



UNIVERSITY OF AGDER

MAS234 PROJECT REPORT

**Marius Egeland,
Kristoffer Hansen Kruithof
Jørgen Strøm Langås,**

Supervisor
Kristian Muri Knausgård

University of Agder, 2018
Faculty of Engineering and Science
Department of Engineering Sciences

class Abstract

```
{  
    public :  
        Virtual char* abstractTextGoesHere() = 0;  
    private :  
        void existence(void);  
};
```

Contents

1	Introduction	1
2	Project parts	2
2.1	Project part 1	2
2.2	Project part 2	7
2.3	Project part 3	9
2.4	Project part 4	11
3	Discussions & Conclusion	13

Chapter 1

Introduction

The project includes 4 parts stretched over 6 weeks. The project parts consists of different hardware configurations and programming these to solve / do a set of tasks given in the project description. This report will describe a set of solutions to each of the parts, and discuss the problems we encountered along the way.

Part 1 consisted of setting up and calculating component values for a power supply unit (PSU) to be used with an Atmega 168 microcontroller and programming this microcontroller using Atmel Studio.

Part 2 and 3 was about programming a Teensy 3.6 to communicate with a GY-521 IMU over I2C, reporting the values over CAN-bus, and in some way implement this data to control basic functionality.

Part 4 included a Raspberry Pi connected to a PiCan CAN-bus shield with IMU and GPS. Due to limited time we didn't implement any particular programming on the Pi, but we successfully sent messages using the PiCan extension and received them with the PEAK systems PCan usb to CAN module. Given more time this would have been interesting to learn more about.

Chapter 2

Project parts

2.1 Project part 1

Task 1, power supply circuit

The power supply we chose was a "LM2931T" and supplies 5V to the power rail on the breadboard.

The key components for the integrated circuit was selected as per the data-sheet [11] namely C1 and C2 seen in 2.1 with values of 100nF and 100uF respectively

The maximum operating range of the input voltage is stated to be 26V [12] and the maximum output current of the IC is in excess of 100mA [13]

The circuit as a whole consumes under 100mA, it consists of two LED's each consuming around 20mA and the ATmega168-20PU consuming 0.3mA at 1MHz, though our implementation runs at a higher clock rate, the ATmega is not expected to consume over 60mA, The PSU IC is therefore capable of supplying the circuit. [23]

To test the power supply circuit we supplied it 7V from a lab bench power supply and measured the result to be around 5V and therefore proceeded to connect the PSU circuit to the ATmega168. It should also be mentioned that the bench power supply was set with OVP to 7.5v and OCP to 100mA to protect the circuit in case of a short circuit connection.

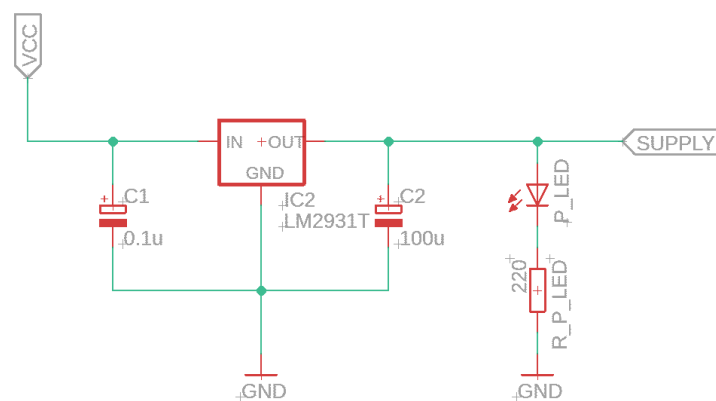


Figure 2.1: Eagle schematic of power supply circuit

Task 2, Schematic of microcontroller and Atmel ICE pin header

The ATmega168 was connected to the power supply and other components where added. XTAL1 and XTAL2 were given pull down resistors of 4.7K [2]. A reset switch was added with a pull up resistor of 100K and a series resistor of 220ohm [3]. A LED was also added with a series resistance of 220ohm to limit the current through the LED to under 20mA.

Lastly the SPI header was connected as described in both the AVR design consideration and the Atmel ICE user manual [4] [22]

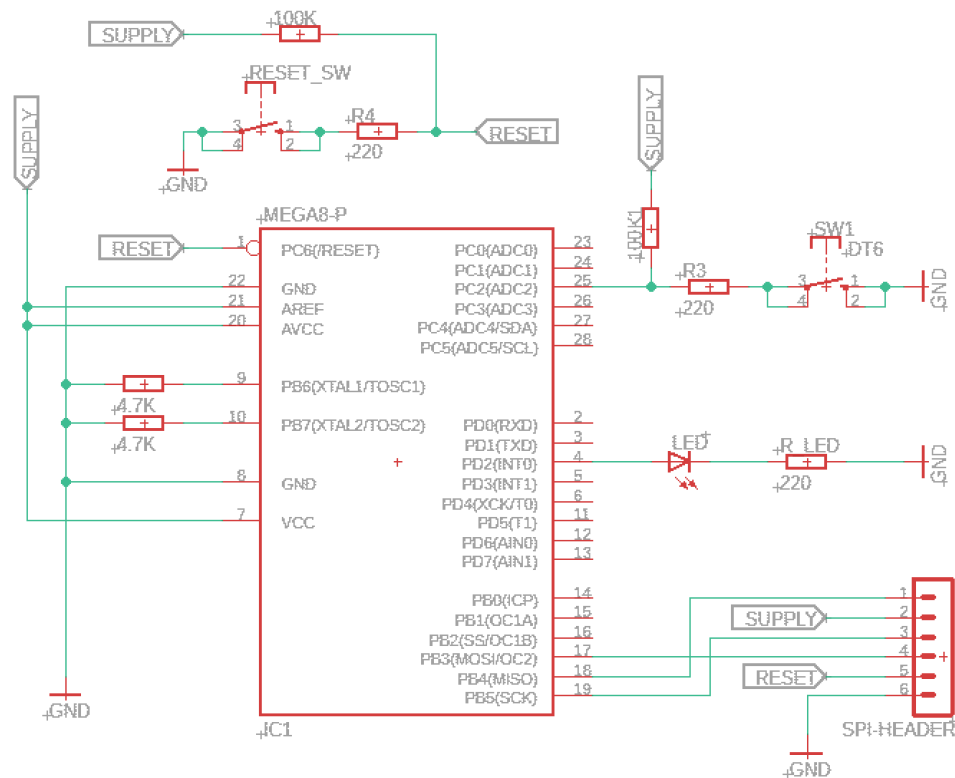


Figure 2.2: ATmega168 with SPI header LED and button

Task 3, A simple "Hello world" program

We started a new "GCC C++ Executable Project" in AtmelStudio and selected ATmega168P as the MCU to program, the example code compiled.

A simple "Hello World" program was created and compiled setting port PD2 as an output, then setting it high, the code as seen in 2.3 sources current from the pin instead of sinking current, for our application it is simpler to troubleshoot if 5V is present on a pin therefore we chose to set it high. Though for an application utilizing multiple LED's or requiring higher currents sinking power is preferred.

```
#include <avr/io.h>

int main(void)
{
    //Dedicating port PD2 as output
    DDRD = 0x04;

    //Setting port PD2 high
    PORTD = 0x04;
}
```

Figure 2.3: AtmelStudio, "Hello World"

Task 4, Power on LED

The schematic for the power on LED can be seen in 2.1, it has a 220ohm series resistor, to prevent over current.

The formula used to calculate the series resistor is ohms law calculated for the voltage drop over the resistor.

$$R_{Series} = \frac{V_{cc} - V_{LED}}{I_{LED}} \quad (2.1)$$

Substituting values gives:

$$\frac{5[V] - 2[V]}{20[mA]} = 150[ohm] \quad (2.2)$$

A series resistor of a value $R > 150\text{ohm}$ is needed to protect the LED from over current.

We selected a 220ohm series resistor.

Task 5, Connecting the circuit and programming the MCU

We connected the LED as per our schematic seen in Figure 2.2

"To sink or not to sink?"

The choice was made to source current from the pin, this was done after consulting the manual manuals provided for the micro controller, it is stated that the mcu portD has the same sink and source capability [1] and can provide a maximum current per pin of 40mA [24]. This meets our current needs.

The Atmel ICE programmer was connected and a co-student checked the connections,

In device programming in AtmelStudio the "ATmega168" was selected as the "Device" and the device signature was read to be "0x1E9406", and target voltage was read to "5V", as seen in figure 2.4

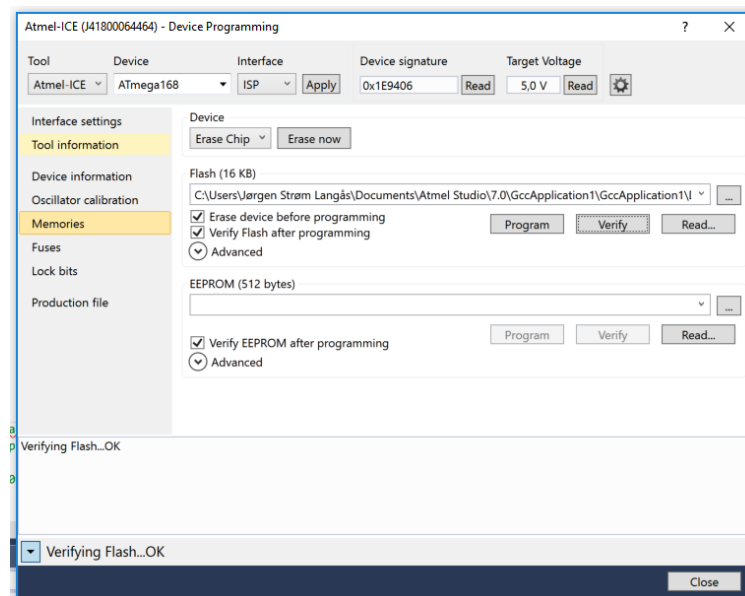


Figure 2.4: AtmelStudio, Device Programming

Under memory in the device interface we selected the ".elf" file to be uploaded to the device "flash". The process of "uploading" a ".elf" file to flash is; the master sets a destination flash address and then sends the information one byte at the time trough the MOSI(Master Out Slave In) line at each plus on the Sck. When one line is sent, it moves to the next address and repeats. [5]

In figure 2.5 we see a simple block representation of the computer, host device and target device. The host device in our case is the Atmel ICE programmer and the target device is the ATmega168

MCU.

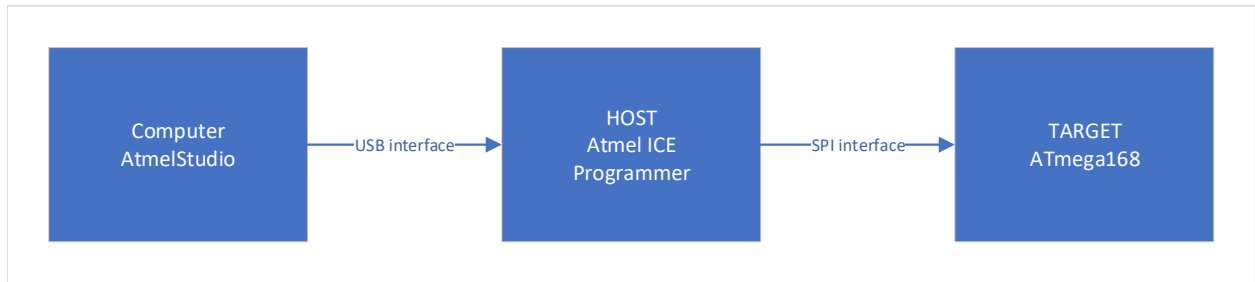


Figure 2.5: Block Diagram of SPI interface

AtmelStudio is for our application cross compiling the code. From wikipedia:

"A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running." [9]

By this definition our code from AtmelStudio is clearly cross compiled.

Task 6, Soft-blink

To get the LED to pulse smoothly we used PWM (Pulse Width Modulation) to vary the light output of the led

In the data sheet for the Kingbright green led we can get values for DC forward current and peak forward current, which correspond to constant applied voltage and a 1/10 duty cycle, with a pulse width of 0.1ms (gives a 1ms cycle time). The absolute maximum values for the two aforementioned modes of operation is 25mA and 140mA, for DC current and PWM modulated current respectively. [20]

Testing the PWM generated by our code:

We implemented the PWM in two different ways; one using delays, and one using interrupts, the implementations will be discussed later. This gave the following results when measured with hand held multimeters.

PWM test and comparison		
PWM duty cycle	Interrupt based code	Delay based code
0%	5.8%	0%
25%	25.2%	25.2%
50%	50.1%	50.2%
75%	75.1%	75.0%
100%	NA	100%

Table 2.1: Measured with UNI-T UT120A, and RICHMETERS 113D

As can be seen in table 2.1 the PWM pulses generated were fairly consistent with a slight bias for higher values then specified. The exceptions where for the interrupt based code, gave a duty cycle of 5.8% when 0% was set in the code, it also gave strange results for a specified duty cycle over 80% where it started to drop off to lower values reaching a bottom of 5.8% when 100% was set in the code.

The luminosity intensity from the LED is a linear relationship between the forward current and light output, as can be seen in the following figure 2.6.

The human eye on the other hand dose not respond linearly to light stimulation, it is a tricky subject and there dose not seem to be a common consensus on the matter. Extract from Wikipedia:

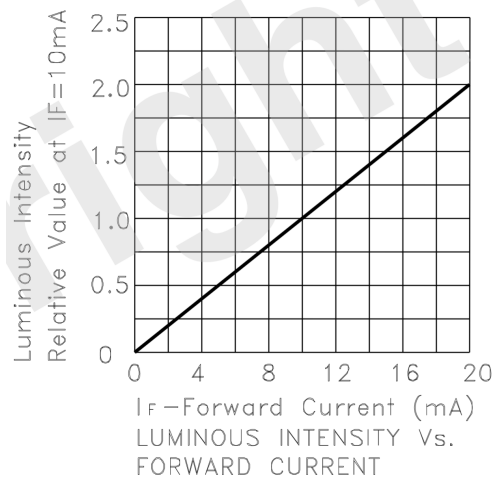


Figure 2.6: Extract from Kingbright data sheet [21]

"At first glance, you might approximate the lightness function by a cube root, an approximation that is found in much of the technical literature. However, the linear segment near black is significant, and so the 116 and 16 coefficients. The best-fit pure power function has an exponent of about 0.42, far from $1/3$." [10]

From this we can get that a good poor approximation for human light perception is a cube root function. Poor because it dose not perfectly represent reality, Good because it is fairly easy to implement in code.

To make a cube root function linear, and thusly make the light perceived from the LED approximate a linear function the LED light intensity need to be incremented with a cubic function. The code implemented is seen figure 2.7.

```
int expAppr(const int xx)
{
    //Approximating a exp function with a third degree polynomial, mapping 0-100, to 0-100
    //Distributing the conversion factor to keep floats small

    const float val = (0.01f*(float)xx)*(0.1f*(float)xx)*(0.1f*(float)xx);
    return (int)val;
}
```

Figure 2.7: Cubic function

The delay base code ran a function called sudoPWM, with a cycle period of 10mS. It ran a for loop iterating 100 values and each iteration of the loop was delayed 100uS, The LED was initially turned on, and then switched off when the counter reached a certain set-point corresponding to the given duty cycle wanted.

The interrupt based code used worked much the same way, though using an internal timer to compare values, and issuing an interrupt command to toggle the LED on or off.

2.2 Project part 2

Task 1, Teensyduino IDE

Testing the *Teensy*

Yet another "Hello World" program was created to check that the teensyduino was installed correctly and to check that communication and uploading of sketches worked. The sketch toggled the on-board LED with a fixed interval.

It blinks !

The program worked as expected and the LED flashed as expected.

To ensure that the serial communication worked as expected we created a "Hello Serial Monitor" program, that printed a string to the serial monitor with a fixed interval. The baud rate selected was 9600baud and this was set in both the Teensy sketch and the serial monitor.

Why 9600baud?

A faster baud rate could have been selected, this would have reduce the time the MCU uses to send the string. However this is not a concern when only sending a simple string and not doing anything else. With a faster baud rate comes more noise and a larger chance of miss-communication.

Task 2, I2C communication with MPU-6050

A simple schematic for the MCU and the MPU-6050 was created in Eagle and wired on a bread board.

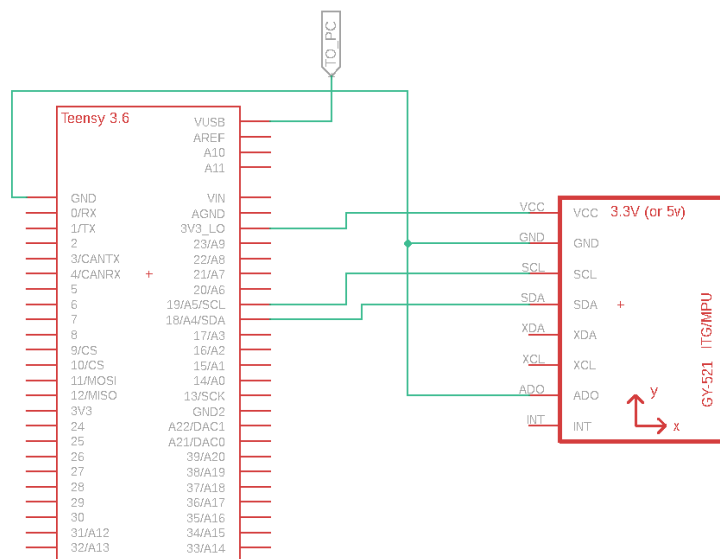


Figure 2.8: Teensy with MPU schematic

As seen in figure 2.8 we did not use pull up resistor on the SDA and SCL bus lines, it would have been wise to use pull up resistors, but we omitted them.

The communication between the MPU-6050 and the MCU is ran over a I2C bus. The communication with the MPU-6050 is fairly simple.

As seen in figure 2.8 pin 18 and 19 is used for SDA and SCL respectively. The MPU-6050 address is either 0x68 or 0x69 depending on whether ADO pin seen in figure 2.8 is set high or low.

```

Wire.beginTransaction(MPU_addr);
Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
Wire.endTransmission(false);
Wire.requestFrom(MPU_addr, 6, true); // request a total of 6 registers

AcX=Wire.read()<<8 | Wire.read(); // 0x3B (ACCEL_XOUT_H) & 0x3C (ACCEL_XOUT_L)
AcY=Wire.read()<<8 | Wire.read(); // 0x3D (ACCEL_YOUT_H) & 0x3E (ACCEL_YOUT_L)
AcZ=Wire.read()<<8 | Wire.read(); // 0x3F (ACCEL_ZOUT_H) & 0x40 (ACCEL_ZOUT_L)

accel.x = (float)AcX/(float)ssf;
accel.y = (float)AcY/(float)ssf;
accel.z = (float)AcZ/(float)ssf;

```

Figure 2.9: I2C, getting acceleration data

First the transmission is began with a `Wire.beginTransaction` and setting the I2C device address as the argument. Then the wanted bit address is sent to the MPU-6050. The `endTransmission` is sent to restart the transmission so we can request data from the selected address instead of writing to it. Next a `requestFrom` is sent, with the MPU address, number of bytes requested and if the slave should be released or not after the requested information is read. [16]

The data is read from the device with a `read` function. The acceleration data stored on the MPU-6050 is a 16 bit int stored as two bytes. First the high byte is read and shifted up 8 bits on a 16 bit int on the Teensy, next the low 8 bits is read. This gives a 16 bit int containing the acceleration data for one axis. This is then done two more times to get the acceleration data for the other two axis. [15]

Task 3, Controlling an LED based on MPU-6050 data

We made two implementations of of the accelerometer data, one where we looked at jerk, and one where we looked at the acceleration data.

They both ues the built in LED on the MCU located on pin 13, the pin sources power as setting it high turns the LED on.

To interpret the data from the MPU-6050 we had to scale the data according to the selected range, the scale factor we used gave the MPU-6050 a range of +/- 2G [14], and set the selected scaling to register 0x1C. The scaling factor was selected in accordance with the data sheet and writen to the appropriate address, and shifted to the appropriate bit location. [14] [15]

The jerk based code suffers from noise issues, a exponential filter or software RLC filter would have helped this issue tremendously, to filter high frequency noise components.

2.3 Project part 3

Task 0 - Installing CAN- Library for Arduino

CAN, or "Controller Area Network" is a bus originally designed for the automotive industry to enable microcontrollers and other devices to communicate with each other without a host computer. It is a message based bus- system designed to be multiplexed in order to save copper in automobiles [8].

The FlexCAN library was downloaded from [Github](#) and added to the Arduino IDE using the "Library management" explorer in the sketch drop-down menu on the toolbar.

As the TeensyDuino extension already contains a FlexCan library, the old library was removed from the extension to make sure no compatibility-errors occurred when trying to compile the examples. An example was then chosen and compiled successfully.

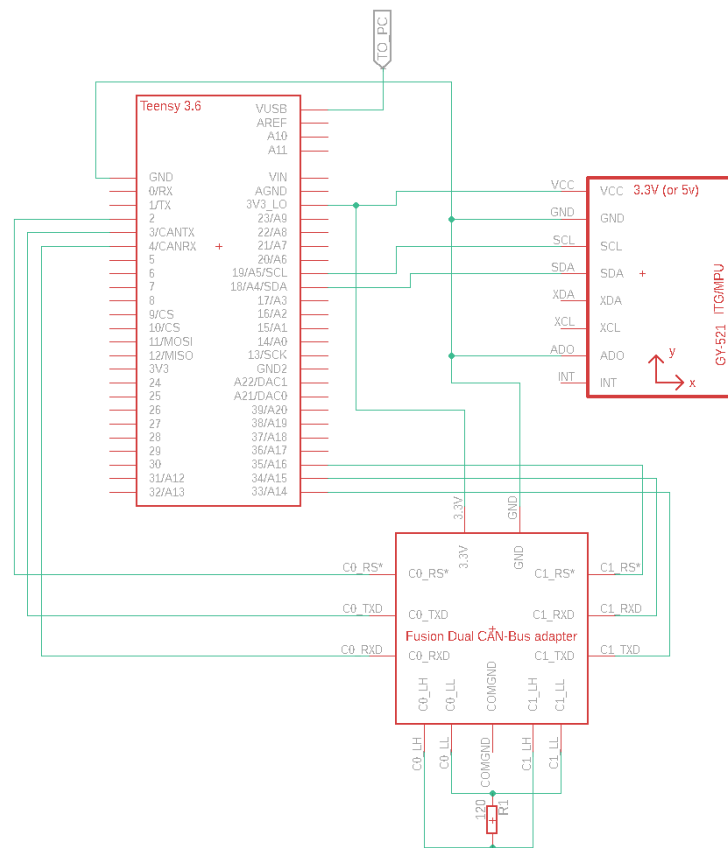


Figure 2.10: Teensy with MPU-6050 and Fusion dual CAN-bus adapter

Task 1 - Connecting CAN-bus adapter for Teensy

The headers were soldered on and visually inspected before being tested for short-circuit by using a multimeter. After determining that the connections were soldered properly, the adapter was connected to the Teensy as shown in figure 2.10, with a 120 ohm termination resistor between the high and low lines on the CAN-bus. The standard CAN0 (3, 4) and CAN1 (33, 34) pins were chosen, as opposed to using the alternative CAN0 pins (29, 30) [26], to make the programming slightly easier. Enable pins (2, 35) were also used for the CAN-bus adapter. The Teensy was then powered on and initialized as normal.

Task 2 - Sending a CAN-message

The CANTest example program was uploaded to the Teensy to check functionality. The example needed to be edited slightly to enable the CAN-bus, as the standard setup is configured with the enable pins high. In order for the CAN-transceivers be able to send and receive data, these need to be set to low. [19]

<pre>48 //if using enable pins on a transceiver they need to be set on 49 pinMode(2, OUTPUT); 50 pinMode(35, OUTPUT); 51 52 digitalWrite(2, HIGH); 53 digitalWrite(35, HIGH);</pre>	<pre>48 //if using enable pins on a transceiver they need to be set on 49 pinMode(2, OUTPUT); 50 pinMode(35, OUTPUT); 51 52 digitalWrite(2, LOW); 53 digitalWrite(35, LOW);</pre>
(a) Original code	(b) Altered code

Figure 2.11: CANTest program alteration

In order to send a CAN-message with the FlexCAN library, you have to set up a message of type `CAN_message_t` and give it an ID, length and content, the message is divided into buffers, each containing 1 byte (8 bit) of data, these buffers are transmitted byte for byte (by utilizing `CanX.write(outMessage)`, where X is the can module you're sending from) and read byte for byte by the receiving CAN module (by utilizing `CanX.read(inMessage)`).

Judging from the FlexCAN.h file the maximum length of CAN-bus messages the library can send is 2040 bit or around 2 Mbit. This is estimated from the `uint8_t` variable `len` (length of data in bytes) and the `uint8_t` `buf[8]` (buffer array).

```
116 CAN_message_t MPU2CAN(MPUData dataRec){ // function for packing MPU data into a CAN message
117
118     static CAN_message_t msgMPU;
119
120     msgMPU.ext = 0;
121     msgMPU.id = 0x20;
122     msgMPU.len = 6;
123
124     msgMPU.buf[0] = lowByte(dataRec.AcX);
125     msgMPU.buf[1] = highByte(dataRec.AcX);
126     msgMPU.buf[2] = lowByte(dataRec.AcY);
127     msgMPU.buf[3] = highByte(dataRec.AcY);
128     msgMPU.buf[4] = lowByte(dataRec.AcZ);
129     msgMPU.buf[5] = highByte(dataRec.AcZ);
```

Figure 2.12: Example of message initialization and buffer filling with MPU-data

The IMU-data was collected via I2C using the same setup as in Day 2, only this time the raw acceleration data was packed into a CAN-bus message named `msgMPU` and sent from `Can1` to `Can0` with id `0x20` and a frequency of 1Hz. The data was here rebuilt into 16bit integers and plotted using `Serial.print`.

The data from the IMU acceleration sensor was also compared in the "sending unit" and if the acceleration was larger in Y-direction than in X-Direction, the unit would send a message with id `0x22` and toggle the onboard LED on or off from the data in the least significant bit (LSB) of the message. This message was only sent at state change (meaning when the data "crosses" over each other), this was done to not clutter the bus with constant messages that do nothing.

2.4 Project part 4

Task 0 -Installing Ubuntu Linux on Virtualbox

The Ubuntu Linux version 16.04.5 was downloaded as an image from Ubuntu. [25] and installed on Virtualbox.

Task 1 - Download and configure buildroot

Buildroot 2017.02.07 was downloaded from Buildroot [7]. The download was then extracted in a project folder.

To get acquainted with the configurator we ran a "make menuconfig" command from the terminal. In order to run "make menuconfig", ncurses-devel had to be installed. After this was done, we ran "make rasperrypi3_defconfig" and found that the rasperrypi had a ARM processor architecture.

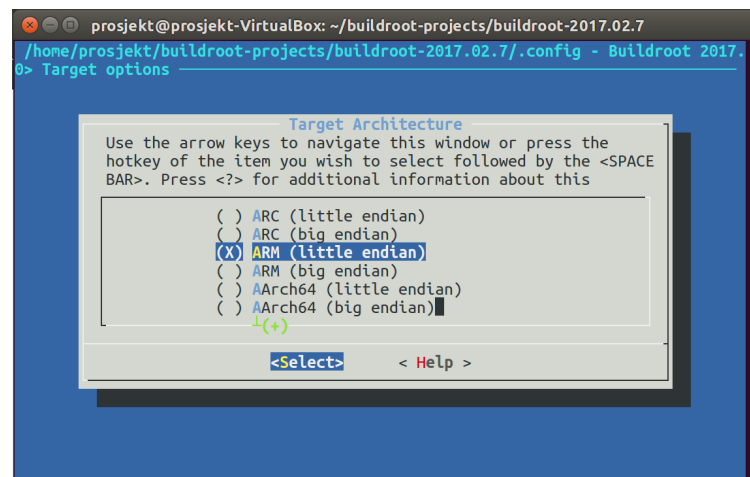


Figure 2.13: The Raspberry Pi's architecture

The "buildroot-project.tar.gz" folder was downloaded from Canvas and unpacked in the same project folder as the "Buildroot 2017.02.07". In the terminal we executed:

```
"make BR2_EXTERNAL=${PWD}/external -C ${PWD} /buildroot  
O=/filbane/til/buildrootproject/buildroot-output/ menuconfig"
```

The configurator was missing the "can_pingpong" package, so the .config file in the "buildroot-output" folder had to be replaced with the "mas234_rpi3..."-configfile. After that we verified "can_pingpong" was checked in the configurator. Then it was time to compile an sd-card image. To be able to compile the image we had to change all of the file paths pre generated in the "mas234_rpi3..."-configfile, with file paths compatible with our project. The image was compiled, then written to a sd-card with a software called Etcher. [6]

Task 2 - Connect hardware

The hardware was already connected. Although there is a termination resistor of 120 ohms present on the PiCan extension board [17], a jumper needed to be soldered on in order for this to be connected. The CAN-bus was instead wired to the breadboard with the Teensy 3.6, as the board already had a connected termination resistor.

Task 3 - Send a CAN message from the terminal

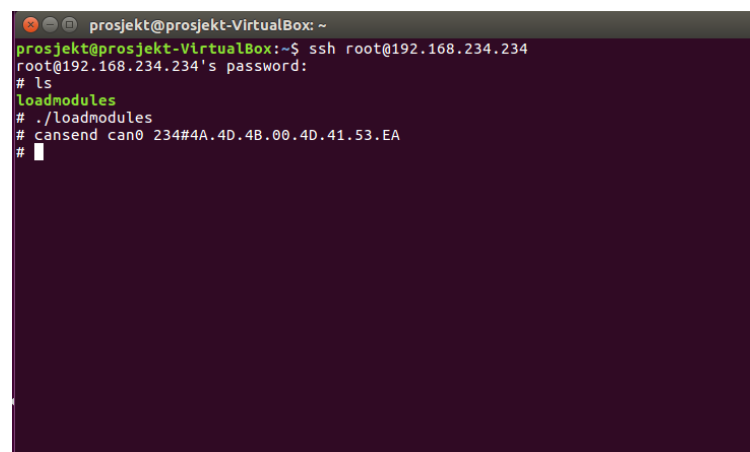
To communicate with the Raspberry Pi there was used a CAT-6 cable between the host computer and the Raspberry Pi, and a static IP had to be set on the host computer (IP:192.168.243.1, MASK:255.255.255.0). From the terminal window on the host computer we executed "ssh root@192.168.234.234", to access the Raspberry Pi. We then executed "ifconfig". There was no can0 visible. We updated the config file on the sd- card with these three commands: [18]

```
dtoverlay=spi=on
```

```
overlay=mcp2515-can0,oscillator=16000000,interrupt=25
```

```
dtoverlay=spi-bcm2835-overlay
```

After rebooting the Raspberry PI we executed "modprobe spi_bcm2835" , "modprobe mcp251x" and "/sbin/ip link set can0 up type can bitrate 250000" to set up the can0 interface (Theese three commands has to be executed after every reboot of the Raspberry PI). We sent CAN-messages to a computer using PEAK canbus-usb adapter. The messages was sent from the Raspberry Pi executing "cansend can0 234#4A.4D.4B.00.4D.41.53.EA" and received with PCAN-View on a computer.



```
projekt@projekt-VirtualBox: ~
projekt@projekt-VirtualBox:~$ ssh root@192.168.234.234
root@192.168.234.234's password:
# ls
loadmodules
# ./loadmodules
# cansend can0 234#4A.4D.4B.00.4D.41.53.EA
#
```

Figure 2.14: The Raspberry Pi's code

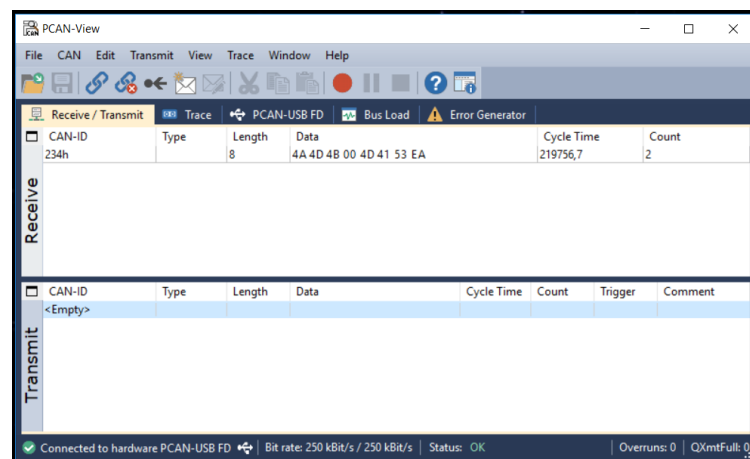


Figure 2.15: The message received in PCAN-View

Chapter 3

Discussions & Conclusion

Discussions

It took some time to get used to the seemingly "low-level" programming that runs on the AT-Mega168, and maneuvering Atmel studio in order to upload the code to the chip. But after doing this a few times and getting to know the shortcut button combinations the process streamlined quite quickly.

Getting the IMU data via I2C proved to be simple, we considered the datasheet for how the data was buffered in the GY-521, checked out an example on how to shift two 8 bit integers into a single 16 bit integer, and got it working with relative ease.

In order to document the Teensy CAN-bus system with Eagle, the Fusion Dual Can-bus adapter had to be created from scratch as no existing eagle library containing the adapter could be found. (This library will be included with the project report.)

We also encountered problems while programming the Teensy 3.6, when the `CAN_message_t` datatypes were moved into functions they seemingly lost their id when they were returned and used. This did not happen all the time! Which was a real issue to troubleshoot. The first messages went through without issue, and on the 8th-10th iteration they started to drop off until they died off completely. The function executed but the messages wouldn't come through. This was remedied by making the messages "static" in the function as seen in figure 2.12 .

The most time consuming part of the project was part 4. We encountered a lot of issues when compiling the software. Due to long compiling time these issues became hard to address. After a lot of trial and error, and cooperation with other groups we found a solution. The solution that worked for us, was to install Ubuntu 16.04 LTS and to use the 2017.02.07 version of Buildroot. We also had to change the file paths in the Buildroot config file to match our file paths. The next issue was to establish connection with the PIKAN card. This issue was solved by updating the Raspberry Pi config file, with code found in the PIKAN user manual.

Conclusion

All told the project has been a fun and at times a frustrating endeavor, we have learned a lot, and hopefully demonstrated that in this report. Though we feel like we would have needed more time with the raspberry-Pi to fully use it in the project for can-bus communication, we see that it has a lot of potential and almost endless implementations in a larger network.

Bibliography

- [1] *ATmega48/V / 88/V / 168/V DATASHEET SUMMARY*. Atmel. 2016. Chap. 5.2.6. Port D (PD[7:0]). URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2545-8-bit-AVR-Microcontroller-ATmega48-88-168_Summary.pdf (visited on 11/09/2018).
- [2] *AVR Microcontroller Hardware Design Considerations*. Microchip. 2018. Chap. 6. Unused XTAL Pins. URL: <http://ww1.microchip.com/downloads/en/AppNotes/AN2519-AVR-Microcontroller-Hardware-Design-Considerations-00002519B.pdf> (visited on 11/08/2018).
- [3] *AVR Microcontroller Hardware Design Considerations*. Microchip. 2018. Chap. 3.1 External RESET Switch. URL: <http://ww1.microchip.com/downloads/en/AppNotes/AN2519-AVR-Microcontroller-Hardware-Design-Considerations-00002519B.pdf> (visited on 11/08/2018).
- [4] *AVR Microcontroller Hardware Design Considerations*. Microchip. 2018. Chap. 4.1 SPI Programming Interface. URL: <http://ww1.microchip.com/downloads/en/AppNotes/AN2519-AVR-Microcontroller-Hardware-Design-Considerations-00002519B.pdf> (visited on 11/08/2018).
- [5] *AVR910: In-System Programming APPLICATION NOTE*. Atmel. 2016. Chap. 1. The Programming Interface. URL: http://ww1.microchip.com/downloads/en/appnotes/atmel-0943-in-system-programming_applicationnote_avr910.pdf (visited on 11/09/2018).
- [6] *BalenaEtcher*. URL: <https://www.balena.io/etcher/> (visited on 11/13/2018).
- [7] *Buildroot*. URL: <https://buildroot.org> (visited on 11/13/2018).
- [8] *CAN-bus*. Chap. CAN bus. URL: https://en.wikipedia.org/wiki/CAN_bus (visited on 11/12/2018).
- [9] *Cross compiler*. URL: https://en.wikipedia.org/wiki/Cross_compiler (visited on 11/09/2018).
- [10] *Lightness*. Chap. Relationship to value and relative luminance. URL: https://en.wikipedia.org/wiki/Lightness#Relationship_to_value_and_relative_luminance (visited on 11/11/2018).
- [11] *LM2931-N Series Low Dropout Regulators*. Version April 2013. Texas Instruments. 2000. Chap. Typical Application. URL: <http://www.ti.com/lit/ds/symlink/lm2931-n.pdf> (visited on 11/08/2018).
- [12] *LM2931-N Series Low Dropout Regulators*. Version April 2013. Texas Instruments. 2000. Chap. Absolute Maximum Ratings. URL: <http://www.ti.com/lit/ds/symlink/lm2931-n.pdf> (visited on 11/08/2018).
- [13] *LM2931-N Series Low Dropout Regulators*. Version April 2013. Texas Instruments. 2000. Chap. Features. URL: <http://www.ti.com/lit/ds/symlink/lm2931-n.pdf> (visited on 11/08/2018).
- [14] *MPU-6000 and MPU-6050 Product Specification Revision 3.4*. InvenSense. 2013. Chap. 6.2 Accelerometer Specifications. URL: <https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf> (visited on 11/13/2018).

- [15] *MPU-6000 and MPU-6050 Register Map and Descriptions Revision 4.2*. InvenSense. 2013. Chap. 3 Register Map. URL: <https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf> (visited on 11/12/2018).
- [16] *MPU-6050 Accelerometer + Gyro*. URL: <https://playground.arduino.cc/Main/MPU-6050> (visited on 11/12/2018).
- [17] *PiCan User manual*. Chap. 2. Hardware installation - 1.3. URL: https://copperhilltech.com/content/PICANGPSACC_V1.pdf (visited on 11/14/2018).
- [18] *PiCan User manual*. Chap. 3. Software Installation. URL: https://copperhilltech.com/content/PICANGPSACC_V1.pdf (visited on 11/14/2018).
- [19] *SN65HVD23x 3.3-V CAN Bus Transceivers*. Texas Instruments. 2016. Chap. 7 Pin Configuration and Functions - Table - Pin Functions. URL: <http://www.ti.com/lit/ds/symlink/sn65hvd230.pdf> (visited on 11/10/2018).
- [20] *T-1 (3mm) SOLID STATE LAMP*. Kingbright. 2013. Chap. Absolute Maximum Ratings at TA=25Å°C. URL: [http://www.kingbright.com/attachments/file/psearch/000/00/watermark00/L-7104GC\(Ver.14B\).pdf](http://www.kingbright.com/attachments/file/psearch/000/00/watermark00/L-7104GC(Ver.14B).pdf) (visited on 11/10/2018).
- [21] *T-1 (3mm) SOLID STATE LAMP*. Kingbright. 2013. Chap. Green L-7104GC, Luminous intensity Vs. Forward current. URL: [http://www.kingbright.com/attachments/file/psearch/000/00/watermark00/L-7104GC\(Ver.14B\).pdf](http://www.kingbright.com/attachments/file/psearch/000/00/watermark00/L-7104GC(Ver.14B).pdf) (visited on 11/10/2018).
- [22] *The Atmel-ICE Debugger*. Atmel. 2016. Chap. 3.7. Connecting to an SPI Target. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42330-Atmel-ICE_UserGuide.pdf (visited on 11/08/2018).
- [23] *Tmega48/V / 88/V / 168/V DATASHEET COMPLETE*. Atmel. 2016. Chap. 13.2.1. Default Clock Source. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2545-8-bit-AVR-Microcontroller-ATmega48-88-168_Datasheet.pdf (visited on 11/14/2018).
- [24] *Tmega48/V / 88/V / 168/V DATASHEET COMPLETE*. Atmel. 2016. Chap. 33.1. Absolute Maximum Ratings. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2545-8-bit-AVR-Microcontroller-ATmega48-88-168_Datasheet.pdf (visited on 11/09/2018).
- [25] *UbuntuBuildroot*. URL: <http://releases.ubuntu.com/16.04/> (visited on 11/13/2018).
- [26] *Welcome to Teensy 3.6*. URL: https://www.pjrc.com/teensy/card9a_rev1.pdf (visited on 11/11/2018).