend = false;
  }
  // Lidar
  if( lidarSend ) {
    MAVSerial->write_message( &lidarMsg );
```

```
    lidarSend = false;
  }
  // Pyranometer
  if( pyranometerSend ) {
    MAVSerial->write_message( &pyranometerMsg );
    pyranometerSend = false;
  }
  // OPLS
  if( oplsSend ) {
    MAVSerial->write_message( &oplsMsg );
    oplsSend = false;
  }
  //-------------------------------------------------------------
  // HERE
  //-------------------------------------------------------------
  // YOUR_SENSOR
  if( YOUR_SENSORSend ) {
    MAVSerial->write_message( &YOUR_SENSORMsg );
    YOUR_SENSORSend = false;
  }
  //-------------------------------------------------------------
}

void Handler::receiveMsgQueue() {
  // Wind sensor
  if( windMsgQueue.receiveData() == RCV_SUCCESS ) {
    setTime();
    windBuild = true;
    angle = windMsgQueue.getAngle();
    windSpeed = windMsgQueue.getSpeed();
    temperature = windMsgQueue.getTemperature();
  }else {
    windBuild = false;
  }
  // Lidar
  if( lidarMsgQueue.receiveData() == RCV_SUCCESS ) {
    setTime();
    lidarBuild = true;
    distance = lidarMsgQueue.getDistance();
    std::cout << "distance : " << distance << std::endl;
  }else {
    lidarBuild = false;
  }
  // Pyranometer
  if( pyranometerMsgQueue.receiveData() == RCV_SUCCESS ) {
    setTime();
    pyranometerBuild = true;
    solarIrradiance = pyranometerMsgQueue.getSolarIrradiance();
  }else {
    pyranometerBuild = false;
  }
  // OPLS
  if( oplsMsgQueue.receiveData() == RCV_SUCCESS ) {
    setTime();
    oplsBuild = true;
    oplsdata  = oplsMsgQueue.getOPLSData();
  }else {
    oplsBuild = false;
  }
  //-------------------------------------------------------------
```

```
  // HERE
  //------------------------------------------------------------
  // YOUR_SENSOR
  if( YOUR_SENSORMsgQueue.receiveData() == RCV_SUCCESS ) {
    setTime();
    YOUR_SENSORBuild = true;
    YOUR_DATA  = YOUR_SENSORMsgQueue.getYOUR_DATA();
  }else {
    YOUR_SENSORBuild = false;
  }
  //------------------------------------------------------------
}

void Handler::sendConfigToSensors() {
  windMsgQueue.setSensorStatus( cfg.windSensorStatus );
  windMsgQueue.setSensorType( cfg.windSensorType );
  windMsgQueue.setSerialPortNum( cfg.windSensorComPortNum );
  lidarMsgQueue.setSensorStatus( cfg.lidarStatus );
  pyranometerMsgQueue.setSensorStatus( cfg.pyranometerStatus );
  pyranometerMsgQueue.setSerialPortNum( cfg.pyranometerComPortNum );
  oplsMsgQueue.setSensorStatus( cfg.oplsStatus );
  oplsMsgQueue.setSerialPortNum( cfg.oplsComPortNum );
  //------------------------------------------------------------
  YOUR_SENSORMsgQueue.setSensorStatus( cfg.YOUR_SENSORStatus ); // HERE
  //------------------------------------------------------------
  windMsgQueue.sendConfig();
  lidarMsgQueue.sendConfig();
  pyranometerMsgQueue.sendConfig();
  oplsMsgQueue.sendConfig();
  //------------------------------------------------------------
  YOUR_SENSORMsgQueue.sendConfig(); // HERE
  //------------------------------------------------------------
}

void Handler::heartBeat() {  // Sends a heartbeat message every seconds
  mavlink_msg_heartbeat_pack( DAUGHTER_BOARD_SYSID, 1, &heartBeatMsg, 0 );
  while( 1 ) {
    usleep( 1000000 );
    std::cout << "heartbeat" << std::endl;
    MAVSerial->write_message( &heartBeatMsg );
  }
}

void Handler::setTime() {
  gettimeofday( &tv, NULL );
  sec  = tv.tv_sec;
  usec = tv.tv_usec;
}
```

Last thing to do: create a MAVLink message specific to your sensor.
Open the *customMessages.xml* file which is located in the “MAVLink_Messages_Definition” folder and add the parts where the “HERE” comment is specified. Another time replace YOUR_SENSOR and YOUR_DATA.

```
<?xml version="1.0"?>
<mavlink>
    <enums>
```

```
    </enums>
    <messages>
    <message id="0" name="HEARTBEAT">
        <description>Daughterboard's heartbeat: sent every seconds to the payload GCS.</descripti
on>
        <field type="uint8_t" name="status">The Daughterboard status.</field>
    </message>
    <message id="150" name="CONFIG">
        <description>This message contains all the configuration parameters for the daughterboard
's sensors.</description>
        <field type="uint8_t" name="windSensorStatus">Tells whether the Wind Sensor is (1) or is
not (0) used in the current configuration.</field>
        <field type="uint8_t" name="windSensorType">The type of wind sensor used: 0 (FT205), 1 (F
T742) or 2 (Trisonica).</field>
      <field type="uint8_t" name="windSensorComPortNum">The serial port number for the wind senso
r.</field>
      <field type="uint8_t" name="lidarStatus">Tells whether the Lidar is (1) or is not (0) used
in the current configuration.</field>                  <field type="uint8_t" name="pyranometerSta
tus">Tells whether the Pyranometer is (1) or is not (0) used in the current configuration.</field>

      <field type="uint8_t" name="pyranometerComPortNum">The serial port number for the Pyranomet
er.</field>
      <field type="uint8_t" name="oplsStatus">Tells whether the OPLS is (1) or is not (0) used in
 the current configuration.</field>
      <field type="uint8_t" name="oplsComPortNum">The serial port number for the OPLS.</field>
        <!-- HERE -->
        <!-- ------------------------------------------------------------------------------- -->
       <field type="uint8_t" name="yOUR_SENSORStatus">Tells whether YOUR_SENSOR is (1) or is not
(0) used in the current configuration.</field>
        <!-- ------------------------------------------------------------------------------- -->
        <!-- IF SERIAL SENSOR UNCOMMENT HERE -->
        <!-- ------------------------------------------------------------------------------- -->
       <!-- <field type="uint8_t" name="yOUR_SENSORComPortNum">The serial port number for YOUR_SE
NSOR.</field>
        <!-- ------------------------------------------------------------------------------- -->
    </message>
    <message id="151" name="WIND_SENSOR">
        <description>This message encodes all of the wind sensor data from the daughter board.</d
escription>
        <field type="uint32_t" name="sec">Epoch number of seconds.</field>
        <field type="uint32_t" name="usec">Number of microseconds. usec divided by 1e6 plus sec f
ield provides current time with microseconds precision</field>
        <field type="uint16_t" name="angle">The angle from the wind (in degrees).</field>
        <field type="float" name="wind_speed">The speed of the wind for the specified angle (in m
/s).</field>
        <field type="float" name="temperature">The temperature measured by the Trisonica sensor (
in degree Celsius).</field>
    </message>
    <message id="152" name="LIDAR">
        <description>This message encodes all of the lidar data from the daughter board.</descrip
tion>
        <field type="uint32_t" name="sec">Epoch number of seconds.</field>
        <field type="uint32_t" name="usec">Number of microseconds. usec divided by 1e6 plus sec f
ield provides current time with microseconds precision</field>
        <field type="uint16_t" name="distance">The distance measured by the Lidar (in cm).</field>

    </message>
    <message id="153" name="PYRANOMETER">
        <description>This message encodes all of the pyranometer data from the daughter board.</d
escription>
```

```
        <field type="uint32_t" name="sec">Epoch number of seconds.</field>
        <field type="uint32_t" name="usec">Number of microseconds. usec divided by 1e6 plus sec f
ield provides current time with microseconds precision</field>
        <field type="float" name="solarIrradiance">The solar iradiance measured by the Pyranomete
r (in W.m-2).</field>
    </message>
    <message id="154" name="OPLS">
        <description>This message encodes all of the OPLS data from the daughter board.</descript
ion>
        <field type="uint32_t" name="sec">Epoch number of seconds.</field>
        <field type="uint32_t" name="usec">Number of microseconds. usec divided by 1e6 plus sec f
ield provides current time with microseconds precision</field>
        <field type="double" name="time">Time.</field>
        <field type="double" name="ch4">CH4.</field>
        <field type="double" name="et">Et.</field>
        <field type="double" name="h2o">H20.</field>
        <field type="double" name="p">Pressure.</field>
        <field type="double" name="t">Temperature.</field>
        <field type="double" name="rf">Range finder.</field>
        <field type="double" name="lon">Longitude.</field>
        <field type="double" name="lat">Latitude.</field>
        <field type="double" name="lsr">Laser power.</field>
    </message>
    <!-- HERE -->
    <!-- ------------------------------------------------------------------------------- -->
       <message id="155" name="YOUR_SENSOR">
        <description>This message encodes all of the YOUR_SENSOR data from the daughter board.</d
escription>
        <field type="uint32_t" name="sec">Epoch number of seconds.</field>
        <field type="uint32_t" name="usec">Number of microseconds. usec divided by 1e6 plus sec f
ield provides current time with microseconds precision</field>
        <field type="Type" name="YOUR_DATA">YOUR_DATA measured by YOUR_SENSOR (in YOUR_UNIT).</fi
eld>
    </message>
    <!-- ------------------------------------------------------------------------------- -->
    </messages>
</mavlink>
```

Now we can generate the headers that will allow us to use functions that build and decode the MAVLink messages defined in the previous xml file.

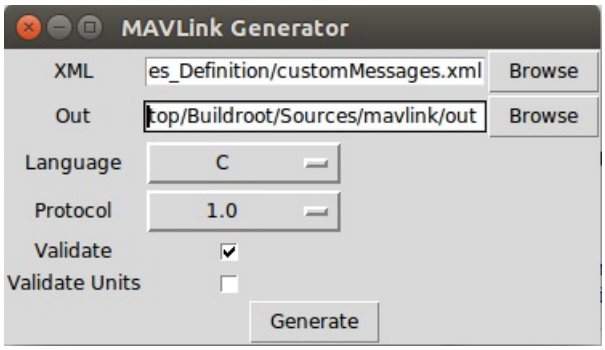
For that purpose, go into the “mavlink” folder that you should have generated in the Prequisites part.

Create a folder called “out”.

Open a terminal and type:

```
$ python mavgenerate.py
```

You should see the following box:



For the XML path, browse to the *customMessages.xml* location.
For the Out path, browse to the “out” folder location.
Select “C” for the Language and click on the “Generate” button.

Now, you should have in your “out” folder all the necessary headers.
Copy the content of the “out” folder in the “MAVLink” folder located in the “Handler” folder (clear the content of the “MAVLink” folder for more safety).

II-4 Building the Linux firmware for the Raspberry Pi 3

For that purpose we will use an open-source embedded Linux build system called Buildroot.
First, you need to install some mandatory packages for Buildroot to be used in checking this link:
https://buildroot.org/downloads/manual/manual.html#requirement-mandatory

Download the latest long term support release of Buildroot available from this link:
https://buildroot.org/download.html

At the time of this tutorial being written, the latest long term support release is the 2018.02.5.

After downloading this archive, extract it in the “Buildroot” folder which is in the “DaughterBoard” folder.

```
$ tar xjf buildroot-20xx.xx.x.tar.bz2
```

We need to configure Buildroot to create a Linux firmware compatible with our application and with the Raspberry Pi 3 hardware specifications.
There are multiple steps to accomplish to get a full usable firmware.
Besides, we want the source codes previously written to be compiled for the target hardware (RPi3) and be executed whenever the DaughterBoard is powered.

First, copy the “Sources” folder in the “Buildroot” folder.
Then, go into the “package” folder and rename the “YOUR_SENSOR” folder by the actual name of your sensor. Enter this folder.
Open now the *Config.in* file and replace every occurences of YOUR_SENSOR. Do the same with the *your_sensor.mk* file.

Under “Buildroot” folder, copy the content of the “package” folder in the “buildroot-20xx.xx.x/package” folder.
Then, go into the “buildroot-20xx.xx.x” folder previously created during the buildroot archive extraction.
Open the *Config.in* file located under “package” and add these lines at the bottom of the file just before the last **end menu** line:

```
1998         source "package/tpm-tools/Config.in"
1999         source "package/unscd/Config.in"
2000         source "package/util-linux/Config.in"
2001         source "package/xen/Config.in"
2002         source "package/xvisor/Config.in"
2003 endmenu
2004
2005 menu "Text editors and viewers"
2006         source "package/ed/Config.in"
2007         source "package/joe/Config.in"
2008         source "package/less/Config.in"
2009         source "package/mc/Config.in"
2010         source "package/nano/Config.in"
2011         source "package/uemacs/Config.in"
2012         source "package/vim/Config.in"
2013 endmenu
2014
2015 menu "Custom"
2016         source "package/Handler/Config.in"
2017         source "package/WindSensor/Config.in"
2018         source "package/Pyranometer/Config.in"
2019         source "package/OPLS/Config.in"
2020         source "package/Lidar/Config.in"
2021         source "package/YOUR_SENSOR/Config.in"
2022 endmenu
2023
2024 endmenu
```

Obviously, replace YOUR_SENSOR by the actual name of the sensor you are adding.

Open a terminal in this folder and type:

```
$ make raspberrypi3_defconfig
```

then

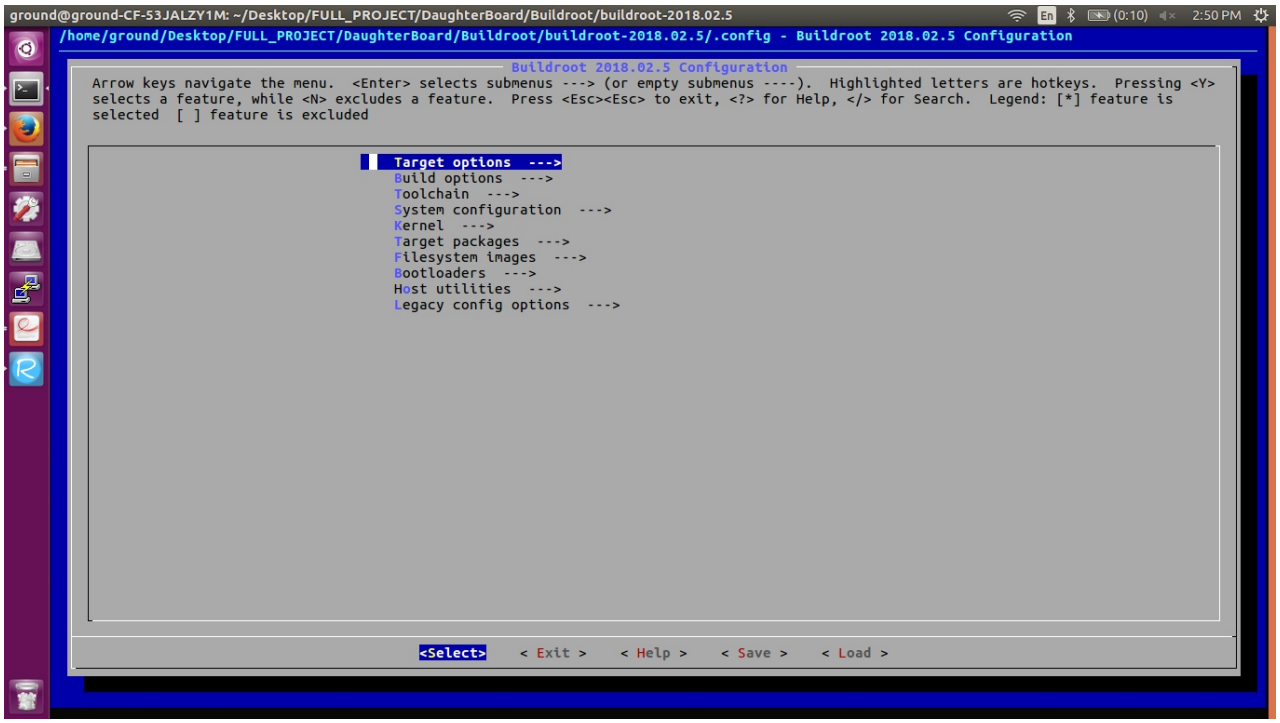
```
$ make menuconfig
```

If this command ends with:

```
...
Your display is too small to run Menuconfig!
It must be at least 19 lines by 80 columns.
Makefile:904: recipe for target 'menuconfig' failed
make[1]: *** [menuconfig] Error 1
Makefile:79: recipe for target '_all' failed
make: *** [_all] Error 2
$
```

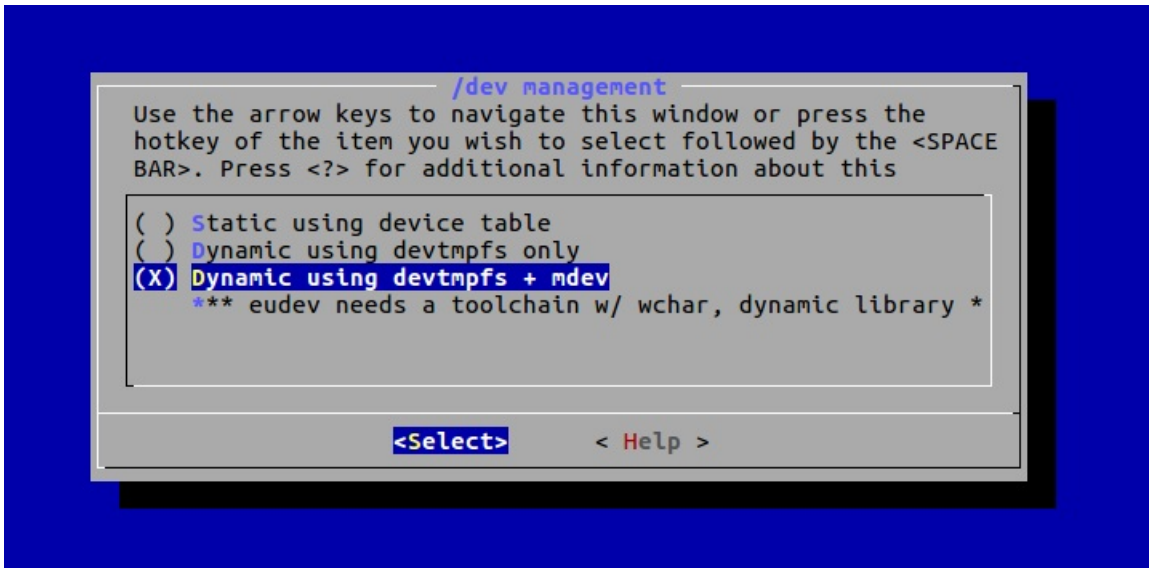
Open your terminal in full screen and try again.

You should see this menu appearing:



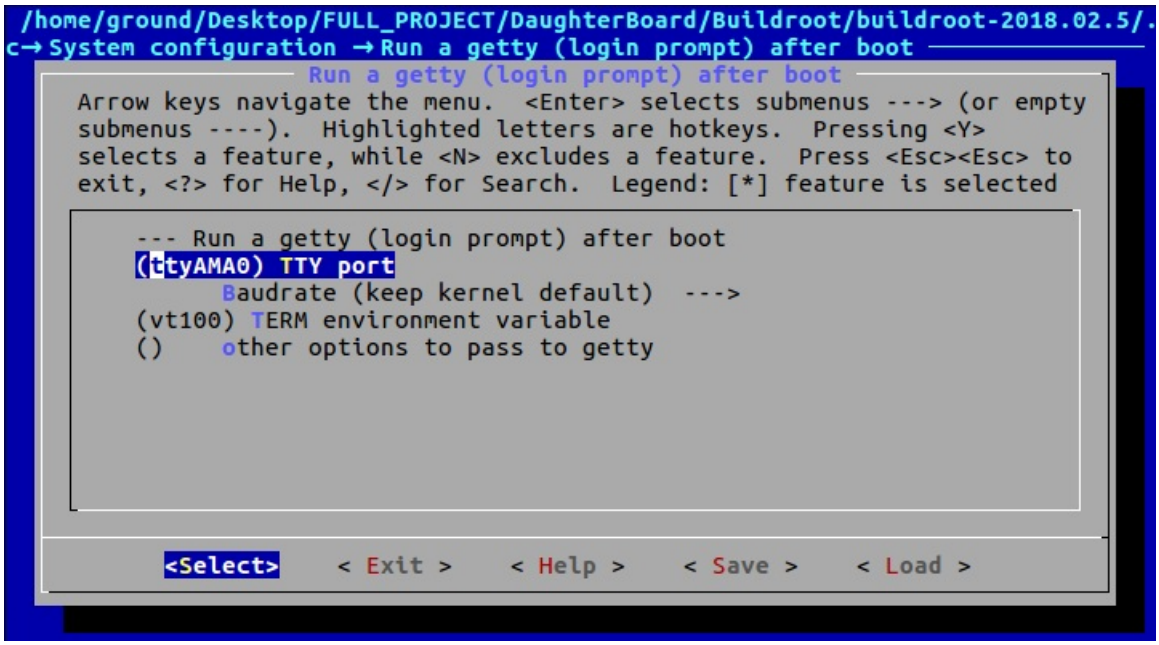
You can navigate through the different sections using the UP/DOWN arrow keys and enter a sub-menu by pressing Enter.

Go into the **System Configuration** menu.
Select the **/dev management** entry, and select the **Dynamic using devtmpfs + mdev** option.



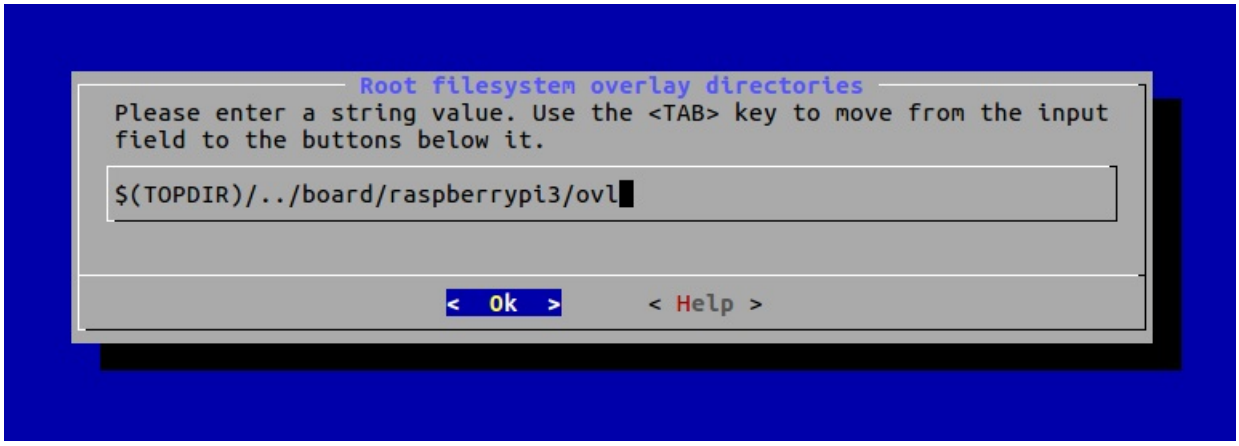
Then, enter **Root password** and choose a password for the root user.

Enter **Run a getty (login prompt) after boot** and replace **console** by **ttyAMA0**.

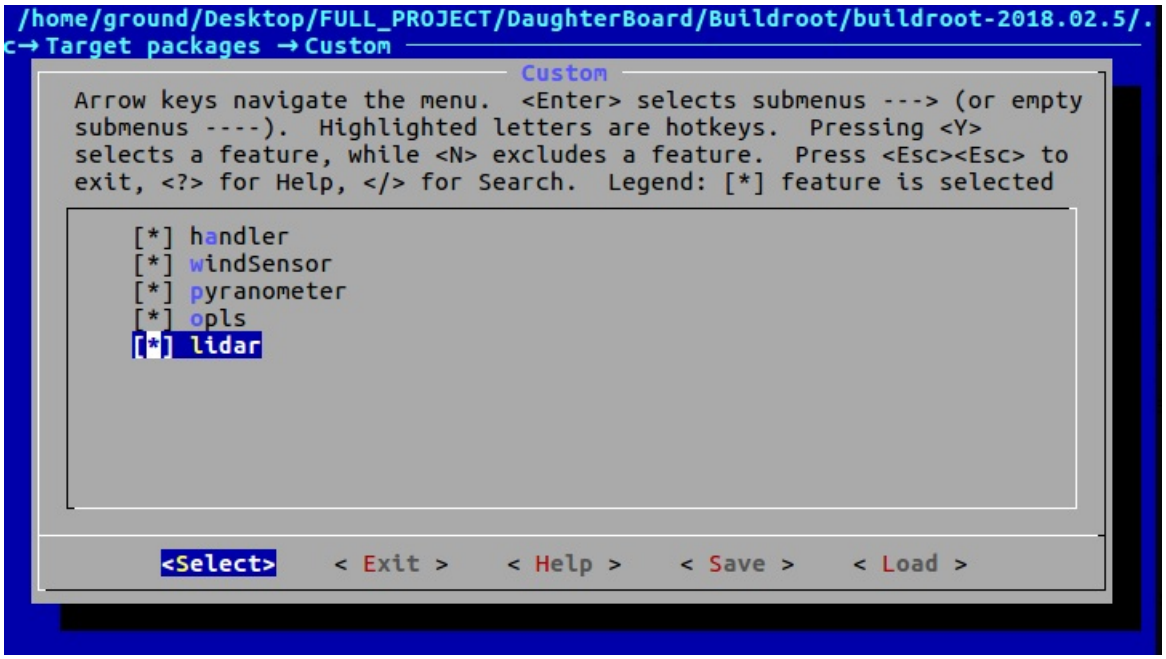


Desactivate the **remount root filesystem read-write during boot** option by pressing the space bar.

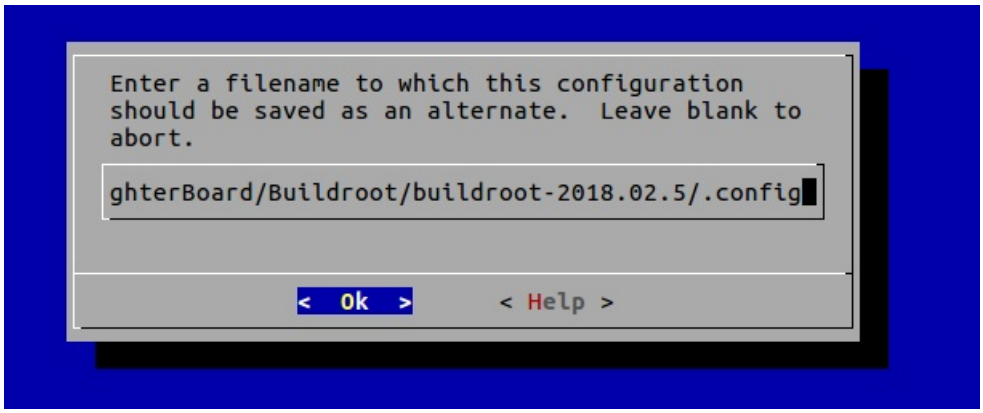
Enter **Root filesystem overlay directories** and type the following path:



Use the RIGHT arrow key and press Enter to exit this menu, and enter **Target packages**. Select the **Custom** entry at the bottom of the menu and press Space key to select the sensors.



Finally use the RIGHT arrow key and press Enter to Save the configuration.



Exit the semi-graphical menu and type:

```
$ make
```

It may take more than 30 minutes to finish generating the firmware.

When this is done, insert a micro SD card in your computer, open a terminal and type:

```
$ dmesg | tail
...
[  447.342265] mmc0: new high speed SDHC card at address 0007
[  447.379945] mmcblk0: mmc0:0007 SL16G 14.5 GiB
[  447.380740] mmcblk0: p1 p2
$
```

The goal is to obtain the device node filename associated with the SD card. The **dmesg** command outputs the messages from the kernel, and as it is the kernel that is responsible for handling the external device, it outputs a message when the SD card is inserted.

That way we know that the device node associated with the SD card is “mmcblk0” which is located in the “/dev” directory as every special file associated with peripheral drivers.

Open the “buildroot-20xx.xx.x/output/images/” folder.
In this folder there should be a file named *sdcard.img*. This is the actual micro-SD card image that you should use on the Raspberry Pi 3 target. We will use the **dd** command which is used copy the image of the firmware.
In this folder, open a terminal and type:

```
$ sudo dd if=sdcard.img of=/dev/mmcblk0
311297+0 records in
311297+0 records out
159384064 bytes (159 MB, 152 MiB) copied, 37.255 s, 4.3 MB/s
$
```

Now this is done, unplug and plug again the SD card.
Two partitions should mount automatically, one of about 34MB (bootable Dos Vfat format) that we can call **BOOT** partition and another one of about 126MB (Linux Ext4 format: root filesystem) that we can call **ROOTFS**.

Open the *config.txt* file and replace the last line *dtoverlay=pi3-miniuart-bt* by those three:

```
dtoverlay=pi3-disable-bt
dtoverlay=i2c-rtc,ds1307
dtparam=i2c_arm=on
```

Close and save the file.
Now open a terminal from the **ROOTFS** partition and type:

```
$ ls -l usr/bin | grep main
-rwxr-xr-x 1 root root 17980 Sep  9 20:02 main_Handler
-rwxr-xr-x 1 root root  9668 Sep  9 20:02 main_Lidar
-rwxr-xr-x 1 root root 18020 Sep  9 20:02 main_OPLS
-rwxr-xr-x 1 root root 13916 Sep  9 20:02 main_Pyranometer
-rwxr-xr-x 1 root root 18048 Sep  9 20:02 main_WindSensor
$
```

This shows that our source codes for handling our sensors have been effectively compiled and the executables are located in “/usr/bin” folder which contains most of the user commands.

We can also check the script which will launch those executables when the Raspberry Pi boots up. From the same directory:

```
$ more etc/init.d/S90rundataloging
#!/bin/sh
main_Handler&
main_Lidar&
main_Pyranometer&
main_WindSensor&
main_OPLS&
$
```

Unmoumt and eject properly the SD-card and insert it in the Raspberry Pi 3.

II- DaughterBoard side

II-1- Writing the source code