Electrical Engineering Department,

Fourth Year - Communications & Electronics.

Graduation Project Report
Embedded Team

Supervised by: Dr. Hania H. Farag

PREPARED BY
Mahmoud Mohamed Kamal Ismail
Mahmoud AbdElHady Mahmoud
Kareem Ibrahim
Hoda Mohamed Farid

1. Introduction

People's lives are valuable. They often spend a great deal of money on stuff like life insurance as they profoundly believe this fact. Everyone wishes to avoid the loss of a dear person due to tragic events such as car accidents. Now annual highway fatalities and injuries keep increasing from year to year. Lots of lives could be saved if drivers paid more attention to some repetitive mistakes on the road. Moreover, most countries are vulnerable to traffic problems and complications as consequences of road accidents. Here comes a thought, what if cars can somehow communicate to fix this mess up? Vehicle-to-vehicle Communication (V2V) technology – which is the wireless transmission of data between motor vehicles - is here to save the day!
Our project's objective is to employ this growing and trending technology, with some modifications, to come up with solutions for these issues, and to behave towards their consequences. Next, we discuss the problem and solution in more details with facts and numbers.

PROBLEM INVESTIGATION
The global epidemic of road crash fatalities and disabilities is gradually being recognized as a major public health concern. The first step to being informed about global road safety and to developing effective road safety interventions is to have access to facts.

Egyptian Central Agency for Public Mobilization and Statistics
As per the latest statistics of the Central Agency for Public Mobilization and Statistics, we can realize how significant the effects of road accidents are. The statistics can be summarized as follows:
- Number of accidents per year = 14700 accidents
- Dangerous rate (Dead and injured per accident) = 1.6
- Cruelty incident rate (deceased / 100 wounded) = 28.7

The Association for Safe International Road Travel (ASIRT)
The Association for Safe International Road Travel (usually abbreviated as ASIRT) is a non-profit, humanitarian organization that promotes road travel safety through education and advocacy. Their annual global road crash statistics are outlined as follows:

• Nearly 1.3 million people die in road crashes each year, on average 3,287 deaths a day. An additional 20-50 million are injured or disabled.
• More than half of all road traffic deaths occur among young adults ages 15-44.
• Road traffic crashes rank as the 9th leading cause of death and account for 2.2% of all deaths globally.
• Road crashes are the leading cause of death among young people ages 15-29, and the second leading cause of death worldwide among young people ages 5-14.
• Each year nearly 400,000 people under 25 die on the world's roads, on average over 1,000 a day.
• Over 90% of all road fatalities occur in low and middle-income countries, which have less than half of the world's vehicles.
• Road crashes cost USD $518 billion globally, costing individual countries from 1- 2% of their annual GDP.
• Road crashes cost low and middle-income countries USD $65 billion annually, exceeding the total amount received in developmental assistance.
• Unless action is taken, road traffic injuries are predicted to become the fifth leading cause of death by 2030.

Car Accidents Major Causes

In order to prevent possible car accidents, it is important to investigate the main causes of most car accidents everywhere. The following list present the very common one.

1. Distracted Driving

The number one cause of car accidents is not a criminal that drove drunk, sped or ran a red light., it's a distracted driver, a motorist that diverts his or her attention from the road, usually to talk on a cell phone, send a text message or eat food.

2. Speeding

We've all seen them on the highway. Many drivers ignore the speed limit and drive over the limit. Speed kills, and traveling above the speed limit is an easy way to cause a car accident. The faster you drive, the slower your reaction time will be if you need to prevent an auto accident.

3. Drunk Driving
When you drink, you lose the ability to focus and function properly and its very dangerous when operating a vehicle. Driving under the influence of alcohol causes car accidents every day.
4. Reckless Driving
Not driving carefully may end up in a needless car accident. That's what often happens to reckless drivers who speed, change lanes too quickly or tailgate before causing a car accident.

5. Rain
If the weather gets bad so do the roads. Car accidents happen very often in the rain because water creates slick and dangerous surfaces for cars, trucks, and motorcycles and often causes automobiles to spin out of control or skid while braking

6. Night Driving
Driving in the daylight can be hazardous, but driving at night nearly doubles the risk of a car accident occurring.

7. Design Defects
No product is ever made perfectly, and cars are no different. Automobiles have hundreds of parts, and any of those defective parts can cause a serious car accident. Many automakers have had problems with design defects in the past, including Ford Explorer rollover accidents and Toyota's unintended acceleration crashes.

8. Unsafe Lane Changes
When drivers don't make safe lane changes properly, it often leads to a car accident. To prevent a needless car accident, blind spots should be checked to proceed carefully into the next lane.

9. Wrong-Way Driving
When people go the wrong way, everyone is in danger heading towards a car accident.

10. Improper Turns
The reason that we have stop lights, turn signals, and lanes designated for moving either right or left as opposed to straight is because when drivers ignore the rules of the road, car accidents are often the result.

11. Tailgating
Many fatal car accidents have occurred when a motorist dangerously tailgated another driver at high speeds

12. Tire Blowouts
Most highways are littered with the scattered remains of a tire blowout. Tire blowouts can cause the loss of control of the vehicle, and they are especially dangerous for bigger automobiles like semi-trucks.

13. Fog
Driving is a skill that requires the ability to see, but fog makes it extremely difficult to see sometimes more than a car length ahead. From the previous facts, we can tell that the car accidents issue needs to be addressed in a way that save people's lives and resources.

PROBLEM STATEMENT
Our problem statement can be described as follow:
"How we can help car drivers on highways get access to emergency services and prevent probable accidents on the road".

Proposed SOLUTION
In our project, we design and implement a device used to notify car drivers about some of the situations that may result in accidents. The device also is used to call for rescue and emergency cars in case of the occurrence of an accident.

2.

3. Microcontrollers

3.1. Tiva-C (TM4C123x Tiva™ C)

Texas Instruments TM4C123x Tiva™ C Series MCUs provide a broad portfolio of connected Cortex™-M4F microcontrollers. Designers who migrate to the Tiva Series MCUs benefit from a balance between the floating-point performance needed to create highly responsive mixed-signal applications and the low power architecture required to enable increasingly aggressive power budgets. Tiva C Series MCUs are supported by TivaWare™ for C Series software, designed specifically for those customers who want to get started easily, write production-ready code quickly, and minimize their overall cost of software ownership. TM4C123x MCUs offer 12-bit ADC accuracy achievable at the full 1 MSPS rating without any hardware averaging, eliminating performance trade-offs. These are the first ARM Cortex-M MCU in advanced 65-nm process technology, providing the right balance between higher performance and low power consumption.

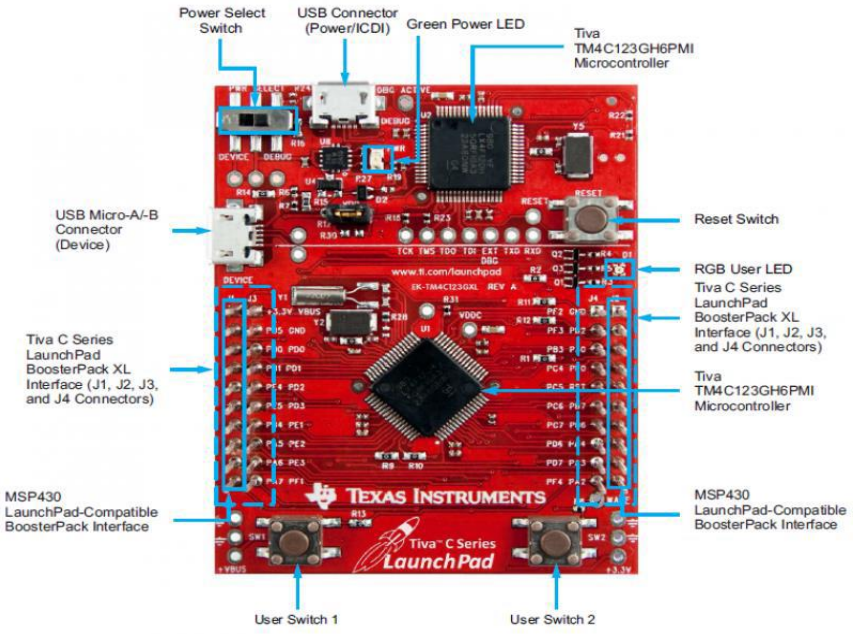
3.1.1. Features

- Motion control PWM.
- USB micro-A and micro-B connector for USB device, host, and on-the-go (OTG) connectivity.
- RGB user LED.
- Two user switches (application/wake).
- Available I/O brought out to headers on a 0.1-in (2.54-mm) grid.
- On-board ICDI.
- Switch-selectable power sources: – ICDI – USB device.
- Reset switch.
- Preloaded RGB quick-start application • Supported by TivaWare for C Series software including the USB library and the peripheral driver library.
- Floating-point capable.
- 256 KB Flash, 32 KB SRAM.

- Two 12-bit, 24-channel, 1-MSPS ADCs.
- Optional full-speed USB 2.0 with device, host, and OTG.
- Advanced motion control capability—up to 40 PWM outputs and two quadrature encoder interfaces.
- Generous serial communication options available:
  o Eight UARTs
  o Six I2C
  o Four SPI / SSI
- Low-power modes, including power-saving hibernate.
- 64-LQFP, 100-LQFP, 144-LQFP, and 157-BGA packages.

Tiva-C is used in many applications, we can't mention all of them of course but here are simple examples:

- Automation and Process Control
- Consumer and Portable Electronics.
- Human Machine Interface.
- Industrial.
- Lighting.
- Sensor Hub.



Tiva C Series TM4C123G LaunchPad Evaluation Board

Figure 3.1: Tiva-C (TM4C123G).

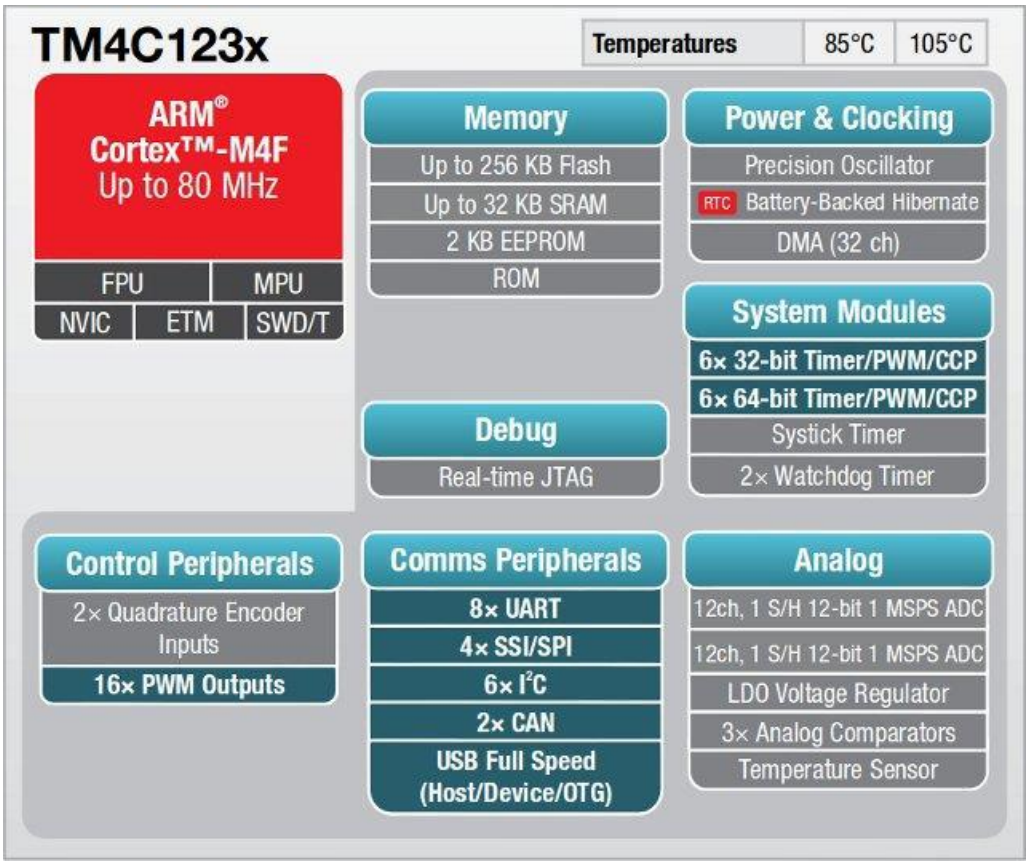


Figure 3.2: Features of TIVA-C.

TM4C123GH6PM Memory

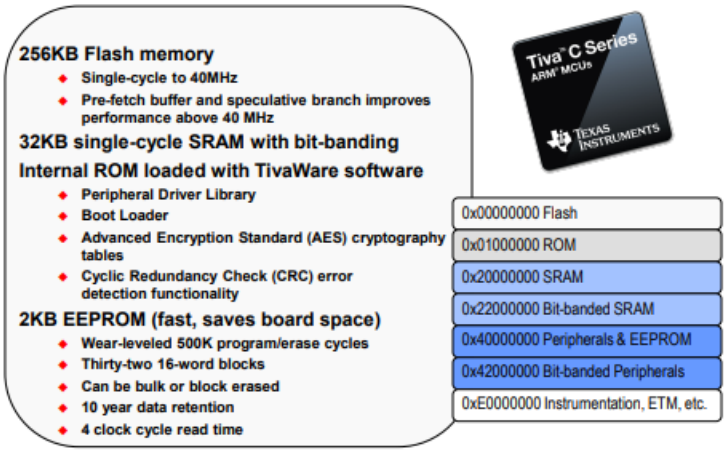


Figure 3.3: TM4C123GH6PM Memory.

3.1.2. Drivers

Software used:

- Code Composer Studio 10
- Eclipse
- Visual Studio 2019
- Putty
- Energia

Layered Architecture:

APP	**main.c - Kalman Filter**
HAL	**IMU – Wheel Encoder – Electronic Compass**
MCAL	**CLOCK – GPIO – STK(Timer) – INTERURRPT – UART – I2C**
LIB	**STD_TYPES – BIT_MATH**

Table 3.1: TIVA-C Drivers.

Every Driver of HAL/MCAL Modules Has 4 files (program.c – interface.h – config.h – private.h)

BIT_MATH: To control the bits of AVR registers.

BIT_MATH.h

```
1
2 #ifndef BIT_MATH_H
3 #define BIT_MATH_H
4
5 /* Bit Math for 1 bit */
6 #define SET_BIT(VAR,BITNO)          (VAR) |=  (   1   << (BITNO))
7 #define CLR_BIT(VAR,BITNO)          (VAR) &= ~(   1   << (BITNO))
8 #define TOG_BIT(VAR,BITNO)          (VAR) ^=  (   1   << (BITNO))
9 #define GIV_BIT(VAR,GVAR,BITNO)     (VAR) |=  ((GVAR) << (BITNO))
10 #define GET_BIT(VAR,BITNO)          (((VAR) >> (BITNO)) & 0x01)
11
12 #endif
```

STD_TYPES: to define new standard types, it could help if change machine.

STD_TYPES.h

```
/*                                                    Standard Types Library
#ifndef STD_TYPES_H
#define STD_TYPES_H

typedef unsigned char      u8;
typedef unsigned short int u16;
typedef unsigned long int  u32;
typedef signed   char      s8;
typedef signed   short int s16;
typedef signed   int       s32;
typedef float              f32;
typedef double             f64;
typedef long     double    f128;

typedef u8*                pu8;
typedef u16*               pu16;
typedef u32*               pu32;
typedef s8*                ps8;
typedef s16*               ps16;
typedef s32*               ps32;
typedef f32*               pf32;
typedef f64*               pf64;
typedef f128*              pf128;

#endif
/*****************************************************************************************************
```

3.1.2.1. Clock

Fundamental Clock Sources

There are multiple clock sources for use in the microcontroller:

■ **Precision Internal Oscillator (PIOSC).** The precision internal oscillator is an on-chip clock source that is the clock source the microcontroller uses during and following POR. It does not require the use of any external components and provides a 16-MHz clock with ±1% accuracy with calibration and ±3% accuracy across temperature (see “PIOSC Specifications” on page 1375).

The PIOSC allows for a reduced system cost in applications that require an accurate clock source. If the main oscillator is required, software must enable the main oscillator following reset and allow the main oscillator to stabilize before changing the clock reference. If the Hibernation Module clock source is a 32.768-kHz oscillator, the precision internal oscillator can be trimmed by software based on a reference clock for increased accuracy. Regardless of whether or not the PIOSC is the source for the system clock, the PIOSC can be configured to be the source for the ADC clock as well as the baud clock for the UART and SSI,

■ **Main Oscillator (MOSC).** The main oscillator provides a frequency-accurate clock source by one of two means: an external single-ended clock source is connected to the OSC0 input pin, or an external crystal is connected across the OSC0 input and OSC1 output pins. If the PLL is being used, the crystal value must be one of the supported frequencies between 5 MHz to 25 MHz (inclusive). If the PLL is not being used, the crystal may be any one of the supported frequencies between 4 MHz to 25 MHz.
. The supported crystals are listed in the XTAL bit field in the **RCC** register. Note that the MOSC provides the clock source for the USB PLL and must be connected to a crystal or an oscillator.

■ **Low-Frequency Internal Oscillator (LFIOSC).** The low-frequency internal oscillator is intended for use during Deep-Sleep power-saving modes. The frequency can have wide variations; refer to “Low-Frequency Internal Oscillator (LFIOSC) Specifications”. This power-savings mode benefits from reduced internal switching and also allows the MOSC to be powered down. In addition, the PIOSC can be powered down while in Deep-Sleep mode.

■ **Hibernation Module Clock Source.** The Hibernation module is clocked by a 32.768-kHz oscillator connected to the XOSC0 pin. The 32.768-kHz oscillator can be used for the system clock, thus eliminating the need for an additional crystal or oscillator. The Hibernation module clock source is intended to provide the system with a real-time clock source and may also provide an accurate source of Deep-Sleep or Hibernate mode power savings. The internal system clock (SysClk), is derived from any of the above sources plus two others: the output of the main internal PLL and the precision internal oscillator divided by four (4 MHz ± 1%). The frequency of the PLL clock reference must be in the range of 5 MHz to 25 MHz (inclusive).

Table 3-2 shows how the various clock sources can be used in a system.

Clock Source	Drive PLL?		Used as SysClk?	
Precision Internal Oscillator	Yes	BYPASS = 0, OSCSRC = 0x1	Yes	BYPASS = 1, OSCSRC = 0x1
Precision Internal Oscillator divide by 4 (4 MHz ± 1%)	No	-	Yes	BYPASS = 1, OSCSRC = 0x2
Main Oscillator	Yes	BYPASS = 0, OSCSRC = 0x0	Yes	BYPASS = 1, OSCSRC = 0x0
Low-Frequency Internal Oscillator (LFIOSC)	No	-	Yes	BYPASS = 1, OSCSRC = 0x3
Hibernation Module 32.768-kHz Oscillator	No	-	Yes	BYPASS = 1, OSCSRC2 = 0x7

Table 3.2: Clock Source Options.

Normally, the execution speed of a microcontroller is determined by an external crystal. The Stellaris® EK-LM4F120XL and EK-TM4C123GXL boards have a 16 MHz crystal. Most microcontrollers include a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second, at the cost of requiring more power to operate and generating more heat.

The default bus speed for the LM4F/TM4C internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. The TExaS real-board grader has been turning on the PLL, and in this section we will explain how it work. If we wish to have accurate control of time, we will activate the external crystal (called the main oscillator) use the PLL to select the desired bus speed.

There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers). However, the objective of the class is to present microcontroller fundamentals. Showing the direct access does illustrate some concepts of the PLL.

Table shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.

The main oscillator for the LM4F/TM4C Launchpad will be 16 MHz This means the reference clock (Ref Clk) input to the phase/frequency detector will be 16 MHz for a 16 MHz crystal, we set the XTAL bits to 10101 (see Table). In this way, a 400 MHz output of the voltage-controlled oscillator (VCO) will yield a 16 MHz clock at the other input of the phase/frequency detector. If the 400 MHz clock is too slow, the **up** signal will add to the charge pump, increasing the input to the VCO, leading to an increase in the 400 MHz frequency. If the 400 MHz clock is too fast, **down** signal will subtract from the charge pump, decreasing the input to the VCO, leading to a decrease in the 400 MHz frequency. Because the reference clock is stable, the feedback loop in the PLL will drive the output to a stable 400 MHz frequency.

Clock_voidFrequency():

```
long int clockFrequency = 16000000.0;
void Clock_voidFrequencySet(double Freq)
{

        char SYSDIV = ceil((400/Freq) - 1);
        // 0) Use RCC2

        RCC2 |=  0x80000000;  // USERCC2

        // 1) bypass PLL while initializing

        RCC2 |=  0x00000800;  // BYPASS2, PLL bypass

        // 2) select the crystal value and oscillator source

        RCC = (RCC &~0x000007C0)   // clear XTAL field, bits 10-6

                        + 0x00000540;   // 10101, configure for 16 MHz crystal

        RCC2 &= ~0x00000070;  // configure for main oscillator source

        // 3) activate PLL by clearing PWRDN

        RCC2 &= ~0x00002000;

        // 4) set the desired system divider

        RCC2 |= 0x40000000;   // use 400 MHz PLL

        RCC2 = (RCC2 &~ 0x1FC00000)  // clear system clock divider

                        + (SYSDIV<<22);      // configure for 80 MHz clock
```

```
        // 5) wait for the PLL to lock by polling PLLLRIS
//        while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
        // 6) enable use of PLL by clearing BYPASS
        RCC2 &= ~0x00000800;
        clockFrequency = (400/(SYSDIV + 1)) * 1000000;
    }
```

The steps 0 to 6 to activate the LM4F123/TM4C123 Launchpad with a 16 MHz main oscillator to run at 80 MHz

0) Use RCC2 because it provides for more options.

1) The first step is to set BYPASS2 (bit 11). At this point the PLL is bypassed and there is no system clock divider.

2) The second step is to specify the crystal frequency in the four XTAL bits using the code in Table. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source.

3) The third step is to clear PWRDN2 (bit 13) to activate the PLL.

4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n**+1. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field.

5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the **SYSCTL_RIS_R** to become high.

6) The last step is to connect the PLL by clearing the BYPASS2 bit. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider.

3.1.2.2. General-Purpose Input/Outputs (GPIOs)

The GPIO module is composed of six physical GPIO blocks, each corresponding to an individual GPIO port (Port A, Port B, Port C, Port D, Port E, Port F). The GPIO module supports up to 43 programmable input/output pins, depending on the peripherals being used.

Features:

Up to 43 GPIOs, depending on configuration.

- Highly flexible pin muxing allows use as GPIO or one of several peripheral functions.
- 5-V-tolerant in input configuration.
- Ports A-G accessed through the Advanced Peripheral Bus (APB).
- Fast toggle capable of a change every clock cycle for ports on AHB, every two clock cycles for ports on APB.
- Programmable control for GPIO interrupts
  - Interrupt generation masking.
  - Edge-triggered on rising, falling, or both.
    - Level-sensitive on High or Low values.
- Can be used to initiate an ADC sample sequence or a µDMA transfer.
- Pin state can be retained during Hibernation mode.
- Pins configured as digital inputs are Schmitt-triggered.
- Programmable control for GPIO pad configuration
  - Weak pull-up or pull-down resistors
  - 2-mA, 4-mA, and 8-mA pad drive for digital communication; up to four pads can sink 18-Ma.

  For high-current applications
  - Slew rate control for 8-mA pad drive.
  - Open drain enables.
  - Digital input enables.

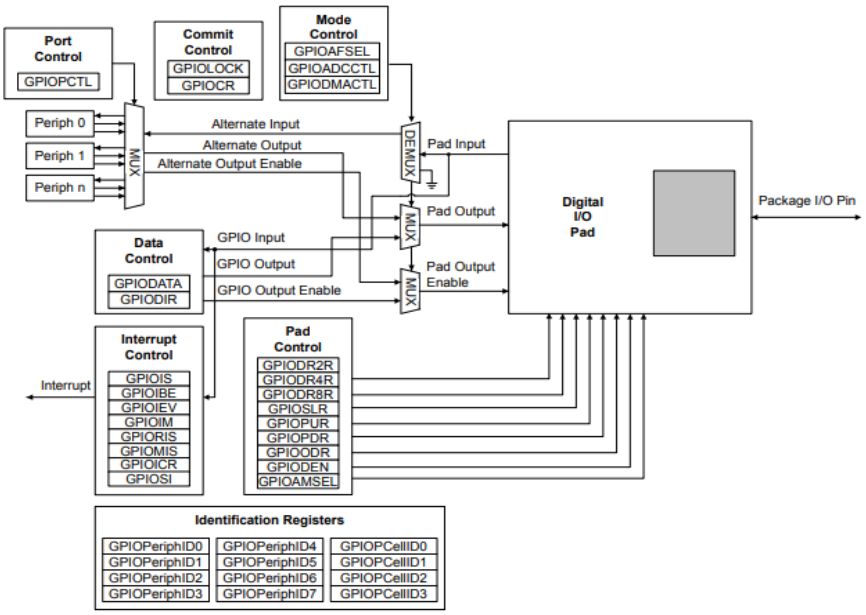


Figure 1.4: Tiva-C GPIO Block diagram.

In order to use GPIO pins we implement initialization and configuration function which include all required function to enable the desired function of the pin

```
void GPIO_voidInitializeDigitalPin(gpio_port_t PORT,char pin,gpio_portDirection_t MODE,gpio_drive_t drive)
{
    GPIO_voidSetBus(PORT, AHB);
    GPIO_voidConnectClock(PORT);
    volatile long int* reg = ((bus&PORT_AHB[PORT]) | ((~bus)&PORT_APB[PORT])) + PORT_DRR + drive;
    *reg = 0xFF;
    volatile long int* reg2 = ((bus&PORT_AHB[PORT]) | ((~bus)&PORT_APB[PORT])) + GPIOCR;
    *reg = 0xFF;
    GPIO_voidEnableDigitalFunction(PORT,pin);
    GPIO_voidDisableAlternateFunction(PORT,pin);
    GPIO_voidSetPinDir(PORT,pin,MODE);
}
```

The initialization function has four input parameters:

- The desired port (A, B, C, ……).
- The desired pin number (0, 1, ….,7).
- The direction of the pin (INPUT, Output, INPUT PULLUP or PULLDOWN, OUTPUT_OPEN_DRAIN).
- The desired current (2mA, 4mA, 8mA).

The GPIO modules may be accessed via two different memory apertures. The legacy aperture, the Advanced Peripheral Bus (APB), is

backwards-compatible with previous devices. The other aperture, the Advanced High-Performance Bus (AHB), offers the same register map but provides better back-to-back access performance than the APB bus. These apertures are mutually exclusive. The aperture enabled for a given GPIO port is controlled by the appropriate bit in the GPIOHBCTL register.

First function in the initialization function is S“GPIO_voidSetBus()” so after calling it we can choose the (APB) or (AHB).

```
void GPIO_voidSetBus(gpio_port_t PORT,gpio_BUS_t busUsed)
{
    bus = busUsed;

    GPIOHBCTL |= ((bus&(1 << PORT)));

}
```

Second function will be enabling the clock to the port by setting the appropriate bits in the RCGCGPIO register.

```
void GPIO_voidConnectClock(gpio_port_t PORT)
{
    RCGCGPIO |= (1 << PORT);

}
```

Configure the GPIOAFSEL register to program each bit as a GPIO or alternate pin. If an alternate pin is chosen for a bit, then the PMCx field must be programmed in the GPIOPCTL register for the specific peripheral required. There are also two registers, GPIOADCCTL and GPIODMACTL, which can be used to program a GPIO pin as ADC or µDMA trigger, respectively.

```
void GPIO_voidEnableDigitalFunction(gpio_port_t port,char pin)
{
    volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + PORT_AFSEL;
    *reg = ((*reg) & (~(1 << pin)));
    volatile long int* reg2 = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + PORT_DEN;
    *reg2 = ((*reg2) & (~(1 << pin))) | ((1 << pin));

}
```

Finally, to enable GPIO pins as digital I/Os, set the appropriate DEN bit in the GPIODEN register. To enable GPIO pins to their analog function (if

available), set the GPIOAMSEL bit in the GPIOAMSEL register.

```
void GPIO_voidSetPinDir(gpio_port_t port,char pin,gpio_portDirection_t MODE)
{

    volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + PORT_DIR;

    *reg = ((*reg) & (~(1 << pin))) | ((1 << pin) & MODE);

}
```

Data Control

There is another important function to call to write data on pins, it takes three parameters (Port, pin number, value high or low)

Then it accesses data register (GPIODATA) and write the value on it.

```
void GPIO_voidSetPin(gpio_port_t port,char pin,gpio_portValue_t value)
{

    volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + PORT_DATA;

    *reg = ((*reg) & (~(1 << pin))) | ((1 << pin) & value);

}
```

Interrupt Control

The interrupt capabilities of each GPIO port are controlled by a set of seven registers. These registers are used to select the source of the interrupt, its polarity, and the edge properties. When one or more GPIO inputs cause an interrupt, a single interrupt output is sent to the interrupt controller for the entire GPIO port. For edge-triggered interrupts, software must clear the interrupt to enable any further interrupts. For a level-sensitive interrupt, the external source must hold the level constant for the interrupt to be recognized by the controller.

Three registers define the edge or sense that causes interrupts:

■ GPIO Interrupt Sense (GPIOIS) register

■ GPIO Interrupt Both Edges (GPIOIBE) register

■ GPIO Interrupt Event (GPIOIEV) register

```
void GPIO_voidSetInterruptEvent(gpio_port_t port,char pin,interrupt_t event)
{
    volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + GPIO_IM;
    *reg |= (1 << pin);
    if(event == BothEdges)
    {
        reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x404;
        *reg |= (1 << pin);
        reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x408;
        *reg |= (1 << pin);
    }
    else if(event == FallingEdge)
    {
    volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x404 + (4 * event);
    *reg |= (1 << pin);
    }
    else
    {
        volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x40C;
        *reg &= ~(1 << pin);
    }

}
```

```
void (*PORTF_ISR_real)();
void serveInterrupt_PORTF(void* PORTF_ISR_ptr)
{
    (*PORTF_ISR_real)();
    volatile long int* reg = PORT_AHB[5] + GPIOICR;
    *reg |= 0xFF;
}
void setISR_PORTF(void* PORTF_ISR_virtual)
{
    RAM_Vector[46] = serveInterrupt_PORTF;
    PORTF_ISR_real = PORTF_ISR_virtual;

}
```

Interrupts are enabled/disabled via the GPIO Interrupt Mask (GPIOIM) register When an interrupt condition occurs, the state of the interrupt signal can be viewed in two locations:

the GPIO Raw Interrupt Status (GPIORIS)and GPIO Masked Interrupt Status (GPIOMIS) registers. As the name implies, the GPIOMIS register only shows interrupt conditions that are allowed to be passed to the interrupt controller.

For a GPIO level-detect interrupt, the interrupt signal generating the interrupt must be held until serviced. Once the input signal desserts from the interrupt generating logical sense, the corresponding RIS bit in the GPIORIS register clears. For a GPIO edge-detect interrupt, the RIS bit in the GPIORIS register is cleared by writing a ‘1’ to the corresponding bit in the GPIO Interrupt Clear (GPIOICR) register.

The corresponding GPIOMIS bit reflects the masked value of the RIS bit.

3.1.2.3. System Time (SysTick) and RTOS

Systick is simply a timer present inside ARM based microcontrollers. Basic purpose is to provide help generate accurate interrupts to different tasks (of RTOS). It has multiple uses aside from that. For example, many developers use it to generate an accurate delay function, Other benefits are portability where you can easily take an RTOS task from one microcontroller to a different one, and not end up changing the scheduling time and time dependent interrupts for tasks, as there can be different clock sources being used on the new microcontroller.

The processor has a 24-bit system timer, SysTick, that counts down from the reload value to zero, reloads (wraps to) the value in the LOAD register on the next clock edge, then counts down on subsequent clocks.

When the processor is halted for debugging the counter does not decrement.

SysTick register map:

Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STK_CTRL	Reserved															COUNTFLAG	Reserved													CLKSOURCE	TICK INT	EN ABLE
Reset Value																0														0	0	0
STK_LOAD	Reserved								RELOAD[23:0]																							
Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
STK_VAL	Reserved								CURRENT[23:0]																							
Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IN "STK_private.h" we save addresses of SysTick regigsters

```
/*****************************************************************************************************************************************/
/*                                                    Registers Definitions                                                              */
/* Address         : 0xE000E010-0xE000E01F                                                                                               */
/* Core peripheral : System timer                                                                                                        */
/* Register map    : Table 3-8 on page 134                                                                                               */
/*****************************************************************************************************************************************/
/* STK Base Address */
#define MSTK_BASE_ADDRESS       0xE000E010

/* STK Register */
typedef struct
{
    volatile uint32_t CTRL   ;          // SysTick Control and Status Register (STCTRL)
    volatile uint32_t RELOAD ;          // SysTick Reload Value Register (STRELOAD)
    volatile uint32_t CURRENT;          // SysTick Current Value Register (STCURRENT)
}MSTK_Type;

#define MSTK                    ((volatile MSTK_Type*)MSTK_BASE_ADDRESS)
/*****************************************************************************************************************************************/

/*****************************************************************************************************************************************/
/*                                                      Registers Bits                                                                   */
/* STK Registers : Section 3.3 Page 137                                                                                                  */
/*****************************************************************************************************************************************/
/* SysTick control and status register (STK_CTRL) */
#define COUNT_BIT               16  // Returns 1 if timer counted to 0 since last time this was read
#define CLK_SRC_BIT             2   // Clock source selection
#define INTEN_BIT               1   // Interrupt Enable
#define ENABLE_BIT              0   // Counter enable
/*****************************************************************************************************************************************/
```

Register 1: SysTick Control and Status Register (STCTRL), offset 0x010

Note: This register can only be accessed from privileged mode.

The SysTick **STCTRL** register enables the SysTick features.

SysTick Control and Status Register (STCTRL)

Base 0xE000.E000
Offset 0x010
Type RW, reset 0x0000.0004

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															COUNT
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved													CLK_SRC	INTEN	ENABLE
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Bit/Field	Name	Type	Reset	Description
31:17	reserved	RO	0x000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
16	COUNT	RO	0	Count Flag Value Description 0 The SysTick timer has not counted to 0 since the last time this bit was read. 1 The SysTick timer has counted to 0 since the last time this bit was read. This bit is cleared by a read of the register or if the **STCURRENT** register is written with any value. If read by the debugger using the DAP, this bit is cleared only if the MasterType bit in the **AHB-AP Control Register** is clear. Otherwise, the COUNT bit is not changed by the debugger read. See the *ARM® Debug Interface V5 Architecture Specification* for more information on MasterType.
15:3	reserved	RO	0x000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
2	CLK_SRC	RW	1	Clock Source Value Description 0 Precision internal oscillator (PIOSC) divided by 4 1 System clock

Bit/Field	Name	Type	Reset	Description
1	INTEN	RW	0	Interrupt Enable Value Description 0 Interrupt generation is disabled. Software can use the COUNT bit to determine if the counter has ever reached 0. 1 An interrupt is generated to the NVIC when SysTick counts to 0.
0	ENABLE	RW	0	Enable Value Description 0 The counter is disabled. 1 Enables SysTick to operate in a multi-shot way. That is, the counter loads the RELOAD value and begins counting down. On reaching 0, the COUNT bit is set and an interrupt is generated if enabled by INTEN. The counter then loads the RELOAD value again and begins counting.

Register 2: SysTick Reload Value Register (STRELOAD), offset 0x014

Note: This register can only be accessed from privileged mode.

The **STRELOAD** register specifies the start value to load into the **SysTick Current Value (STCURRENT)** register when the counter reaches 0. The start value can be between 0x1 and 0x00FF.FFFF. A start value of 0 is possible but has no effect because the SysTick interrupt and the COUNT bit are activated when counting from 1 to 0.

SysTick can be configured as a multi-shot timer, repeated over and over, firing every N+1 clock pulses, where N is any value from 1 to 0x00FF.FFFF. For example, if a tick interrupt is required every 100 clock pulses, 99 must be written into the RELOAD field.

Note that in order to access this register correctly, the system clock must be faster than 8 MHz.

SysTick Reload Value Register (STRELOAD)

Base 0xE000.E000
Offset 0x014
Type RW, reset -

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved								RELOAD							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELOAD															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:24	reserved	RO	0x00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
23:0	RELOAD	RW	0x00.0000	Reload Value Value to load into the **SysTick Current Value (STCURRENT)** register when the counter reaches 0.

Register 3: SysTick Current Value Register (STCURRENT), offset 0x018

Note: This register can only be accessed from privileged mode.

The **STCURRENT** register contains the current value of the SysTick counter.

SysTick Current Value Register (STCURRENT)

Base 0xE000.E000
Offset 0x018
Type RWC, reset -

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved								CURRENT							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CURRENT															
Type	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:24	reserved	RO	0x00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
23:0	CURRENT	RWC	0x00.0000	Current Value This field contains the current value at the time the register is accessed. No read-modify-write protection is provided, so change with care. This register is write-clear. Writing to it with any value clears the register. Clearing this register also clears the COUNT bit of the **STCTRL** register.

In “STK_interface.h” there to types CLKSource and TimeUnit, we may use them in the functions arguments.

```
typedef enum
{
    MSTK_SYS_CLK_DIV_4,
    MSTK_SYS_CLK
}MSTK_CLKSource_Type;

typedef enum
{
    MSTK_TIME_MS,
    MSTK_TIME_US
}MSTK_TimeUnit_Type;
```

In “STK_config.h” file we could Choose (Clock Source, SysTick State enable or not, Interrupt state, SysTick Clock and Number of tasks for RTOS)

```
/* MSTK_CLKSOURCE:  MSTK_SYS_CLK_DIV_4
                    MSTK_SYS_CLK       */
#define MSTK_CLKSOURCE      MSTK_SYS_CLK

/* MSTK_STATE:  MSTK_DISABLE
                MSTK_ENABLE  */
#define MSTK_STATE          MSTK_DISABLE

/* MSTK_INTERRUPT:  MSTK_DISABLE
                    MSTK_ENABLE  */
#define MSTK_INTERRUPT      MSTK_DISABLE

/* SYS Clock */
#define MSTK_SYS_CLK_CLK    80000000

/* RTOS */
#define MSTK_NUMBER_OF_TASKS 1
```

Initialization function to initialize (Clock Source, SysTick State enable or not and Interrupt state)

```
/* Initialization */
void MSTK_voidInit(void)
{
    /* Apply Clock Source */
    #if   MSTK_CLKSOURCE == MSTK_SYS_CLK_DIV_4
    SET_BIT(MSTK->CTRL,CLK_SRC_BIT);
        MSTK_u32Clk = MSTK_10000 / (MSTK_SYS_CLK_CLK / ( MSTK_1000000));
    #elif MSTK_CLKSOURCE == MSTK_SYS_CLK
        SET_BIT(MSTK->CTRL,CLK_SRC_BIT);
        MSTK_u32Clk = MSTK_10000 / (MSTK_SYS_CLK_CLK / MSTK_1000000);
    #else
        #error("Configuration Error :: Wrong MSTK_CLKSOURCE")
    #endif

    /* SysTick State */
    #if   MSTK_STATE == MSTK_DISABLE
        CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    #elif MSTK_STATE == MSTK_ENABLE
        SET_BIT(MSTK->CTRL,ENABLE_BIT);
    #else
        #error("Configuration Error :: Wrong MSTK_STATE")
    #endif

    /* SysTick Interrupt State */
    #if   MSTK_INTERRUPT == MSTK_DISABLE
        CLR_BIT(MSTK->CTRL,INTEN_BIT);
    #elif MSTK_INTERRUPT == MSTK_ENABLE
        SET_BIT(MSTK->CTRL,INTEN_BIT);
    #else
        #error("Configuration Error :: Wrong MSTK_INTERRUPT")
    #endif
}
```

Functions to Control and read the clock source

```
/* Select Clock Source */
void MSTK_voidSelectCLKSource(MSTK_CLKSource_Type Copy_CLKSource)
{
    switch (Copy_CLKSource)
    {
        case MSTK_SYS_CLK_DIV_4:
            CLR_BIT(MSTK->CTRL,CLK_SRC_BIT);
            MSTK_u32Clk = MSTK_SYS_CLK_CLK / MSTK_4;
            break;
        case MSTK_SYS_CLK       :
            SET_BIT(MSTK->CTRL,CLK_SRC_BIT);
            MSTK_u32Clk = MSTK_SYS_CLK_CLK;
            break;
        default                 :
            /* Do Nothing */
            break;
    }
}

/* Read Clock Source */
MSTK_CLKSource_Type MSTK_ReadCLKSource(void)
{
    MSTK_CLKSource_Type Local_CLKSource = GET_BIT(MSTK->CTRL,CLK_SRC_BIT);
    return Local_CLKSource;
}
```

Delay Function (Synchronous without Interrupt)

```
/* Synchronous */
void MSTK_voidSetBusyWait(uint32_t Copy_u32Time)
{
    uint32_t Local_u32Load = MSTK_INITIAL_VALUE;
    /* Restart Timer */
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    MSTK->CURRENT = MSTK_0;
    /* Calculation and Load Ticks to STK_LOAD register */
    Local_u32Load = (Copy_u32Time)-1;// - (MSTK_u32Clk/MSTK_10000);
    MSTK->RELOAD  = Local_u32Load;
    /* Start Timer */
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
    /* Wait till flag is raised when timer counted to 0 */
    while(!(GET_BIT(MSTK->CTRL,COUNT_BIT)));
    /* Stop Timer */
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    /* Load Zero to STK_LOAD */
    MSTK->RELOAD  = MSTK_0;
    MSTK->CURRENT = MSTK_0;
}
```

Interval Modes Functions

MSTK_voidSetIntervalSingle: At under-flow -> Disable STK Interrupt - Stop Timer - Load Zero to STK_LOAD.

MSTK_voidSetIntervalPeriodic: At under-flow -> Doesn’t Disable STK Interrupt - Doesn’t Stop Timer - Doesn’t Load Zero to STK_LOAD.

MSTK_voidStart: Doesn't use interrupt and Loads Maximum Value to STK_LOAD (0xFFFFFF).

```c
/* Asynchronous */
void MSTK_voidSetIntervalSingle(uint32_t Copy_u32Time, void (*Copy_ptr)(void))
{
    uint32_t Local_u32Load = MSTK_INITIAL_VALUE;
    /* Restart Timer */
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    MSTK->CURRENT  = MSTK_0;
    /* Calculation and Load Ticks to STK_LOAD register */
    /* Calculation and Load Ticks to STK_LOAD register */
    Local_u32Load = (Copy_u32Time/MSTK_10000) - (MSTK_u32Clk);
    MSTK->RELOAD  = Local_u32Load;
    /* Save CallBack */
    MSTK_CallBack = Copy_ptr;
    /* Start Timer */
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
    /* Set Mode to Single */
    MSTK_u8IntervalMode = MSTK_SINGLE_INTERVAL;
    /* Enable STK Interrupt */
    SET_BIT(MSTK->CTRL,INTEN_BIT);
}

void MSTK_voidSetIntervalPeriodic(uint32_t Copy_u32Time, void (*Copy_ptr)(void))
{
    uint32_t Local_u32Load = MSTK_INITIAL_VALUE;
    /* Restart Timer */
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    MSTK->CURRENT  = MSTK_0;
    /* Calculation and Load Ticks to STK_LOAD register */
    /* Calculation and Load Ticks to STK_LOAD register */
    Local_u32Load = (Copy_u32Time/MSTK_10000) - (MSTK_u32Clk);
    MSTK->RELOAD  = Local_u32Load;
    /* Save CallBack */
    MSTK_CallBack = Copy_ptr;
    /* Start Timer */
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
    /* Set Mode to Single */
    MSTK_u8IntervalMode = MSTK_PERIOD_INTERVAL;
    /* Enable STK Interrupt */
    SET_BIT(MSTK->CTRL,INTEN_BIT);
}
void MSTK_voidStart(void)
{
    /* Load Maximum Value to STK_LOAD */
    MSTK->RELOAD = MSTK_0xFFFFFF;
    /* Start Timer */
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
}
```

Get Timer Value Functions

```c
/* Get Time Value */
uint32_t  MSTK_u32GetElapsedTime(MSTK_TimeUnit_Type Copy_Unit)
{
    uint32_t Local_u32TicksValue  = MSTK->RELOAD - MSTK->CURRENT;
    uint32_t Local_u32ElapsedTime = MSTK_INITIAL_VALUE;
    switch (Copy_Unit)
    {
        case MSTK_TIME_MS: Local_u32ElapsedTime = Local_u32TicksValue/(MSTK_u32Clk/MSTK_1000);      break;
        case MSTK_TIME_US: Local_u32ElapsedTime = Local_u32TicksValue/(MSTK_u32Clk/MSTK_1000000);   break;
        default          : /* Do Nothing */                                                        break;
    }
    return Local_u32ElapsedTime;
}

uint32_t  MSTK_u32GetRemainingTime(MSTK_TimeUnit_Type Copy_Unit)
{
    uint32_t Local_u32TicksValue    = MSTK->CURRENT;
    uint32_t Local_u32RemainingTime = MSTK_INITIAL_VALUE;
    switch (Copy_Unit)
    {
        case MSTK_TIME_MS: Local_u32RemainingTime = Local_u32TicksValue/(MSTK_u32Clk/MSTK_1000);     break;
        case MSTK_TIME_US: Local_u32RemainingTime = Local_u32TicksValue/(MSTK_u32Clk/MSTK_1000000); break;
        default          : /* Do Nothing */                                                        break;
    }
    return Local_u32RemainingTime;
}
```

SysTick Interrupts Functions

```c
/* Interrupt */
void MSTK_voidEnableInterrupt(void)
{
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
}
void MSTK_voidDisableInterrupt(void)
{
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
}
void MSTK_voidClearInterruptFlag(void)
{
    u8 Local_u8InterruptFlag = GET_BIT(MSTK->CTRL,COUNT_BIT);
}
u8   MSTK_u8ReadInterruptFlag(void)                          /* This Function Clears Interrupt Flag */
{
    u8 Local_u8InterruptFlag = GET_BIT(MSTK->CTRL,COUNT_BIT);
    return Local_u8InterruptFlag;
}
void MSTK_voidSetCallBack(void (*Copy_ptr)(void))
{
    MSTK_CallBack = Copy_ptr;
}
```

RTOS

A Real Time Operating System, commonly known as an RTOS, is a software component that rapidly switches between tasks, giving the impression that multiple programs are being executed at the same time on a single processing core.

In actual fact the processing core can only execute one program at any one time, and what the RTOS is actually doing is rapidly switching between individual programming threads (or Tasks) to give the impression that multiple programs are executing simultaneously.

OS or RTOS?

The difference between an OS (Operating System) such as Windows or Unix and an RTOS (Real Time Operating System) found in embedded systems, is the response time to external events. OS's typically provide a non-deterministic, soft real time response, where there are no guarantees as to when each task will complete, but they will try to stay responsive to the user. An RTOS differs in that it typically provides a hard real time response, providing a fast, highly deterministic reaction to external events. The difference between the two can be highlighted through examples – compare, for example, the editing of a document on a PC to the operation of a precision motor control.

Scheduling Algorithms

When switching between Tasks the RTOS has to choose the most appropriate task to load next. There are several scheduling algorithms available, including Round Robin, Co-operative and Hybrid scheduling. However, to provide a responsive system most RTOS's use a pre-emptive scheduling algorithm.

Tasks and Priorities

In a pre-emptive system each Task is given an individual priority value. The faster the required response, the higher the priority level assigned. When working in pre-emptive mode, the task chosen to execute is the highest priority task that is able to execute. This results in a highly responsive system.

Implementation of RTOS in STK driver

```
struct task
{
    void (*taskISR)(void);
    uint32_t periodicity;
};

struct task MyTasks[MSTK_NUMBER_OF_TASKS] ;
```

```
void MSTK_voidStartScheduler()
{
    MSTK_voidInit();
    uint32_t Local_u32Load = MSTK_INITIAL_VALUE;
    /* Restart Timer */
    CLR_BIT(MSTK->CTRL,ENABLE_BIT);
    MSTK->CURRENT  = MSTK_0;
    /* Calculation and Load Ticks to STK_LOAD register */
    /* Calculation and Load Ticks to STK_LOAD register */
    Local_u32Load = 80000;
    MSTK->RELOAD  = Local_u32Load;
    /* Save CallBack */
    MSTK_CallBack = MSTK_voidScheduleTasks;
    /* Start Timer */
    SET_BIT(MSTK->CTRL,ENABLE_BIT);
    /* Set Mode to Period */
    MSTK_u8IntervalMode = MSTK_PERIOD_INTERVAL;
    /* Enable STK Interrupt */
    SET_BIT(MSTK->CTRL,INTEN_BIT);

}
void MSTK_voidScheduleTasks()
{
    static uint32_t counter = 0;
    u8 i;
    for(i = 0 ; i < 10 ; i++)
    {
        if((!(counter % MyTasks[i].periodicity)) && (counter > 0))
        {
            (*MyTasks[i].taskISR)();
        }
    }
    counter++;
}

void MSTK_voidCreateTask(u8 Copy_u8TaskID, u16 Copy_u16Period, void (*Copy_voidFPtr)(void))
{
    MyTasks[Copy_u8TaskID].periodicity =Copy_u16Period;
    MyTasks[Copy_u8TaskID].taskISR     =Copy_voidFPtr;
}
```

3.1.2.4. Interrupt

In TM4C123GH6PM Architecture, we have to enable the required interrupt vectors of the NVIC from the Cortex M4 Peripheral

Register 4: Interrupt 0-31 Set Enable (EN0), offset 0x100

Register 5: Interrupt 32-63 Set Enable (EN1), offset 0x104

Register 6: Interrupt 64-95 Set Enable (EN2), offset 0x108

Register 7: Interrupt 96-127 Set Enable (EN3), offset 0x10C

Note: This register can only be accessed from privileged mode.

The **ENn** registers enable interrupts and show which interrupts are enabled. Bit 0 of **EN0** corresponds to Interrupt 0; bit 31 corresponds to Interrupt 31. Bit 0 of **EN1** corresponds to Interrupt 32; bit 31 corresponds to Interrupt 63. Bit 0 of **EN2** corresponds to Interrupt 64; bit 31 corresponds to Interrupt 95. Bit 0 of **EN3** corresponds to Interrupt 96; bit 31 corresponds to Interrupt 127. Bit 0 of **EN4** (see page 143) corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138.

See Table 2-9 on page 104 for interrupt assignments.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

Registers:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Enable

Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, enables the interrupt.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **DISn** register.

Register 8: Interrupt 128-138 Set Enable (EN4), offset 0x110

Note: This register can only be accessed from privileged mode.

The **EN4** register enables interrupts and shows which interrupts are enabled. Bit 0 corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138. See Table 2-9 on page 104 for interrupt assignments.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

Interrupt 128-138 Set Enable (EN4)
Base 0xE000.E000
Offset 0x110
Type RW, reset 0x0000.0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved					INT										
Type	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:11	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
10:0	INT	RW	0x0	Interrupt Enable

Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, enables the interrupt.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **DIS4** register.

- *Registers Load Function:*

In interrupt driver, we added Interrupt_voidEnable () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to enable it.

```
24  void Interrupt_voidEnable(unsigned char INT_ID)
25  {
26      volatile long int* reg = 0xE000E000 + 0x100 + (INT_ID/32);
27
28      *reg |= (1 << (INT_ID%32));
29  }
```

In line 26 we access the correct register from the five EN Registers by the mathematical pattern INT_ID/32 + 0x100, where 0x100 represents the offset of EN0, and we divide the ID by 32 because every register is 32 bits.

while in line 28 we use the remainder of division to set the correct bit in the chosen register.

Same logic exactly is applied to the disable function, set pending, clear pending, and set active functions, but with different offsets for every register:

Register 14: Interrupt 0-31 Set Pending (PEND0), offset 0x200

Register 15: Interrupt 32-63 Set Pending (PEND1), offset 0x204

Register 16: Interrupt 64-95 Set Pending (PEND2), offset 0x208

Register 17: Interrupt 96-127 Set Pending (PEND3), offset 0x20C

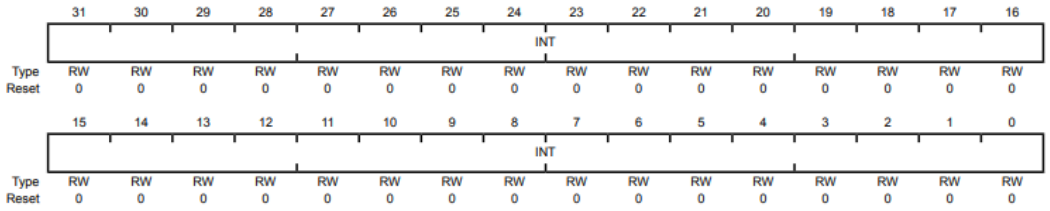
Note: This register can only be accessed from privileged mode.

The **PENDn** registers force interrupts into the pending state and show which interrupts are pending. Bit 0 of **PEND0** corresponds to Interrupt 0; bit 31 corresponds to Interrupt 31. Bit 0 of **PEND1** corresponds to Interrupt 32; bit 31 corresponds to Interrupt 63. Bit 0 of **PEND2** corresponds to Interrupt 64; bit 31 corresponds to Interrupt 95. Bit 0 of **PEND3** corresponds to Interrupt 96; bit 31 corresponds to Interrupt 127. Bit 0 of **PEND4** (see page 147) corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138.

See Table 2-9 on page 104 for interrupt assignments.

Interrupt 0-31 Set Pending (PEND0)

Base 0xE000.E000
Offset 0x200
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Set Pending

Value	Description
0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, the corresponding interrupt is set to pending even if it is disabled.

If the corresponding interrupt is already pending, setting a bit has no effect.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **UNPEND0** register.

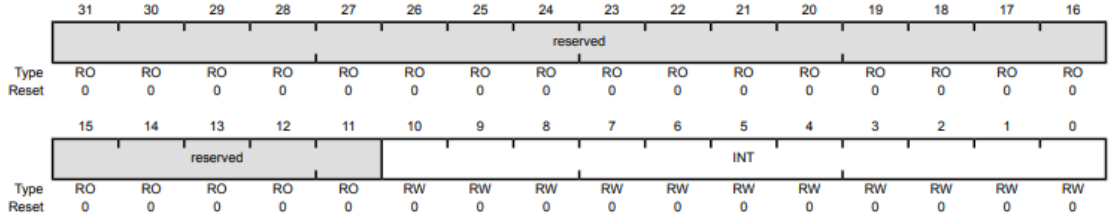
Register 18: Interrupt 128-138 Set Pending (PEND4), offset 0x210

Note: This register can only be accessed from privileged mode.

The **PEND4** register forces interrupts into the pending state and shows which interrupts are pending. Bit 0 corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138. See Table 2-9 on page 104 for interrupt assignments.

Interrupt 128-138 Set Pending (PEND4)

Base 0xE000.E000
Offset 0x210
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:11	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
10:0	INT	RW	0x0	Interrupt Set Pending

Value	Description
0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, the corresponding interrupt is set to pending even if it is disabled.

If the corresponding interrupt is already pending, setting a bit has no effect.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **UNPEND4** register.

- *Registers Load Function:*

In interrupt driver, we added Interrupt_voidSetPending () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to set it pending.

```
37  void Interrupt_voidSetPending(unsigned char INT_ID)
38  {
39      volatile long int* reg = 0xE000E000 + 0x200 + (INT_ID/32);
40      *reg |= (1 << (INT_ID%32));
41
42  }
```

Register 19: Interrupt 0-31 Clear Pending (UNPEND0), offset 0x280

Register 20: Interrupt 32-63 Clear Pending (UNPEND1), offset 0x284

Register 21: Interrupt 64-95 Clear Pending (UNPEND2), offset 0x288

Register 22: Interrupt 96-127 Clear Pending (UNPEND3), offset 0x28C

Note: This register can only be accessed from privileged mode.

The **UNPENDn** registers show which interrupts are pending and remove the pending state from interrupts. Bit 0 of **UNPEND0** corresponds to Interrupt 0; bit 31 corresponds to Interrupt 31. Bit 0 of **UNPEND1** corresponds to Interrupt 32; bit 31 corresponds to Interrupt 63. Bit 0 of **UNPEND2** corresponds to Interrupt 64; bit 31 corresponds to Interrupt 95. Bit 0 of **UNPEND3** corresponds to Interrupt 96; bit 31 corresponds to Interrupt 127. Bit 0 of **UNPEND4** (see page 149) corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138.

See Table 2-9 on page 104 for interrupt assignments.

Interrupt 0-31 Clear Pending (UNPEND0)

Base 0xE000.E000
Offset 0x280
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Clear Pending

Value	Description
0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, clears the corresponding `INT[n]` bit in the **PEND0** register, so that interrupt [n] is no longer pending. Setting a bit does not affect the active state of the corresponding interrupt.

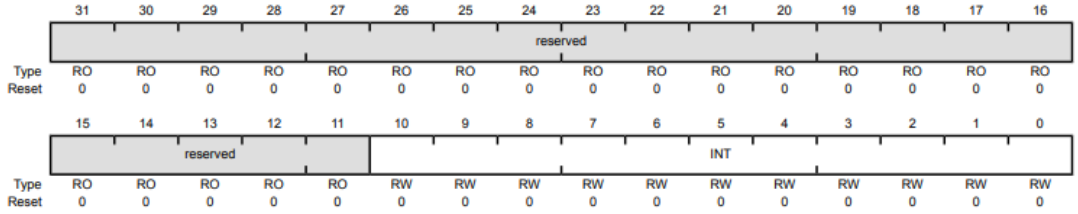
Register 23: Interrupt 128-138 Clear Pending (UNPEND4), offset 0x290

Note: This register can only be accessed from privileged mode.

The **UNPEND4** register shows which interrupts are pending and removes the pending state from interrupts. Bit 0 corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138. See Table 2-9 on page 104 for interrupt assignments.

Interrupt 128-138 Clear Pending (UNPEND4)

Base 0xE000.E000
Offset 0x290
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:11	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
10:0	INT	RW	0x0	Interrupt Clear Pending

Value	Description
0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, clears the corresponding `INT[n]` bit in the **PEND4** register, so that interrupt [n] is no longer pending. Setting a bit does not affect the active state of the corresponding interrupt.

- *Registers Load Function:*

  In interrupt driver, we added Interrupt_voidClearPending () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to clear it from pending states.

```
44  void Interrupt_voidClearPending(unsigned char INT_ID)
45  {
46      volatile long int* reg = 0xE000E000 + 0x280 + (INT_ID/32);
47      *reg |= (1 << (INT_ID%32));
48
49  }
```

Register 24: Interrupt 0-31 Active Bit (ACTIVE0), offset 0x300

Register 25: Interrupt 32-63 Active Bit (ACTIVE1), offset 0x304

Register 26: Interrupt 64-95 Active Bit (ACTIVE2), offset 0x308

Register 27: Interrupt 96-127 Active Bit (ACTIVE3), offset 0x30C

Note: This register can only be accessed from privileged mode.

The **UNPENDn** registers indicate which interrupts are active. Bit 0 of **ACTIVE0** corresponds to Interrupt 0; bit 31 corresponds to Interrupt 31. Bit 0 of **ACTIVE1** corresponds to Interrupt 32; bit 31 corresponds to Interrupt 63. Bit 0 of **ACTIVE2** corresponds to Interrupt 64; bit 31 corresponds to Interrupt 95. Bit 0 of **ACTIVE3** corresponds to Interrupt 96; bit 31 corresponds to Interrupt 127. Bit 0 of **ACTIVE4** (see page 151) corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138.

See Table 2-9 on page 104 for interrupt assignments.

Caution – Do not manually set or clear the bits in this register.

Interrupt 0-31 Active Bit (ACTIVE0)

Base 0xE000.E000
Offset 0x300
Type RO, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INT															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INT															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RO	0x0000.0000	Interrupt Active

Value	Description
0	The corresponding interrupt is not active.
1	The corresponding interrupt is active, or active and pending.

- *Registers Load Function:*

  In interrupt driver, we added Interrupt_voidClearPending () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to clear it from pending states.

Register 28: Interrupt 128-138 Active Bit (ACTIVE4), offset 0x310

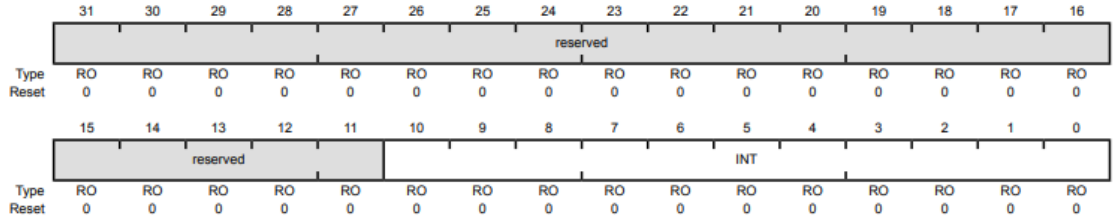
Note: This register can only be accessed from privileged mode.

The **ACTIVE4** register indicates which interrupts are active. Bit 0 corresponds to Interrupt 128; bit 10 corresponds to Interrupt 131. See Table 2-9 on page 104 for interrupt assignments.

Caution – Do not manually set or clear the bits in this register.

Interrupt 128-138 Active Bit (ACTIVE4)

Base 0xE000.E000
Offset 0x310
Type RO, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:11	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
10:0	INT	RO	0x0	Interrupt Active Value Description 0 The corresponding interrupt is not active. 1 The corresponding interrupt is active, or active and pending.

- *Registers Load Function:*

  In interrupt driver, we added Interrupt_voidSetActive () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to set it active.

```
57  void Interrupt_voidSetActive(unsigned char INT_ID)
58  {
59      volatile long int* reg = 0xE000E000 + 0x300 + (INT_ID/32);
60      *reg |= (1 << (INT_ID%32));
61  }
```

Register 9: Interrupt 0-31 Clear Enable (DIS0), offset 0x180

Register 10: Interrupt 32-63 Clear Enable (DIS1), offset 0x184

Register 11: Interrupt 64-95 Clear Enable (DIS2), offset 0x188

Register 12: Interrupt 96-127 Clear Enable (DIS3), offset 0x18C

Note: This register can only be accessed from privileged mode.

The **DISn** registers disable interrupts. Bit 0 of **DIS0** corresponds to Interrupt 0; bit 31 corresponds to Interrupt 31. Bit 0 of **DIS1** corresponds to Interrupt 32; bit 31 corresponds to Interrupt 63. Bit 0 of **DIS2** corresponds to Interrupt 64; bit 31 corresponds to Interrupt 95. Bit 0 of **DIS3** corresponds to Interrupt 96; bit 31 corresponds to Interrupt 127. Bit 0 of **DIS4** (see page 145) corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138.

See Table 2-9 on page 104 for interrupt assignments.

Interrupt 0-31 Clear Enable (DIS0)

Base 0xE000.E000
Offset 0x180
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Disable

Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, clears the corresponding `INT[n]` bit in the **EN0** register, disabling interrupt [n].

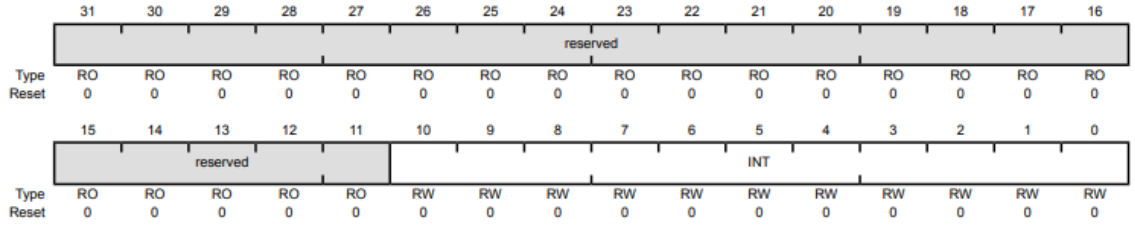
Register 13: Interrupt 128-138 Clear Enable (DIS4), offset 0x190

Note: This register can only be accessed from privileged mode.

The **DIS4** register disables interrupts. Bit 0 corresponds to Interrupt 128; bit 10 corresponds to Interrupt 138. See Table 2-9 on page 104 for interrupt assignments.

Interrupt 128-138 Clear Enable (DIS4)

Base 0xE000.E000
Offset 0x190
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:11	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
10:0	INT	RW	0x0	Interrupt Disable Value Description 0 On a read, indicates the interrupt is disabled. On a write, no effect. 1 On a read, indicates the interrupt is enabled. On a write, clears the corresponding `INT[n]` bit in the **EN4** register, disabling interrupt [n].

- *<u>Registers Load Function:</u>*

  In interrupt driver, we added Interrupt_voidDisable () function that accepts a parameter INT_ID which represents the Interrupt Vector Number IRQ to disable.

```
30  void Interrupt_voidDisable(unsigned char INT_ID)
31  {
32      volatile long int* reg = 0xE000E000 + 0x180 + (INT_ID/32);
33      *reg |= (1 << (INT_ID%32));
34
35  }
```

```
62  unsigned char Interrupt_voidGetState(unsigned char INT_ID)
63  {
64      volatile long int* reg = 0xE000E000 + 0x300 + (INT_ID/32);
65      return (*reg & (1 << INT_ID%32));
66
67  }
68
```

This function reads the ACTIVE Register of the required interrupt vector by the user and returns its state by the same logic explained in the previous functions.

```
51  unsigned char Interrupt_voidGetPending(unsigned char INT_ID)
52  {
53      volatile long int* reg1 = 0xE000E000 + 0x200 + (INT_ID/32);
54      volatile long int* reg2 = 0xE000E000 + 0x280 + (INT_ID/32);
55      return ((*reg1 & (1 << INT_ID%32)) - (*reg2 & (1 << INT_ID%32)));
56  }
```

This function reads the PEND Register of the required interrupt vector by the user and subtracts it from the UNPEND Register then returns the result by the same logic explained in the previous functions.

So if both bits in PEND and UNPEND are set, then it will return 0 (because 1 – 1 = 0), which is true because if the bit is set in UNPEND Register is set then the IRQ will be cleared from pending state even if it is set in PEND Register.

If bit of the PEND Register is only set then it will return 1 which means set pending.

If both cleared, it will return 0 because it is not pending.

If the UNPEND bit only is set it will return 255 (unsigned char), which is equivalent to -1 in signed char data type.

In order to make the interrupt driver dynamic, we must load the NVIC table from the flash memory to the RAM so as we can modify it in run time without the need to modify the startup code.

```
7   #pragma DATA_ALIGN(RAM_Vector,1024)
8   #pragma DATA_SELECTION(RAM_Vector,".vTable")
9   extern void (* const g_pfnVectors[])(void);
10  void (*RAM_Vector[155])(void);
11  void Load_Vector_Table(void)
12  {
13
14      volatile unsigned long int* reg;
15      int i = 0;
16      for(i = 0 ; i < 155 ; i++)
17      {
18          RAM_Vector[i] = g_pfnVectors[i];
19      }
20      reg = 0xE000ED08;
21      *reg = (unsigned long int)RAM_Vector;
22  }
```

In lines 7 and 8 we choose a place in RAM with zeros in Least significant bits.

This is the function of the loop starting in line 16 that loops 156 to transfer all interrupt vectors even the reserved ones from the flash memory in array g_pfnVectors to RAM in array RAM_Vector to protect the program from crash during runtime or reset.

We must then tell the compiler where is the NVIC table after transfer to call ISRs from it instead of that in the flash memory. This can be done via VTABLE Register.

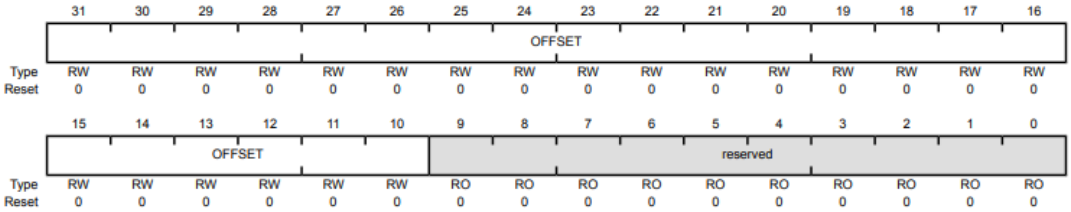
Register 68: Vector Table Offset (VTABLE), offset 0xD08

Note: This register can only be accessed from privileged mode.

The **VTABLE** register indicates the offset of the vector table base address from memory address 0x0000.0000.

Vector Table Offset (VTABLE)

Base 0xE000.E000
Offset 0xD08
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:10	OFFSET	RW	0x000.00	Vector Table Offset When configuring the `OFFSET` field, the offset must be aligned to the number of exception entries in the vector table. Because there are 138 interrupts, the offset must be aligned on a 1024-byte boundary.
9:0	reserved	RO	0x00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.

So we just need to access VTABLE Register and load it with the address or pointer to our RAM_Vector array in lines 20 and 21 respectively.

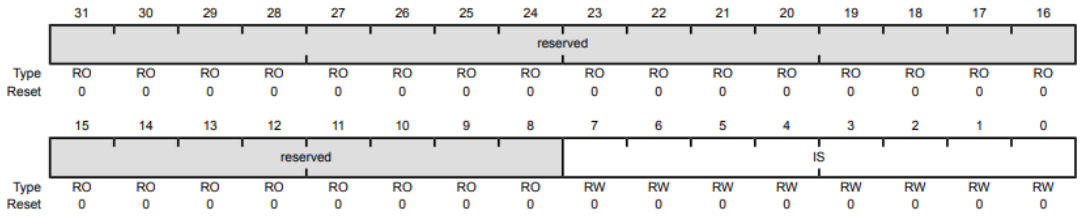
- In TM4C123GH6PM Architecture, we can control the GPIO interrupt through 5 registers:

a) GPIO Interrupt Sense (GPIOIS), offset 0x404:

The **GPIOIS** register is the interrupt sense register. Setting a bit in the **GPIOIS** register configures the corresponding pin to detect levels, while clearing a bit configures the corresponding pin to detect edges. All bits are cleared by a reset.

Note: To prevent false interrupts, the following steps should be taken when re-configuring GPIO edge and interrupt sense registers:

1. Mask the corresponding port by clearing the `IME` field in the **GPIOIM** register.
2. Configure the `IS` field in the **GPIOIS** register and the `IBE` field in the **GPIOIBE** register.
3. Clear the **GPIORIS** register.
4. Unmask the port by setting the `IME` field in the **GPIOIM** register.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

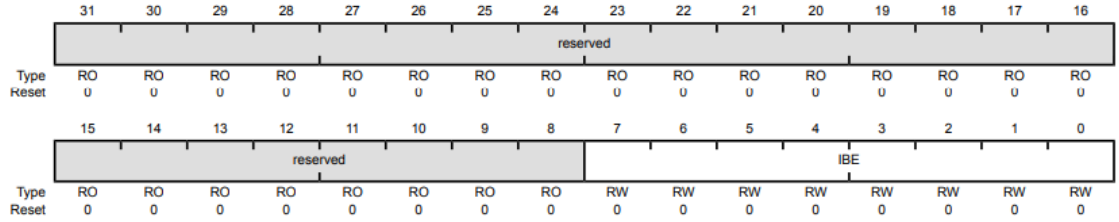
Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

b) GPIO Interrupt Both Edges (GPIOIBE), offset 0x408:

The **GPIOIBE** register allows both edges to cause interrupts. When the corresponding bit in the **GPIO Interrupt Sense (GPIOIS)** register (see page 664) is set to detect edges, setting a bit in the **GPIOIBE** register configures the corresponding pin to detect both rising and falling edges, regardless of the corresponding bit in the **GPIO Interrupt Event (GPIOIEV)** register (see page 666). Clearing a bit configures the pin to be controlled by the **GPIOIEV** register. All bits are cleared by a reset.

Note: To prevent false interrupts, the following steps should be taken when re-configuring GPIO edge and interrupt sense registers:

1. Mask the corresponding port by clearing the `IME` field in the **GPIOIM** register.
2. Configure the `IS` field in the **GPIOIS** register and the `IBE` field in the **GPIOIBE** register.
3. Clear the **GPIORIS** register.
4. Unmask the port by setting the `IME` field in the **GPIOIM** register.

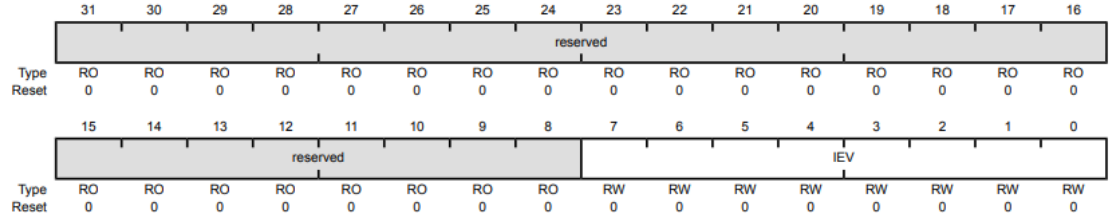


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 666).
1	Both edges on the corresponding pin trigger an interrupt.

c) GPIO Interrupt Event (GPIOIEV), offset 0x40C:

The **GPIOIEV** register is the interrupt event register. Setting a bit in the **GPIOIEV** register configures the corresponding pin to detect rising edges or high levels, depending on the corresponding bit value in the **GPIO Interrupt Sense (GPIOIS)** register (see page 664). Clearing a bit configures the pin to detect falling edges or low levels, depending on the corresponding bit value in the **GPIOIS** register. All bits are cleared by a reset.

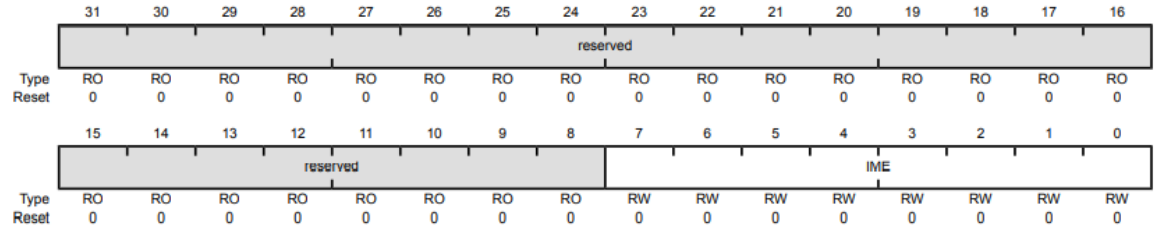


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

d) GPIO Interrupt Mask (GPIOIM), offset 0x410:

The **GPIOIM** register is the interrupt mask register. Setting a bit in the **GPIOIM** register allows interrupts that are generated by the corresponding pin to be sent to the interrupt controller on the combined interrupt signal. Clearing a bit prevents an interrupt on the corresponding pin from being sent to the interrupt controller. All bits are cleared by a reset.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable

Value	Description
0	The interrupt from the corresponding pin is masked.
1	The interrupt from the corresponding pin is sent to the interrupt controller.

- *Registers Load Function:*

  GPIO_voidSetInterruptEvent() function receives two parameters from the user, which are the port that the user wants to mask its interrupt, the pin needed to enable its interrupt, and the interrupt event.

```
105  void GPIO_voidSetInterruptEvent(gpio_port_t port,char pin,interrupt_t event)
106  {
107      volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + GPIO_IM;
108      *reg |= (1 << pin);
109      if(event == BothEdges)
110      {
111          reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x404;
112          *reg |= (1 << pin);
113          reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x408;
114          *reg |= (1 << pin);
115      }
116      else if(event == FallingEdge)
117      {
118      volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x404 + (4 * event);
119      *reg |= (1 << pin);
120      }
121      else
122      {
123          volatile long int* reg = ((bus&PORT_AHB[port]) | ((~bus)&PORT_APB[port])) + 0x40C;
124          *reg &= ~(1 << pin);
125      }
126
127  }
```

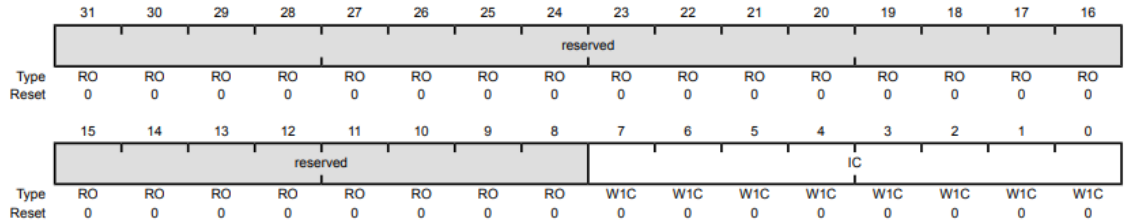
  In lines 107 and 108 in GPIO Driver, we access GPIOIM Register and set the required pin.

  Then in line 109, we checked if the event required is BothEdges to access the GPIOIS and the GPIOIBE Registers to set the required pins. Then in line 116, we checked if event is falling edge to access GPIOIEV Register in line 118 then set the required pin in line 119.

  Else the event is rising edge, so accessed GPIOIEV Register in line 123 then cleared the required pin in line 124.

e) GPIO Interrupt Clear (GPIOICR), offset 0x41C:

The **GPIOICR** register is the interrupt clear register. For edge-detect interrupts, writing a 1 to the `IC` bit in the **GPIOICR** register clears the corresponding bit in the **GPIORIS** and **GPIOMIS** registers. If the interrupt is a level-detect, the `IC` bit in this register has no effect. In addition, writing a 0 to any of the bits in the **GPIOICR** register has no effect.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

The main problem we wanted to solve was that we have to clear the interrupt flag in every ISR, which will cause a program crash if the user forgot and also will force the user to read the datasheet which is not of his business.

So in our problem we focused on this problem, and here is how we solved it:

We divided the ISR function called by the hardware into 2 parts, the first is a pointer to the function that the user want to be called when the interrupt event occurs, and the second is the clear of the flag. So the user now need only to write his ISR function without bothering himself with the datasheet.

```
144 void serveInterrupt_PORTF(void* PORTF_ISR_ptr)
145 {
146     (*PORTF_ISR_real)();
147     volatile long int* reg = PORT_AHB[5] + GPIOICR;
148     *reg |= 0xFF;
149 }
```

This is the ISR function called by the hardware. Notice that in line 146 we call pointer to the real ISR the user wants only to be executed. Then in lines 147 and 148 we mask and clear the flag which does not concern the user.

```
150 void setISR_PORTF(void* PORTF_ISR_virtual)
151 {
152     RAM_Vector[46] = serveInterrupt_PORTF;
153     PORTF_ISR_real = PORTF_ISR_virtual;
154 
155 }
```

This is the function that accepts a pointer to his ISR function and can be called in application layer drivers. You can see that in line 153 it sets the virtual ISR written by the user to the real ISR called by the hardware in serveInterrupt_PORTF() function. While in line 152 it loads the vector location of the interrupt in NVIC table with a pointer to the serveInterrupt_PORTF() function.

3.1.2.5. UART

UART_config.h and UART_config.c: To config UART State enable or disable, data word length, parity, receive or transmission, stop bits 1 or 2, UART mode and Select baud rate.

UART_config.c | UART_config.h

```
/* Number of UART Channels
#define MUART_MAX_CH      8

/* System Clock */
#define MUART_SYS_CLK     160

/* UART State Type */
typedef enum
{
      DISABLE,
      ENABLE
}MUART_State_Type;

/* Data Length Type */
typedef enum
{
      DATA_5BIT,
      DATA_6BIT,
      DATA_7BIT,
      DATA_8BIT
}MUART_WordLength_Type;

/* Parity Type */
typedef enum
{     NO_PARITY    = 0   ,
      EVEN_PARITY  = 6   ,
      ODD_PARITY   = 2   ,
      STICK0       = 134,
      STICK1       = 130
}MUART_Parity_Type;

/* RT Control Type */
typedef enum
{
      ENABLE_RECEIVER    ,
      ENABLE_TRANSMITTER,
      ENABLE_BOTH
}MUART_RTControl_Type;

/* Stop Bits Type */
typedef enum
{
    STOP_1BIT,
    STOP_2BITS
}MUART_StopBits_Type;

/* Configuration Type */
typedef struct
{
    MUART_State_Type       UART_State ;
    MUART_State_Type       FIFO_State ;
    MUART_WordLength_Type  WordLength ;
    MUART_Parity_Type      ParityMode ;
    MUART_RTControl_Type   RTControl  ;
    MUART_StopBits_Type    StopBits   ;
    u32                    u32BaudRate;
}MUART_config_Type;
```

UART_config.h | UART_config.c

```c
/*****************************************************************************
/*                                                            Header Files Inclu
/*****************************************************************************
/* Libraries */
#include "STD_TYPES.h"

/* Configuration */
#include "UART_config.h"
/*****************************************************************************

/*****************************************************************************
/*                                                                Configuration
/* UART Registers : Section
/*****************************************************************************
const MUART_config_Type MUART_Config[MUART_MAX_CH] =
{
    {ENABLE , ENABLE , DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {ENABLE , ENABLE , DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600},
    {DISABLE, DISABLE, DATA_8BIT, NO_PARITY, ENABLE_BOTH, STOP_1BIT, 9600}
};
```

UART_interface: Some of UART functions

- UARTR_voidInit(): Initialization function to Set the defines of the Config files.

```c
/* Initialization */
void MUART_voidInit(void)
{
    MUART_CH_Type Local_Counter = UART0;
    for (Local_Counter=UART0; Local_Counter<MAX_UARTS; Local_Counter++)
    {
        if      (MUART_Config[Local_Counter].UART_State == ENABLE)
        {
            /* Clear Control Registers */
//          MUART[Local_Counter]->LCRH = MUART_0;
//          MUART[Local_Counter]->CTL  = MUART_0;
            /* Connect Clock */
            SET_BIT(RCGCUART,Local_Counter);
            /* Configure Baud Rate */
            MUART_voidSetBaudRate(Local_Counter,MUART_Config[Local_Counter].u32BaudRate);
            /* Configure Word Length */
            GIV_BIT(MUART[Local_Counter]->LCRH,MUART_Config[Local_Counter].WordLength,WLEN0_BIT);
            /* Configure Parity */
            GIV_BIT(MUART[Local_Counter]->LCRH,MUART_Config[Local_Counter].ParityMode,MUART_0);
            /* Configure Stop Bits */
            GIV_BIT(MUART[Local_Counter]->LCRH,MUART_Config[Local_Counter].StopBits,STP2_BIT);
            /* Enable FIFO */
            SET_BIT(MUART[Local_Counter]->LCRH,FEN_BIT);
            /* GPIO */
            GPIO_voidConnectClock(PINS[Local_Counter].PORT1);
            GPIO_voidConnectClock(PINS[Local_Counter].PORT2);
            GPIO_voidSetBus(PINS[Local_Counter].PORT1,AHB);
            GPIO_voidEnableDigitalFunction(PINS[Local_Counter].PORT1,PINS[Local_Counter].pin1);
            GPIO_voidEnableAlternateFunction(PINS[Local_Counter].PORT1,PINS[Local_Counter].pin1);
            GPIO_voidSetBus(PINS[Local_Counter].PORT2,AHB);
            GPIO_voidEnableDigitalFunction(PINS[Local_Counter].PORT2,PINS[Local_Counter].pin2);
            GPIO_voidEnableAlternateFunction(PINS[Local_Counter].PORT2,PINS[Local_Counter].pin2);
            GPIO_voidPinMUltiplexerSet(PINS[Local_Counter].PORT1,PINS[Local_Counter].pin1,1);
            GPIO_voidPinMUltiplexerSet(PINS[Local_Counter].PORT2,PINS[Local_Counter].pin2,1);
            /* Configure RT State */
            switch (MUART_Config[Local_Counter].RTControl)
            {
                case ENABLE_RECEIVER    :
                    SET_BIT(MUART[Local_Counter]->CTL,RXE_BIT);
                    //CLR_BIT(MUART[Local_Counter]->CTL,TXE_BIT);
                    break;
                case ENABLE_TRANSMITTER:
                    //CLR_BIT(MUART[Local_Counter]->CTL,RXE_BIT);
```

- MUART_voidSendDataSynch(): To send data Synchronous without interrupt.

```
/* Send Data Synchronous */
void MUART_voidSendDataSynch(MUART_CH_Type Copy_UARTChannel, pu8 Copy_pu8Data)
{
    if (Copy_UARTChannel < MAX_UARTS)
    {
        while (*Copy_pu8Data)
        {
            MUART[Copy_UARTChannel]->DR = *Copy_pu8Data;
            while(GET_BIT(MUART[Copy_UARTChannel]->FR,TXFF_BIT));
            Copy_pu8Data++;
        }
    }
    else
    {
        /* Do Nothing */
    }
}
```

- MUART_voidReceiveDataSynchWithDataLength(): To receive data with an expecting length.

```
/* Receive Data Synchronous with Data Length */
void MUART_voidReceiveDataSynch(MUART_CH_Type Copy_UARTChannel, pu8 Copy_pu8Data, u8 Copy_u8DataLength)
{
    if (Copy_UARTChannel < MAX_UARTS)
    {
        while (Copy_u8DataLength--)
        {
            while(GET_BIT(MUART[Copy_UARTChannel]->FR,RXFE_BIT));
            *Copy_pu8Data = MUART[Copy_UARTChannel]->DR;
            Copy_pu8Data++;
        }
    }
    else
    {
        /* Do Nothing */
    }
}
```

- MUART_voidWriteNum16Bit(): To write negative numbers.

```
12 void MUART_voidWriteNum16Bit(MUART_CH_Type Copy_UARTChannel, s16 Copy_u16Num)
13 {
14     u8  Local_u8DigitNum = 0;
15     u16 Local_u16Num     = Copy_u16Num;
16     u8  Local_u8Counter  = 0;
17     u8 y =0;
18     u8 Local_arru8Num[50];
19     if (Copy_u16Num < 0)
20     {
21         Copy_u16Num *= -1;
22         Local_u16Num *= -1;
23         MUART_voidSendDataSynch(Copy_UARTChannel, "-");
24     }
25     if (!Copy_u16Num)
26     {
27         Local_u8DigitNum = 1;
28     }
29     else
30     {
31         while (Local_u16Num)
32         {
33             Local_u16Num /= 10;
34             ++Local_u8DigitNum;
35         }
36     }
37
38     for (Local_u8Counter=Local_u8DigitNum; Local_u8Counter>0; Local_u8Counter--)
39     {
40         Local_u16Num = Copy_u16Num%10;
41         Local_arru8Num[Local_u8Counter-1] = Local_u16Num;
42         Copy_u16Num = Copy_u16Num/10;
43     }
44
45     for (Local_u8Counter=0; Local_u8Counter<Local_u8DigitNum; Local_u8Counter++)
46     {
47
48         y = Local_arru8Num[Local_u8Counter]+'0';
49         MUART_voidSendDataSynch(Copy_UARTChannel, &(y));
50     }
51 }
```

3.1.2.6. I2C

We needed a driver for the I2C module in Tiva C for the IMU and the Compass. A quick glance on the I2C module block diagram of TM4C123GH6PM, we noticed that we need only to access 5 registers only for tiva C master communication with our two sensors (slaves).

- Driver Code:

```
12  struct I2C_PORT
13  {
14      gpio_port_t PORT;
15      char pin1;
16      char pin2;
17  };
18
19  struct I2C_PORT I2C_PINS[4] = {{PORTB,2,3},{PORTA,6,7},{PORTE,4,5},{PORTD,0,1}};
20
```

For easier control of all I2C Ports in tiva C, we defined an array of structures and filled it with GPIO port and pins of every I2C port in line 19.

But first, we have to define this structure in line 12 with name I2C_PORT and having 3 fields, the GPIO port, pin 1, and pin 2.

```
21  void I2C_voidConnectClock(I2C_t I2C)
22  {
23      volatile long int* reg = 0x400FE620;
24      *reg |= (1 << I2C);
25  }
```

Like every module in TM4C123GH6PM architecture, we have to connect clock before addressing it. This can be done from RCGI2C Register of address 0x400FE620 in System Control Registers.

This function takes I2C port number as a parameter. In line 23, we access RCGI2C Register, and in line 24 we set the bit which corresponds to the number of the chosen I2C port.

```
26  void I2C_voidSetMode(I2C_t I2C, mode_t mode)
27  {
28      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMCR;
29      *reg |= (1 << mode);
30
31  }
```

This function takes 2 arguments, the first is the I2C port number as in the previous function, while the second is the mode that the user want to put the required I2C port in.

In line 28, we access I2CMCR Register. In line 29, we set the bit that corresponds to the mode we want.

```
33  void I2C_voidSetClockFrequency(I2C_t I2C, long int freq)
34  {
35      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMTPR;
36      *reg |= (clockFrequency/20*freq) - 1;
37
38  }
```

This function takes 2 arguments, the first is the I2C Port number, and the second is the required frequency. In line 35, we accessed I2CMTPR Register. We set it according to equation given in datasheet in line 36.

```
39  /*static*/ int I2C_intWaitTillDone(I2C_t I2C)
40  {
41      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMCS;
42      while((*reg) & 1);   /* wait until I2C master is not busy */
43      return (*reg) & 0xE; /* return I2C error code, 0 if no error*/
44  }
```

This function just takes 1 argument, which must be the I2C Port number. It returns the status of the communication process, 0 means no error, 1 means in error. However, we must first access I2CMCS Register in line 41, and polling on its flag (bit 1) to make sure I2C module is not busy doing another thing.

```
120 void I2C_voidInit(I2C_t I2C, mode_t mode, long int clock)
121 {
122     I2C_voidConnectClock(I2C);
123     GPIO_voidConnectClock(I2C_PINS[I2C].PORT);
124     GPIO_voidSetBus(I2C_PINS[I2C].PORT, AHB);
125     GPIO_voidEnableDigitalFunction(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin1);
126     GPIO_voidEnableAlternateFunction(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin1);
127     GPIO_voidEnableDigitalFunction(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin2);
128     GPIO_voidEnableAlternateFunction(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin2);
129     GPIO_voidPinMUltiplexerSet(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin1,3);
130     GPIO_voidPinMUltiplexerSet(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin2,3);
131     GPIO_voidSetOpenDrain(I2C_PINS[I2C].PORT,I2C_PINS[I2C].pin2);
132     I2C_voidSetMode(I2C, mode);
133     I2C_voidSetClockFrequency(I2C, clock);
134
135
136 }
```

This function combines all previous I2C activation steps, and for this it needs 3 arguments; the I2C Port number, the required mode, and the clock frequency.

In line 122 we connected the clock to the I2C module. From line 123 to line 131, we use GPIO driver to enable the pins of the chosen I2C module. In line 132, we set the required mode. Finally in line 133, we set the clock frequency.

```
137  void I2C_voidCallSlave(I2C_t I2C, long int slave_address)
138  {
139      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMSA;
140      *reg = slave_address;
141  }
```

This function calls the required slave from its address. Therefore, it must take 2 parameters, the I2C Port number, and the slave address.

In line 139, we accessed I2CMSA. In line 140, we load the register with the required slave address.

```
143  void I2C_voidSetDataRegister(I2C_t I2C, char data)
144  {
145      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMDR;
146      *reg = data;
147
148  }
```

This function loads the I2C data register (I2CMDR) by the required data to be sent. It takes 2 parameters, the I2C port number, and the required data.

```
150  void I2C_voidSetCommunication(I2C_t I2C, char commMode)
151  {
152      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMCS;
153      *reg = commMode;
154  }
```

This function chooses communication program with the slave according to the explanation in the datasheet. Its control is just like the previous function just with different registers.

```
155   char I2C_charReadFlagRegister(I2C_t I2C)
156   {
157       volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMCS;
158       return *reg;
159   }
```

This function reads the I2C module flag of I2CMCS and returns it. In line 157, we accessed this register. In line 158, we return it.

```
161  char I2C_charReadDataRegister(I2C_t I2C)
162  {
163      volatile long int* reg = I2C_BASE + (I2C << 12) + I2CMDR;
164      return *reg;
165  }
```

This function reads and returns the value stored in I2CMDR register. Its control is the same as the previous function.

```
46  char I2C_charWrite(I2C_t I2C, int slave_address, char slave_memory_address, char data)
47  {
48
49      char error;
50
51      /* send slave address and starting address */
52      I2C_voidCallSlave(I2C, (slave_address << 1));
53      I2C_voidSetDataRegister(I2C,slave_memory_address);
54      I2C_voidSetCommunication(I2C, 3);                       /* S-(saddr+w)-ACK-maddr-ACK */
55
56      error = I2C_intWaitTillDone(I2C);   /* wait until write is complete */
57      if (error) return error;
58
59      /* send data */
60      I2C_voidSetDataRegister(I2C, data);
61      I2C_voidSetCommunication(I2C, 5);                          /* -data-ACK-P */
62      error = I2C_intWaitTillDone(I2C);       /* wait until write is complete */
63      while((I2C_charReadFlagRegister(I2C)) & 0x40);             /* wait until bus is not busy */
64      error = (I2C_charReadFlagRegister(I2C)) & 0xE;
65      if (error) return error;
66
67      return 0;       /* no error */
68  }
```

This function takes 4 arguments, the first is the I2C Port number, slave address, and the data wanted to be sent. We called the required slave by its address in line 52. We loaded data register with the slave address. We called the required slave first, then loaded data register I2CMDR with the slave memory address. Then, we set communication mode to 3 which is “S-(saddr+w)-ACK-maddr-ACK”. Then, checks error with the previously mentioned functions. We loaded data register with our message, then we chose mode 5 which corresponds to “-data-ACK-P”

```
char I2C_charRead(I2C_t I2C, int slaveAddr, char slave_memory_address, int byteCount, char* data)
{
    char error;

   if (byteCount <= 0)
       return -1;         /* no read was performed */

   /* send slave address and starting address */
   I2C_voidCallSlave(I2C, (slaveAddr << 1));
   I2C_voidSetDataRegister(I2C,slave_memory_address);
   I2C_voidSetCommunication(I2C, 3);       /* S-(saddr+w)-ACK-maddr-ACK */
   error = I2C_intWaitTillDone(I2C);
   if (error)
       return error;

   /* to change bus from write to read, send restart with slave addr */
   I2C_voidCallSlave(I2C, ((slaveAddr << 1) + 1));   /* restart: -R-(saddr+r)-ACK */

   if (byteCount == 1)             /* if last byte, don't ack */
       I2C_voidSetCommunication(I2C, 7);              /* -data-NACK-P */
   else                            /* else ack */
       I2C_voidSetCommunication(I2C, 0xB);            /* -data-ACK- */
   error = I2C_intWaitTillDone(I2C);
   if (error) return error;

   *data++ = I2C_charReadDataRegister(I2C);            /* store the data received */


   if (--byteCount == 0)           /* if single byte read, done */
   {
       while((I2C_charReadFlagRegister(I2C)) & 0x40);    /* wait until bus is not busy */
       return 0;       /* no error */
   }

   /* read the rest of the bytes */
   while (byteCount > 1)
   {
       I2C_voidSetCommunication(I2C, 9);          /* -data-ACK- */
       error = I2C_intWaitTillDone(I2C);
       if (error) return error;
       byteCount--;
       *data++ = I2C_charReadDataRegister(I2C);       /* store data received */
   }

   I2C_voidSetCommunication(I2C, 5);               /* -data-NACK-P */
   error = I2C_intWaitTillDone(I2C);
   *data = I2C_charReadDataRegister(I2C);          /* store data received */
   while((I2C_charReadFlagRegister(I2C)) & 0x40);       /* wait until bus is not busy */

   return 0;       /* no error */
}
```

This function reads from slave, and its arguments must be changed as we must add the slave memory address, the byte counter, and the buffer instead of data also. In line 73, we checked if user entered a valid byte count. From lines 77 to 84, we repeated steps of transmission to set communication between the 2 devices. In line 85, we used this logic to leave bit 0 in I2CMSA Register, because it is not part of the slave address. In line 87, we checked if byte count equals 1, then we send NACK and end communication this is communication mode 7 “-data-NACK-P”. If the byte count is greater than one, then we need to send ACK after data. This is communication mode 0xB “-data-ACK- “. If so, then we checked error in every iteration as usual, and filled buffer byte by byte in line 94. We waited until data received in line 98 by polling on busy flag and exit function if 1 byte receive only. In line 103, we continue rest of data for byte count larger than 1, but this time with communication mode equals 9 “-data-ACK- “. Finally, the rest of the function is the ending of communication process, and it is exactly like master transmit operation.

3.2. Atmega32 (AVR)

The AVR is a modified Harvard Architecture 8-bit RISC single-chip microcontroller, which was developed by Atmel in 1996. The AVR was one of the first microcontroller families to use on-chip Flash memory for program storage, as opposed to one-time programmable Memory, EPROM, or EEPROM used by other microcontrollers at the time.

Figure 3.4: ATmega32 Controller kit.

3.2.1. Features

- Multifunction, bi-directional general-purpose I/O ports with configurable, built-in pull-up resistor.
- Multiple internal oscillators, including RC oscillator without external parts.
- Internal, self-programmable instruction flash memory up to 256 KB (384 KB on Xmega) In-system programmable using serial/parallel low-voltage proprietary interfaces or JTAG Optional boot code section with independent lock bits for protection.
- On-chip debugging (OCD) support through JTAG or Debug Wire on most devices The JTAG signals (TMS, TDI, TDO, and TCK) are multiplexed on GPIOs. These pins can be configured to function as JTAG or GPIO depending on the setting of a fuse bit, which can be programmed via ISP or HVSP. By default, AVRs with JTAG come with the JTAG interface enabled. Debug WIRE uses the /RESET pin as a bi-directional communication channel to access on-chip debug circuitry. It is present on devices with lower pin counts, as it only requires one pin.

- Internal data EEPROM up to 4 Kbyte.
- Internal SRAM up to 16 KB (32 KB on Xmega).
- External 64 KB little endian data space on certain models, including the Mega8515 and Mega162.The external data space is overlaid with the internal data space, such that the full 64 KB address space does not appear on the external bus and accesses to e.g. address 010016 will access internal RAM, The Xmega series, the external data space has been enhanced to support both SRAM and SDRAM. As well, the data addressing modes have been expanded to allow up to 16 MB of data memory to be directly addressed. AVRs generally do not support executing code from external memory. Some ASSPs using the AVR core do support external program memory.
- 8-bit and 16-bit timers PWM output (some devices have an enhanced PWM peripheral which includes a dead-time generator) Input Capture that record a time stamp triggered by a signal edge.
- Analog comparator.
- 10 or 12-bit A/D Converters, with multiplex of up to 16 channels.
- 12-bit D/A Converters.
- A variety of serial interfaces, including I2C compatible Two-Wire Interface (TWI)Synchronous/asynchronous serial peripherals (UART/USART) (used with RS-232, RS-485, and more) Serial Peripheral Interface (SPI)Universal Serial Interface (USI): a multi-purpose hardware communication module that can be used to implement an SPI, I2C or UART interface.
- Brownout detection.
- Watchdog Timer (WDT).
- Multiple power-saving sleep modes.
- Lighting and motor control (PWM-specific) controller models.
- CAN controller support.
- USB controller support Proper full-speed (12 Mbit/s) hardware & Hub controller with embedded AVR. Also, freely available low-speed (1.5 Mbit/s) (HID) bit banging software emulations.
- Ethernet controller support.
- LCD controller support.
- Low-voltage devices operating down to 1.8 V (to 0.7 V for parts with built-in DC–DC up converter).
- Pico Power devices.

- DMA controllers and “event system” peripheral communication.
- Fast cryptography support for AES and DES

AVR has more than one family the ones used in this project is Atmega32.

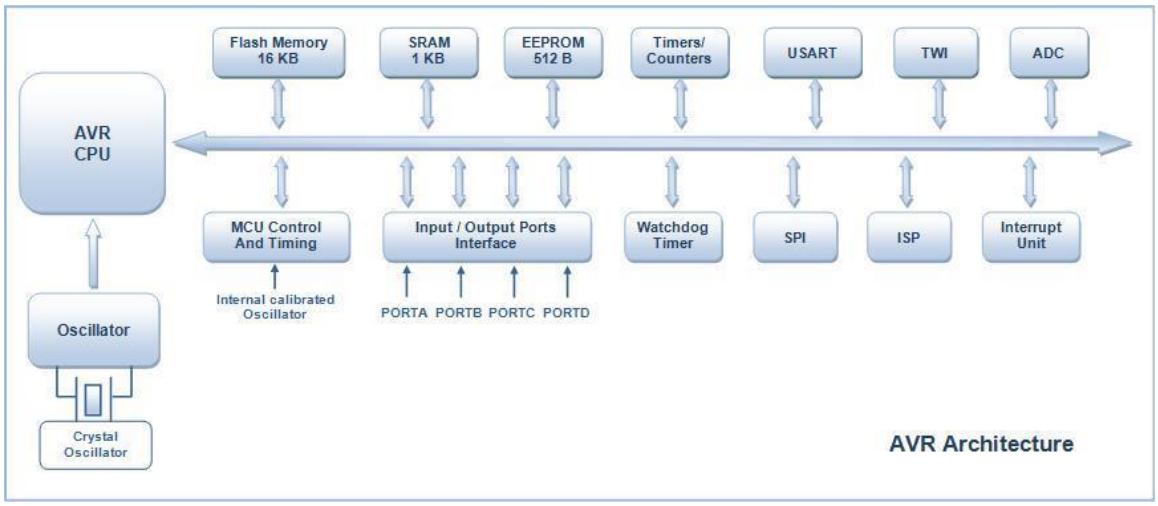


Figure 3.5: The architecture of AVR.

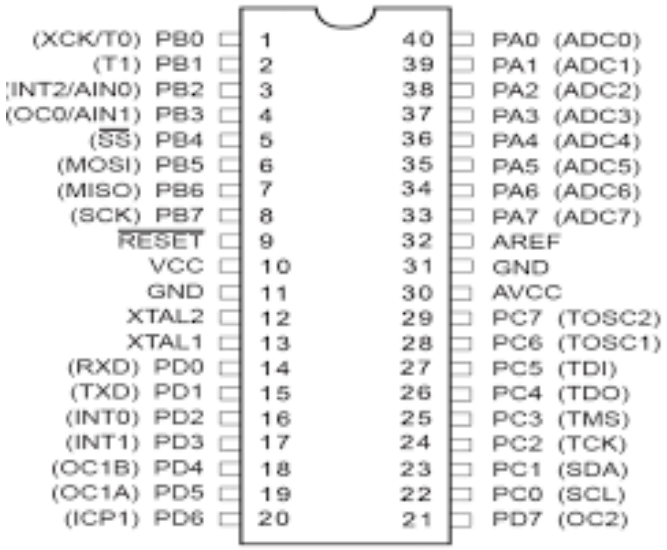


Figure 3.6: Atmega32 Pinout.

3.2.2. Drivers

Software used:

- Eclipse
- Putty
- Energia

Layered Architecture:

APP	**Main.c**
HAL	**7-Segments**
MCAL	**DIO – UART**
LIB	**STD_TYPES – BIT_MATH**

Table 3.3: AVR Drivers.

Every Driver of HAL/MCAL Modules Has 4 files (program.c – interface.h – config.h – private.h)

BIT_MATH: To control the bits of AVR registers.

BIT_MATH.h

```
/*                                                    Bit Math Library
#ifndef BIT_MATH_H
#define BIT_MATH_H

/* Bit Math for 1 bit */
#define SET_BIT(VAR,BITNO)                  (VAR) |=  (   1   << (BITNO))
#define CLR_BIT(VAR,BITNO)                  (VAR) &= ~(   1   << (BITNO))
#define TOG_BIT(VAR,BITNO)                  (VAR) ^=  (   1   << (BITNO))
#define GIV_BIT(VAR,VAL,BITNO)              (VAR) |=  ( (VAL) << (BITNO))
#define EQU_BIT(VAR,VAL)                    (VAR) |=  VAL
#define GET_BIT(VAR,BITNO)                  (((VAR) >> (BITNO)) & 0x01)

/* Bit Math for 3 bit */
#define CLR_3BIT(VAR,BITNO)                 (VAR) &= ~((0b111) << (BITNO))

/* Bit Math for 5 bit */
#define CLR_5BIT(VAR,BITNO)                 (VAR) &= ~((0b11111) << (BITNO))

/* Bit Math for 8 bit */
#define TOG_8BIT(VAR)                       (VAR) ^= 0xFF

/* Concatenate */
/* Note: It can Conc more than 8-bit but it takes only 8 Numbers, Parameters can more than 1 digi
#define CONC(X7,X6,X5,X4,X3,X2,X1,X0)           CONC_HELPER(X7,X6,X5,X4,X3,X2,X1,X0)
#define CONC_HELPER(X7,X6,X5,X4,X3,X2,X1,X0)    0b##X7##X6##X5##X4##X3##X2##X1##X0

#endif
/*****************************************************************************************************
```

STD_TYPES: to define new standard types, it could help if change machine.

STD_TYPES.h

```
/*                                                          Standard Types Library
#ifndef STD_TYPES_H
#define STD_TYPES_H

typedef unsigned char      u8;
typedef unsigned short int u16;
typedef unsigned long int  u32;
typedef signed   char      s8;
typedef signed   short int s16;
typedef signed   int       s32;
typedef float              f32;
typedef double             f64;
typedef long     double    f128;

typedef u8*                pu8;
typedef u16*               pu16;
typedef u32*               pu32;
typedef s8*                ps8;
typedef s16*               ps16;
typedef s32*               ps32;
typedef f32*               pf32;
typedef f64*               pf64;
typedef f128*              pf128;

#endif
/*****************************************************************************************************
```

3.2.2.1. DIO

DIO_config.h: To config all pins Direction, Value and Select Floating or Pullup if the pin is input.

DIO_config.h

```
/*                                                                  DIO Driver
#ifndef DIO_CONFIG_H
#define DIO_CONFIG_H

/*****************************************************************************************************
 /*                                                                Configuration
 /* DIO Registers : Page 62
 /*****************************************************************************************************
/* MDIO_PORTx_PINx_DIR Opitions:                INPUT
                                                OUTPUT

   MDIO_PORTx_PINx_VALUE Options for Output:    LOW
                                                HIGH

   MDIO_PORTx_PINx_MODE Opitions for Input:     FLOATING
                                                PULLUP      */

/* PORTA - PIN0 */
#define MDIO_PORTA_PIN0_DIR     INPUT
#define MDIO_PORTA_PIN0_VALUE   HIGH
#define MDIO_PORTA_PIN0_MODE    PULLUP

/* PORTA - PIN1 */
#define MDIO_PORTA_PIN1_DIR     INPUT
#define MDIO_PORTA_PIN1_VALUE   HIGH
#define MDIO_PORTA_PIN1_MODE    PULLUP

/* PORTA - PIN2 */
#define MDIO_PORTA_PIN2_DIR     INPUT
#define MDIO_PORTA_PIN2_VALUE   HIGH
#define MDIO_PORTA_PIN2_MODE    PULLUP
```

DIO_interface.h: Enums for Ports,Pins,Pin's value and Floating or Pullup if the pin is input

DIO_interface.h

```
/* Ports */
typedef enum
{
    PORTA,              // 0
    PORTB,              // 1
    PORTC,              // 2
    PORTD               // 3
}MDIO_PORT_Type;

/* Pins */
typedef enum
{
    PIN0,               // 0
    PIN1,               // 1
    PIN2,               // 2
    PIN3,               // 3
    PIN4,               // 4
    PIN5,               // 5
    PIN6,               // 6
    PIN7                // 7
}MDIO_PIN_Type;

/* Objects Like Macro  */
/* Port Directions Options */
#define INPUT_ALL       0x00
#define OUTPUT_ALL      0xFF

/* Pin Directions Options */
#define INPUT           0
#define OUTPUT          1

/* Port Mode Options for Input */
#define FLOATING_ALL    0x00
#define PULLUP_ALL      0xFF

/* Pin Mode Options for Input */
#define FLOATING        0
#define PULLUP          1

/* Port Value Options for Output */
#define LOW_ALL         0x00
#define HIGH_ALL        0xFF

/* Pin Value Options for Output */
#define LOW             0
#define HIGH            1
/*************************************
```

DIO_private.h: Registers of DIO.

DIO_private.h

```
/*                                                                    DIO Driver
#ifndef DIO_PRIVATE_H
#define DIO_PRIVATE_H

/*********************************************************************************************************
/*                                                                Registers Definitions
/* Register Summary : Page 299
/*********************************************************************************************************
/* DIO PORT A Registers */
#define PORTA_REGISTER          *((volatile pu8)0x3B)   // Port A Data Register â€“ PORTA
#define DDRA_REGISTER           *((volatile pu8)0x3A)   // Port A Data Direction Register â€“ DDRA
#define PINA_REGISTER           *((volatile pu8)0x39)   // Port A Input Pins Address â€“ PINA

/* DIO PORT B Registers */
#define PORTB_REGISTER          *((volatile pu8)0x38)   // Port B Data Register â€“ PORTB
#define DDRB_REGISTER           *((volatile pu8)0x37)   // Port B Data Direction Register â€“ DDRB
#define PINB_REGISTER           *((volatile pu8)0x36)   // Port B Input Pins Address â€“ PINB

/* DIO PORT C Registers */
#define PORTC_REGISTER          *((volatile pu8)0x35)   // Port C Data Register â€“ PORTC
#define DDRC_REGISTER           *((volatile pu8)0x34)   // Port C Data Direction Register â€“ DDRC
#define PINC_REGISTER           *((volatile pu8)0x33)   // Port C Input Pins Address â€“ PINC

/* DIO PORT D Registers */
#define PORTD_REGISTER          *((volatile pu8)0x32)   // Port D Data Register â€“ PORTD
#define DDRD_REGISTER           *((volatile pu8)0x31)   // Port D Data Direction Register â€“ DDRD
#define PIND_REGISTER           *((volatile pu8)0x30)   // Port D Input Pins Address â€“ PIND

```

DIO_program.c: Some of DIO functions

- MDIO_voidInit(): Initialization function to Set Config Direction and Value of pins

DIO_program.c

```
#include "DIO_private.h"
#include "DIO_config.h"
/*************************************************************

/*************************************************************
/*
/*************************************************************
/* Initialization */
void MDIO_voidInit(void)
{
    /* PORTA Direction */
    DDRA_REGISTER  = CONC(MDIO_PORTA_PIN7_DIR,
                          MDIO_PORTA_PIN6_DIR,
                          MDIO_PORTA_PIN5_DIR,
                          MDIO_PORTA_PIN4_DIR,
                          MDIO_PORTA_PIN3_DIR,
                          MDIO_PORTA_PIN2_DIR,
                          MDIO_PORTA_PIN1_DIR,
                          MDIO_PORTA_PIN0_DIR);
    /* PORTB Direction */
    DDRB_REGISTER  = CONC(MDIO_PORTB_PIN7_DIR,
                          MDIO_PORTB_PIN6_DIR,
                          MDIO_PORTB_PIN5_DIR,
                          MDIO_PORTB_PIN4_DIR,
                          MDIO_PORTB_PIN3_DIR,
                          MDIO_PORTB_PIN2_DIR,
                          MDIO_PORTB_PIN1_DIR,
                          MDIO_PORTB_PIN0_DIR);
    /* PORTC Direction */
```

- MDIO_voidSetPinDierction(): Set pin direction function: Select port, pin and direction “INPUT” or “OUTPUT”.

```
/* IO Pins */
void MDIO_voidSetPinDirection(MDIO_PORT_Type Copy_Port, MDIO_PIN_Type Copy_Pin, u8 Copy_u8Direction)
{
    if (Copy_Pin <= PIN7)
    {
        switch (Copy_u8Direction)
        {
            case INPUT :
                switch (Copy_Port)
                {
                    case PORTA: CLR_BIT(DDRA_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTB: CLR_BIT(DDRB_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTC: CLR_BIT(DDRC_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTD: CLR_BIT(DDRD_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    default   : /* return MDIO_WRONG_PORT; */                               break;
                }
                break;
            case OUTPUT:
                switch (Copy_Port)
                {
                    case PORTA: SET_BIT(DDRA_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTB: SET_BIT(DDRB_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTC: SET_BIT(DDRC_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    case PORTD: SET_BIT(DDRD_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
                    default   : /* return MDIO_WRONG_PORT; */                               break;
                }
                break;
            default    : /* return MDIO_WRONG_DIRECTION; */                                 break;
        }
    }
    else
    {
        /* return MDIO_WRONG_PIN; */
    }
}
```

- MDIO_voidSetPinValue(): Select port, pin and value “HIGH” or “LOW”.

```
void MDIO_voidSetPinValue(MDIO_PORT_Type Copy_Port, MDIO_PIN_Type Copy_Pin, u8 Copy_u8Value)
{
    if (Copy_Pin <= PIN7)
    {
        switch (Copy_u8Value)
        {
            case LOW:
                switch (Copy_Port)
                {
                    case PORTA: CLR_BIT(PORTA_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTB: CLR_BIT(PORTB_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTC: CLR_BIT(PORTC_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTD: CLR_BIT(PORTD_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    default   : /* return MDIO_WRONG_PORT; */                               break;
                }
                break;
            case HIGH:
                switch (Copy_Port)
                {
                    case PORTA: SET_BIT(PORTA_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTB: SET_BIT(PORTB_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTC: SET_BIT(PORTC_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    case PORTD: SET_BIT(PORTD_REGISTER,Copy_Pin);   /* return MDIO_DONE; */ break;
                    default   : /* return MDIO_WRONG_PORT; */                               break;
                }
                break;
            default  : /* return MDIO_WRONG_VALUE; */                                      break;
        }
    }
    else
    {
        /* return MDIO_WRONG_PIN; */
    }
}
```

- MDIO_u8GetPinValue(); To read the value of pin if the pin is input

```
u8   MDIO_u8GetPinValue(MDIO_PORT_Type Copy_Port, MDIO_PIN_Type Copy_Pin)
{
    u8 Local_u8Value = MDIO_INITIAL_VALUE;

    if (Copy_Pin <= PIN7)
    {
        switch (Copy_Port)
        {
            case PORTA: Local_u8Value = GET_BIT(PINA_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
            case PORTB: Local_u8Value = GET_BIT(PINB_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
            case PORTC: Local_u8Value = GET_BIT(PINC_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
            case PORTD: Local_u8Value = GET_BIT(PIND_REGISTER,Copy_Pin); /* return MDIO_DONE; */    break;
            default   : /* return MDIO_WRONG_PORT; */                                               break;
        }
    }
    else
    {
        /* return MDIO_WRONG_PIN; */
    }

    return Local_u8Value;
}
```

- Same function but for all pins of the port instead of single pin:

```
/* IO Ports */
void MDIO_voidSetPortDirection(MDIO_PORT_Type Copy_Port, u8 Copy_u8Direction)
{
    switch (Copy_Port)
    {
        case PORTA: DDRA_REGISTER = Copy_u8Direction; /* return MDIO_DONE; */   break;
        case PORTB: DDRB_REGISTER = Copy_u8Direction; /* return MDIO_DONE; */   break;
        case PORTC: DDRC_REGISTER = Copy_u8Direction; /* return MDIO_DONE; */   break;
        case PORTD: DDRD_REGISTER = Copy_u8Direction; /* return MDIO_DONE; */   break;
        default   : /* return MDIO_WRONG_PORT; */                               break;
    }
}

void MDIO_voidSetPortValue(MDIO_PORT_Type Copy_Port, u8 Copy_u8Value)
{
    switch (Copy_Port)
    {
        case PORTA: PORTA_REGISTER = Copy_u8Value; /* return MDIO_DONE; */  break;
        case PORTB: PORTB_REGISTER = Copy_u8Value; /* return MDIO_DONE; */  break;
        case PORTC: PORTC_REGISTER = Copy_u8Value; /* return MDIO_DONE; */  break;
        case PORTD: PORTD_REGISTER = Copy_u8Value; /* return MDIO_DONE; */  break;
        default   : /* return MDIO_WRONG_PORT; */                           break;
    }
}

u8   MDIO_u8GetPortValue(MDIO_PORT_Type Copy_Port)
{
    u8 Local_u8Value = MDIO_INITIAL_VALUE;

    switch (Copy_Port)
    {
        case PORTA: Local_u8Value = PINA_REGISTER; /* return MDIO_DONE; */  break;
        case PORTB: Local_u8Value = PINB_REGISTER; /* return MDIO_DONE; */  break;
        case PORTC: Local_u8Value = PINC_REGISTER; /* return MDIO_DONE; */  break;
        case PORTD: Local_u8Value = PIND_REGISTER; /* return MDIO_DONE; */  break;
        default   : /* return MDIO_WRONG_PORT; */                           break;
    }

    return Local_u8Value;
}
```

3.2.2.2. UART

UART_config.h: To config UART State enable or disable, data word length, parity, receive or transmission, stop bits 1 or 2, UART mode and Select baud rate.

UART_config.h

```
/*                                                                    Configuration
/* DIO Registers : Page 157
/*****************************************************************************
/* MUART_STATE: MUART_DISABLE
                MUART_ENABLE  */
#define MUART_STATE         MUART_ENABLE

/* MUART_WORD_LENGTH:   MUART_DATA_5BIT
                        MUART_DATA_6BIT
                        MUART_DATA_7BIT
                        MUART_DATA_8BIT
                        MUART_DATA_9BIT */
#define MUART_WORD_LENGTH   MUART_DATA_8BIT

/* MUART_PARITY:    MUART_NO_PARITY
                    MUART_EVEN_PARITY
                    MUART_ODD_PARITY  */
#define MUART_PARITY        MUART_NO_PARITY

/* MUART_RT_CONTROL:    MUART_ENABLE_RECEIVER
                        MUART_ENABLE_TRANSMITTER
                        MUART_ENABLE_BOTH        */
#define MUART_RT_CONTROL    MUART_ENABLE_BOTH

/* MUART_STOP_BITS: MUART_STOP_1BIT
                    MUART_STOP_2BITS */
#define MUART_STOP_BITS     MUART_STOP_1BIT

/* MUART_MODE:  MUART_ASYNCHRONOUS
                MUART_SYNCHRONOUS  */
#define MUART_MODE          MUART_ASYNCHRONOUS

/* Configure Baud Rate: Page 163 */
#define MUART_UBRR          51      // fosc = 8 MHz, BaudRate = 9600

/* MUART_CLK_POLARITY:  MUART_RISING_TX
                        MUART_RISING_RX */
#define MUART_CLK_POLARITY  MUART_RISING_TX
/*****************************************************************************
```

UART_private.h: Registers of UART.

UART_private.h

```
/*                                                                              UART Driver
#ifndef UART_PRIVATE_H
#define UART_PRIVATE_H

/*****************************************************************************************************************
/*                                                                          Registers Definitions
/* Register Summary : Page 299
/*****************************************************************************************************************
#define UDR_REGISTER        *((pu8)0x2C)    // USART I/O Data Register
#define UCSRA_REGISTER      *((pu8)0x2B)    // USART Control and Status Register A
#define UCSRB_REGISTER      *((pu8)0x2A)    // USART Control and Status Register B
#define UCSRC_REGISTER      *((pu8)0x40)    // USART Control and Status Register C
#define UBRRL_REGISTER      *((pu8)0x29)    // USART Baud Rate Registers
#define UBRRH_REGISTER      *((pu8)0x40)    // USART Baud Rate Registers
/*****************************************************************************************************************
```

UART_program.c: Some of UART functions

- UARTR_voidInit(): Initialization function to Set the defines of the Config file

UART_program.c

```
/* Initialization */
void MUART_voidInit(void)
{
    if      (MUART_STATE == MUART_ENABLE)
    {
        /* Clear Control Registers */
        UCSRA_REGISTER = MUART_0;
        UCSRB_REGISTER = MUART_0;
        UCSRC_REGISTER = MUART_0X80;
        /* Configure Parity */
        #if   ((MUART_PARITY != MUART_NO_PARITY) && (MUART_PARITY != MUART_EVEN_PARITY) && (MUART_PARITY != MUART_ODD_PARITY))
            #error("Configuration Error :: Wrong MUART_PARITY")
        #endif
        /* Configure Stop BITs */
        #if   ((MUART_STOP_BITS != MUART_STOP_1BIT) && (MUART_STOP_BITS != MUART_STOP_2BITS))
            #error("Configuration Error :: Wrong MUART_STOP_BITS")
        #endif
        /* Configure UART Mode */
        #if   ((MUART_MODE != MUART_ASYNCHRONOUS) && (MUART_MODE != MUART_SYNCHRONOUS))
            #error("Configuration Error :: Wrong MUART_MODE")
        #endif
        /* Configure Clock Polarity */
        #if   ((MUART_CLK_POLARITY != MUART_RISING_TX) && (MUART_CLK_POLARITY != MUART_RISING_RX))
            #error("Configuration Error :: Wrong MUART_CLK_POLARITY")
        #endif
        /* Configure Word Length */
        #if   ((MUART_WORD_LENGTH ==  MUART_DATA_5BIT) || (MUART_WORD_LENGTH ==  MUART_DATA_6BIT) || \
               (MUART_WORD_LENGTH ==  MUART_DATA_7BIT) || (MUART_WORD_LENGTH ==  MUART_DATA_8BIT))
            //CLR_BIT(UCSRB_REGISTER,UCSZ2_BIT);
            EQU_BIT(UCSRC_REGISTER,MUART_WORD_LENGTH | MUART_PARITY | MUART_STOP_BITS | MUART_MODE | MUART_CLK_POLARITY);
        #elif (MUART_WORD_LENGTH == MUART_DATA_9BIT)
            SET_BIT(UCSRB_REGISTER,UCSZ2_BIT);
            EQU_BIT(UCSRC_REGISTER,(MUART_WORD_LENGTH-MUART_1));
        #else
            #error("Configuration Error :: Wrong MUART_WORD_LENGTH")
        #endif
        /* Configure RT State */
        #if   (MUART_RT_CONTROL == MUART_ENABLE_RECEIVER)
            SET_BIT(UCSRB_REGISTER,RXEN_BIT);
            //CLR_BIT(UCSRB_REGISTER,TXEN_BIT);
        #elif (MUART_RT_CONTROL == MUART_ENABLE_TRANSMITTER)
            //CLR_BIT(UCSRB_REGISTER,RXEN_BIT);
            SET_BIT(UCSRB_REGISTER,TXEN_BIT);
```

```
73          #elif (MUART_RT_CONTROL == MUART_ENABLE_BOTH)
74              SET_BIT(UCSRB_REGISTER,RXEN_BIT);
75              SET_BIT(UCSRB_REGISTER,TXEN_BIT);
76          #else
77              #error("Configuration Error :: Wrong MUART_RT_CONTROL")
78          #endif
79          /* Configure Baud Rate */
80          if (MUART_UBRR < MUART_UBRRL_Full)
81          {
82              UBRRL_REGISTER = MUART_UBRR;
83          }
84          else
85          {
86  //          CLR_BIT(UBRRL_REGISTER,URSEL_BIT);
87  //          MUART_UBRR = ((UBRRH_REGISTER << MUART_8) | UBRRL_REGISTER);
88          }
89          /* Clear Data Register */
90          MUART_voidClearDataRegister();
91          /* Configure DIO */
92          MDIO_voidSetPinDirection(MUART_PORT,MUART_RXD,INPUT );
93          MDIO_voidSetPinDirection(MUART_PORT,MUART_TXD,OUTPUT);
94      }
95      else if (MUART_STATE == MUART_DISABLE)
96      {
97          /* Clear Control Registers */
98          UCSRA_REGISTER = MUART_0;
99          UCSRB_REGISTER = MUART_0;
100         UCSRC_REGISTER = MUART_0X80;
101     }
102     else
103     {
104         /* Do Nothing */
105     }
106     /* return MUART_DONE; */
107 }
```

- MUART_voidSendDataSynch(): To send data Synchronous without interrupt.

```
211 /* Send Data Synchronous */
212 void MUART_voidSendDataSynch(pu8 Copy_pu8Data)
213 {
214     while (*Copy_pu8Data)
215     {
216         while(GET_BIT(UCSRA_REGISTER,UDRE_BIT) == MUART_0);
217         UDR_REGISTER = *Copy_pu8Data;
218         Copy_pu8Data++;
219     }
220     /* return MUART_DONE; */
221 }
222
```

- MUART_voidReceiveDataSynch(): To receive data with waiting until TimeOut if there's no data send 255 which a flag for no data.

```
/* Receive Data Synchronous */
void MUART_voidReceiveDataSynch(pu8 Copy_pu8Data, u32 Copy_u32TimeOut)
{
    u32 Local_u32TimeOutCounter = MUART_0;
    *Copy_pu8Data               = MUART_0;

    while (GET_BIT(UCSRA_REGISTER,RXC_BIT) == MUART_0)
    {
        Local_u32TimeOutCounter++;
        if (Local_u32TimeOutCounter == Copy_u32TimeOut)
        {
            *Copy_pu8Data = MUART_255;
            break;
        }
    }
    if (*Copy_pu8Data == MUART_0)
    {
        *Copy_pu8Data = UDR_REGISTER;
    }
    else
    {
        /* Do Nothing */
    }
    /* return MUART_DONE; */
}
```

- MUART_voidReceiveDataSynchWithDataLength(): To receive data with an expecting length.

```
/* Receive Data Synchronous with Data Length */
void MUART_voidReceiveDataSynchWithDataLength(pu8 Copy_pu8Data, u8 Copy_u8DataLength)
{
    while (Copy_u8DataLength--)
    {
        while(GET_BIT(UCSRA_REGISTER,RXC_BIT) == MUART_0);
        *Copy_pu8Data = UDR_REGISTER;
        Copy_pu8Data++;
    }
    /* return MUART_DONE; */
}
```

3.3. Raspberry Pi

Figure 3.7: Raspberry Pi 3 Kit.

The Raspberry Pi 3 will cost the same as its predecessor but feature much more powerful hardware. Bluetooth will be built into the board for the first time and is powered by a Quad Core Broadcom BCM2837 64bit ARMv8 processor.

The Pi 3 runs at 1.2 GHz, compared to the Pi 2's 900MHz, and also has an upgraded power system, and the same four USB ports and extendable 'naked board' design as the Pi 2. "Four years ago today, we launched the first Raspberry Pi with our friends at Premier Farnel," Pi founder Eben Upton said.

So successful has the board been, it is now almost certainly the best-selling British-made and designed computer in history, Upton told the BBC. "Today we're launching Raspberry Pi 3: it's still $35 and it's still the size of your credit card, but now it comes with on-board wireless LAN and Bluetooth, 50 percent more processing power, and a Quad Core 64bit processor. The new Raspberry Pi opens up even more possibilities for IoT and embedded projects; we hope you like it as much as we do."

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.

What's more, the Raspberry Pi has the ability to interact with the outside world and has been used in a wide array of digital maker projects, from music machines and parent detectors to weather stations and tweeting birdhouses with infra-red cameras. We want to see the Raspberry Pi being

used by people all over the world to learn to program and understand how computers work.

Why Raspberry pi?

- Complete computing platform.
- Cheap.
- Raw.
- It doesn't come hidden away in a box; you have access to the GPIO.
- Runs Linux.

3.3.1. Features

	Raspberry Pi 3 Model B	Raspberry Pi 2 Model B	Raspberry Pi Model B+
Processor Chipset	**Broadcom BCM2837 64Bit Quad Core ARM Cortex A53 at 1.2GHz**	Broadcom BCM2836 32Bit Quad Core ARMv7 at 900MHz	Broadcom BCM2835 32Bit ARMv6k at 700MHz
GPU	**Videocore IV @ 400MHz**	Videocore IV @ 250MHz	Videocore IV @ 250MHz
Processor Speed	**QUAD Core @1.2 GHz**	QUAD Core @900 MHz	Single Core @700 MHz
RAM	1GB SDRAM @ 400 MHz	1GB SDRAM @ 400 MHz	512 MB SDRAM @ 400 MHz
Storage	MicroSD	MicroSD	MicroSD
USB 2.0	4x USB Ports	4x USB Ports	4x USB Ports
Max Power Draw/voltage	2.5A @ 5V	1.8A @ 5V	1.8A @ 5V
GPIO	40 pin	40 pin	40 pin
Ethernet Port	Yes	Yes	Yes
WiFi	**Built in**	No	No
Bluetooth LE	**Built in**	No	No
Video Output	HDMI/Composite via RCA Jack	HDMI/Composite via RCA Jack	HDMI/Composite via RCA Jack
Audio Output	3.5mm Jack	3.5mm Jack	3.5mm Jack

Table 3.4: RPI Features.

We used **Raspberry Pi 3 Model B**.

- CPU: Quad-core 64-bit ARM Cortex A53 clocked at 1.2 GHz.
- GPU: 400MHz VideoCore IV multimedia.
- Memory: 1GB LPDDR2-900 SDRAM (i.e. 900MHz).
- USB ports: 4.
- Video outputs: HDMI, composite video (PAL and NTSC) via 3.5 mm jack.
- Network: 10/100Mbps Ethernet and 802.11n Wireless LAN.
- Peripherals: 17 GPIO plus specific functions, and HAT ID bus.
- Bluetooth: 4.1.
- Power source: 5 V via MicroUSB or GPIO header.
- Size: 85.60mm × 56.5mm.
- Weight: 45g (1.6 oz).

3.3.2. Raspberry Pi 3 Components and Pinout

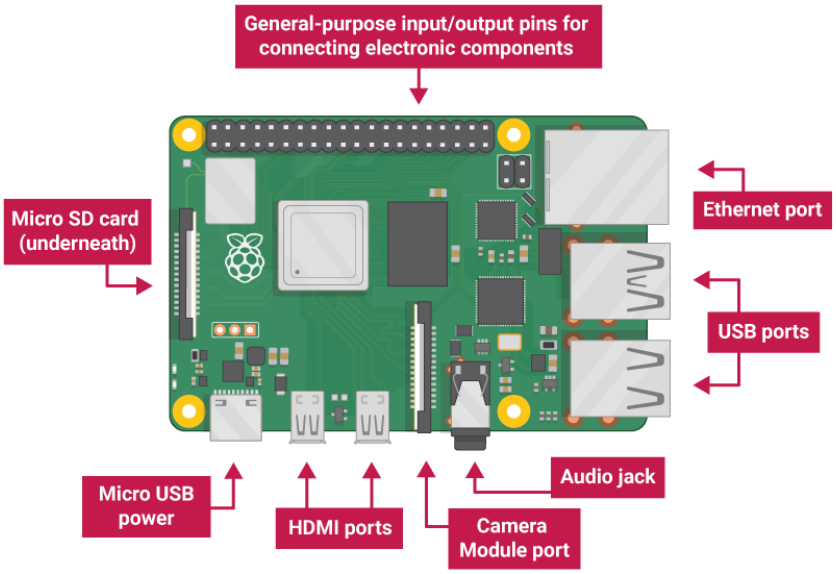


Figure 3.8: Raspberry Pi 3 Components (1).

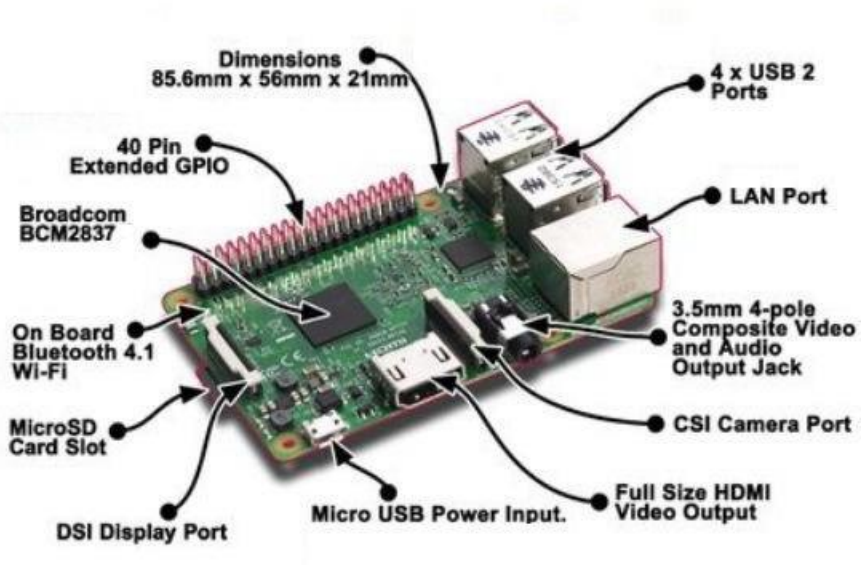


Figure 3.9: Raspberry Pi 3 Components (2).

- **USB ports**—these are used to connect a mouse and keyboard. You can also connect other components, such as a USB drive.
- **SD card slot**—you can slot the SD card in here. This is where the operating system software and your files are stored.
- **Ethernet port**—this is used to connect Raspberry Pi to a network with a cable. Raspberry Pi can also connect to a network via wireless LAN.
- **Audio jack**—you can connect headphones or speakers here.
- **HDMI port**—this is where you connect the monitor (or projector) that you are using to display the output from the Raspberry Pi. If your monitor has speakers, you can also use them to hear sound.
- **Micro USB power connector**—this is where you connect a power supply. You should always do this last, after you have connected all your other components.
- **GPIO ports**—these allow you to connect electronic components such as LEDs and buttons to Raspberry Pi.

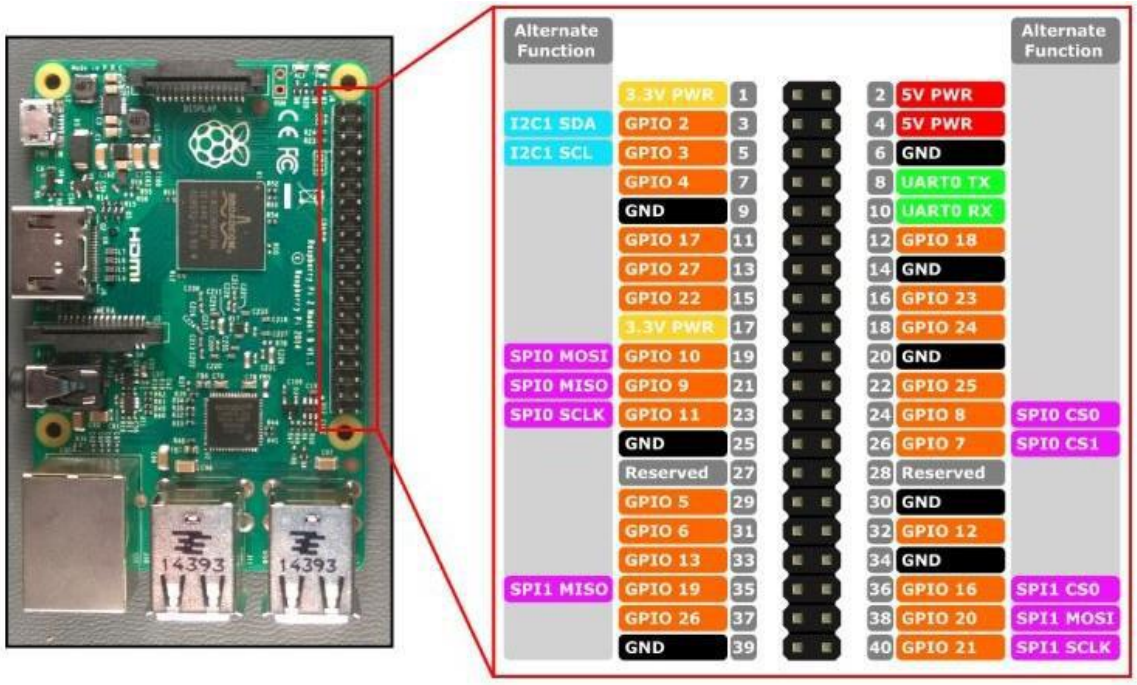


Figure 3.10: Raspberry Pi 3 Pinout.

3.3.3. OS Installation

Step 1: Insert a microSD card into your computer. Your card should be 8 GB or Larger.

Step 2: Install Etcher, Click the Select image button and choose the Raspbian image to flash.

Figure 3.11: OS Installation Step 2.

Step 3: Select the SD card and click flash.

Figure 3.12: OS Installation Step 3.

Step 4: Etcher will take a few minutes to install Raspbian on your microSD card When it's done at least in Windows you'll see a number of alerts prompting you to format the card Close these dialog boxes or hit cancel on them (you will format over the OS).

Figure 3.13: OS Installation Step 4.

Step 5 Write an empty text file named "ssh"(no file extension) to the root of the directory of the card When it sees the " on its first boot up, Raspbian will automatically enable SSH (Secure Socket Shell), which will allow you to remotely access the Pi command line from your PC.

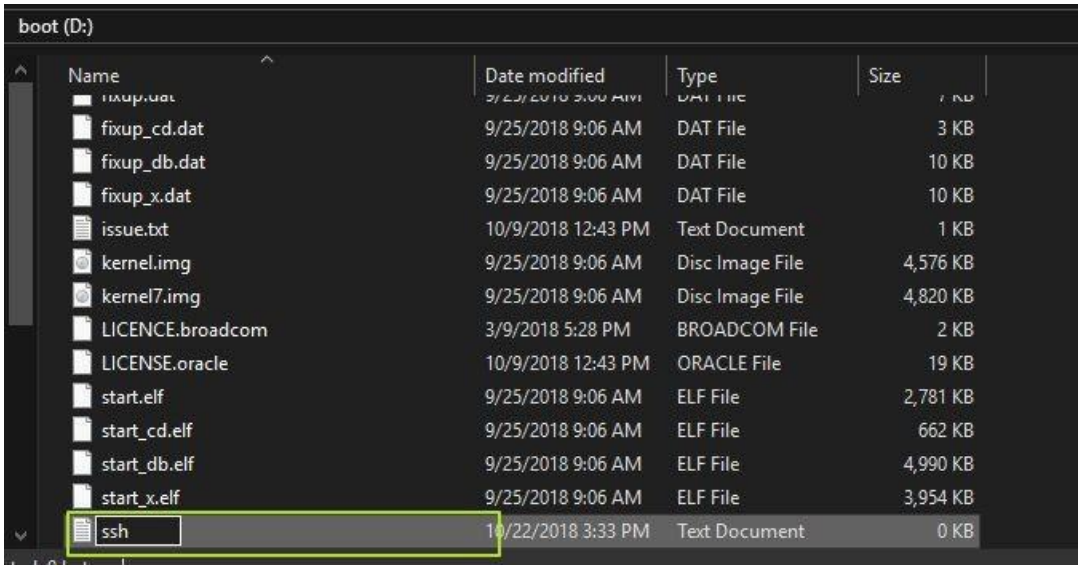


Figure 3.14: OS Installation Step 5.

Step 6: For headless mode using Wi-Fi create file called “wpa_supplicant.conf” in boot partition and add the following configuration of your Wi-Fi.

```
country=ie
update_config=1
ctrl_interface=/var/run/wpa_supplicant

network={
 scan_ssid=1
 ssid="[redacted]"
 psk="88888888"
}
```

Figure 3.15: OS Installation Step 6.

Step 7: Navigate to the Network Connections menu, which is part of the old school Control Panel You can get to this screen by going to Settings -->Network Internet -->Wi Fi and then clicking "Change Adapter Settings" on the right side of the screen This works whether you are sharing an Internet connection that comes to your PC from Wi Fi or from Ethernet.

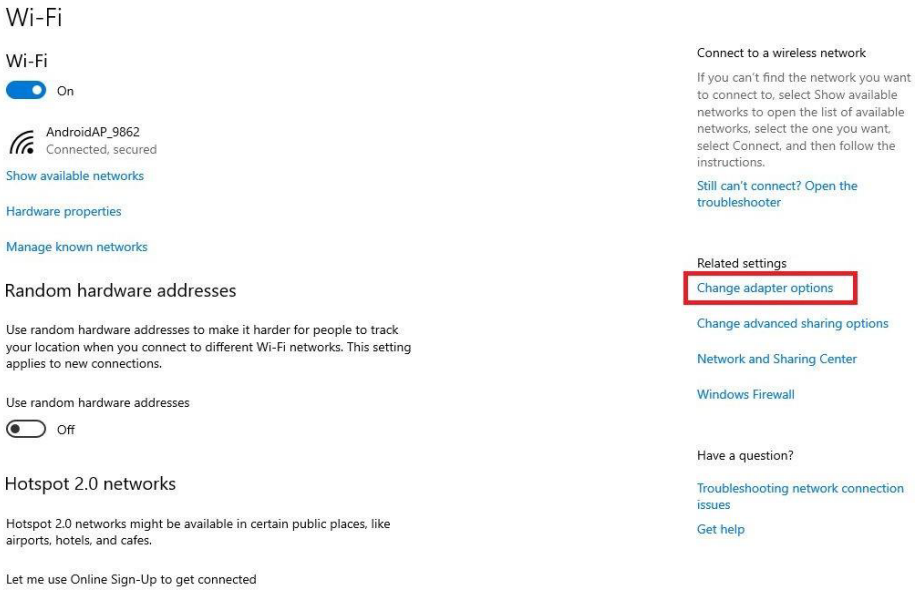


Figure 3.16: OS Installation Step 7.

Step 8: Right click on the adapter that's connected to the Internet and select properties.

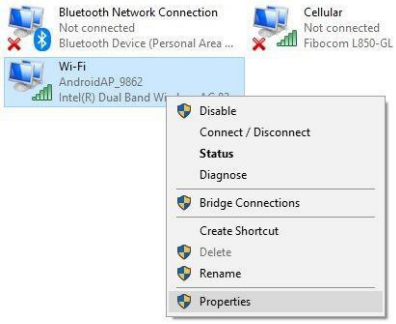


Figure 3.17: OS Installation Step 8.

Step 9: Enable "Allow other network users to connect" on the " tab.

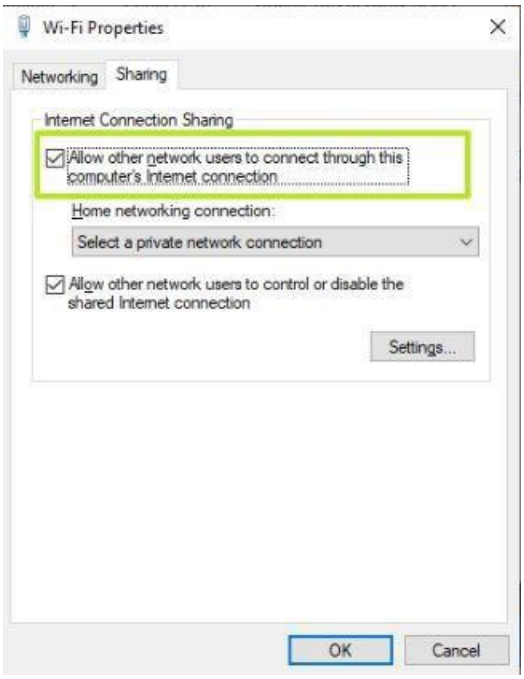


Figure 3.18: OS Installation Step 9.

Step 10: Now we can communicate through **putty** to the raspberry pi Enter raspberrypi or raspberrypi.local as the address you wish to connect to in Putty, and click Open.

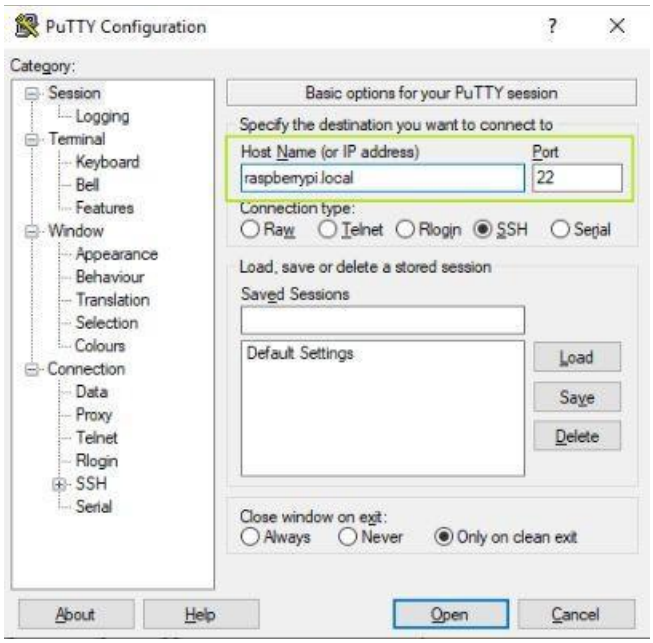


Figure 3.19: OS Installation Step 10.

Step 11: Enter “pi” as your username and “raspberry” as your password.

Figure 3.20: OS Installation Step 11.

Step 12: after successful connection to Pi, Enter “sudo raspi-config” at the command prompt. Then choose Interfacing Options.

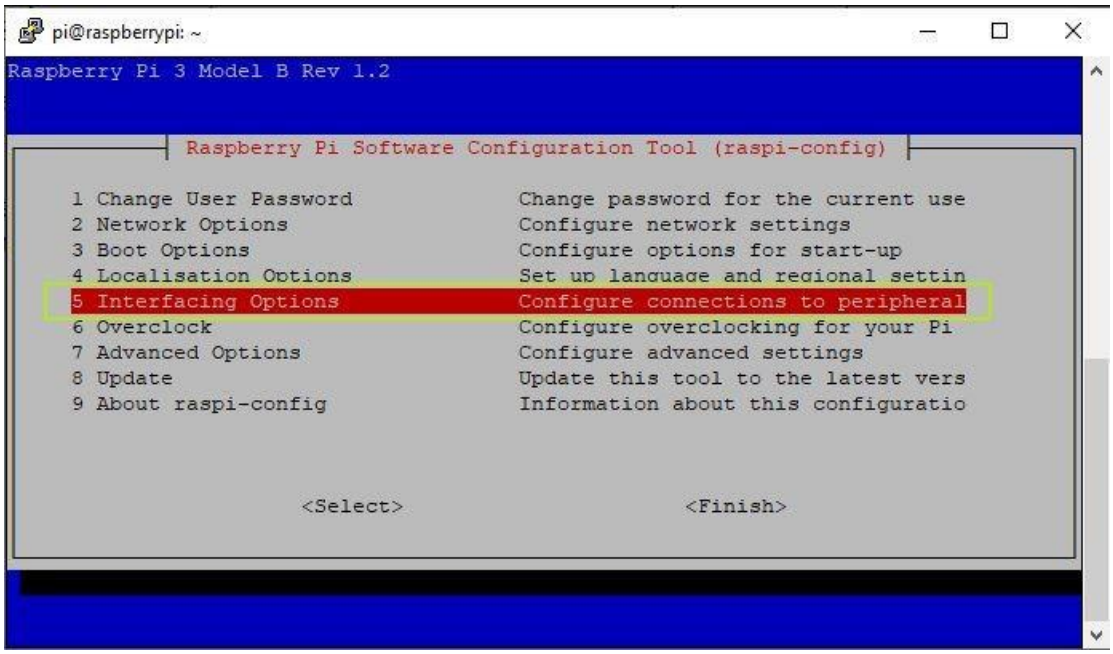


Figure 3.21: OS Installation Step 12.

Step 13: Enable VNC (Virtual Network Computing).

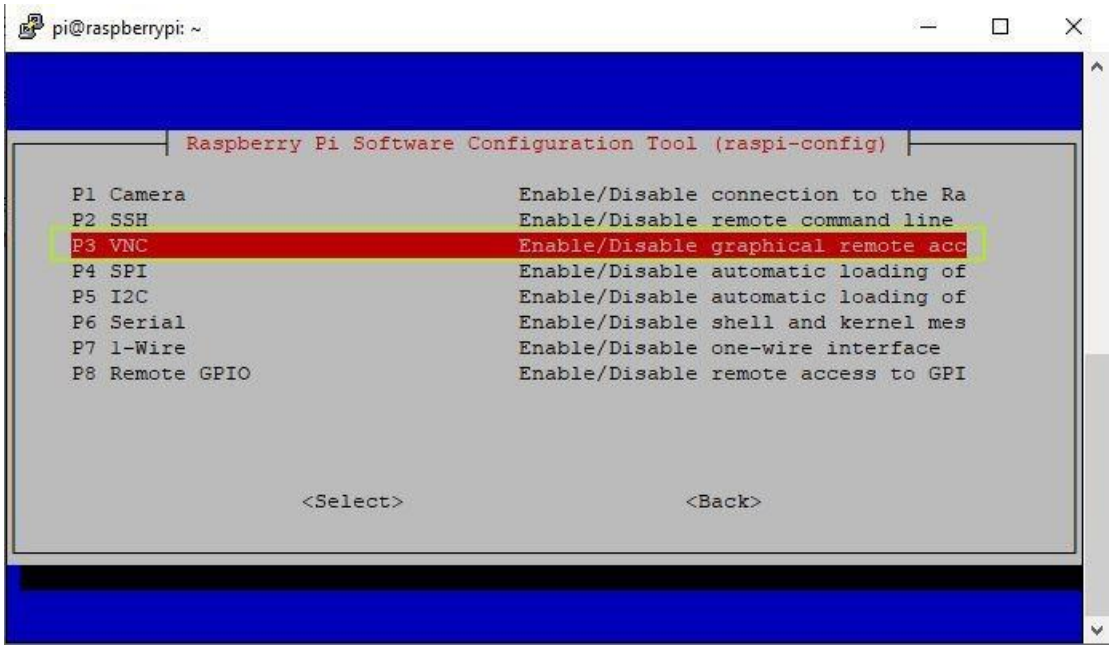


Figure 3.22: OS Installation Step 13.

Step 14: launch VNC Viewer and Select New connection from the File menu.

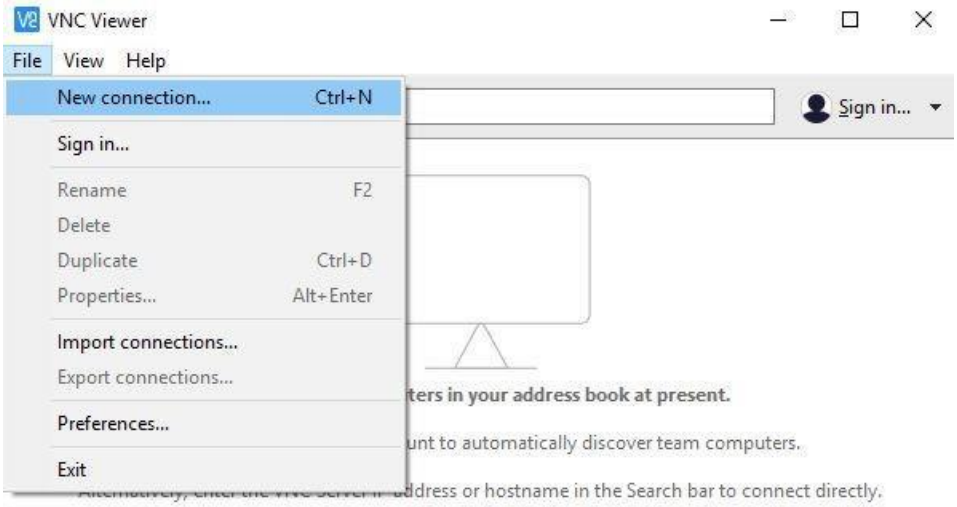


Figure 3.23: OS Installation Step 14.

Step 15: Enter “raspberry.local” in the "VNC Server" field. If this does not work, try again with the name "raspberrypi" without “.local” then click ok.

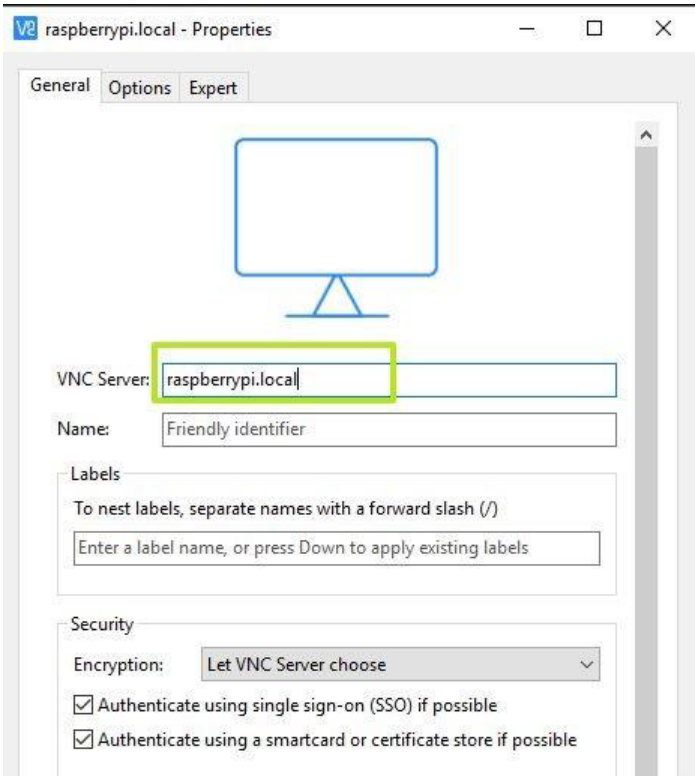


Figure 3.24: OS Installation Step 15.

Step 16: Double click on the connection icon to connect and Click Ok if you are shown a security warning.

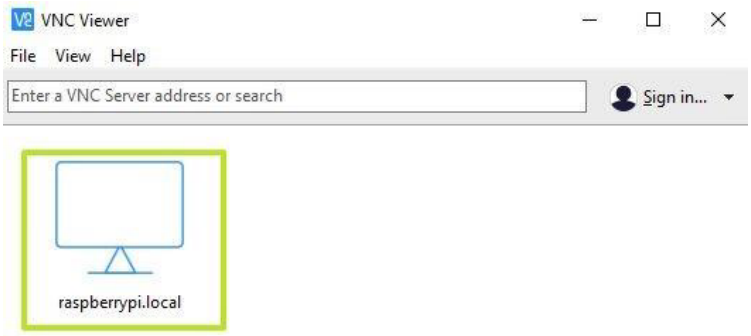


Figure 3.25: OS Installation Step 16.

Step 17: Enter the Pi's username and password when prompted. The Defaults are username: pi and password: raspberry.

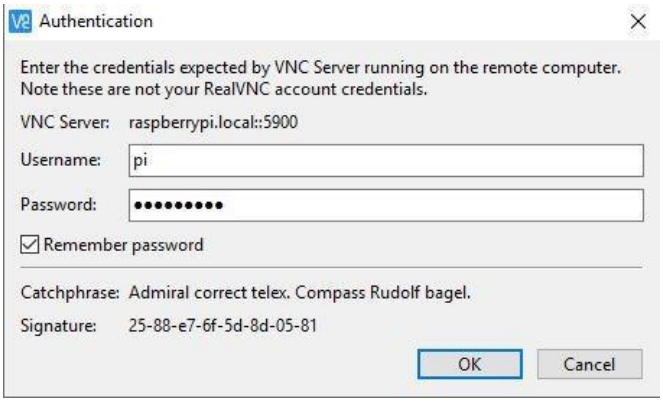


Figure 3.26: OS Installation Step 17.

Step 18: Adjust resolution.

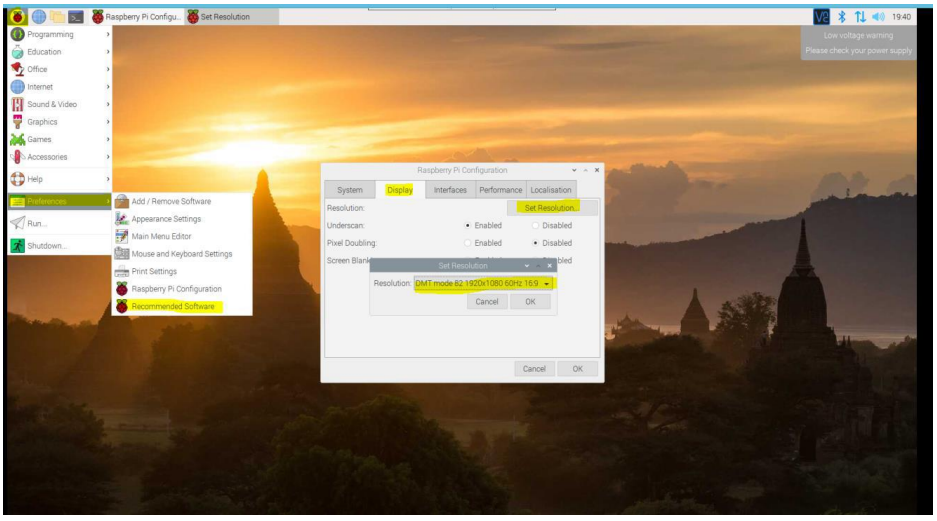


Figure 3.27: OS Installation Step 18.

3.3.4. Configuration

3.3.4.1. UART Configuration

Enable UART in the Raspberry Pi, Open the Raspberry Pi terminal and insert the following commands.

Step 1: Open the config.txt file in the nano editor using the following command.

```
sudo nano /boot/config.txt
```

Add the following lines to the end of the file.

```
# Enable UART
enable_uart=1
```

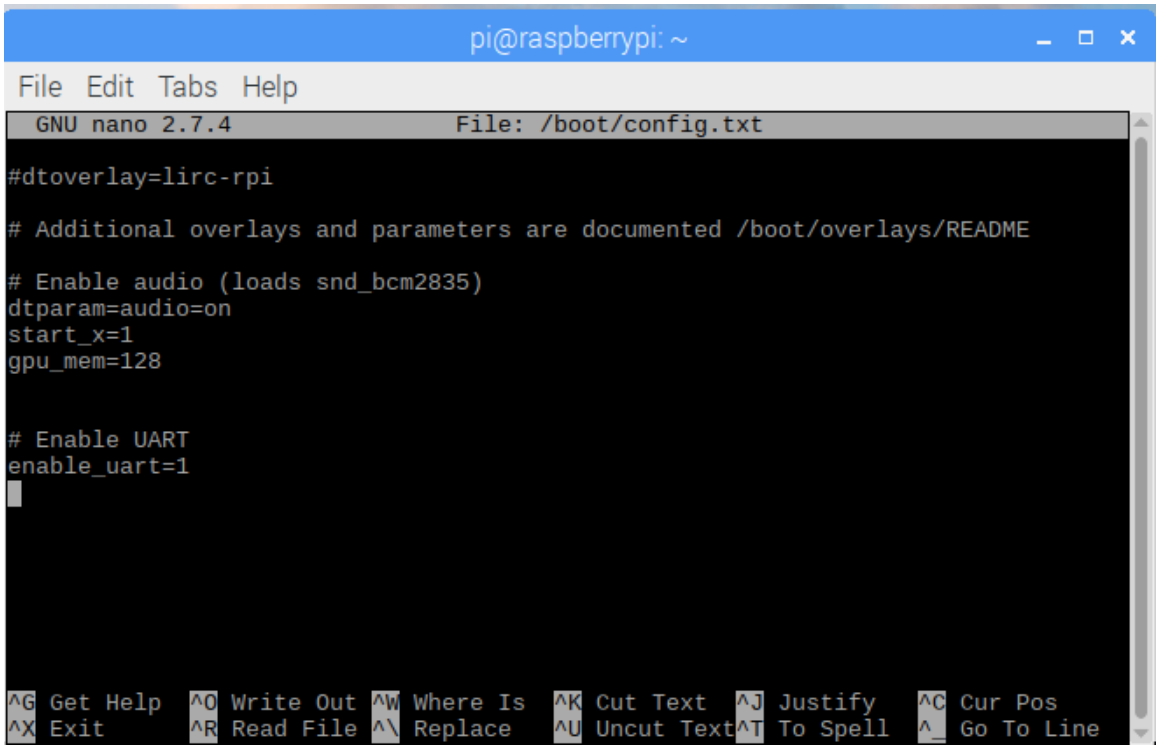


Figure 3.28: UART Configuration Step 1.

Then press Ctrl+x and press Y to save the file and close it.

Step 2: Disconnect the serial communication between the Raspberry Pi and the Bluetooth module.

```
sudo systemctl disable serial-getty@ttyS0.service
```

Step 3: Open cmdline.txt in nano editor.

Delete the “console=serial0,115200” line and save the file.

sudo nano /boot/cmdline.txt

Step 4: Reboot your Raspberry Pi.

sudo reboot

Step 5: Install python-serial.

sudo apt-get install python-serial
sudo apt-get install python3-serial

Step 5: Use python to receive data from Microcontrollers.

UART.py *

```
1  import serial
2
3  ser = serial.Serial("/dev/ttyS0",baudrate=9600,timeout=0.5)
4  ser.write('Connected\r\n')
5
6  while True:
7      data = ser.readline()
8      print(data)
```

3.3.4.2. Bluetooth Configuration

We used it for GPS.

Enable Bluetooth in the Raspberry Pi, Open the Raspberry Pi terminal and insert the following commands.

Step 1: Add the SP profile to the Pi. Edit this file:

sudo nano /etc/systemd/system/dbus-org.bluez.service

Step 2: Add the compatibility flag, ' -C', at the end of the 'ExecStart=' line. Add a new line after that to add the SP profile. The two lines should look like this:

```
ExecStart=/usr/lib/bluetooth/bluetoothd -C
ExecStartPost=/usr/bin/sdptool add SP
NotifyAccess=main
```

Figure 3.29: Bluetooth Configuration Step 2.

Step 3: Save the file and reboot.

Sudo reboot

Step 4: Pair and trust your Pi and phone with bluetoothctl or with GUI.

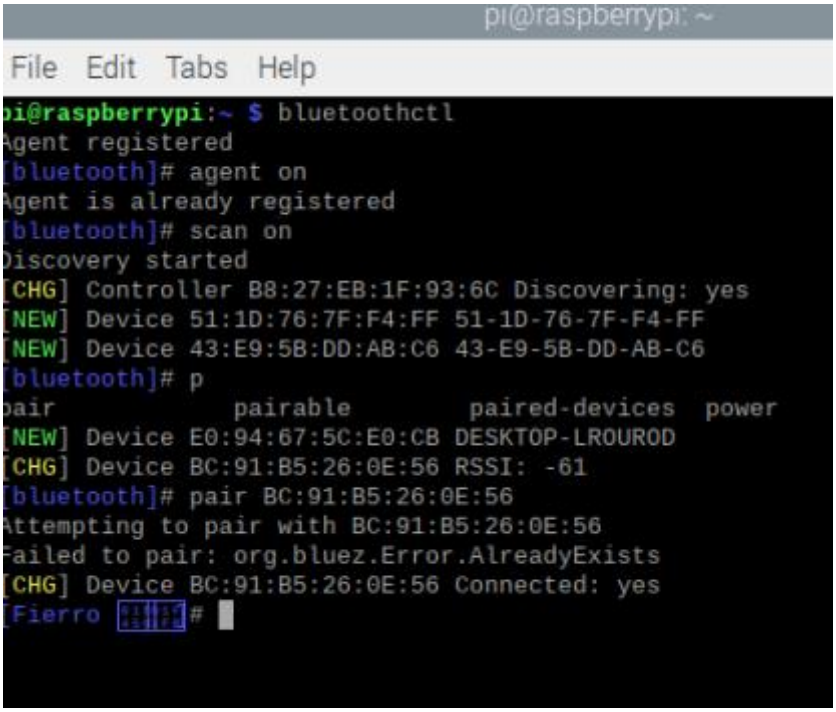


Figure 3.30: Bluetooth Configuration Step 4 - Terminal.

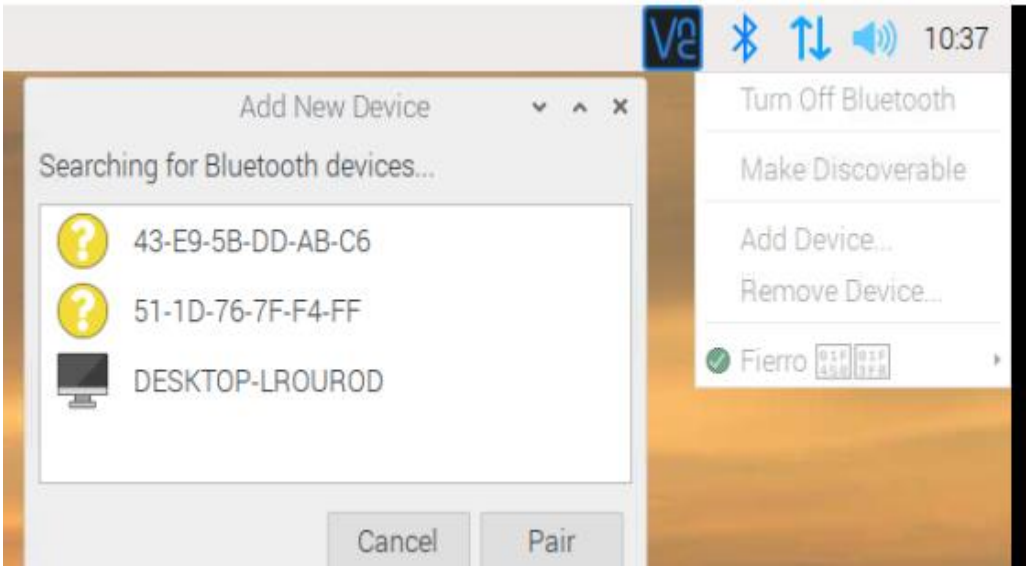


Figure 1.31: Bluetooth Configuration Step 4 - GUI.

4. Communication Protocols

4.1. UART

4.1.1. Introduction

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. UART is also a common integrated feature in most microcontrollers. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. Serial transmission of digital information (bits) through a single wire or other medium is much more cost effective than parallel transmission through multiple wires. Communication can be “full duplex” (both send and receive at the same time) or “half duplex” (devices take turns transmitting and receiving).

4.1.2. The Asynchronous Receiving and Transmitting Protocol

When a word is passed to the UART for asynchronous transmissions, the Start bit is added at beginning of the word. The Start bit is used to inform the receiver that a

Word of data is about to be send, thereby forcing the clock in the receiver to be in

Sync with the clock in the transmitter. It is important to note that the frequency drift Between these two clocks must not exceed 10%. In other words, both the transmitter and receiver must have identical baud rate. After the Start bit, the individual bits of the word of data are sent, beginning with the Least Significant Bit (LSB). When data is fully transmitted, an optional Parity bit is sent to the transmitter. This bit is

usually used by receiver to perform simple error checking. Lastly, stop bit will be sent to indicate the end of transmission. When the receiver has received all the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver searches for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. Common reason for the occurrence of Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted.

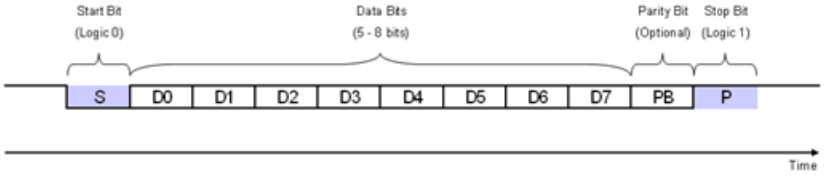


Figure 4.1: The Frame of UART.

Every operation of the UART hardware is controlled by a clock signal which runs at much faster rate than the baud rate. For example, the popular 16450 UART has an internal clock that runs 16 times faster than the baud rate. This allows the UART receiver to sample the incoming data with granularity of 1/16 the baud-rate period and has greater immunity towards baud rate error. The receiver detects the Start bit by detecting the voltage transition from logic 1 to logic 0 on the transmission line. In the case of 16450 UART, once the Start bit is detected, the next data bit's "center" can be assured to be 24 ticks minus 2 (worse case synchronizer uncertainty) later. From then on, every next data bit center is 16 clock ticks later. Transmitting and receiving UARTs must be set at the same baud rate, character length, parity, and stop bits for proper operation. The typical format for serial ports used with PC connected to modems is 1 Start bit, 8 data bits, no Parity and 1 Stop bit.

4.2. I2C

I²C (**Inter-Integrated Circuit**), pronounced *I-squared-C*, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus invented in 1982 by Philips Semiconductor (now NXP Semiconductors). It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.
The Inter-Integrated Circuit (I2C) bus provides bi-directional data transfer through a two-wire design (a serial data line SDA and a serial clock line SCL), and interfaces to external I2C devices such as serial memory (RAMs and ROMs), networking devices, LCDs, tone generators, and so on. The I2C bus may also be used for system testing and diagnostic purposes in product development and manufacturing. The TM4C123GH6PM microcontroller includes providing the ability to communicate (both transmit and receive) with other I2C devices on the bus.

The TM4C123GH6PM controller includes I2C modules with the following features:

■ Devices on the I2C bus can be designated as either a master or a slave
– Supports both transmitting and receiving data as either a master or a slave
– Supports simultaneous master and slave operation
■ Four I2C modes
– Master transmit.
– Master receive.
– Slave transmit.
– Slave receive.
■ Four transmission speeds:
– Standard (100 Kbps).
– Fast-mode (400 Kbps).
– Fast-mode plus (1 Mbps).
– High-speed mode (3.33 Mbps).
■ Clock low timeout interrupt.
■ Dual slave address capability.

■ Glitch suppression.

■ Master and slave interrupt generation.

– Master generates interrupts when a transmit or receive operation completes (or aborts due to an error).

– Slave generates interrupts when data has been transferred or requested by a master or when a START or STOP condition is detected.

■ Master with arbitration and clock synchronization, multimaster support, and 7-bit addressing Mode.

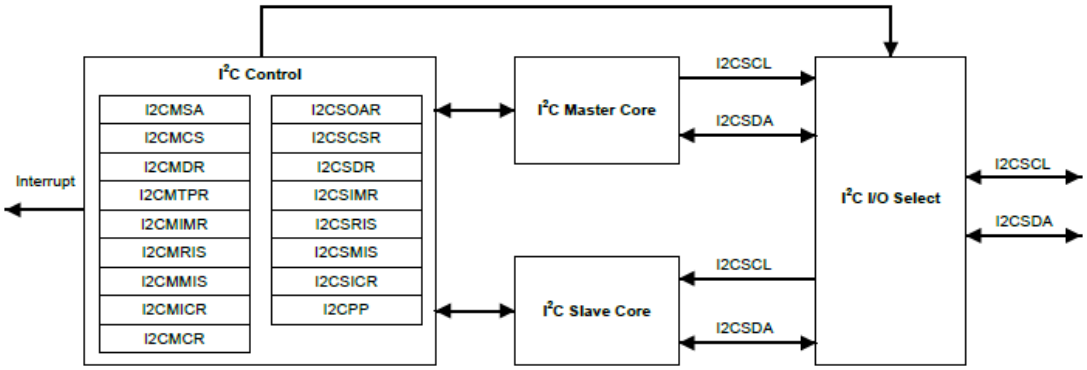


Figure 4.2: The block diagram of I2C protocol.

4.2.1. Signal Description

Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type[a]	Description
I2C0SCL	47	PB2 (3)	I/O	OD	I²C module 0 clock. Note that this signal has an active pull-up. The corresponding port pin should not be configured as open drain.
I2C0SDA	48	PB3 (3)	I/O	OD	I²C module 0 data.
I2C1SCL	23	PA6 (3)	I/O	OD	I²C module 1 clock. Note that this signal has an active pull-up. The corresponding port pin should not be configured as open drain.
I2C1SDA	24	PA7 (3)	I/O	OD	I²C module 1 data.
I2C2SCL	59	PE4 (3)	I/O	OD	I²C module 2 clock. Note that this signal has an active pull-up. The corresponding port pin should not be configured as open drain.
I2C2SDA	60	PE5 (3)	I/O	OD	I²C module 2 data.
I2C3SCL	61	PD0 (3)	I/O	OD	I²C module 3 clock. Note that this signal has an active pull-up. The corresponding port pin should not be configured as open drain.
I2C3SDA	62	PD1 (3)	I/O	OD	I²C module 3 data.

a. The TTL designation indicates the pin has TTL-compatible voltage levels.

Figure 4.3: I2C's signal description.

4.2.2. I2C Bus Functional Overview

The I2C bus uses only two signals: SDA and SCL, named I2CSDA and I2CSCL on TM4C123GH6PM Microcontrollers. SDA is the bi-directional serial data line and SCL is the bi-directional serial clock line. The bus is considered idle when both lines are High.
Every transaction on the I2C bus is nine bits long, consisting of eight data bits and a single acknowledge bit. The number of bytes per transfer (defined as the time between a valid START and STOP condition, described in “START and STOP conditions” on page 999) is unrestricted, but each data byte has to be followed by an acknowledge bit, and data must be transferred MSB first.
When a receiver cannot receive another complete byte, it can hold the clock line SCL Low and force the transmitter into a wait state. The data transfer continues when the receiver releases the clock SCL.

- **START and STOP Conditions:**
  The protocol of the I2C bus defines two states to begin and end a transaction: START and STOP.
  A High-to-Low transition on the SDA line while the SCL is High is defined as a START condition, and a Low-to-High transition on the SDA line while SCL is High is defined as a STOP condition.
  The bus is considered busy after a START condition and free after a STOP condition.

Figure 4.4: The start and stop conditions of the I2C.

The STOP bit determines if the cycle stops at the end of the data cycle or continues to a repeated START condition.
To generate a single, transmit cycle, the I2C Master Slave Address (I2CMSA) register is written with the desired address, the R/S bit is cleared, and the Control register is written with ACK=X (0 or 1), STOP=1, START=1, and RUN=1 to perform the operation and stop. When the operation is completed (or aborted due an error), the interrupt pin

becomes active and the data may be read from the I2C Master Data (I2CMDR) register. When the I2C module operates in Master receiver mode, the ACK bit is normally set causing the I2C bus controller to transmit an acknowledge automatically after each byte. This bit must be cleared when the I2C bus controller requires no further data to be transmitted from the slave transmitter.

When operating in slave mode, the STARTRIS and STOPRIS bits in the I2C Slave Raw Interrupt Status (I2CSRIS) register indicate detection of start and stop conditions on the bus and the I2C Slave Masked Interrupt Status (I2CSMIS) register can be configured to allow STARTRIS and STOPRIS to be promoted to controller interrupts (when interrupts are enabled).

- **Data Format with 7-Bit Address:**

  The protocol of the I2C bus defines two states to begin and end a transaction: START and STOP.

  A High-to-Low transition on the SDA line while the SCL is High is defined as a START condition, and a Low-to-High transition on the SDA line while SCL is High is defined as a STOP condition.

  The bus is considered busy after a START condition and free after a STOP condition.

  master, however, a master can initiate communications with another device on the bus by generating a repeated START condition and addressing another slave without first generating a STOP condition. Various combinations of receive/transmit formats are then possible within a single transfer.

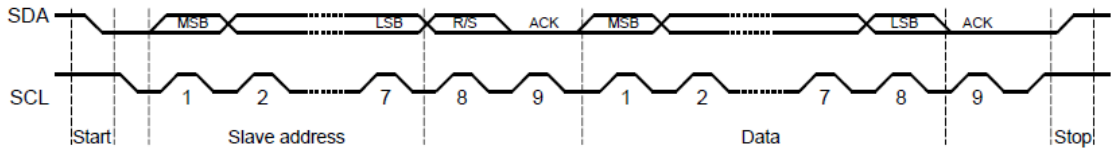


Figure 4.5: The data format with 7-Bit address.

The first seven bits of the first byte make up the slave address. The eighth bit

determines the direction of the message. A zero in the R/S position of the first byte means that the master transmits (sends) data to the

selected slave, and a one in this position means that the master receives data from the slave.

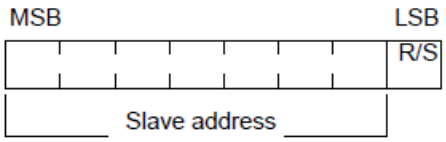


Figure 4.6: The slave address in the I2C.

4.2.3. Acknowledge

All bus transactions have a required acknowledge clock cycle that is generated by the master. During the acknowledge cycle, the transmitter (which can be the master or slave) releases the SDA line.

To acknowledge the transaction, the receiver must pull down SDA during the acknowledge clock cycle. The data transmitted out by the receiver during the acknowledge cycle must comply with the data validity requirements described in “Data Validity” on page 1000.

When a slave receiver does not acknowledge the slave address, SDA must be left High by the slave so that the master can generate a STOP condition and abort the current transfer. If the master device is acting as a receiver during a transfer, it is responsible for acknowledging each transfer made by the slave. Because the master controls the number of bytes in the transfer, it signals the end of data to the slave transmitter by not generating an acknowledge on the last data byte. The slave transmitter must then release SDA to allow the master to generate the STOP or a repeated START condition. If the slave is required to provide a manual ACK or NACK, the I2C Slave ACK Control (I2CSACKCTL) register allows the slave to NACK for invalid data or command or ACK for valid data or command. When this operation is enabled, the MCU slave module I2C clock is pulled low after the last data bit until this register is written with the indicated response.

Available Speed Modes:

The I2C bus can run in Standard mode (100 kbps), Fast mode (400 kbps), Fast mode plus (1 Mbps) or High-Speed mode (3.33 Mbps). The selected mode should match the speed of the other I2C devices on the bus.

Interrupts:

The I2C can generate interrupts when the following conditions are observed:

- Master transaction completed
- Master arbitration lost
- Master transaction error
- Master bus timeout
- Slave transaction received
- Slave transaction requested
- Stop condition on bus detected
- Start condition on bus detected

The I2C master and I2C slave modules have separate interrupt signals. While both modules can generate interrupts for multiple conditions, only a single interrupt signal is sent to the interrupt controller.

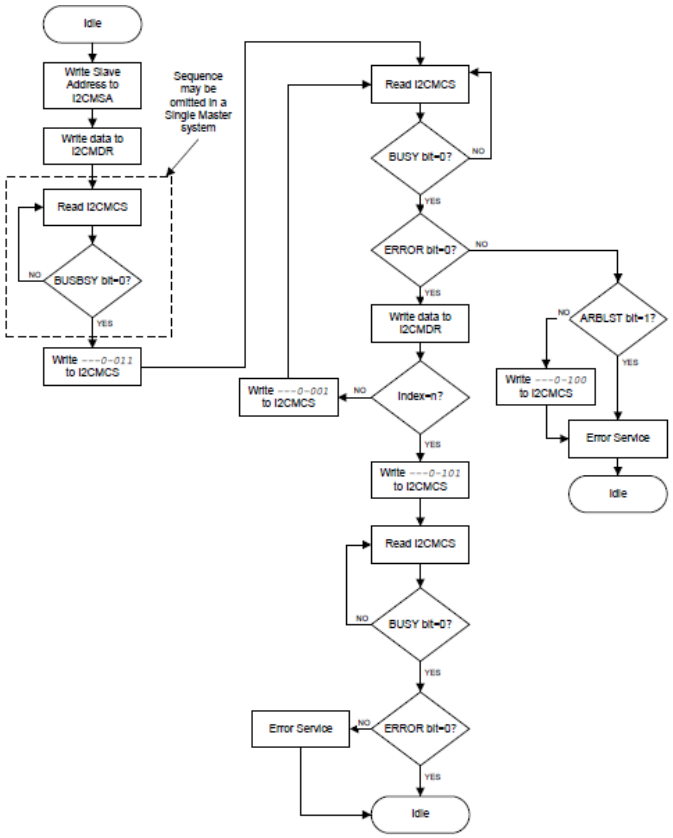


Figure 4.7: The Master TRANSMIT of Multiple Data Bytes.

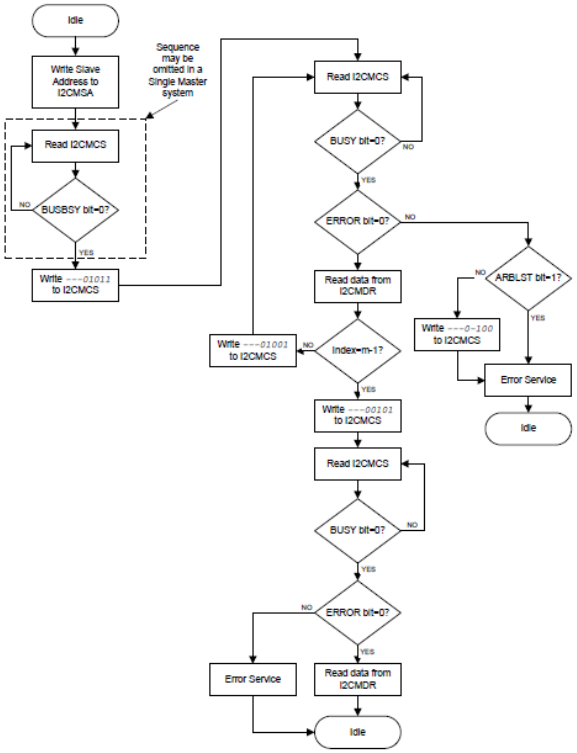


Figure 4.8: The Master Receive of Multiple Data Bytes.

4.3. CAN

4.3.1. Overview

Controller Area Network (CAN), Development of the CAN bus started in 1983 at Robert Bosch GmbH The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) conference in Detroit, Michigan. The first CAN controller chips, produced by Intel and Philips, came on the market in 1987. Released in 1991 the Mercedes-Benz W140 was the first production vehicle to feature a CAN-based multiplex wiring system. Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. A CAN device that uses 11-bit identifiers is commonly called CAN 2.0A and a CAN device that uses 29-bit identifiers is commonly called CAN 2.0B. These standards are freely available from Bosch along with other specifications and white papers. In 1993, the International Organization for Standardization (ISO) released the CAN standard ISO 11898 which was later restructured into two parts; ISO 11898-1 which covers the data link layer, and ISO 11898-2 which covers the CAN physical layer for high-speed CAN. ISO 11898-3 was released later and covers the CAN physical layer for low-speed, fault-tolerant CAN. The physical layer standards ISO 11898-2 and ISO, 11898-3 are not part of the Bosch CAN 2.0 specification. These standards may be purchased from the ISO.

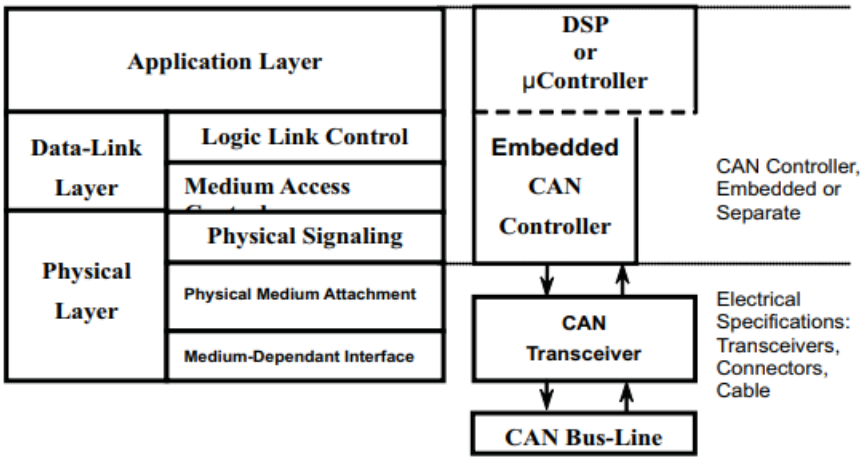


Figure 4.9: the layered ISO 11898 standard architecture.

Bosch is still active in extending the CAN standards. In 2012, Bosch released CAN FD 1.0 or CAN with Flexible Data-Rate. This specification uses a different frame format that allows a different data length as well as optionally switching to a faster bit rate after the arbitration is decided. CAN FD is compatible with existing CAN 2.0 networks so new CAN FD devices can coexist on the same network with existing CAN devices. CAN bus is one of five protocols used in the on-board diagnostics (OBD)-II vehicle diagnostics standard. The OBD-II standard has been mandatory for all cars and light trucks sold in the United States since 1996. The EOBD standard has been mandatory for all petrol vehicles sold in the European Union since 2001 and all diesel vehicles since 2004.

4.3.2. Applications

- Passenger vehicles, trucks, buses (gasoline vehicles and electric vehicles).
- Electronic equipment for aviation and navigation.
- Industrial automation and mechanical control.
- Elevators, escalators Building automation.
- Medical instruments and equipment.

4.3.3. Operation of CAN bus

Think of the CAN bus as a simple network where any system in the car can listen and send commands to. It integrates all of these complex components in an elegant way, allowing for many of the modern features we all love in vehicles today. Physically: CAN is a multi-master serial bus standard for connecting Electronic Control Units [ECUs] also known as nodes. Two or more nodes are required on the CAN network to communicate. The complexity of the node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software. The node may also be a gateway allowing a general purpose computer (such as a laptop) to communicate over a USB or Ethernet port to the devices on a CAN network. All nodes are connected to each other through a two wire bus. The wires are a twisted pair with a 120 Ω (nominal) characteristic impedance. ISO 11898-2, also called high speed CAN (bit speeds up to 1Mb/s on CAN, 5Mb/s on CAN-FD), uses a linear bus terminated at each end with 120 Ω resistors. High speed CAN signaling drives the CAN high wire towards 5 V and the CAN low wire towards 0 V when transmitting a dominant (0), and does not drive either wire when transmitting a recessive (1). Designating "0" as dominant gives the nodes with the lower ID numbers priority on the bus. The dominant differential voltage is a nominal 2 V. The termination resistor passively returns the two wires to a nominal differential voltage of 0 V. The dominant common mode voltage must be within 1.5 to 3.5 V of common and the recessive common mode voltage must be within +/-12 of common. ISO 11898-3, also called low speed or fault tolerant CAN (up to 125 Kbps), uses a linear bus, star bus or multiple star buses connected by a linear bus and is terminated at each node by a fraction of the overall termination resistance. The overall termination resistance should be about 100 Ω, but not less than 100 Ω. Low speed/Fault tolerant CAN signaling drives the CAN high wire towards 5 V and the CAN low wire towards 0 V when transmitting a dominant (0), and does not drive either wire when transmitting a recessive (1). The dominant differential voltage must be greater than 2.3 V (with a 5 V Vcc) and the

recessive differential voltage must be less than 0.6 V The termination resistors passively return the CAN low wire to RTH where RTH is a minimum of 4.7 V (Vcc - 0.3 V where Vcc is 5 V nominal) and the CAN high wire to RTL where RTL is a maximum of 0.3 V. Both wires must be able to handle -27 to 40 V without damage.

4.3.4. Electrical properties

With both high speed and low speed CAN, the speed of the transition is faster when a recessive to dominant transition occurs since the CAN wires are being actively driven. The speed of the dominant to recessive transition depends primarily on the length of the CAN network and the capacitance of the wire used. High speed CAN is usually used in automotive and industrial applications where the bus runs from one end of the environment to the other. Fault tolerant CAN is often used where groups of nodes need to be connected together. The specifications require the bus be kept within a minimum and maximum common mode bus voltage, but do not define how to keep the bus within this range. The CAN bus must be terminated. The termination resistors are needed to suppress reflections as well as return the bus to its recessive or idle state. High speed CAN uses a 120 Ω resistor at each end of a linear bus. Low speed CAN uses resistors at each node. Other types of terminations may be used such as the Terminating Bias Circuit defined in ISO11783. A terminating bias circuit provides power and ground in addition to the CAN signaling on a four-wire cable. This provides automatic electrical bias and termination at each end of each bus segment. An ISO11783 network is designed for hot plug-in and removal of bus segments and ECUs.

Nodes Each node requires a

- Central processing unit, microprocessor, or host processor
  - The host processor decides what the received messages mean and what messages it wants to transmit.

- Sensors, actuators and control devices can be connected to the host processor.

- CAN controller; often an integral part of the microcontroller.
  - Receiving: the CAN controller stores the received serial bits from the bus until an entire message is available, which can then be fetched by the host processor (usually by the CAN controller triggering an interrupt).

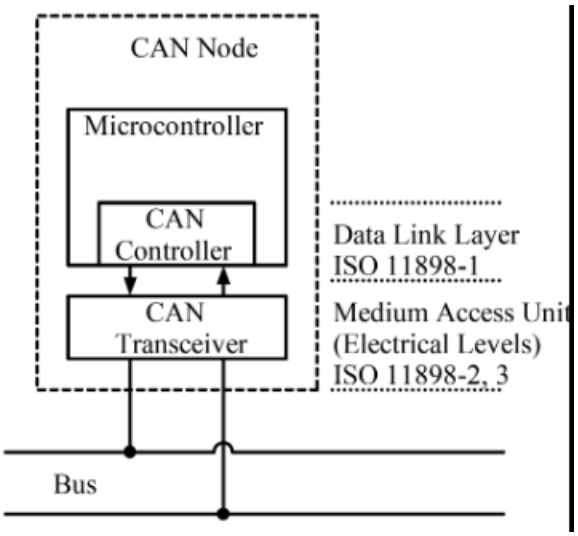


Figure 4.10: CAN node.

  - Sending: the host processor sends the transmit message(s) to a CAN controller, which transmits the bits serially onto the bus when the bus is free.

- Transceiver Defined by ISO 11898-2/3 Medium Access Unit [MAU] standards.
  - Receiving: it converts the data stream from CAN bus levels to levels that the CAN controller uses. It usually has protective circuitry to protect the CAN controller.
  - Transmitting: it converts the data stream from the CAN controller to CAN bus levels.

Each node is able to send and receive messages, but not simultaneously. A message or Frame consists primarily of the ID (identifier), which represents the priority of the message, and up to eight data bytes. A CRC,

acknowledge slot [ACK] and other overhead are also part of the message. The improved CAN FD extends the length of the data section to up to 64 bytes per frame. The message is transmitted serially onto the bus using a non-return-to-zero (NRZ) format and may be received by all nodes.
The devices that are connected by a CAN network are typically sensors, actuators, and other control devices. These devices are connected to the bus through a host processor, a CAN controller, and a CAN transceiver.

4.4. Bluetooth

Bluetooth wireless technology is a short-range communications technology intended to replace the cables connecting portable and/or fixed devices while maintaining high levels of security.
The key features of Bluetooth technology are robustness, low power, and low cost.
The Bluetooth specification defines a uniform structure for a wide range of devices to connect and communicate with each other. Bluetooth technology has achieved global acceptance such that any Bluetooth enabled device, almost everywhere in the world, can connect to other Bluetooth enabled devices in proximity. Bluetooth enabled electronic devices connect and communicate wirelessly through short-range, ad hoc networks known as piconets. Each device can simultaneously communicate with up to seven other devices within a single piconet. Each device can also belong to several piconets simultaneously. Piconets are established dynamically and automatically as Bluetooth enabled devices enter and leave radio proximity.

Most Bluetooth devices are described as 'Class 2'. These are very low power (typically 1 milliwatt - 1/1000th of a watt) and have a range of about 10 m (33 ft).Some devices - for example, some plug in'dongles' that can be added to to laptop computers - are Class 1. These have range comparable to that of Wi-Fi, ie, 100 m or 330 ft. With Bluetooth, short range is actually a

benefit, because it reduces the chance of interference between your Bluetooth devices and those belonging to other people nearby.

4.4.1. Link and Channel Management Protocols

- Control Layers
  Above the physical channel there is a layering of links and channels and associated control protocols. The hierarchy of channels and links from the physical channel upwards is physical channel, physical link, logical transport, logical link and L2CAP channel.

- Physical Links
  Within a physical channel, a physical link is formed between any two devices that transmit packets in either direction between them. In a piconet physical channel there are restrictions on which devices may form a physical link. There is a physical link between each slave and the master. Physical links are not formed directly between the slaves in a piconet.

- Logical Links
  The physical link is used as a transport for one or more logical links that support unicast synchronous, asynchronous and isochronous traffic, and broadcast traffic. Traffic on logical links is multiplexed onto the physical link by occupying slots assigned by a scheduling function in the resource manager.

- Link Manager Protocol (LMP)
  A control protocol for the baseband and physical layers is carried over logical links in addition to user data. This is the link manager protocol (LMP). Devices that are active in a piconet have a default asynchronous connection-oriented logical transport that is used to transport the LMP protocol signaling. For historical reasons this is known as the ACL logical transport. The default ACL logical transport is the one that is created

whenever a device joins a piconet. Additional logical transports may be created to transport synchronous data streams when this is required. The link manager function uses LMP to control the operation of devices in the piconet and provide services to manage the lower architectural layers (radio layer and baseband layer). The LMP protocol is only carried on the default ACL logical transport and the default broadcast logical transport.

- Logical Link Control And Adaptation Protocol (L2CAP)
  Above the baseband layer the L2CAP layer provides a channel-based abstraction to applications and services. It carries out segmentation and reassembly of application data and multiplexing and de-multiplexing of multiple channels over a shared logical link. L2CAP has a protocol control channel that is carried over the default ACL logical transport. Application data submitted to the L2CAP protocol may be carried on any logical link that supports the L2CAP protocol.

4.4.2. Core Architectural Blocks

This section describes the function and responsibility of each of the blocks.

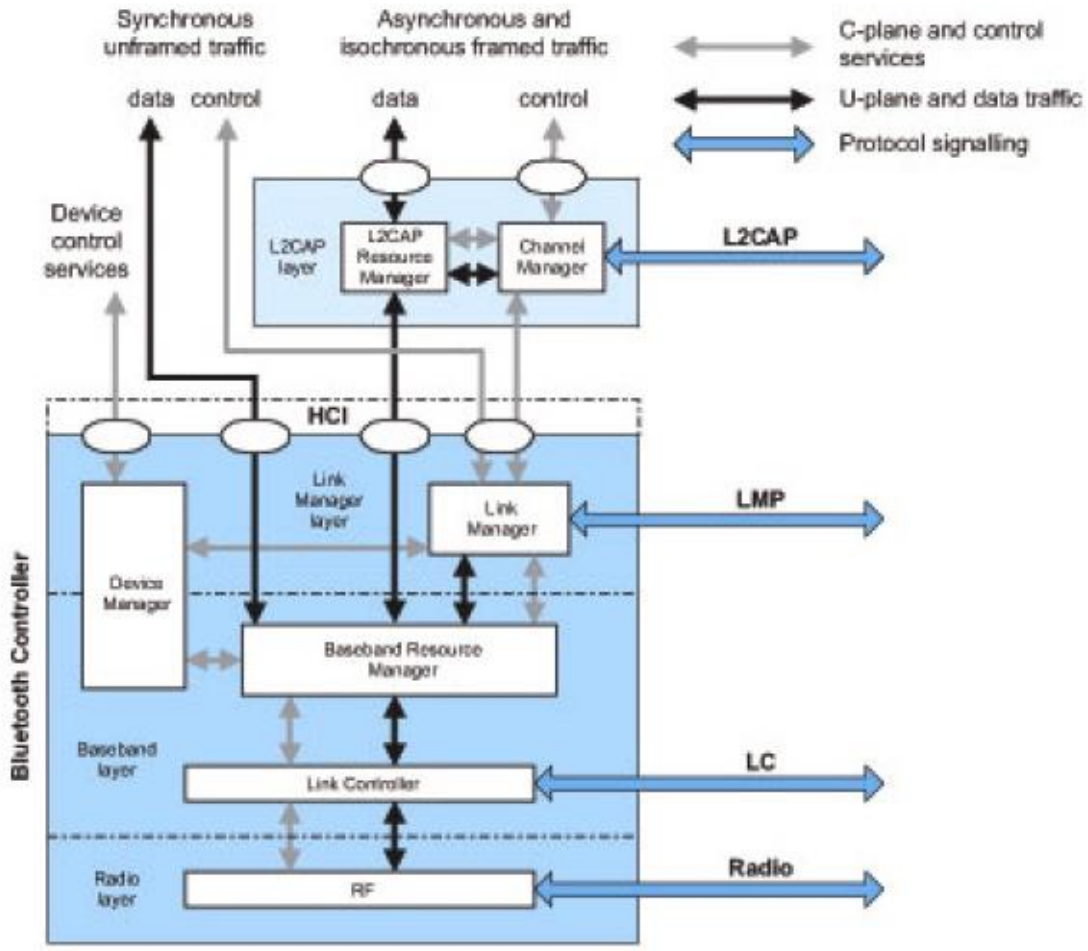


Figure 4.11: Bluetooth Architecture.

Several new features are introduced in Bluetooth Core Specification 2.1 +EDR. The major areas of improvement are:

- Erroneous Data Reporting.
- Encryption Pause and Resume.
- Extended Inquiry Response.
- Link Supervision Timeout Changed Event.
- Non-Automatically-Flushable Packet Boundary Flag.
- Secure Simple Pairing.
- Sniff Subrating.
- Security Mode 4.

4.4.3. Erroneous Data Reporting

The Erroneous Data Reporting configuration parameter shall be used for SCO and eSCO connections only. This parameter determines if the Controller is required to provide data to the Host for every (e)SCO interval, with the Packet Status Flag in HCI Synchronous Data Packets set according to demand.

- Encryption Pause and Resume
  The Encryption Key Refresh Complete event is used to indicate to the Host that the encryption key was refreshed on the given Connection Handle any time encryption is paused and then resumed. The Controller shall send this event when the encryption key has been refreshed due to encryption being started or resumed If the Encryption Key Refresh Complete event was generated due to an encryption pause and resume operation embedded within a change connection link key procedure, the Encryption Key Refresh Complete event shall be sent prior to the Change Connection Link Key Complete event.If the Encryption Key Refresh Complete event was generated due to an encryption pause and resume operation embedded within a role switch procedure,the Encryption Key Refresh Complete event shall be sent prior to the Role Change event.

- Extended Inquiry Response
  The Extended Inquiry Response provides information about the local device in response to inquiry from remote devices. The configuration parameter has two parts, a significant part followed by a non-significant part. The non-significant part contains only zero octets. The length of the extended inquiry response configuration parameter is 240 octets.

- Link Supervision Timeout Changed Event
  The Link Supervision Timeout Changed event is used to notify the slave's Host when the Link Supervision Timeout parameter is changed in the

slave Controller. This event shall only be sent to the Host by the slave controller upon receiving an LMP supervision timeout PDU from the master.

- Sniff Subrating
  Sniff subrating provides a mechanism for further reducing the active duty cycle ,thereby enhancing the power-saving capability of sniff mode. Sniff subrating allows a Host to create a guaranteed access-like connection by specifying maximum transmit and receive latencies. This allows the basebands to optimize the low power performance without having to exit and re-enter sniff mode using Link Manager commands.

- Security mode 4 (service level enforced security)
  A Bluetooth device in security mode 4 shall classify the security requirements of its services using at least the following attributes (in order of decreasing security):
  Authenticated link key required.
  Unauthenticated link key required.
  No security required.
  An authenticated link key is a link key where either the numeric comparison ,out-ofband or passkey entry simple pairing association models were used. An authenticated link key has protection against man-in-the-middle (MITM) attacks. To ensure that an authenticated link key is created during the Simple Pairing procedure, the Authentication Requirements parameter should be set to one of the MITM Protection Required options. An unauthenticated link key is a link key where the just works Secure Simple Pairing association model was used. An unauthenticated link key does not have protection against MITM attacks.

  When both devices support Secure Simple Pairing, GAP shall default to requiring an unauthenticated link key and enabling encryption. A profile or protocol may define services that require more security (e.g. an

authenticated link key)or no security. To allow an unauthenticated link key to be created during the Simple Pairing procedure, the Authentication Requirements parameter may be set to one of the MITM Protection Not Required options. When the device is in Bondable Mode, it shall enable Secure Simple Pairing mode prior to entering Connectable Mode or establishing a link.

A Bluetooth device in security mode 4 shall respond to authentication requests during link establishment when the remote device is in security mode 3 for backwards compatibility reasons. A Bluetooth device in security mode 4 enforces its security requirements before it attempts to access services offered by a remote device and before it grants access to services it offers to remote devices.

5. Sensors

5.1. IMU (Inertial Measurement Unit)

MPU6050 Sensor Introduction

MPU6050 gyroscope and accelerometer sensor consist of a digital motion sensor that performs all complex processing and computations and provides sensor data output to other MCUs over I2C communication. It has been widely adopted and become very popular in smartphones and tablets for gesture control applications.

Figure 5.1: MPU6050 Sensor.

5.1.1. Features

- Digital-output X-, Y-, and Z-Axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ±250, ±500, ±1000, and ±2000°/sec.
- Digital-output triple-axis accelerometer with a programmable full-scale range of ±2g, ±4g, ±8g and ±16g.
- It has a built-in I2C sensor bus which is used to provide gyroscope, accelerometer, and temperature sensor data microcontroller.
- This sensor module has onboard pull-up resistors.so we do not need to connect external pull resistors which are a requirement for the I2C bus interface.
- User Programmable gyroscope and accelerometer with the help of 16-bit analog to digital converter.
- 1024 Byte FIFO buffer to provide data to the connected microcontroller in high speed and enters the low power mode afterwards.
- Built-in temperature sensor.

5.1.2. Hardware

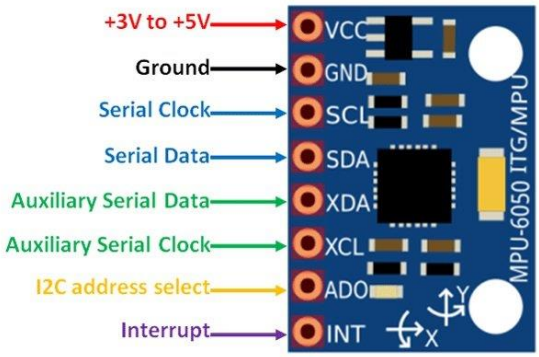


Figure 5.2: MPU6050 Pinout.

The IMU comes in the form of modules that interface with the microcontroller by I2C protocol.

Hardware disadvantages:
Its readings are too noisy. Which mean we have to apply alman filter or complementary filter to get better results from it.
The rotational acceleration is also used to improve the readings besides alerting from high rotational acceleration around axes.

5.1.3. Euler's angles

The IMU returns 6 values that can be used to determine Euler's angles.

Euler's angles are a method used to determine the orientation of the vehicle where:

1) Angle Yaw: The angle of rotation around z-axis from the starting position. So, it is the most used since we assume car is moving on a 2D plane.
2) Angle pitch: The angle of rotation around y-axis from the starting position. Used when car or vehicle is ascending or descending an inclined plane.
3) Angle roll: The angle of rotation around y-axis from the starting position. So, it is rarely used. Because it means that the vehicle is flipping on one of its sides!

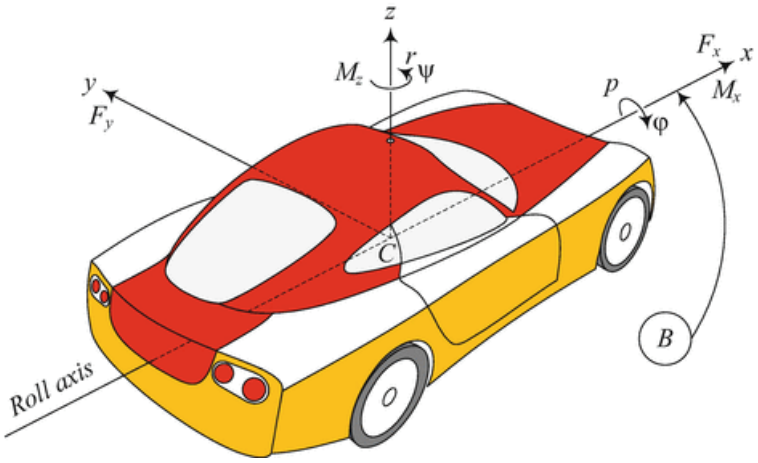


Figure 5.3: The returns 6 values of the IMU.

However unfortunately, the IMU does not return the euler's angles directly.

The values given by the IMU:

1) GyroZ: it is the gyroscopic or angular velocity around Z- axis.
2) GyroY: it is the gyroscopic or angular velocity around Y- axis.
3) GyroX: it is the gyroscopic or angular velocity around X- axis.
4) AccZ: it is the gravitational acceleration around Z-axis.
5) AccY: it is the gravitational acceleration around Y-axis.
6) AccX: it is the gravitational acceleration around X-axis.

So, to find Euler's angles, we have to integrate the all the Gyro values given by the IMU.

5.1.4. Why IMU?

The Inertial Measuring Unit (IMU) is a sensor embedding a gyroscope, accelerometer, and a temperature sensor in one package. So, it can measure the vehicle's orientation accurately.

Determining the orientation of our vehicle helps us to:

1) **State *Estimation with Wheel Encoders, GPS and Compass by means of sensor fusion technique:***
   Most of GPS sensors are not that accurate to depend on only on a long car journey. In fact, their accuracies are too poor to the degree

that the most common type in the market is of error up to 6m!
Wheel Encoders only can determine longitudinal distance accurately, but they give no information about direction or orientation. Wheel encoders with IMU can give accurate estimation of position. However, they give the position in relative or inertial coordinates not in a global reference frame as GPS sensors do. IMU's gyroscopes also are not very accurate, and the integration of their readings causes error accumulation with time. So, we also used an electronic compass to correct IMU readings. In other words, we need to fuse data of GPS, wheel encoder, GPS, and magnetometer (compass).

2) **PID lateral control of vehicle position:**
In order to drive our car autonomously according to data obtained from V2X communication server, we will use pure pursuit or Stanley controller. Both of these algorithms accept the current position (orientation) and the desired position (orientation) to output the optimum steering angle to get to the desired location quickly and smoothly where we need the IMU to send the proper input to the PID controller.
3) **Lane keeping and straight driving or PID longitudinal control:**
Consider the driver accurately positioned the car on the lane of the road. Isn't keeping car from drift and keeping a perfect velocity too tiresome for highways and long roads?
In our project we solved this problem by V2V communication with the nearest car ahead of the driver on the same lane. Where the car ahead of us is continuously sending us its orientation which is the same as the lane geometric graph. So the car can follow it autonomously by means of PID longitudinal control. However, for a long road, all mechanical cars drift from the straight lane without continuous correcting this drift by the driver. This small drift angle can be computed by the IMU and corrected by the PID controller.

5.1.5. Software

C Code implementation

First, we define the names of MPU6050 addresses using “#define” preprocessing directive in IMU_priv.h We will use these register addresses to initialize sensor and to read gyro and acceleration outputs from their respective registers by using their addresses.

MPU6050 Initialization Function

“void IMU_voidMPU6050Init(I2C_t)” function initializes the sensor by writing values declared in “commands [6]” array to respective registers of MPU6050 over the I2C bus by “I2C_charWrite()”

```
void IMU_voidMPU6050Init(I2C_t I2C)
{
    char commands[6] = {0x07, 0x01, 0x00, 0x00, 0x18, 0x01};
    I2CBUS = I2C;
    I2C_voidInit(I2C, MASTER, 100000);
    I2C_charWrite(I2C,0x68,SMPLRT_DIV, 0x07);
    I2C_charWrite(I2C,0x68,PWR_MGMT_1,  0x01);
    I2C_charWrite(I2C,0x68,CONFIG, 0x00);
    I2C_charWrite(I2C,0x68,ACCEL_CONFIG,0x00);
    I2C_charWrite(I2C,0x68,GYRO_CONFIG,0x18);
    I2C_charWrite(I2C,0x68,INT_ENABLE, 0x01);
}
```

MPU6050 Calculate Angle Function

“void IMU_CalculateAngle(void)” function declare Gyro variables and calculate Eular’s angles from Gyro which has been read from “short IMU_u16GetGyroZ(void)” &“short IMU_u16GetGyroX(void)” & “short IMU_u16GetGyroY(void)” functions .

```
void IMU_CalculateAngle(void)
{
    short GyroX = IMU_u16GetGyroZ();
    short GyroY = IMU_u16GetGyroZ();
    short GyroZ = IMU_u16GetGyroZ();

    static float angleYAW = 0;
    static float anglePITCH = 0;
    static float angleROLL = 0;

    angleYAW += GyroZ*135.0/(400000.0);
    anglePITCH += GyroZ*135.0/(400000.0);
    angleROLL += GyroZ*135.0/(400000.0);

    YAW = (short) angleYAW;
    PITCH = (short) anglePITCH;
    ROLL = (short) angleROLL;
}



short IMU_u16GetGyroZ(void)
{




    I2C_charRead(I2CBUS, 0x68, ACCEL_XOUT_H, 14, sensordata);
    short GyroZ = (short) ( (sensordata[12] << 8 ) |sensordata[13] );
    GyroZ -= gyro_z_cal;
    return GyroZ;

}
```

MPU6050 Gyro Calibrate Function

```
void IMU_Gyro_Calibrate(void){
    int cal_int;
    short gyro_x = 0;
    short gyro_y = 0;
    short gyro_z = 0;

    for (cal_int = 0; cal_int < 2000 ; cal_int ++){
              I2C_charRead(I2CBUS, 0x68, ACCEL_XOUT_H, 14, sensordata);
              gyro_x = (short) ( (sensordata[8] << 8 ) |sensordata[9] );
              gyro_y = (short) ( (sensordata[10] << 8 ) |sensordata[11] );
              gyro_z = (short) ( (sensordata[12] << 8 ) |sensordata[13] );
              gyro_x_cal += gyro_x;
              gyro_y_cal += gyro_y;
              gyro_z_cal += gyro_z;
              Delay(3);
             }
                gyro_x_cal /= 2000;
                gyro_y_cal /= 2000;
                gyro_z_cal /= 2000;


                MSTK_voidInit();
                MSTK_voidCreateTask(0, 1, IMU_CalculateAngle);
                MSTK_voidStartScheduler();
```

We use this function in testing process only because it takes few seconds to calibrate the readings.

MPU6050 Get Acceleration Function

```
short IMU_u16GetACCZ(void)
{
    short accZ;
    I2C_charRead(I2CBUS, 0x68, ACCEL_XOUT_H, 14, sensordata);
    accZ = (int) ( (sensordata[4] << 8 ) |sensordata[5] );
    return accZ;
}
```

MPU6050 Get Euler's angles Function.

```
short IMU_u16GetYAW(void)
{

    return YAW;
}
short IMU_u16GetPITCH(void)
{

    return PITCH;
}
short IMU_u16GetROLL(void)
{

    return ROLL;
}
```

Errors in coding:

1) Type Casting in TI compiler.

A very serious problem we faced was that we were testing our code on Arduino board and it was completely successful. However, when we tried to mimic the Arduino code on CCS for Tiva C, it prints 0 as 65535 and sometimes it jumps suddenly from 1 to 65535 or a bit lower number. A good observation leaded us to that this number is the upper bound of short integer (16- bit unsigned variable).

So, we concluded that the problem is in casting bytes of data received from IMU into a data type. On Arduino, we simply cast them into integer. However, in Arduino compiler, the integer is 16 bits, while in TI compiler it is 32 bits.

And that leads to a problem in casting, because according to C99:

C99 6.3.1.3-p3

> Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

C99 6.3.1.3-p1

> When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.

So, we solved that by simply replacing int type casting into short because it is 16 -bit variable as Arduino.

2) Display for Debugging:

For the debugging and testing, we needed a virtual terminal to display data collected from IMU. So, we used Tera Term at the beginning, but it sometimes gives random characters and repeated or missing characters, also it does not support printing characters on different lines as columns and rows format. After some search we found that the energia IDE serial terminal is better since it supports all that.

Another problem concerning display is the function to convert integer to a string to be sent via UART. We always prefer itoa function. However, we discovered that It is not supported by TI compilers. So, we turned to alman, but we found that the stdio library needs huge stack which is difficult to guarantee in our application and it may crash due to stack overflow if we used it. Then we decided to make the function ourselves, but we found that it will enlarge and increase the complexity of code besides its successfulness for all integers is not guaranteed. Finally, we found the best function, which is the ltoa. Since it is supported by TI and does not need to include stdio.h plus it does not consume much memory or complicates the code, it is surely the best solution.

5.2. Electronic Compass (HMC5883L Or QMC5883L)

HMC5883L and QMC5883L don't have too much difference. HMC5883L is made by Honeywell. But because the production is about to be stopped, so we use a similar compass module QMC5883L instead. QMC5883L is made by a Chinese company, So QMC5883L and HMC5883L basically are the same with little bit different in their registers. Mechanical

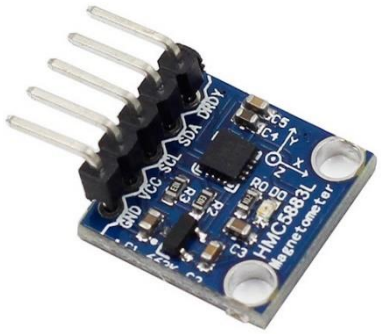


Figure 5.4: Electronic Compass (HMC5883L).

compasses measure heading by pointing to north however electronic compasses measure earth magnetic field component on each axis.
The QMC5883L is a multi-chip three-axis magnetic sensor. I2C based.

5.2.1. Features

- I2C Interface with Standard and Fast Modes.
- Wide Range Operation Voltage (2.16V To 3.6V) and Low Power Consumption (75μA).
- Max 116 Hz output rate.
- Wide Magnetic Field Range (±8 Gauss).
- High heading accuracy.
- High-Speed Interfaces for Fast Data Communications. Maximum 200Hz Data Output Rate.

5.2.2. Hardware

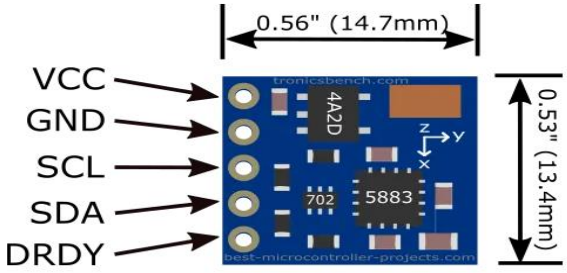


Figure 5.5: HMC5883L Pinout.

5.2.3. Why QMC5883L?

IMU's gyroscopes also are not very accurate, and the integration of their readings causes error accumulation with time. So, we also used an electronic compass to correct IMU readings.
the electronic compass measures the magnetic field strength and has a maximum output along a specific axis when the axis is aligned, and pointing to the Earth's North pole and return the heading.

$$Heading = \text{atan}\left(\frac{\text{y}}{\text{x}}\right)$$

Now the heading angle refers to the orientation around the north pole the IMU uses it as an initial value to determine the orientation of our vehicle. So, the final result that no error accumulation due to integration of noise or bias errors.

5.2.4. Magnetometer Problems

1) Electromagnetic interference noise from motors, mobile phones, …etc.
2) Magnetic materials as iron and permeant magnets (Hard Iron distortion) causes noise, this distortion is by far the most important to be eliminated and must be removed before the magnetometer can be used as a compass.
3) Soft Iron Distortion field caused by ferro-magnetic material. This distortion warps the Earth's magnetic field causing an ellipsoid magnetometer output.
4) Misalignments errors so Sensor must be fixed on a horizontal plane. In other words, pitch and roll lean causes errors in yaw.
5) Temperature dependence.

5.2.5. Magnetometer Calibration

No Distortions

In the ideal case that there are no hard or soft iron distortions present, the measurements should form a circle centered at X=0, Y=0. The radius of the circle equals the magnitude of the magnetic field.

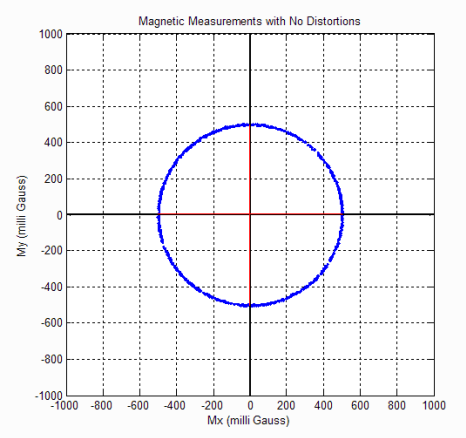


Figure 5.6: Magnetometer Calibration - No Distortions.

Hard Iron Distortions

Hard iron distortions will cause a permanent bias to be present in the outputs. Hard iron distortions will only shift the center of the circle away from the origin.

These distortions will not distort the shape of the circle.

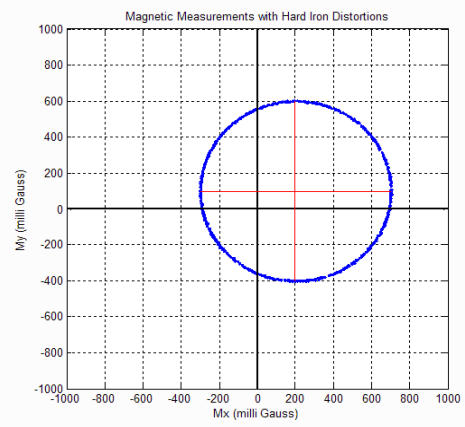


Figure 5.7: Magnetometer Calibration - Hard Iron Distortions

Hard and Soft Iron Distortions

Soft iron distortions distort and warp the existing magnetic fields. When you plot the magnetic output soft iron distortions are easy to recognize since they will distort the circular output. Soft Iron effects warp the circle into an elliptical shape. As shown in the above magnetometer output, the circle has been distorted into an ellipse and shifted from the center.

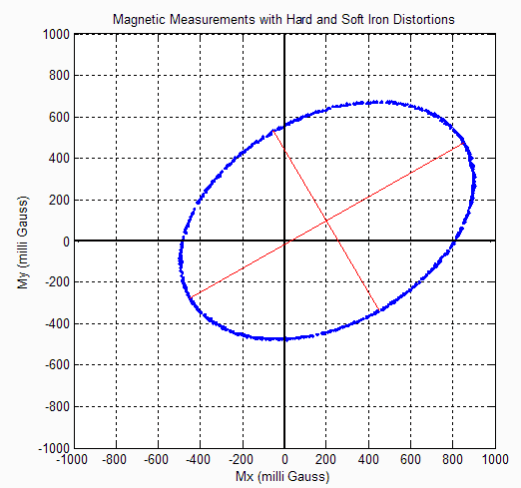


Figure 5.8: Magnetometer Calibration - Hard and Soft Iron Distortions

Calibration procedure

Calibrated measurements (3x1)

Sensor measurements (3x1)

Hard iron corrections (3x1)

$$\vec{m}_{calib} = A\left(\vec{m}_{meas} - \vec{b}\right)$$

Soft iron, scale factor, and misalignment corrections (3x3, symmetric)

So, we have 9 parameters that control the shape of the ellipse.

Now we have a problem that it is difficult to get the calibration parameters analytically using c programming because of matrices so we have two options:

1) Python codes which are complicated too.
2) Magneto Software

Figure 5.9: Magnetometer Calibration - Magneto Software.

Magneto Software calibration parameters:

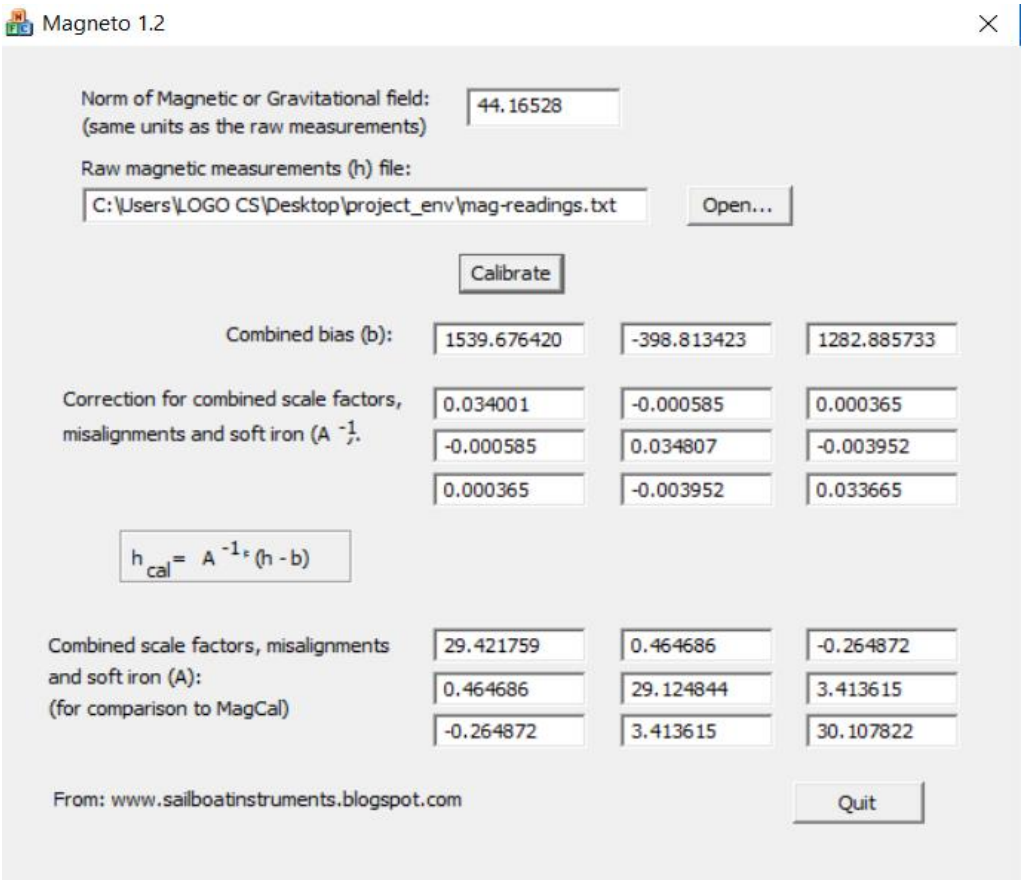


Figure 5.10: Magnetometer Calibration - Magneto Software calibration parameters.

Results after calibration will be spherical shape around zero like this 3D plot.

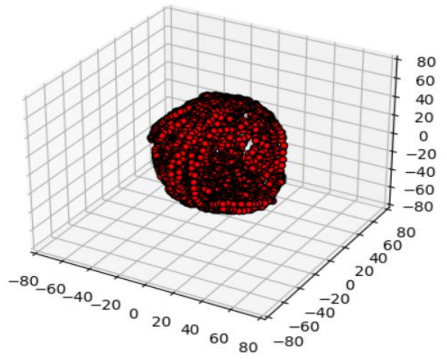


Figure 5.11: Magnetometer Calibration - Results 3D Plot.

This is MATLAB simulation show the measurements in 2D plot before and after the calibration process.

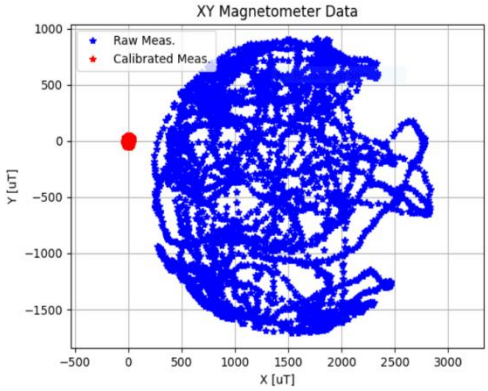


Figure 5.12: Magnetometer Calibration - Results 2D Plot (XY).

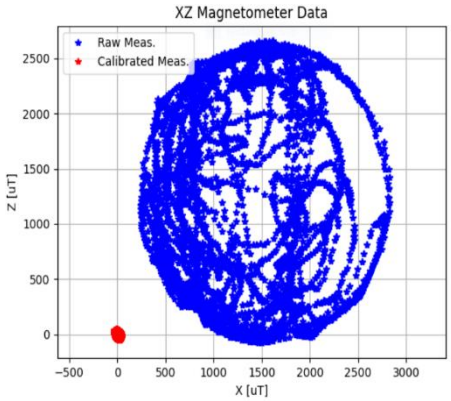


Figure 5.13: Magnetometer Calibration - Results 2D Plot (XZ).

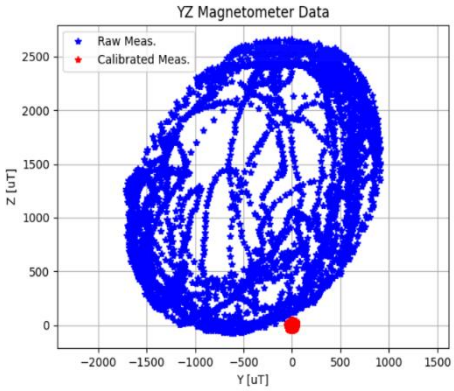


Figure 5.14: Magnetometer Calibration - Results 2D Plot (YZ).

5.2.6. Software

C Code implementation

Using I2C communication protocol, the configuration of the compass and the mode of operation are determined through accessing the registers of the compass that responsible of that (Configuration register A, Configuration register B, Mode register) By using the I2C_send function to send the address of write operation from master to slave (digital compass) to access the configuration registers ,then sending the required data through the same function and defining a pointer to reference the location of registers. - Doing the same steps to make the configuration of the 2 registers A and B and the mode register. - After that to read the data from the data output registers,by sending the address of the first data output register through I2C_send function then using the I2C_Receive function to receive the data, so by sending the address of the read operation and determining the number of data bytes to be read. - To get the heading of the compass after reading the 16-bit data (x, y and z) in 2's complement form from data output registers, the following steps are done:

```
void QMC5883_f32GetInitialHeading()
{
    short x,y,z;
    QMC5883_u16Read(&x, &y, &z);

    x = (0.029975 * (x - 1358.944534) + 0.002137 * (y + 366.800852) - 0.005795 * (z - 1175.440578));
    y = (0.002137 * (x - 1358.944534) + 0.036549 * (y + 366.800852) - 0.001469 * (z - 1175.440578));
    z = (-0.005795 * (x - 1358.944534) - 0.001469 * (y + 366.800852) + 0.031086 * (z - 1175.440578));

    x = (1.177870 * (x - 7.020898) - 0.037572 * (y - 2.888513) + 0.195462 * (z - 2.089576));
    y = (-0.037572 * (x - 7.020898) + 0.968025 * (y - 2.888513) - 0.072147 * (z - 2.089576));
    z = (0.195462 * (x - 7.020898) - 0.072147 * (y - 2.888513) + 1.084548 * (z - 2.089576));

    initial=atan2(x, y)/0.0174532925;
}
f32 QMC5883_f32GetHeading()
{
    short x,y,z;
    f32 heading;
    QMC5883_u16Read(&x, &y, &z);

    x = (0.029975 * (x - 1358.944534) + 0.002137 * (y + 366.800852) - 0.005795 * (z - 1175.440578));
    y = (0.002137 * (x - 1358.944534) + 0.036549 * (y + 366.800852) - 0.001469 * (z - 1175.440578));
    z = (-0.005795 * (x - 1358.944534) - 0.001469 * (y + 366.800852) + 0.031086 * (z - 1175.440578));

    x = (1.177870 * (x - 7.020898) - 0.037572 * (y - 2.888513) + 0.195462 * (z - 2.089576));
    y = (-0.037572 * (x - 7.020898) + 0.968025 * (y - 2.888513) - 0.072147 * (z - 2.089576));
    z = (0.195462 * (x - 7.020898) - 0.072147 * (y - 2.888513) + 1.084548 * (z - 2.089576));

    heading=(atan2(x, y)/0.0174532925);
    heading -= initial;
    if(abs(heading) > 180)
    {
        if(heading > 0)
        {
            heading -= 360;
        }
        else if(heading < 0)
        {
            heading += 360;
        }
    }
    heading *= -1;
    return heading;
}
```

5.3. Electronic Compass (HMC5883L Or QMC5883L)

You can use a Kalman filter in any place where you have uncertain information about some dynamic system, and you can make an educated guess about what the system is going to do next. Even if messy reality comes along and interferes with the clean motion you guessed about, the Kalman filter will often do a very good job of figuring out what actually happened. And it can take advantage of correlations between crazy phenomena that you maybe wouldn't have thought to exploit!
Kalman filters are ideal for systems which are continuously changing. They have the advantage that they are light on memory (they don't need to keep any history other than the previous state), and they are very fast, making them well suited for real time problems and embedded systems.

5.3.1. Why Kalman Filter?

Let's make a toy example: You've built a little robot that can wander around in the woods, and the robot needs to know exactly where it is so that it can navigate.

We'll say our robot has a state $\vec{x_k}$, which is just a position and a velocity:

$$\vec{x_k} = (\vec{p}, \vec{v})$$

Note that the state is just a list of numbers about the underlying configuration of your system; it could be anything. In our example it's position and velocity, but it could be data about the amount of fluid in a tank, the temperature of a car engine, the position of a user's finger on a touchpad, or any number of things you need to keep track of.
Our robot also has a GPS sensor, which is accurate to about 10 meters, which is good, but it needs to know its location more precisely than 10 meters. There are lots of gullies and cliffs in these woods, and if the robot is wrong by more than a few feet, it could fall off a cliff. So, GPS by itself is not good enough.
We might also know something about how the robot moves: It knows the commands sent to the wheel motors, and it knows that if it's headed in one direction and nothing interferes, at the next instant it will likely be further along that same direction. But of course, it doesn't know everything about

its motion: It might be buffeted by the wind, the wheels might slip a little bit, or roll over bumpy terrain; so, the amount the wheels have turned might not exactly represent how far the robot has actually traveled, and the prediction won't be perfect.
The GPS sensor tells us something about the state, but only indirectly, and with some uncertainty or inaccuracy. Our prediction tells us something about how the robot is moving, but only indirectly, and with some uncertainty or inaccuracy.
But if we use all the information available to us, can we get a better answer than either estimate would give us by itself? Of course, the answer is yes, and that's what a Kalman filter is for.

5.3.2. How a Kalman filter sees your problem?

Let's look at the landscape we're trying to interpret. We'll continue with a simple state having only position and velocity.

$$\vec{x} = \begin{bmatrix} p \\ v \end{bmatrix}$$

We don't know what the *actual* position and velocity are; there are a whole range of possible combinations of position and velocity that might be true, but some of them are more likely than others:

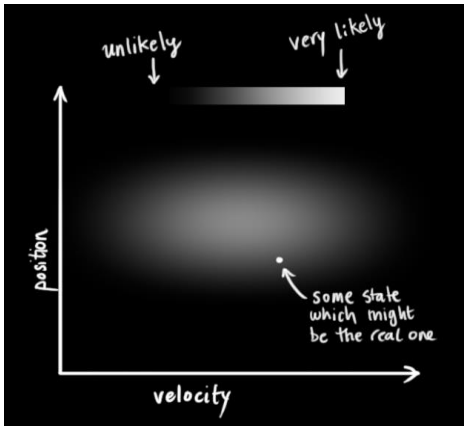


Figure 5.15: Kalman Filter - Plot 1.

The Kalman filter assumes that both variables (position and velocity, in our case) are random, and *Gaussian distributed.* Each variable has a mean value μ, which is the center of the random distribution (and its most likely state), and a variance σ2, which is the uncertainty:

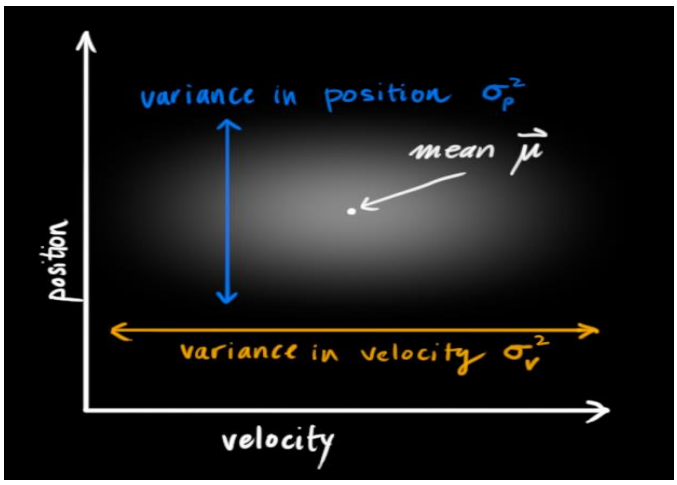


Figure 5.16: Kalman Filter - Plot 2.

In the above picture, position and velocity are uncorrelated, which means that the state of one variable tells you nothing about what the other might be.

The example below shows something more interesting: Position and velocity are correlated. The likelihood of observing a particular position depends on what velocity you have:

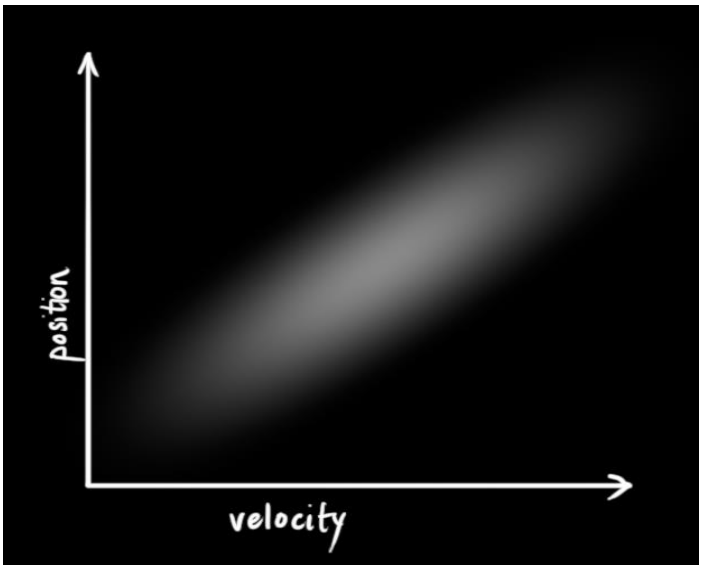


Figure 5.17: Kalman Filter - Plot 3.

This kind of situation might arise if, for example, we are estimating a new position based on an old one. If our velocity was high, we probably moved farther, so our position will be more distant. If we're moving slowly, we didn't get as far.

This kind of relationship is really important to keep track of, because it gives us more information: One measurement tells us something about what the others could be. And that's the goal of the Kalman filter, we want to squeeze as much information from our uncertain measurements as we possibly can!

This correlation is captured by something called a covariance matrix. In short, each element of the matrix Σij is the degree of correlation between the *ith* state variable and the *jth* state variable. (You might be able to guess that the covariance matrix is symmetric, which means that it doesn't matter if you swap *i* and *j*). Covariance matrices are often labelled "Σ", so we call their elements "Σij".

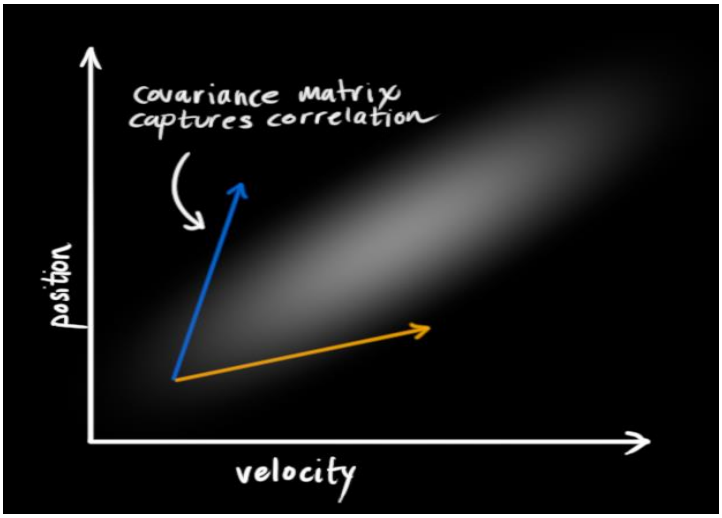


Figure 5.18: Kalman Filter - Plot 4.

5.3.3. Describing the problem with matrices

We're modeling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time k: We'll call our best estimate x^k (the mean, elsewhere named μ), and its covariance matrix Pk.

$$\hat{\mathbf{x}}_k = \begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix} \qquad (1)$$

$$\mathbf{P}_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix}$$

(Of course, we are using only position and velocity here, but it's useful to remember that the state can contain any number of variables, and represent anything you want).

Next, we need some way to look at the current state (at time k-1) and **predict the next state** at time k. Remember, we don't know which

state is the “real” one, but our prediction function doesn’t care. It just works on *all of them*, and gives us a new distribution:

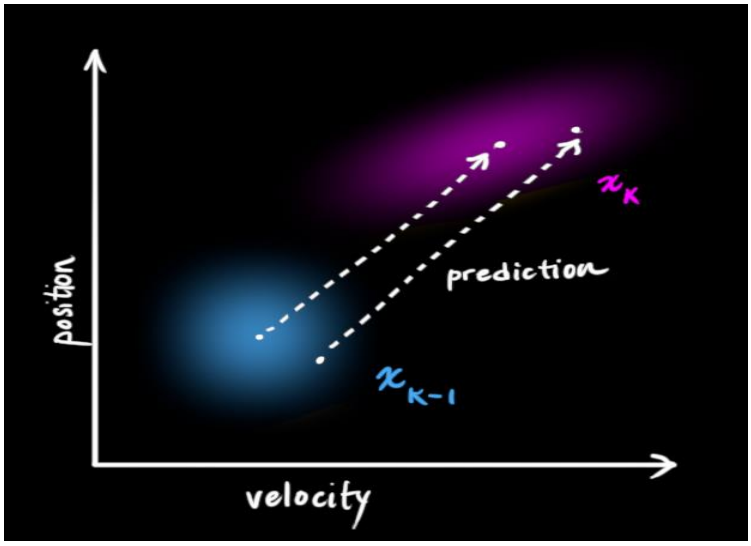


Figure 5.19: Kalman Filter - Plot 5.

We can represent this prediction step with a matrix, Fk:

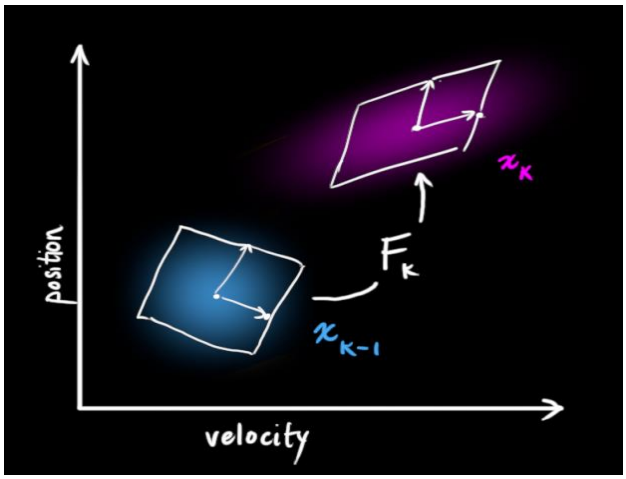


Figure 5.20: Kalman Filter - Plot 6.

It takes *every point* in our original estimate and moves it to a new predicted location, which is where the system would move if that original estimate was the right one.

Let’s apply this. How would we use a matrix to predict the position and velocity at the next moment in the future? We’ll use a really basic kinematic formula:

$$p_k = p_{k-1} + \Delta t v_{k-1}$$
$$v_k = v_{k-1}$$

In other words:

$$\hat{\mathbf{x}}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{\mathbf{x}}_{k-1} \quad (2)$$
$$= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} \quad (3)$$

:

We now have a prediction matrix which gives us our next state, but we still don't know how to update the covariance matrix.

This is where we need another formula. If we multiply every point in a distribution by a matrix A, then what happens to its covariance matrix Σ?

Well, it's easy. I'll just give you the identity:

$$Cov(x) = \Sigma$$
$$Cov(\mathbf{A}x) = \mathbf{A}\Sigma\mathbf{A}^T \quad (4)$$

So combining (4) with equation (3):

$$\hat{\mathbf{x}}_k = \mathbf{F}_k \hat{\mathbf{x}}_{k-1}$$
$$\mathbf{P}_k = \mathbf{F}_\mathbf{k} \mathbf{P}_{k-1} \mathbf{F}_k^T \quad (5)$$

5.3.4. External influence

Everything is fine if the state evolves based on its own properties. Everything is *still* fine if the state evolves based on external forces, so long as we know what those external forces are.

But what about forces that we *don't* know about? If we're tracking a quadcopter, for example, it could be buffeted around by wind. If we're tracking a wheeled robot, the wheels could slip, or bumps on the ground could slow it down. We can't keep track of these things, and if any of this happens, our prediction could be off because we didn't account for those extra forces.

We can model the uncertainty associated with the "world" (i.e., things we aren't keeping track of) by adding some new uncertainty after every prediction step:

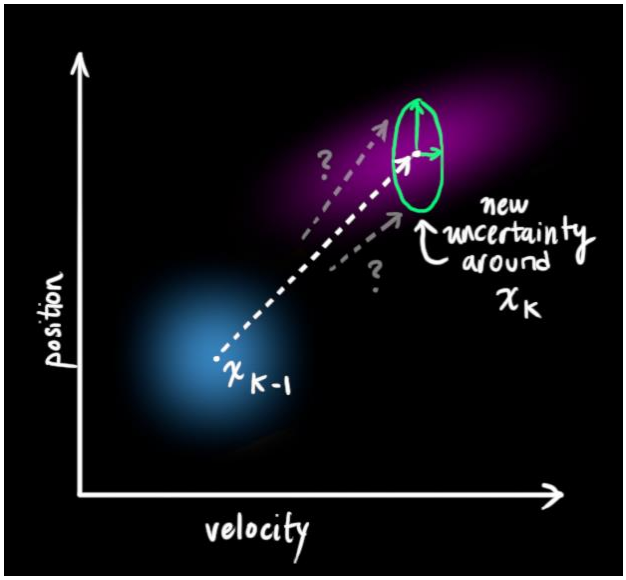


Figure 5.21: Kalman Filter - Plot 7.

Every state in our original estimate could have moved to a *range* of states. Because we like Gaussian blobs so much, we'll say that each point in x^k−1 is moved to somewhere inside a Gaussian blob with covariance Qk. Another way to say this is that we are treating the untracked influences as noise with covariance Qk.

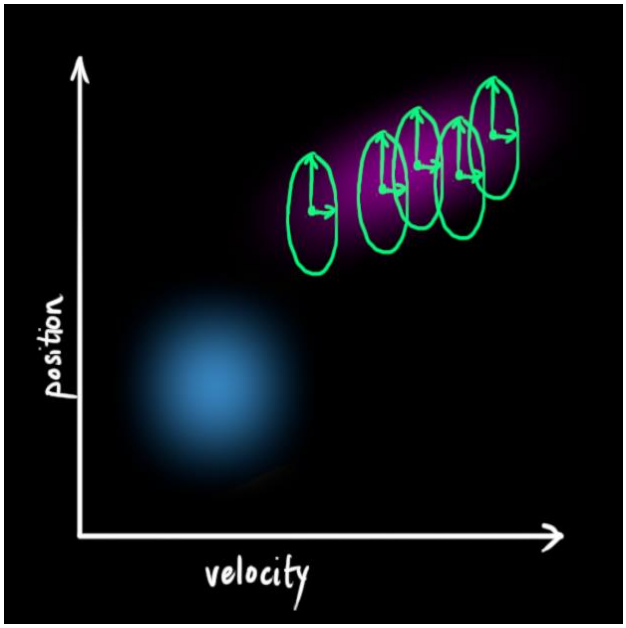


Figure 5.22: Kalman Filter - Plot 8.

This produces a new Gaussian blob, with a different covariance (but the same mean):

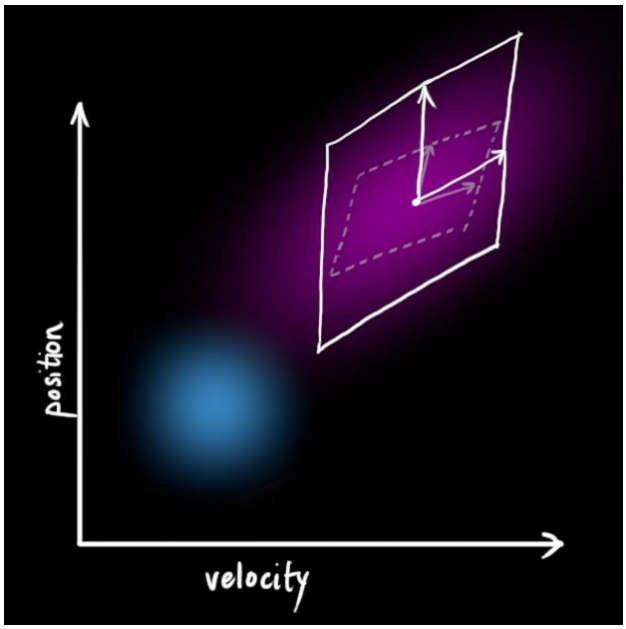


Figure 5.23: Kalman Filter - Plot 9.

We get the expanded covariance by simply adding Qk, giving our complete expression for the prediction step:

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k \vec{u}_k$$
$$P_k = F_k P_{k-1} F_k^T + Q_k \qquad (7)$$

In other words, the new best estimate is a prediction made from previous best estimate, plus a correction for known external influences.

And the new uncertainty is predicted from the old uncertainty, with some additional uncertainty from the environment**.**

All right, so that's easy enough. We have a fuzzy estimate of where our system might be, given by x^k and Pk. What happens when we get some data from our sensors?

Refining the estimate with measurements

We might have several sensors which give us information about the state of our system. For the time being it doesn't matter what they measure; perhaps one reads position and the other reads velocity. Each sensor tells us something indirect about the state— in other words, the sensors operate on a state and produce a set of readings.

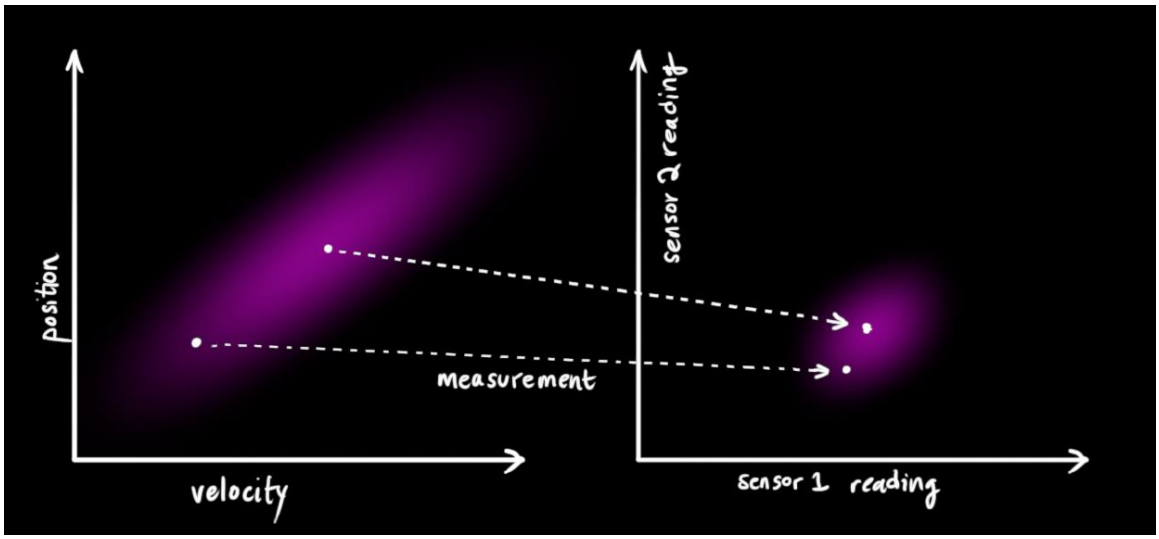


Figure 5.24: Kalman Filter - Plot 10.

Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, Hk.

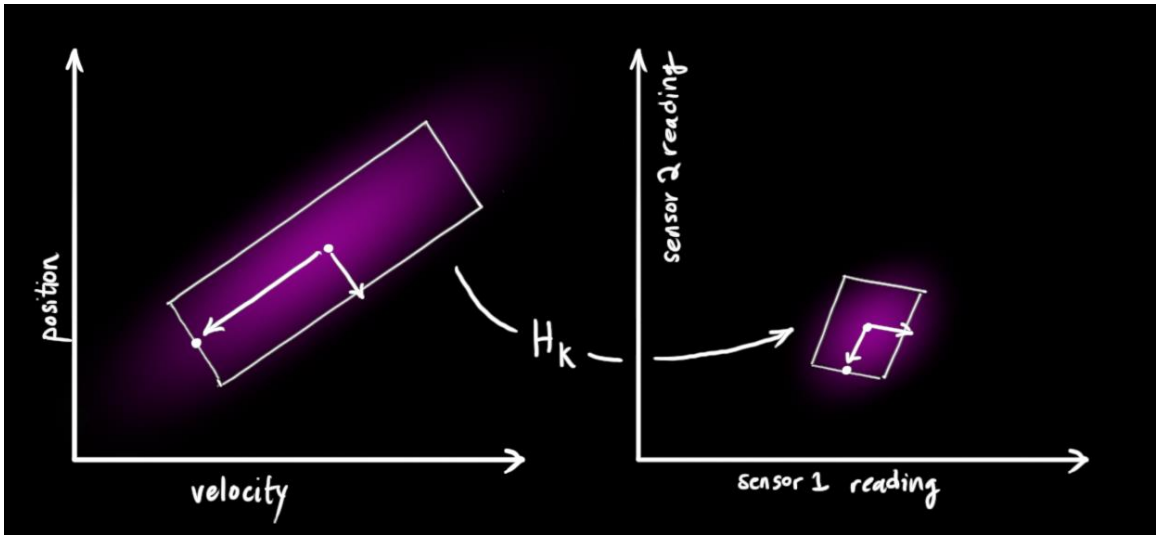


Figure 5.25: Kalman Filter - Plot 11.

We can figure out the distribution of sensor readings we'd expect to see in the usual way:

$$\begin{aligned} \vec{\mu}_{\text{expected}} &= \mathbf{H}_k \hat{\mathbf{x}}_k \\ \mathbf{\Sigma}_{\text{expected}} &= \mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T \end{aligned} \tag{8}$$

One thing that Kalman filters are great for is dealing with *sensor noise*. In other words, our sensors are at least somewhat unreliable, and every state in our original estimate might result in a *range* of sensor readings.

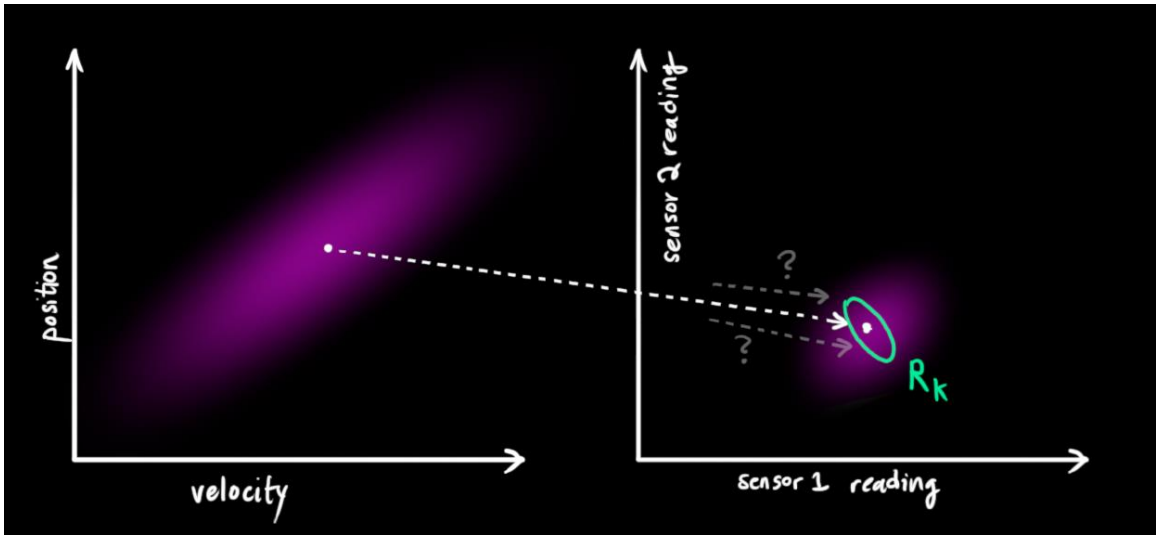


Figure 5.26: Kalman Filter - Plot 12.

From each reading we observe, we might guess that our system was in a particular state. But because there is uncertainty, some states are more likely than others to have produced.

the reading we saw:

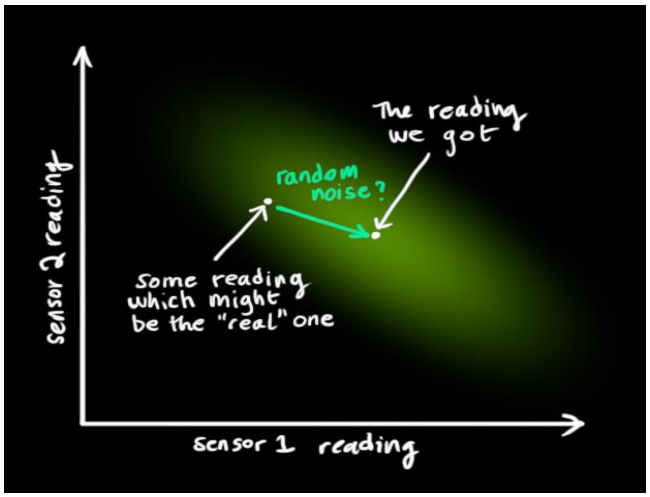


Figure 5.27: Kalman Filter - Plot 13.

We'll call the covariance of this uncertainty (i.e., of the sensor noise) Rk. The distribution has a mean equal to the reading we observed, which we'll call $\overrightarrow{Zk}$.

So now we have two Gaussian blobs: One surrounding the mean of our transformed prediction, and one surrounding the actual sensor reading we got.

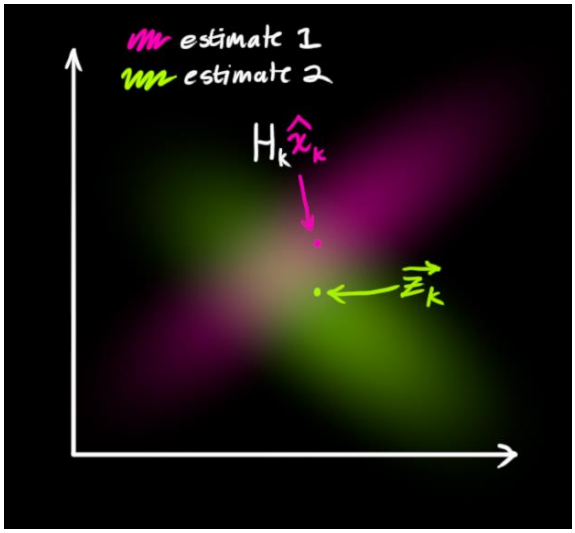


Figure 5.28: Kalman Filter - Plot 14.

We must try to reconcile our guess about the readings we'd see based on the predicted state (pink) with a *different* guess based on our sensor readings (green) that we actually observed.

So, what's our new most likely state? For any possible reading (z1,z2), we have two associated probabilities: (1) The probability that our sensor reading vector) is a (mis-)measurement of (z1,z2), and (2) the probability that our previous estimate thinks (z1,z2) is the reading we should see.

If we have two probabilities and we want to know the chance that *both* are true, we just multiply them together. So, we take the two Gaussian blobs and multiply them:

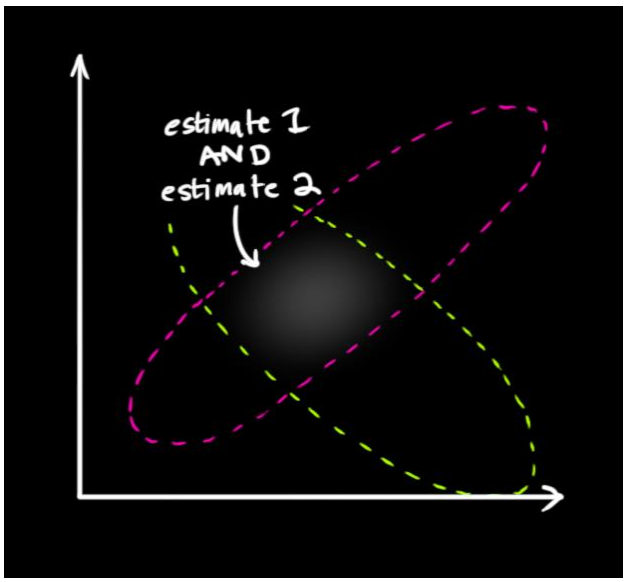


Figure 5.29: Kalman Filter - Plot 15.

What we're left with is the overlap, the region where *both* blobs are bright/likely. And it's a lot more precise than either of our previous estimates. The mean of this distribution is the configuration for which both estimates are most likely, and is therefore the best guess of the true configuration given all the information we have.

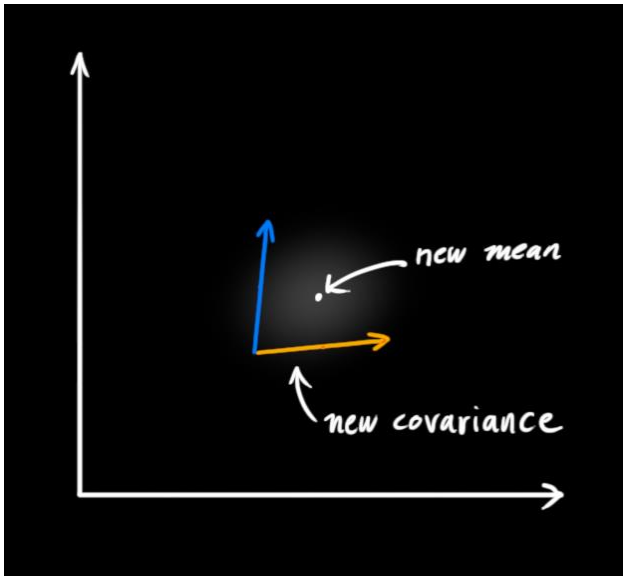


Figure 5.30: Kalman Filter - Plot 16.

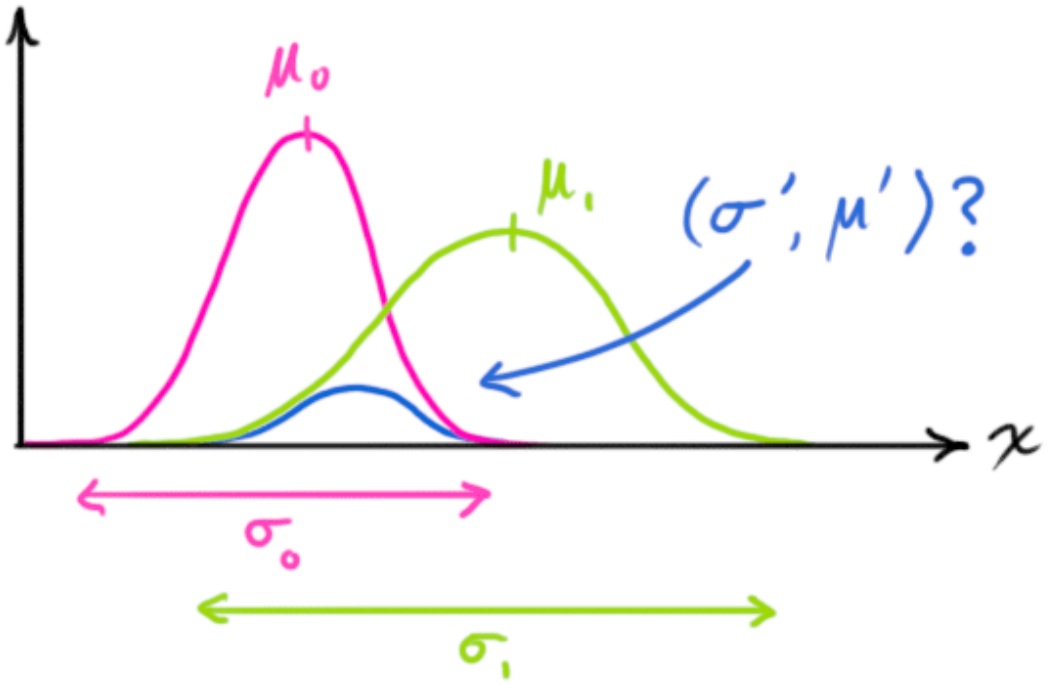
As it turns out, when you multiply two Gaussian blobs with separate means and covariance matrices, you get a *new* Gaussian blob with its own mean and covariance matrix! Maybe you can see where this is going: There's got to be a formula to get those new parameters from the old ones!

5.3.5. Combining Gaussians

Let's find that formula. It's easiest to look at this first in **one dimension**. A 1D Gaussian bell curve with variance σ2 and mean μ is defined as:

$$N(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (9)$$

We want to know what happens when you multiply two Gaussian curves together. The blue curve below represents the (unnormalized) intersection of the two Gaussian populations:



$$\mathcal{N}(x,\mu_0,\sigma_0)\cdot\mathcal{N}(x,\mu_1,\sigma_1)\stackrel{?}{=}\mathcal{N}(x,\mu',\sigma') \qquad (10)$$

You can substitute equation (9) into equation (10) and do some algebra (being careful to renormalize, so that the total probability is 1) to obtain:

$$\mu' = \mu_0 + \frac{\sigma_0^2(\mu_1-\mu_0)}{\sigma_0^2+\sigma_1^2}$$
$$\sigma'^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2+\sigma_1^2} \qquad (11)$$

We can simplify by factoring out a little piece and calling it k:

$$\mathbf{k} = \frac{\sigma_0^2}{\sigma_0^2+\sigma_1^2} \qquad (12)$$

$$\mu' = \mu_0 + \mathbf{k}(\mu_1-\mu_0)$$
$$\sigma'^2 = \sigma_0^2 - \mathbf{k}\sigma_0^2 \qquad (13)$$

Take note of how you can take your previous estimate and add something to make a new estimate. And look at how simple that formula is!

But what about a matrix version? Well, let's just re-write equations (12) and (13) in matrix form. If Σ is the covariance matrix of a Gaussian blob, and $\vec{\mu}$ its mean along each axis, then:

$$\mathbf{K} = \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \tag{14}$$

$$\begin{aligned} \vec{\mu}' &= \vec{\mu_0} + \mathbf{K}(\vec{\mu_1} - \vec{\mu_0}) \\ \Sigma' &= \Sigma_0 - \mathbf{K}\Sigma_0 \end{aligned} \tag{15}$$

K is a matrix called the Kalman gain, and we'll use it in just a moment.

Easy! We're almost finished!

Putting it all together

We have two distributions: The predicted measurement with $(\mu_0, \Sigma_0) = (\mathbf{H}_k\hat{\mathbf{x}}_k, \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T)$, and the observed measurement with $(\mu_1, \Sigma_1) = (\vec{\mathbf{z}_k}, \mathbf{R}_k)$. We can just plug these into equation (15) to find their overlap:

$$\begin{aligned} \mathbf{H}_k\hat{\mathbf{x}}'_k &= \mathbf{H}_k\hat{\mathbf{x}}_k &&+\mathbf{K}(\vec{\mathbf{z}_k} - \mathbf{H}_k\hat{\mathbf{x}}_k) \\ \mathbf{H}_k\mathbf{P}'_k\mathbf{H}_k^T &= \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T &&-\mathbf{K}\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T \end{aligned} \tag{16}$$

And from (14), the Kalman gain is:

$$\mathbf{K} = \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \tag{17}$$

We can knock an $\mathbf{H}_k$ off the front of every term in (16) and (17) (note that one is hiding inside $\mathbf{K}$), and an $\mathbf{H}_k^T$ off the end of all terms in the equation for $\mathbf{P}'_k$.

$$\begin{aligned} \hat{\mathbf{x}}'_k &= \hat{\mathbf{x}}_k + \mathbf{K}'(\vec{\mathbf{z}_k} - \mathbf{H}_k\hat{\mathbf{x}}_k) \\ \mathbf{P}'_k &= \mathbf{P}_k - \mathbf{K}'\mathbf{H}_k\mathbf{P}_k \end{aligned} \tag{18}$$

$$\mathbf{K}' = \mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \tag{19}$$

giving us the complete equations for the update step.

And that's it! $\hat{\mathbf{x}}'_k$ is our new best estimate, and we can go on and feed it (along with $\mathbf{P}'_k$) back into another round of **predict** or **update** as many times as we like.

5.3.6. Software

C Code implementation

After finishing the Kalman filter driver for the heading angle with the fusion of gyroscope and compass, we found that we need another two filters for the rest of euler's angles (Pitch and Roll) , another three filters for the position vectors (x, y and z) and another three filters for velocity vectors (x_dot, y_dot and z_dot). Wait This means we may need another EIGHT Kalman filter drivers!. Those drivers will be very similar to the heading angle kalman filter with just very small differences in the tunable constants of errors. So, we needed another approach to make a universal Kalman filter driver rather than making a driver for each quantity or complicating the driver of the heading angle to make it general.

The approach we thought of was the OOP driver. In other words, we thought of making use of the reusability property of OOP programming languages as C++. So we just need to make a C++ driver containing a class called FilteredAngle or FilteredQuantity. Then if we need to filter another quantity, we will simply make an object from it with different parameters for every quantity.

Structure of the Code:

- The header file of the Class

```
17  class FilteredAngle
18  {
19  private:
20
21      f32 P[2][2] = {{0.0, 0.0}, {0.0, 0.0}};
22      f32 R = 200;
23      f32 sigma1 = 0.001f, sigma2 = 0.003f;
24      f32 Corrected_Angle = 0.0f;
25      f32 Corrected_Rate = 0.0f;
26      f32 bias = 0.0f;
27      f32 deltaT;
28      u8 ID;
29
30      f32(*MeasuringSensor_function)();
31      short(*PredictionSensor_function)();
```

The private section of the class contains the fields specifying each quantity needed to be filtered:

1) The Uncertainty Matrix P (2x2): Because all the state variables we need to filter are of size (2x1).
2) The variance of measuring sensor R
3) The covariance of the first and the second quantity sigma1 and sigma2.
4) The output of the filter: Corrected_Angle.
5) The corrected rate of change of the quantity: Corrected_Angle.
6) The bias of the predicting sensor: bias.
7) The time step of calculation of Kalman Filter: deltaT.
8) The ID of the input Quantity (user determined): ID.
9) Pointer to the measuring sensor output function (as QMC5883_f32GetAngle): f32(*MeasuringSensor_function)().
10) Pointer to the predicting or rate measuring sensor output function (as IMU_u16GetGyroZ): f32(*PredictingSensor_function)().

And the methods which are only called by the driver not the user:

```
33      static void KalmanFilter_voidGetReadings()
34      {
35
36
37          u8 ID = MSTK_voidGetLastCalled();
38          extern FilteredAngle All_Angles[3];
39
40          f32 newAngle = All_Angles[ID - 1].MeasuringSensor_function();
41          f32 newRate = All_Angles[ID - 1].PredictionSensor_function();
42          All_Angles[ID - 1].KalmanFilter_voidUpdateAngle(newAngle, newRate);
43
44
45      }
```

The function KalmanFilter_voidGetReadings() is defined as a static because it is called by the Timer ISR, so it must be of type void* () not a non-static member function depending on object.

This function reads from the measuring and predicting sensor and call the updateAngle function giving it the measured values as parameters.
Now one thing remains. How can this function determine which object it should call its methods?
For this we made a function that returns the last task that was served by the timer, which is the same as the quantity ID. And we will talk about the stack All_Angles later.

```
46      void KalmanFilter_voidUpdateAngle(f32 newAngle, f32 newRate)
47      {
48
49
50          this->Corrected_Angle = KalmanFilter_f32GetPredictedState(this->Corrected_Angle, newRate);
51          f32* K = KalmanFilter_f32CalculateKalmanGain();
52
53          f32 y = newAngle - this->Corrected_Angle; // Angle difference
54
55          this->Corrected_Angle += K[0] * y;
56          this->bias += K[1] * y;
57
58
59          f32 P00_temp = this->P[0][0];
60          f32 P01_temp = this->P[0][1];
61
62          this->P[0][0] -= K[0] * P00_temp;
63          this->P[0][1] -= K[0] * P01_temp;
64          this->P[1][0] -= K[1] * P00_temp;
65          this->P[1][1] -= K[1] * P01_temp;
66      }
```

This function is the same as that of the normal C driver with just using this keyword for the class fields.

```
67      f32 KalmanFilter_f32GetPredictedState(f32 newAngle, f32 newRate)
68      {
69          f32 corrected_rate = newRate - this->bias;
70          newAngle += corrected_rate * 0.3 * this->deltaT;
71
72          this->P[0][0] += this->deltaT * (this->deltaT * this->P[1][1] - this->P[0][1] - this->P[1][0] + this->sigma1);
73          this->P[0][1] -= this->deltaT * this->P[1][1];
74          this->P[1][0] -= this->deltaT * this->P[1][1];
75          this->P[1][1] += this->sigma2 * this->deltaT;
76
77          return newAngle;
78      }
79      f32 KalmanFilter_f32CalculateInnovation(void)
80      {
81          f32 S = this->P[0][0] + this->R;
82          return S;
83      }
84      f32* KalmanFilter_f32CalculateKalmanGain(void)
85      {
86          f32 S = this->KalmanFilter_f32CalculateInnovation();
87
88          static f32 K[2];
89          K[0] = this->P[0][0] / S;
90          K[1] = this->P[1][0] / S;
91          return K;
92      }
```

These functions are also the same as that of the normal C driver with just using this keyword for the class fields.

```
93  public:
94
95      FilteredAngle(u16 angle_ID, f32 delta_t, f32(*Sensor_function)(), short(*Predictor_function)())
96      {
97          // TODO Auto-generated constructor stub
98          this->ID = angle_ID;
99          this->deltaT = delta_t;
100         MSTK_voidInit();
101         MSTK_voidCreateTask(angle_ID, delta_t * 1000, KalmanFilter_voidGetReadings);
102         MSTK_voidStartScheduler();
103
104         this->MeasuringSensor_function = Sensor_function;
105         this->PredictionSensor_function = Predictor_function;
106
107     }
```

Now it is the time to talk about the public section of the class:

The first function is the constructor. It accepts the ID, the time and the pointer to measuring and predicting sensors.

The function assigns the fields ID, deltaT, MeasuringSensor_function and PredictingSensor_function according to user input. Then it creates a systick timer task with the same ID and deltaT whose ISR function is KalmanFilter_voidGetReadings.

```
108     void PushInStack()
109     {
110         extern u8 stackPtr;
111         extern FilteredAngle All_Angles[3];
112         All_Angles[stackPtr] = *this;
113         stackPtr++;
114     }
```

This function pushes every object created by the user in a predefined stack to keep track of every quantity simultaneously and we will talk about this stack later.

```
115     f32 KalmanFilter_f32GetAngle(void)
116     {
117         extern FilteredAngle All_Angles[3];
118         return All_Angles[(this->ID) - 1].Corrected_Angle;
119     }
```

This function returns the required quantity output through its index in the All_Angles stack.

```
120     FilteredAngle()
121     {
122
123     }
```

This is the default constructor of the class.

```
124     virtual ~FilteredAngle()
125     {
126
127     }
128 };
```

And finally, this is the destructor of the class.

- The .cpp File of the Class:

```
1  #include<FilteredAngle.h>
2  #include"STD_TYPES.h"
3  FilteredAngle All_Angles[3];
4  u8 stackPtr = 0;
```

Here is the declaration of the stack All_Angles and its pointer stackPtr. The stack size is defined to be 3 because the driver was made for euler's angles only. However, it can be extended for all other quantities by increasing its size.

Now, Let's use the driver in the main:

```
26      FilteredAngle angleYaw = FilteredAngle(1, 0.004, QMC5883_f32GetHeading, IMU_u16GetGyroZ);
27      FilteredAngle anglePitch = FilteredAngle(2, 0.004, IMU_u16GetPITCH_ACC, IMU_u16GetGyroY);
28      FilteredAngle angleRoll = FilteredAngle(3, 0.004, IMU_u16GetROLL_ACC, IMU_u16GetGyroY);
```

Here, we created three objects through the class constructor, each refers to one of the euler's angles.

We set theirs IDs to be 1, 2, 3. And their all their time steps to be 4 ms. Then we set the pointers to measuring and predicting sensors. Where:

1) angleYaw measuring sensor is the compass so the measuring sensor pointer is QMC5883_f32GetHeading. While the predicting sensor is the gyroscope component in z direction whose pointer is IMU_u16GetGyroZ.
2) anglePitch measuring sensor is the accelerometer so the measuring sensor pointer is IMU_u16GetPITCH_ACC. While the predicting sensor is the gyroscope component in y direction whose pointer is IMU_u16GetGyroY.
3) angleRoll measuring sensor is the accelerometer so the measuring sensor pointer is IMU_u16GetROLL_ACC. While the predicting sensor is the gyroscope component in x direction whose pointer is IMU_u16GetGyroX.

```
30      angleYaw.PushInStack();
31      anglePitch.PushInStack();
32      angleRoll.PushInStack();
```

Now, we push all the three objects in the stack.

```
34      while(1)
35      {
36          heading = angleYaw.KalmanFilter_f32GetAngle();
37
38          ltoa(heading, msg, 10);
39          UART_voidTransmitString(UART0, msg);
40          UART_voidTransmitChar(UART0, ' ');
41
42          heading = anglePitch.KalmanFilter_f32GetAngle();
43
44          ltoa(heading, msg, 10);
45          UART_voidTransmitString(UART0, msg);
46          UART_voidTransmitChar(UART0, ' ');
47
48          heading = angleRoll.KalmanFilter_f32GetAngle();
49
50          ltoa(heading, msg, 10);
51          UART_voidTransmitString(UART0, msg);
52          UART_voidTransmitChar(UART0, '\n');
53          strcpy(msg, "");
54       Delay(4);
55      }
```

Finally, we print all these angles every 4 ms in the while(1) loop.

5.4. Wheel Encoder

DC5V-24V 600P/R Encoder Incremental Rotary 360 Pulses – AB 2 Phase 6mm Shaft

A rotary encoder, also called a shaft encoder, is an electro-mechanical device that converts the angular position or motion of a shaft or axle to analog or digital output signals.

Rotary encoders are used in a wide range of applications that require monitoring or control, or both, of mechanical systems, including industrial controls, robotics, photographic lenses, computer input devices such as optomechanical mice and trackballs, controlled stress rheometers, and rotating radar platforms.

5.4.1. Features

- Part Number: E38S6G5-600B-G24N.
- 600 p/r.
- Power source: DC5-24V.
- Shaft: 6*13mm.
- Size: 38*35.5mm.
- Output: AB 2phase output rectangular orthogonal pulse circuit, the output for the NPN open-collector output type
- Maximum mechanical speed: 5000 R / min.
- Response frequency: 0-20KHz.
- Cable length: 1.5 meter.
- Notice: AB 2phase output must not be directly connected with VCC, otherwise, will burn the output triode, because different batches, and may not have the terminal.

5.4.2. Hardware

DC5V-24V 600P/R Encoder Incremental Rotary 360 Pulses – AB 2 Phase 6mm Shaft:

Pinout

- Green → A phase.
- White → B phase.
- Red → VCC power “+”.
- Black → V0 “GND”.

Figure 5.31: DC5V-24V 600P/R Encoder Incremental Rotary 360 Pulses - AB 2 Phase 6mm Shaft.

SN-360:

- Type: Incremental Encoder
- Cycles per revolution (CPR): 20
- Working voltage: 0 – 5V
- Material: PCB + Brass
- Weight: 10g
- Size: 32 x 19 x 30mm

Figure 5.32: Rotary Encoder (SN-360).

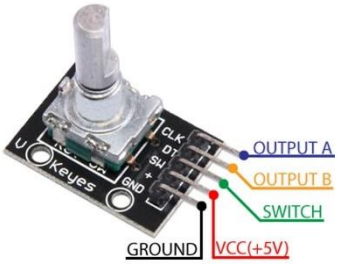


Figure 5.33: Rotary Encoder (SN-360) Pinout.

5.4.3. Why Wheel Encoder?

It is used to measure the rotational speed, Angle and acceleration of the object and the length measurement Suitable for intelligent control of various displacement measurement, automatic fixed-length leather automatic guillotine machines, steel cut length control, civil measured height human scale, Students racing robots.

DC5V-24V 600P/R Encoder Incremental Rotary 360 Pulses – AB 2 Phase 6mm Shaft:

We can used to measure the distance of the vehicle and with the TIMER of microcontroller we can get the velocity of the vehicle.
Only with measure the rotation of wheel and calibration number or rotations in 1 meter for example we can get the distance.
It connected with wheel for forward and back direction.

SN-360:

We can used to measure the angel of driving wheel for right and left direction.

5.4.4. The operation of Wheel Encoder

- As we see in figure 3.6 when we rotate in clockwise the falling edge of A phase meets with high B phase, and if we rotate in counter-clockwise the falling edge of A phase meets with low B phase.
- We use in our application only one phase to fire the interrupt when the wheel of vehicle rotates and count up the numbers of rotation with calibration we can get the distance of this vehicle in meter.

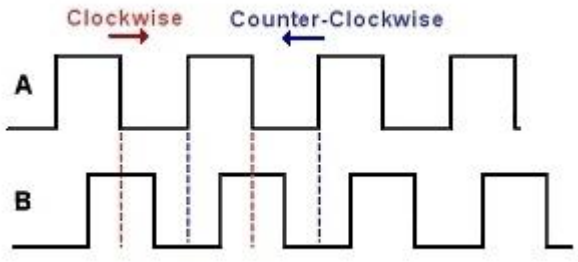


Figure 5.34: Rotary Encoder (SN-360) Pinout.

5.4.5. Calibration of Wheel Encoder

To calibrate the encoder, we follow these steps:

1) Measure 2-meter distance and let the car move.
2) Read the result of encoder through UART.
3) We retry the first two steps 5 times.

Try	Encoder's Read
#1	1187
#2	1270
#3	1208
#4	1275
#5	1182

Table 5.1: Wheel Encoder Calibration.

4) $Average = \frac{1187+1270+1208+1275+1182}{5} = 1224.4$
5) Then for 2-meter encoder rotate about 1224 times.
6) So, for 1-meter $\frac{1224}{2} = 612$
7) We do same for driving wheel and measure the read of encoder to get driving wheel to left and right.

5.4.6. Software

- ENCODER_config.h:

```
/* Set A Phase PORT/PIN */
#define HENCODER_A_PHASE_PORT    PORTF
#define HENCODER_A_PHASE_PIN     4
```

This file to set the PORT and PIN of A Phase in TIVA-C.

- ENCODER_private.h:

```
/* 1 meter = 612 */
#define HENCODER_CALIBRATION_VALUE      612
```

This value of calibration.

```
/* Interrupt Number */
#if HENCODER_A_PHASE_PORT == PORTA
    #define HENCODER_INT_ENABLE 0
#elif HENCODER_A_PHASE_PORT == PORTB
    #define HENCODER_INT_ENABLE 1
#elif HENCODER_A_PHASE_PORT == PORTC
    #define HENCODER_INT_ENABLE 2
#elif HENCODER_A_PHASE_PORT == PORTD
    #define HENCODER_INT_ENABLE 3
#elif HENCODER_A_PHASE_PORT == PORTE
    #define HENCODER_INT_ENABLE 4
#elif HENCODER_A_PHASE_PORT == PORTF
    #define HENCODER_INT_ENABLE 30
#else
    #error("You Enter A Wrong HENCODER_A_PHASE_PORT")
#endif
```

After we set PORT of A Phase this file will choose the right Interrupt Number.

- ENCODER_interface.h:

  This file contains function declaration.

```
/* Initialization */
void HENCODER_voidInit(void);
/* ISR */
void HEncoder_voidOperationISR(void);
/* Read the Distance in Meter */
u32 HENCODER_u32ReadDistanceInMeter(void);

/* Reset */
void HENCODER_voidResetCounter(void);
void HENCODER_voidResetDistance(void);
void HENCODER_voidResetEncoder(void);
/* Turn off Encoder */
void HENCODER_voidTurnOffEncoder(void);
```

- ENCODER_program.c: This file contains function of the driver.

```
void HENCODER_voidInit(void)
{
    Load_Vector_Table();
    /* A Phase Initialization */
    GPIO_voidInitializeDigitalPin(HENCODER_A_PHASE_PORT,HENCODER_A_PHASE_PIN,INPUT_PULLUP,Drive_8mA);
    GPIO_voidSetPULL_UP_DN(HENCODER_A_PHASE_PORT,HENCODER_A_PHASE_PIN,PULLUP);
    GPIO_voidSetInterruptEvent(HENCODER_A_PHASE_PORT,HENCODER_A_PHASE_PIN,FallingEdge);
    Interrupt_voidEnable(HENCODER_INT_ENABLE);
}
```

Initialization Function to set the direction of A Phase pin as pullup

input, enable the interrupt in this pin and set the interrupt event at Failing Edge.

```
/* ISR */
void HENCODER_voidOperationISR(void)
{
    Global_u32Counter++;

    if ((Global_u32Counter%HENCODER_CALIBRATION_VALUE) == 0)
    {
        Global_u32Distance++;
    }
    else
    {
        /* Do Nothing */
    }

    MUART_voidWriteNum16Bit(Global_u32Counter);
    MUART_voidSendDataSynch(UART0, "\n");
}
```

ISR Function: this function consists of the operation of encoder it happen when the interrupt fired by rotating the wheel encoder the Global_u32Counter will increase by 1 until it reach to the value reminder of it with Calibration value equal to zero in this condition the distance will increase by 1 meter we can read this value with next function by reading Global_u32Distance.

```
/* Read the Distance in Meter */
u32 HENCODER_u32ReadDistanceInMeter(void)
{
    return Global_u32Distance;
}
```

```c
/* Reset */
void HENCODER_voidResetCounter(void)
{
    Global_u32Counter = 0;
}
void HENCODER_voidResetDistance(void)
{
    Global_u32Distance = 0;
}
void HENCODER_voidResetEncoder(void)
{
    Global_u32Counter = 0;
    Global_u32Distance = 0;
}

/* Turn off Encoder */
void HENCODER_voidTurnOffEncoder(void)
{
    Global_u32Counter = 0;
    Global_u32Distance = 0;
    Interrupt_voidDisable(HENCODER_INT_ENABLE);
}
```

These function use for Reset or Turn off the encoder.

5.5. GPS

5.5.1. Mobile GPS

We tried to use a GPS module, but it was so slow and not accurate.
So, we create a new idea to get location of vehicle, Traffic light or the garage, by using GPS of Mobile and share its reads via Bluetooth and serial communication between the Mobile phone and Raspberry Pi.

Figure 5.35: NEO6MV2 GPS module with active antenna.

5.5.2. Share GPS via Bluetooth:

Step 1: Pair your mobile phone with Raspberry pi.

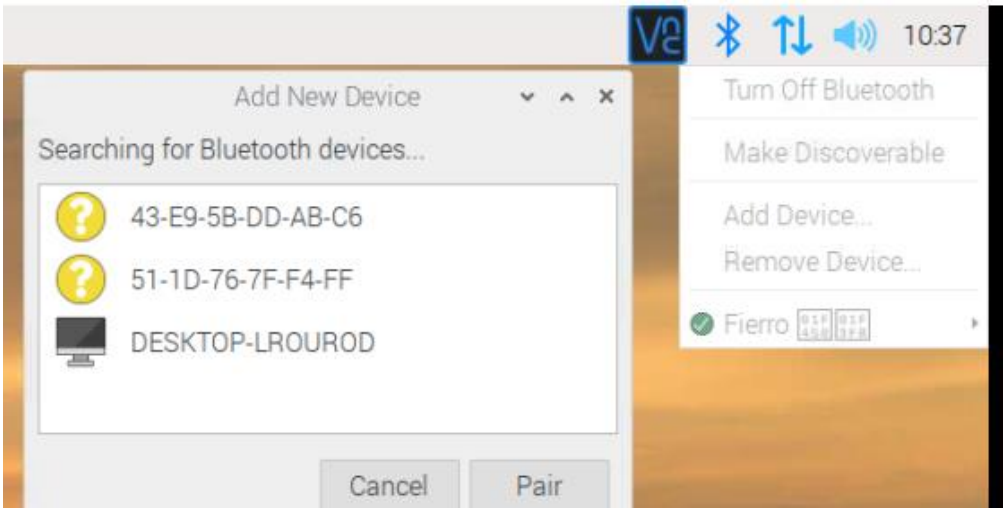


Figure 5.36: Share GPS via Bluetooth Step 1.

Step 2: Install Share GPS on your phone.

Figure 5.37: Share GPS via Bluetooth Step 2.

Step 3: Turn on Location on your phone and the app will start getting your latitude and longitude.

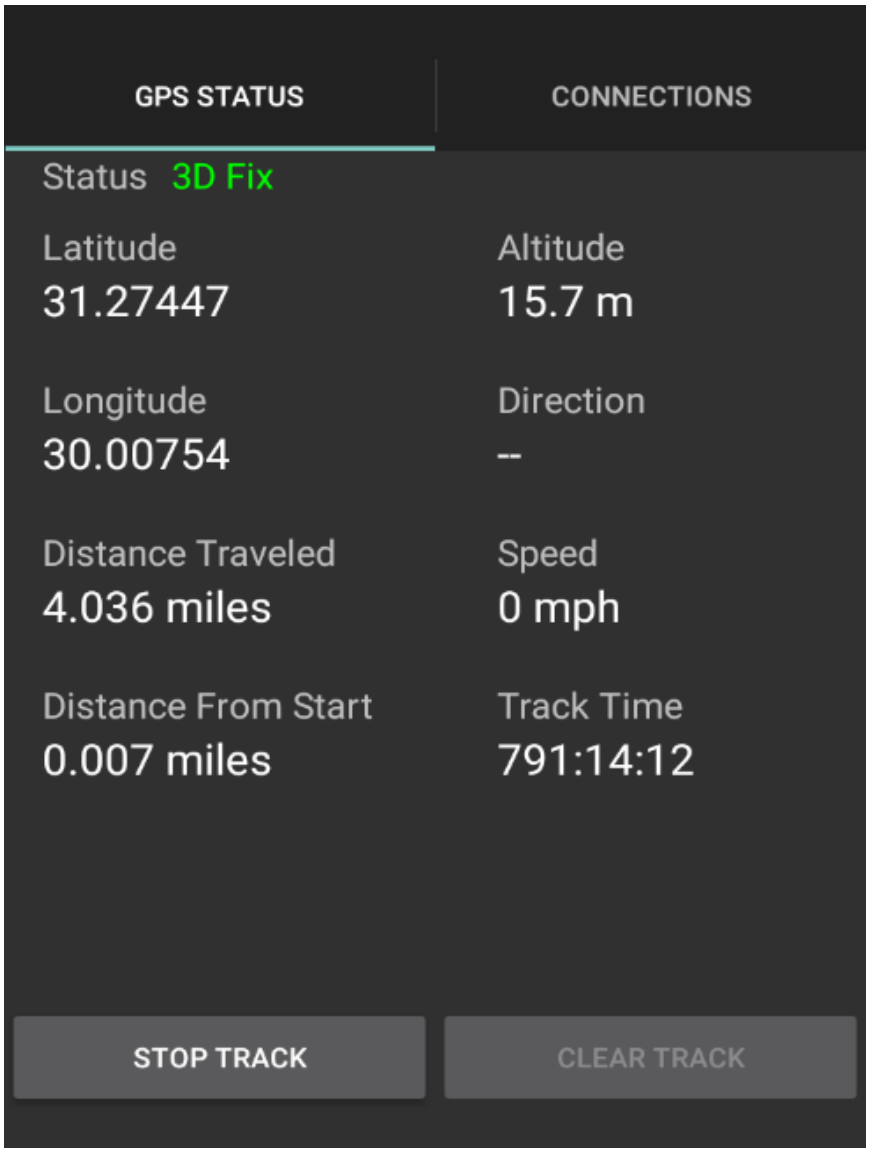


Figure 5.38: Share GPS via Bluetooth Step 3.

Step 4: wait serial connection by using this command:

sudo rfcomm watch hci0

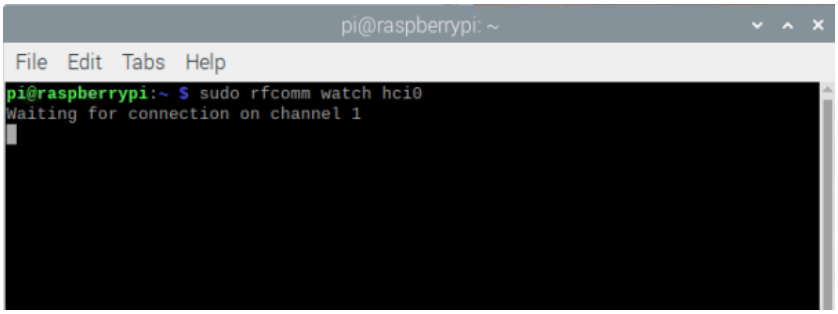


Figure 5.39: Share GPS via Bluetooth Step 4.

Step 5: Connect the app with raspberry pi.

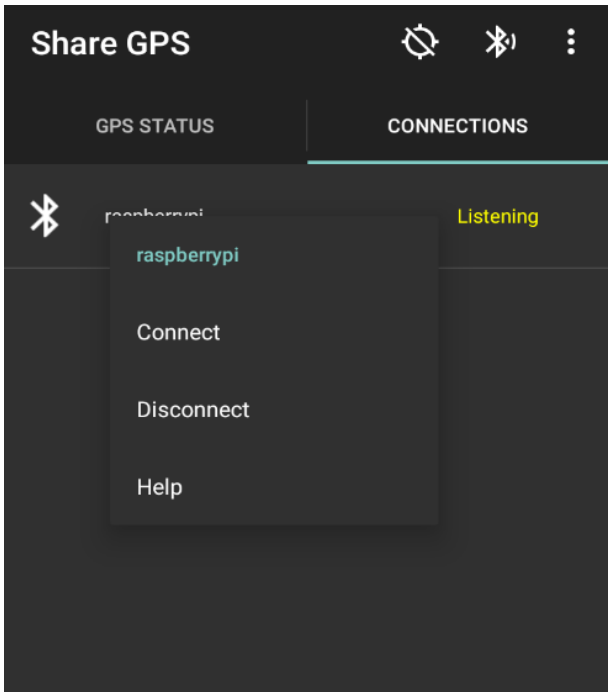


Figure 5.40: Share GPS via Bluetooth Step 5.

Step 6: Now app starts to connect and share location to raspberry pi.
And you can receive the reads by serial communication of /dev/rfcomm0.

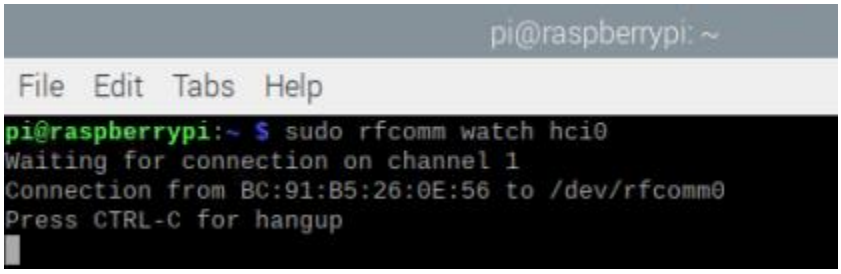


Figure 5.41: Share GPS via Bluetooth Step 6 a.

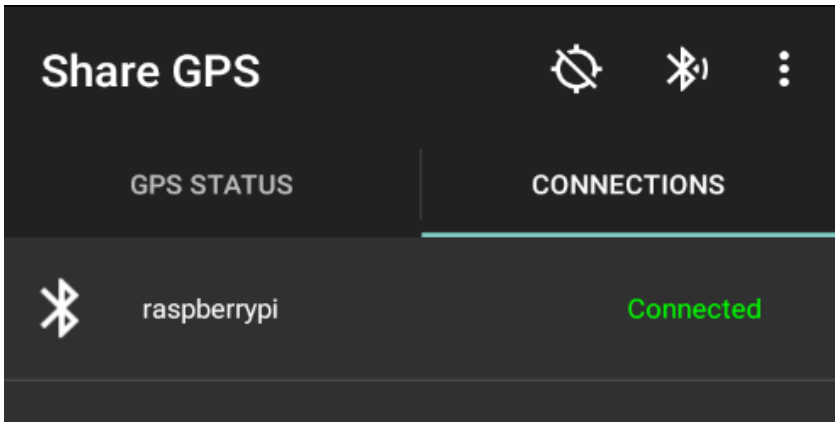


Figure 5.42: Share GPS via Bluetooth Step 6 b.

Step 7: By python and serial module we can get NMEA code that had your latitude and longitude.

GPS.py

```
1  import serial
2
3  ser = serial.Serial("/dev/rfcomm0",baudrate=9600,timeout=0.5)
4
5
6  while True:
7      data = ser.readline()
8      print(data)
```

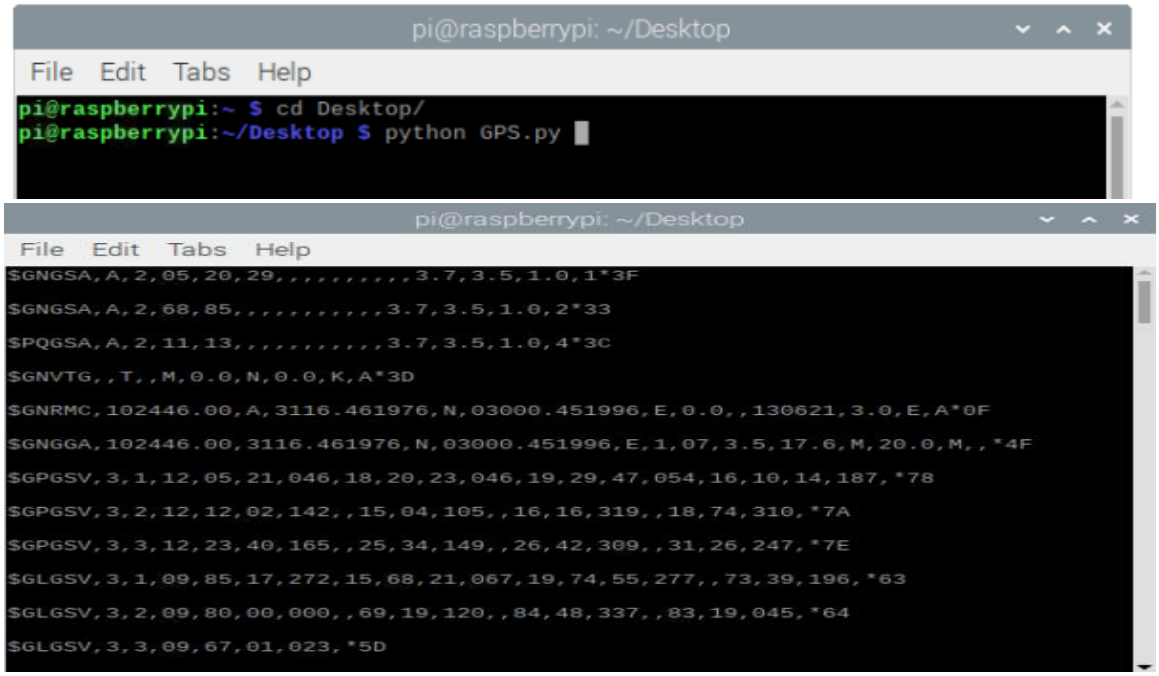


Figure 5.43: Share GPS via Bluetooth Step 7.

5.5.3. NMEA:

At the end we received a message that called NMEA code or NMEA message.

```
$GNRMC,055557.00,A,3116.459486,N,03000.462794,E,0.0,,170721,3.0,E,A*0D
```

All we need is this line that has GMT, your latitude, and your longitude.

- GMT: `055557` This means the time of GMT 05:55:57.
- Position status: `A,` (A = data valid, V = data invalid)
- Latitude:

  `,3116.461976,N,`

  This means $31°16.461976'N = 31.27438°N$
- Longitude:

  `03000.451996,E`

  This means $30°00.451996'e = 30.00755°N$

So, the app all his works is sharing GPS of mobile with Raspberry pi and sending NMEA code with your time and position.

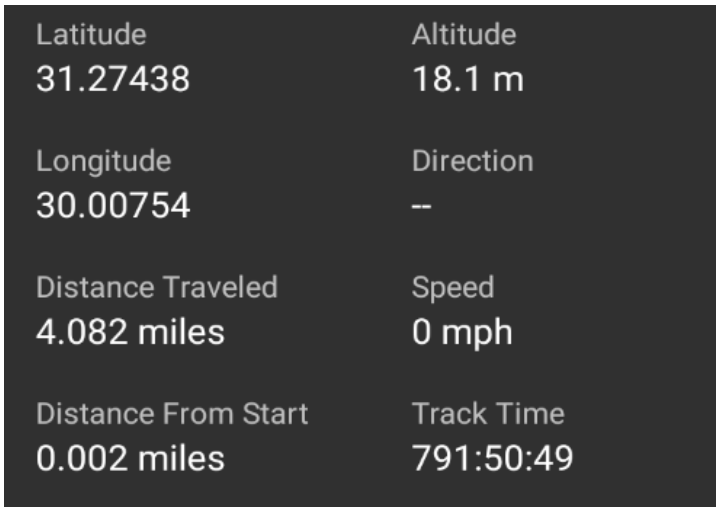


Figure 5.44: NMEA Checking.

5.5.4. Advantages

- So accurate using many stiles to get your location.
- Indoor.
- Suitable for our project.
- Using Bluetooth of Raspberry pi.

5.5.5. Software

Using serial module to get data from Bluetooth of mobile and the parsing NMEA code to get specific line start with “$GNRMC” sprite it to time, latitude and longitude.

We can send the data after parsing to another microcontroller through UART.

GPS.py

```
import serial

GPS  = serial.Serial("/dev/rfcomm0",baudrate=9600,timeout=0.5)
#UART = serial.Serial("/dev/ttyS0",baudrate=9600,timeout=0.5)
#UART.write("ddd")
while True:
    line = GPS.readline()
    data = line.split(",")
    ''' Parsing NMEA Code '''
    if data[0] == "$GNRMC":
        #UART.write(line)
        #UART.write("\n")
        ''' Get UTC of postion '''
        time = data[1]
        time = time.split(".")
        utc  = time[0]
        utc2 = str(int(utc[1])+2)
        hour = utc[0] + utc2
        mins = utc[2:4]
        secs = utc[4:6]
        print(line)
        print ("Time(UTC+2): "+hour+":"+mins+":"+secs)
        #UART.write("Time(UTC+2): "+hour+":"+mins+":"+secs)
        #UART.write("\n")
        ''' Check Position status (A = data valid, V = data invalid) '''
        if data[2] == "A":
            ''' Get UTC of Latitude '''
            Latitude_deg = data[3][0:2]
            Latitude_min = data[3][2::]
            Latitude_dir = data[4]
            Latitude     = Latitude_deg+Latitude_min+Latitude_dir
            print("Latitude: "+Latitude)
            #UART.write("Latitude: "+Latitude)
            #UART.write("\n")
            ''' Get UTC of Longtitude '''
            Longtitude_deg = data[5][0:3]
            Longtitude_min = data[5][3::]
            Longtitude_dir = data[6]
            Longtitude     = Longtitude_deg+Longtitude_min+Longtitude_dir
            print("Longtitude: "+Longtitude)
            # UART.write("Longtitude: "+Longtitude)
            #UART.write("\n")
        else:
            print "Not Fix - Data Invalid"
```

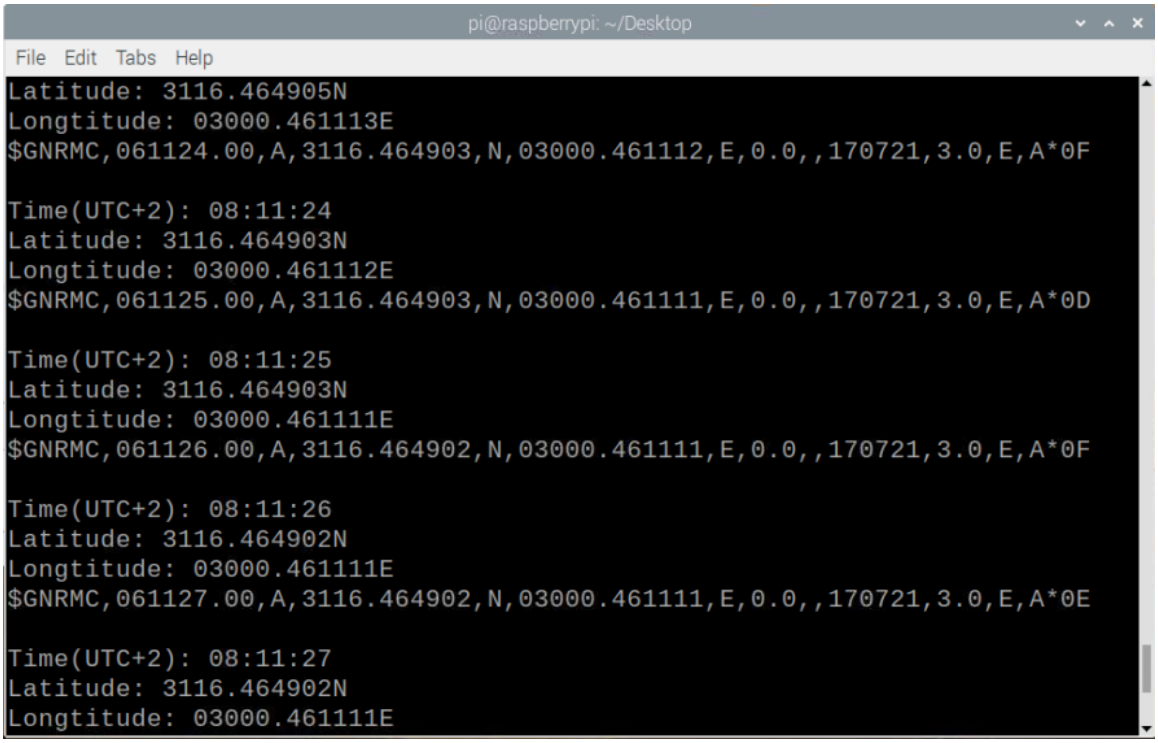


Figure 5.45: GPS code output.

As shown refresh data every 1 second.

6. Vehicle Unit

6.1. Introduction

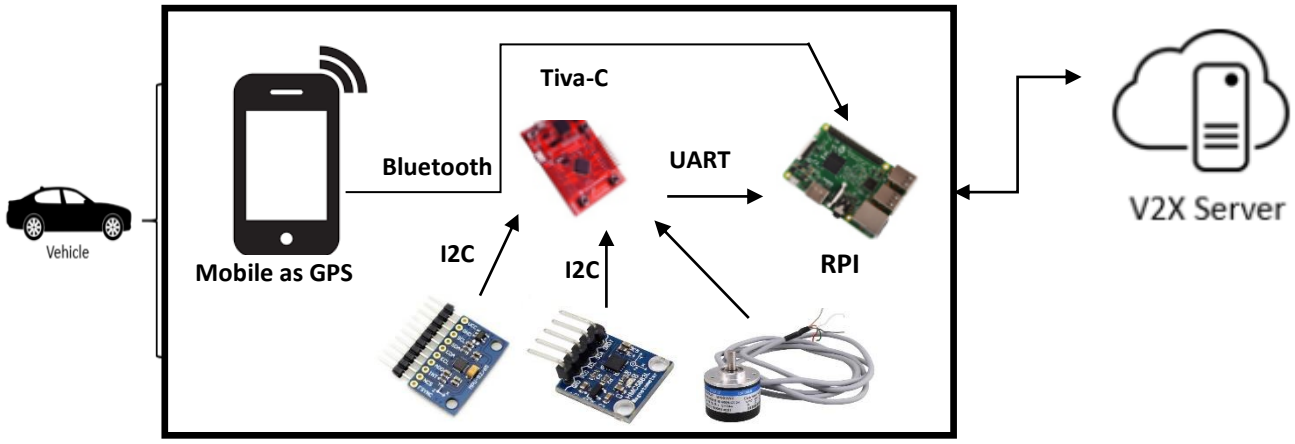


Figure 6.1: Vehicle Unit Diagram.

vehicle block consists of 4 main components (Tiva-C (ARM Based) microcontroller- Raspberry Pi – Mobile as GPS – Sensors of orientation and distance).

The GPS used to send the vehicle location to the Raspberry through Bluetooth.

Sensors:

- Encoders we have two as we told one for distance and other one for driving wheel deviation.
- IMU and Electronic Compass with Kalman filter we get the orientation of the car and correct the deviation with the encoder which connect with driving wheel.

The Tiva-C (ARM Based) microcontroller is used for controlling the sensors and collect the output to be sent finally to the Raspberry pi through UART Protocol.

The Raspberry pi is used here for the receiving from the Tiva-c and sending the data to the server and display the speed and location of the car on the monitor of the PC.

6.2. PCB Design

We design PCB KIT for Vehicle Unit to let the connection of the circuit more stable for calibration of sensors and send data to RPI with less losses or garbage.

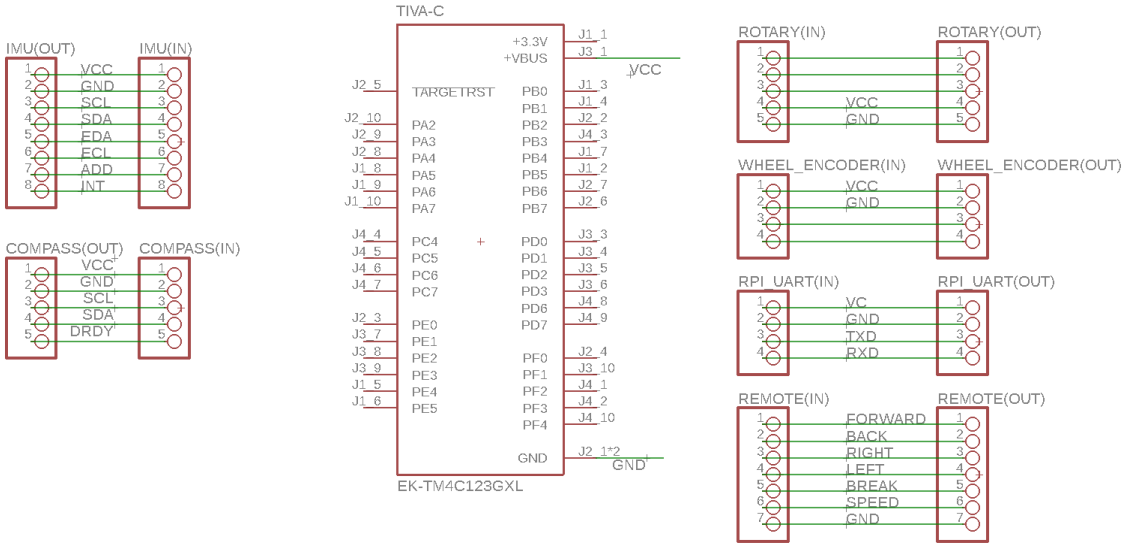


Figure 6.2: The PCB Schematic of the Vehicle Unit by Eagle.

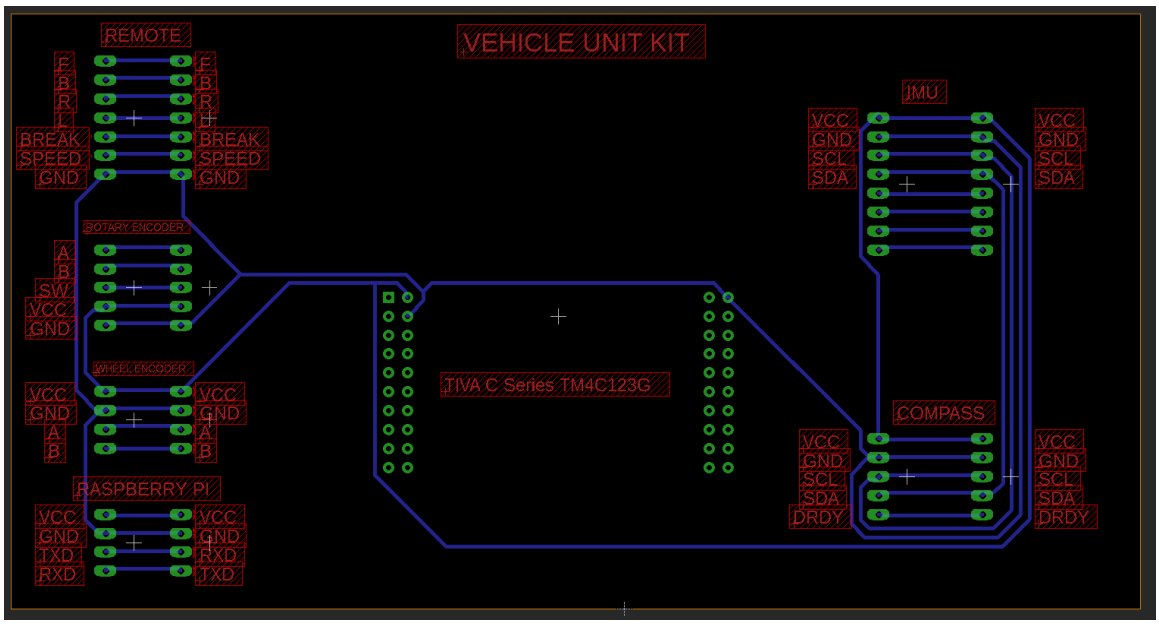


Figure 6.3: The PCB Board of the Vehicle Unit by Eagle.

6.3. Mechanical Work

To install the wheel encoder next to the wheel of the car to reflect the rotation of the wheel, and the number of turns of the encoder is calculated then do mathematical operations on it to determine the distance.

Figure 6.4: Mechanical Work of the vehicle (1).

We have designed a gear for the orientation encoder.

Figure 6.5: Mechanical Work of the vehicle (2).

6.4. Scenarios

We have to send speed, state (on or off) and location (latitude and longitude) to the V2X server and receive the state of any traffic light in the same area, garage state and communicate with other cars.

We have

- Optional semi self-driving car in long distance or highway:
  The driver has the length of the way and V2X server automatically sends the distance to the microcontroller.

- Follow another car which has same destination:
  V2X server read the states of the master car then send the same states to the slave car which send them to Tiva-C to control the car.

- Traffic Light: car stops when the state is red and keeps moving if state green (If both car and traffic light has the same location) and sends the state of car which have been moved or stopped to the V2X server so the cars behind this car.

- Sudden brake: When car stops suddenly sends state of this car to V2X server then the server sends to all cars in the area, so any car takes car.

- Garage Unit: Receive the state of garage full or empty.

7. Traffic Light Unit

7.1. Introduction

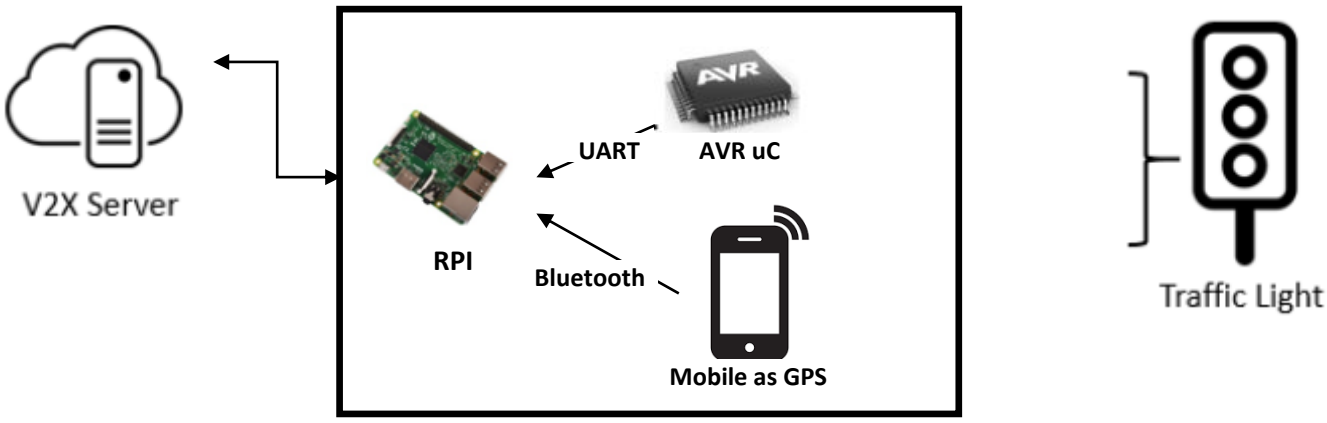
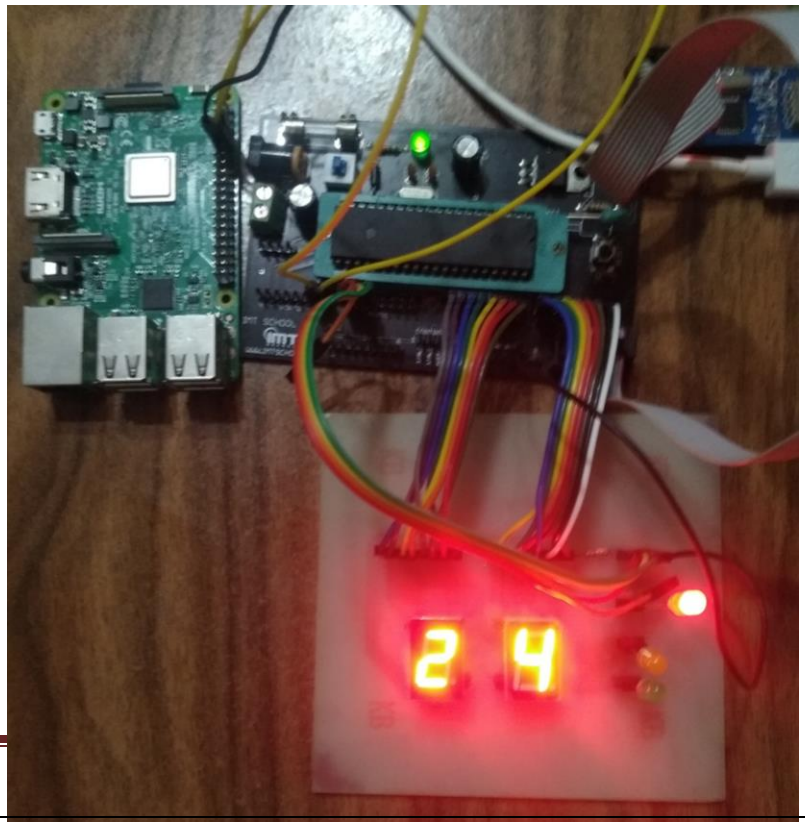


Figure 7.1: Traffic Light Unit Diagram.

The Traffic Light Unit consists of 3 main components (Mobile as GPS - AVR microcontroller- Raspberry Pi).
The GPS is used to send traffic location to the Raspberry Pi through Bluetooth.
The AVR microcontroller is used to count down for the traffic status and change color between green, yellow and red.
The Raspberry Pi is used here for the receiving from AVR microcontroller and sending the data to the server.



Raspberry Pi also display the data on monitor of the PC and you could send data of GPS to AVR through UART.

Figure 7.2: The Design of the Traffic Light Unit.

7.2. PCB Design

We design PCB KIT for Traffic Light Unit as maquette to let the connection of the circuit more stable and send data to RPI with less losses or garbage.

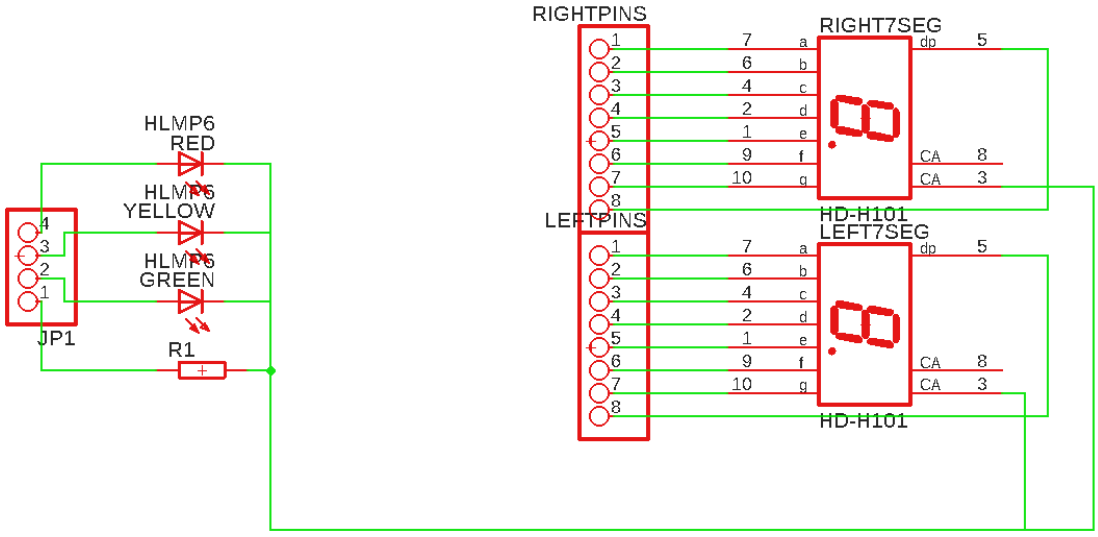


Figure 7.3: The PCB Schematic of the Traffic Light Unit by Eagle.

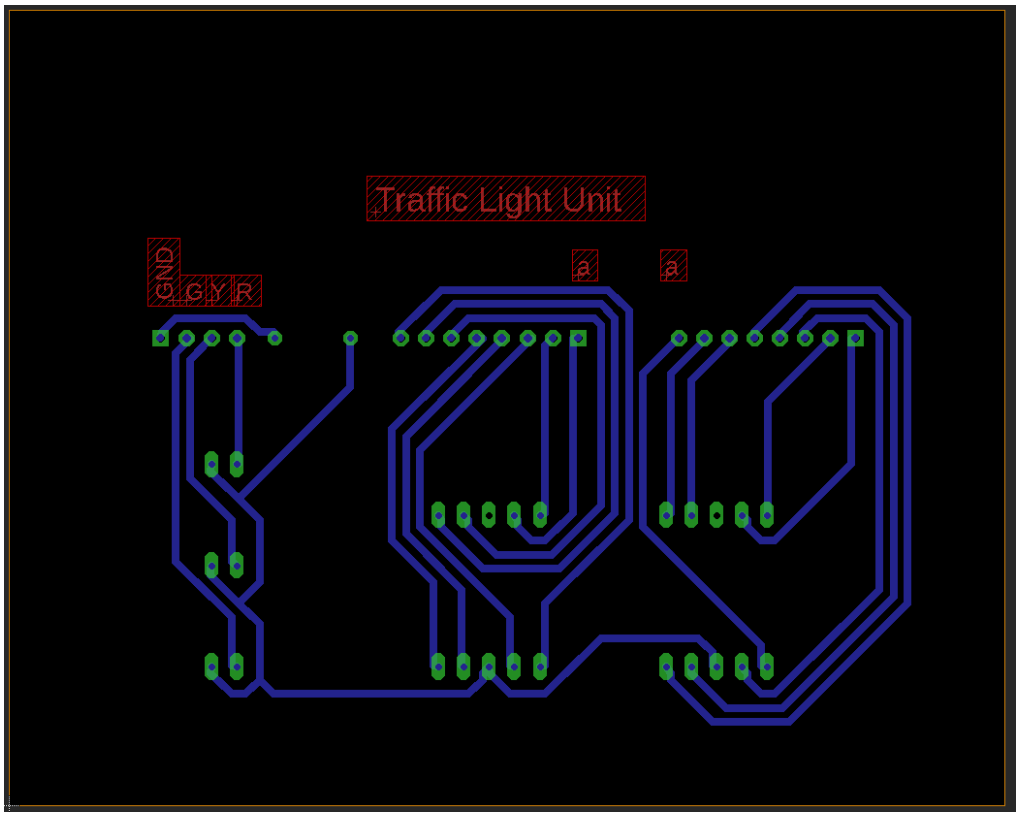


Figure 7.4: The PCB Board of the Traffic Light Unit by Eagle.

7.3. The operation of the unit

1- The red light will be turn on for 30 seconds the 7-segments count down from 29 to 00 Second.
2- The Green light will turn on for 55 seconds the 7-segments count down from 59 to 05 Second.
3- When the 7-segments value 05 the yellow light will turn on for 5 seconds and 7-segments will count down until reach 00.

7-segments:

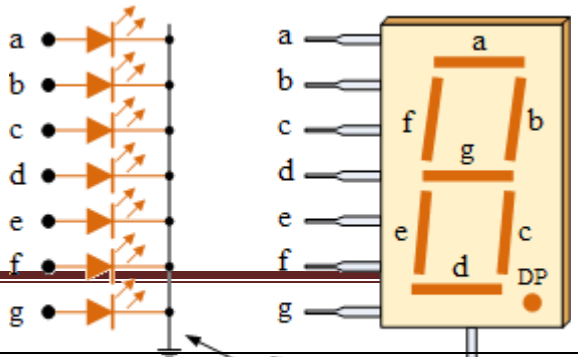


Figure 7.5: Common Cathode 7-Segments.

We use 2 ”Common cathode 7-segments” one for units and the other for tens.

7.4. Software

AVR:

First, The configuration of the pins of 7-segments and LED.

```
/* Configuration */
#define RED_LIGHT_PIN        PORTC,PIN0
#define YELLOW_LIGHT_PIN     PORTC,PIN1
#define GREEN_LIGHT_PIN      PORTC,PIN2
#define COUNTER_LEFT_PORT    PORTA
#define COUNTER_LEFT_COMMON  HSEVSEG_CATHODE
#define COUNTER_RIGHT_PORT   PORTB
#define COUNTER_LEFT_COMMON  HSEVSEG_CATHODE
```

Initialization to set the Direction of pins all pins will be output.

```c
/* Functions */
/* Set PINs/PORTs Direction */
void voidInit(void)
{
    MDIO_voidSetPinDirection(RED_LIGHT_PIN,OUTPUT);      // Red Light
    MDIO_voidSetPinDirection(YELLOW_LIGHT_PIN,OUTPUT);   // Yellow Light
    MDIO_voidSetPinDirection(GREEN_LIGHT_PIN,OUTPUT);    // Green Light
    MDIO_voidSetPortDirection(COUNTER_LEFT_PORT,OUTPUT_ALL);
    MDIO_voidSetPortDirection(COUNTER_RIGHT_PORT,OUTPUT_ALL);
}
```

Functions to turn on only one LED (red or yellow or green) and turn off the others.

```c
/* Red Light Function */
void voidOnRedLight(void)
{
    MDIO_voidSetPinValue(RED_LIGHT_PIN,HIGH);
    MDIO_voidSetPinValue(YELLOW_LIGHT_PIN,LOW);
    MDIO_voidSetPinValue(GREEN_LIGHT_PIN,LOW);
}

/* Yellow Light Function */
void voidOnYellowLight(void)
{
    MDIO_voidSetPinValue(RED_LIGHT_PIN,LOW);
    MDIO_voidSetPinValue(YELLOW_LIGHT_PIN,HIGH)
    MDIO_voidSetPinValue(GREEN_LIGHT_PIN,LOW );
}

/* Green Light Function */
void voidOnGreenLight(void)
{
    MDIO_voidSetPinValue(RED_LIGHT_PIN,LOW);
    MDIO_voidSetPinValue(YELLOW_LIGHT_PIN,LOW);
    MDIO_voidSetPinValue(GREEN_LIGHT_PIN,HIGH);
}
```

Main function and operation of traffic light.

```c
/* Main */
void main(void)
{
    /* DIO Initialization */
    MDIO_voidInit();

    /* Set PINs/PORTs Direction */
    voidInit();

    /* UART Initialization */
    MUART_voidInit();

    /* Counter Variable */
    s8 Local_s8CounterLeft  = 0;
    s8 Local_s8CounterRight = 0;
    u8 Local_u8Time         = 0;

    while(1)
    {
        voidOnRedLight();

        for (Local_s8CounterLeft=2;Local_s8CounterLeft>=0;Local_s8CounterLeft--)
        {
            HSEVSEG_voidDisplay(COUNTER_LEFT_PORT,COUNTER_LEFT_COMMON,Local_s8CounterLeft);
            for (Local_s8CounterRight=9;Local_s8CounterRight>=0;Local_s8CounterRight--)
            {
                Time = 10*Local_s8CounterLeft+Local_s8CounterRight;
                MUART_voidSendDataSynch("State: RED\r\n");
                MUART_voidSendDataSynch("Time: ");
                Local_u8Time =Local_s8CounterLeft + '0';
                MUART_voidSendDataSynch(&Local_u8Time);
                Local_u8Time =Local_s8CounterRight + '0';
                MUART_voidSendDataSynch(&Local_u8Time);
                MUART_voidSendDataSynch(" second\r\n");
                HSEVSEG_voidDisplay(COUNTER_RIGHT_PORT,HSEVSEG_CATHODE,Local_s8CounterRight);
                _delay_ms(1000);
            }
        }

        voidOnGreenLight();


        for (Local_s8CounterLeft=5;Local_s8CounterLeft>=0;Local_s8CounterLeft--)
        {
            HSEVSEG_voidDisplay(COUNTER_LEFT_PORT,COUNTER_LEFT_COMMON,Local_s8CounterLeft);
            for (Local_s8CounterRight=9;Local_s8CounterRight>=0;Local_s8CounterRight--)
            {
                if ((Local_s8CounterLeft<=0) && (Local_s8CounterRight<=5))
                {
                    voidOnYellowLight();
                    MUART_voidSendDataSynch("State: YELLOW\r\n");
                }
                else
                {
                    MUART_voidSendDataSynch("State: GREEN\r\n");
                }

                MUART_voidSendDataSynch("Time: ");

                Local_u8Time = Local_s8CounterLeft + '0';
                MUART_voidSendDataSynch(&Local_u8Time);
                Local_u8Time = Local_s8CounterRight + '0';
                MUART_voidSendDataSynch(&Local_u8Time);
                MUART_voidSendDataSynch(" second\r\n");
                HSEVSEG_voidDisplay(COUNTER_RIGHT_PORT,COUNTER_LEFT_COMMON,Local_s8CounterRight);
                _delay_ms(1000);
            }
        }
    }
}
```

This function from 7-segments driver and it set the 7-segment state if common cathode or anode and set its value from next array.

```
void HSEVSEG_voidDisplay(u8 Copy_u8Port, u8 Copy_u8Common, u8 Copy_u8Number)
{
    /* Set Port Mode */
    //MDIO_voidSetPortDirection(Copy_u8Port,OUTPUT_ALL);

    if (Copy_u8Number < HSEVSEG_INDEX)
    {
        switch (Copy_u8Common)
        {
            case HSEVSEG_CATHODE: MDIO_voidSetPortValue(Copy_u8Port,   arru87Seg[Copy_u8Number] );  /* return HSEVSEG_DONE; */  break;
            case HSEVSEG_ANODE  : MDIO_voidSetPortValue(Copy_u8Port, ~(arru87Seg[Copy_u8Number]));  /* return HSEVSEG_DONE; */  break;
            default             : /* return HSEVSEG_WRONG_COMMON; */                                                          break;
        }
    }
    else
    {
        /* return HSEVSEG_WRONG_NUMBER; */
    }
}
```

You could find this array in SEVSEG_private.h.

```
/* 7-Segment Numbers */
const u8 arru87Seg[HSEVSEG_INDEX] = {0b00111111,
                                     0b00000110,
                                     0b01011011,
                                     0b01001111,
                                     0b01100110,
                                     0b01101101,
                                     0b01111101,
                                     0b00000111,
                                     0b01111111,
                                     0b01101111};
```

The AVR send data through UART and Raspberry Pi get location through Bluetooth

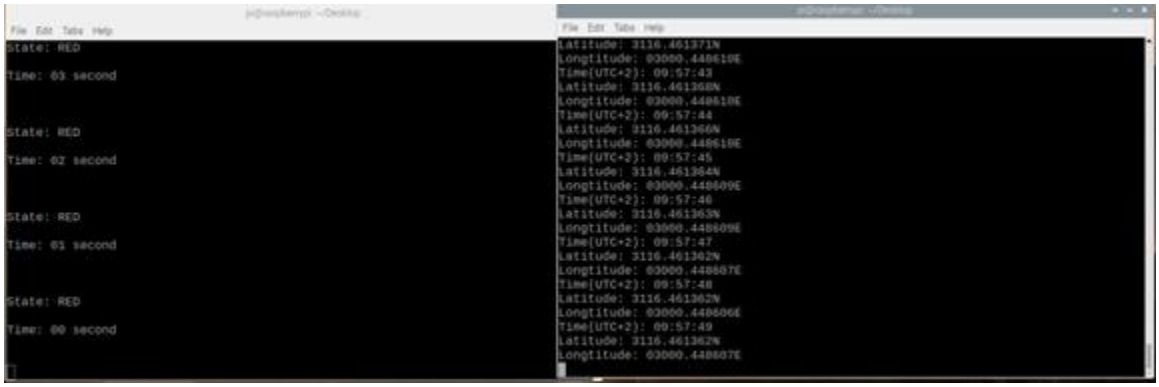


Figure 7.6: Output of The Traffic Light display on RPI.

8. Garage Unit

8.1. Introduction

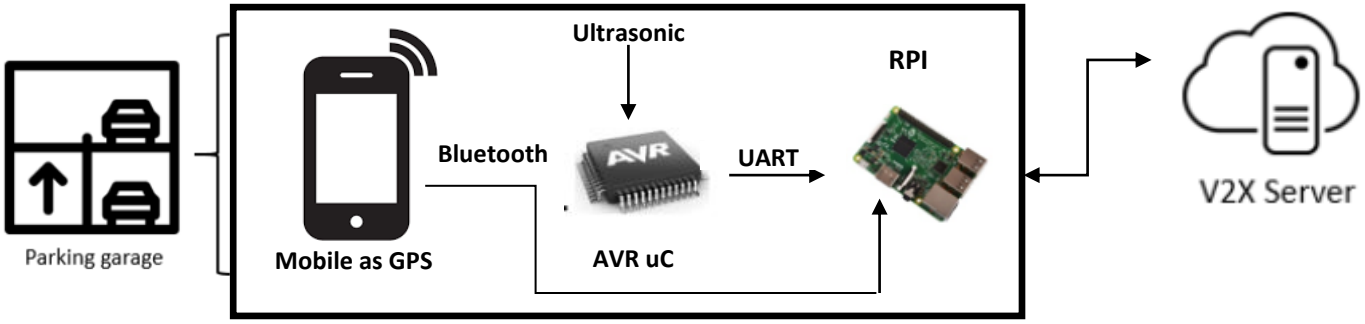


Figure 8.1: Garage Unit Diagram.

The Garage Unit consists of 4 main components (Mobile as GPS - AVR microcontroller - Raspberry pi - Ultrasonic sensors).
The GPS is used to send garage location to the Raspberry Pi through Bluetooth.
The Ultrasonic sensors are used for detecting the presence of vehicle or not to be sent to AVR.
The AVR microcontroller is used to get the data from Ultrasonic for the reserved places in the garage and if there is a space for new vehicles.
The Raspberry pi is used here for the receiving from AVR microcontroller and sending the data to the server and display the data to the PC monitor.

8.2. The operation of the unit

There is 3 Ultrasonic in the garage left, right and behand if the 3 sensors read the distance that cover the area of garage, then the garage is full else the garage is empty and car can park in this garage.
Raspberry Pi send data to the server to the car.