Rupesh Basnet

Wireless Control Mechanism for a Trolling Motor

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Bachelor Thesis

Date: 13 January 2013

Helsinki Metropolia University of Applied Sciences

Author Title Number of Pages Date	Rupesh Basnet Wireless Control Mechanism for a Trolling Motor 44 pages + 3 appendices 14 January 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded System Engineering
Instructor(s)	Kimmo Sauren, Lecturer

The goal of the project was to develop a wireless control mechanism for a trolling motor. Widely used trolling motors are expensive ranging from 100 to over 1000 Euros. Manual control trolling motors are relatively less expensive than wireless trolling motor. The project was carried out to minimize the cost involved in wireless trolling motors. A goal was that having this mechanism developed, one could enjoy unconfined mobility within a boat while fishing at a lower cost.

A manual control trolling motor was bought and a wireless control system for the motor was developed. Microcontrollers and radio transceivers were used to communicate wirelessly. Three input buttons were used for users to feed in the speed inputs. A simple analog electronic circuit was designed to interface the low power microcontroller with high power trolling motor. An N-type transistor was used for the purpose.

A smoothly running wireless trolling motor was developed as a result of the project. It was discovered that developing a wireless control system for a trolling motor, one could minimize the cost involved in buying wireless trolling motors which could save hundreds of Euros. The speed and direction could be customized as required. The user interface for the input could be made more user-friendly.

The project could be further developed to have a global positioning system (GPS) to keep on track of where the boat is going and an auto steering system for a boat to steer itself. Also, a remote pilot that navigates the boat through the GPS system could be implemented. In addition, some external position detecting sensors should be used to locate the exact position so that the auto pilot would know where exactly the boat is and where it should be navigated.

Keywords	PSoC, nRF24L01, UART , SPI, trolling motor, VSD-22YMB

Contents

Appendices

Appendix 1: Program code - Transmit side

Appendix 2: Program Code - Receive Side

Appendixm 3: Function Prototypes Declaration

List of Abbreviations

ADC - analog to digital converter

API - application programming interface

CE - chip enable

CIS - complex instruction set

CPHA - clock phase

CPOL - clock polarity

CRC - cyclic redundancy check

CSN - chip select signal

DAC - digital to analog converter

FIFO - first in first out

GPIO - general purpose input output

GPS - global positioning system

I2C - inter integrated circuit

IC - integrated circuit

ISR - interrupt service routine

MAC - multiply and accumulator

MCU - microcontroller

MISO - master in slave out

MOSFET - metal oxide semiconductor field effect transistor

MOSI - master out slave in

POR - power on reset

PRS - pseudo random sequence

PSoC - programmable system on chip

PWM - pulse width modulation

RF - radio frequency

RFID - radio frequency identity

SCLK - system clock

SMP - symmetric multiprocessor

SPIM - serial peripheral interface master

SRAM - static random access memory

UART - universal asynchronous receiver and transmitter

WDR - watch dog reset

XRES - external reset

Acknowledgements

I would like to extend my sincere thanks to Kimmo Sauren, who not only proposed me the topic but also provided his valuable time in debugging and suggesting me the idea to get the project done to the extent I did. Many thanks to Dr. Antti Piironen for accepting the idea and providing administrative support. I would also like to thank Juho Vesanen and Joseph Hotchkiss for their valuable suggestions and feedback.

1 Introduction

The goal of the project was to develop a wireless control mechanism for a trolling motor. The purpose was to develop a prototype which would embed a wireless control system in a manual control trolling motor. A manual control trolling motor in any boat would make fishing difficult in a way that one has to control the boat sitting close to the motor which confines the individual's movement within the boat. If the motor is installed in the rear deck, one cannot go to the front deck for fishing. Therefore the idea was proposed to develop a wireless control mechanism for the trolling motor that would ease one's movement onboard.

An aim of the project was to put forth an answer to how a manual control trolling motor could be further developed to work as a wireless control trolling motor. One of the key reasons why the project was carried out was the cost issue. How a wireless trolling motor can be developed using a manual control trolling motor with minimum cost is described in this thesis. A block diagram consisting of major electronic parts is illustrated in figure 1.

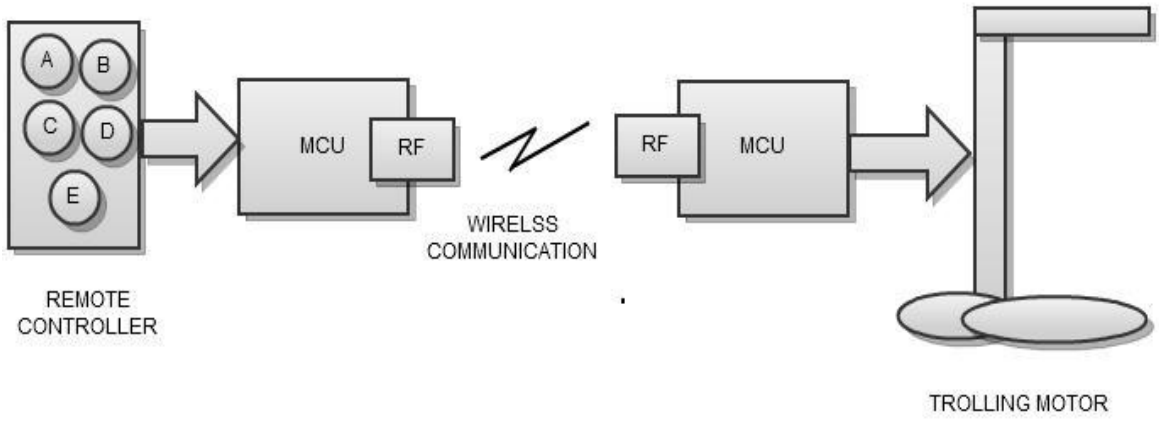

Figure 1: Block diagram of the wireless control system for the trolling motor
(Designed by using web platform, Gliffy [1])

Figure 1 shows five different input possible buttons which are fed in to the microcontroller (MCU). The MCU reads inputs through some connecting wires and based on the inputs, it transmits a unique value to the remote MCU through radio transceivers (RF). The remote MCU reads the received value and outputs a unique pulse with a unique

duty cycle to the trolling motor. The serial peripheral interface master (SPIM) is the protocol to communicate between the MCU and RF.

Trolling motors are used in boats for steering which is a replacement for manual rowing of the boat particularly for fishing purposes. The scope of the project was limited to the wireless steering of a boat. This project did not incorporate auto steering or the global positioning (GPS) system. The trolling motors could be placed in the front or rear deck and steer the boat wirelessly sitting anywhere in the boat. The range of the control system is confined within the area of the boat.

2 Hardware Overview

The project involved programmable system on chip (PSoC) microcontrollers, RF transceivers, a manual control trolling motor and a winch servo motor. PSoC was used to read the fed-in inputs and process the inputs according to the requirements. The RF sends the signals from one module to the other and the trolling motor was the load that the output signals work on and bring the desired consequence.

2.1 Programmable System on Chip (PSoC)

PSoC is a family of integrated circuits (IC) designed and manufactured by Cypress Semiconductor. The application of PSoC ranges from a simple toothbrush to the advanced field of robotics. The PSoC microcontroller is the most powerful microcontroller chip which allows graphical chip-level design and C code implementation or assembly language programming. It also supports inline assembly that is embedded within the C code implementation and has configurable digital and analog blocks. Modules can be placed to these blocks according to the requirements and parameters to these modules can be selected in a graphical environment.

PSoC fascinates users because it lays forth a platform to add in the required modules and edit accordingly in a graphical environment. Additionally a well-documented guide and the built- in datasheets for each module with an illustrated application programming interface (API) assist beginners. Figure 2 shows how a graphical chip-level design environment for a PSoC looks like.

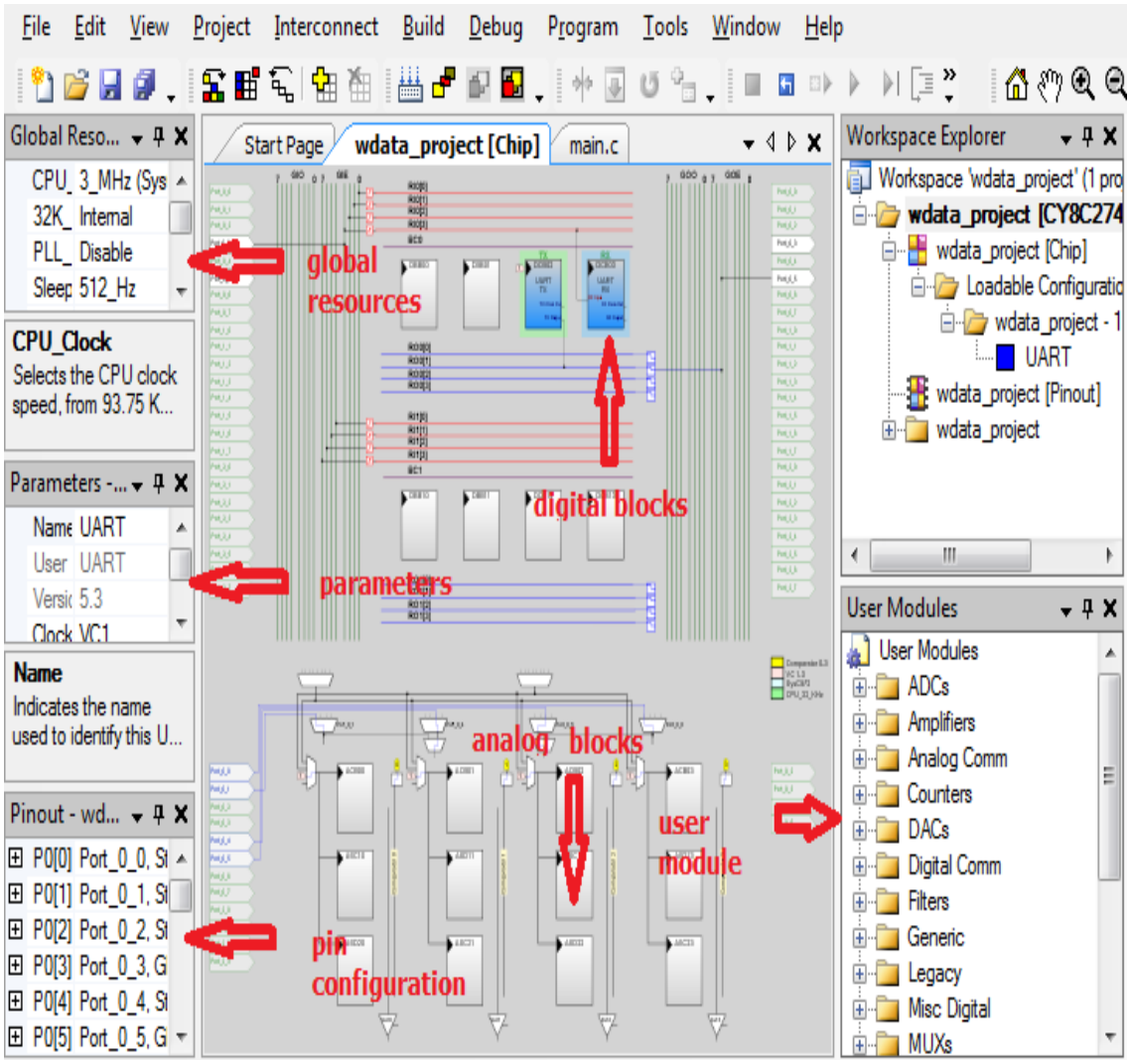

Figure 2: Screenshot of PSoC chip level design

Figure 2 shows how the PSoC chip level graphical window looks like. One can drag (or simply double click) the required components to either the digital blocks or analog blocks as per the module's functionality from the user's modules window. Digital modules like a digital to analog converter (DAC), liquid crystal display (LCD) or light emitting diode (LED) go to the digital block while the analog modules like an analog to digital converter (ADC) or amplifiers etc go to the analog blocks. The global resources window is used to customize the clock source and define the other hardware settings like voltage levels, sleep time rate for the processor. The parameters window allows to change the name of the module, select the input clock and row input and output buses. Pins can be configured through the pin configuration window.

2.1.1 PSoC Architecture

CY8C29466-24PXI was the microcontroller used in the project. It is an 8-bit complex instruction set (CIS) computer. The selection of a particular microcontroller depends on the functionality of the project. However the basic differences among the PSoC families are the number of available programmable blocks and number of input and output pins. Depending on the microcontroller family, PSoC chips have 4-16 programmable digital blocks and 3-12 analog blocks. The prominent features of PSoC are the following:

- Multiply and accumulator (MAC) unit, hardware 8x8 multiplication with result stored in 32-bit accumulator,
- Changeable working voltage, 3.3V or 5V,
- Possibility of small voltage supply, to 1V,
- Programmable frequency choice [2].

PSoC microcontrollers have programmable voltage, inverting amplifiers and non-inverting amplifiers. Accessing the peripheral device through PSoC is convenient from the code implementation point of view as one can easily find the sample firmware in the datasheets. The reason for the PSoC success is its ability to allow peripheral device configuration inside the MCU itself and flexible pin assignment.

Fast prototyping is another significant feature of the PSoC MCU. Developing a prototype using the PSoC MCU is convenient and fast because of its graphical user interface on which the required user module could simply be dragged and dropped and the available sample firmware in the datasheets makes coding fast. The PSoC standard library includes almost all the features of C programming and additionally it also supports inline assembly programming, which enlarges the algorithm implementation scope. Introducing sleep modes in the PSoC helps it to ease its performance. If the MCU is not performing anything special, it can be set to the sleep mode, so that it draws minimal current and hence power. It is useful mainly in a battery operated application. The use of sleep modes varies from application to application. If an application is such that the MCU keeps on doing something all the time, then it is wise to use the suitable clock frequency to keep the MCU up on running all the time. Unlike this, if the MCU does some specific jobs for a period of time and does nothing the rest of the time, it is better to use the sleep mode that lets MCU work until a required period of time and then drives the MCU to the sleep mode.

The usual way to enter into the sleep mode is use the API function CyPmAltAct() [3]. However, there are different wakeup sources for the MCU to get it to the pre-sleep mode state. A pending interrupt service routine (ISR) could be one of the wakeup sources. There are three available categories of wakeup events: periodic, asynchronous and reset. A sleep timer, comparator and XRES are the respective examples of wakeup sources.

In the PSoC, timers and counters are more flexible than in a usual implementation. A sample C code implementation in a datasheet for each module makes it easy to write the code. PSoC also supports inline assembly, which consequently enhances the code implementation. If a certain function is not defined in the standard library file of the PSoC, an inline assembly of complex instruction set computer can be used.

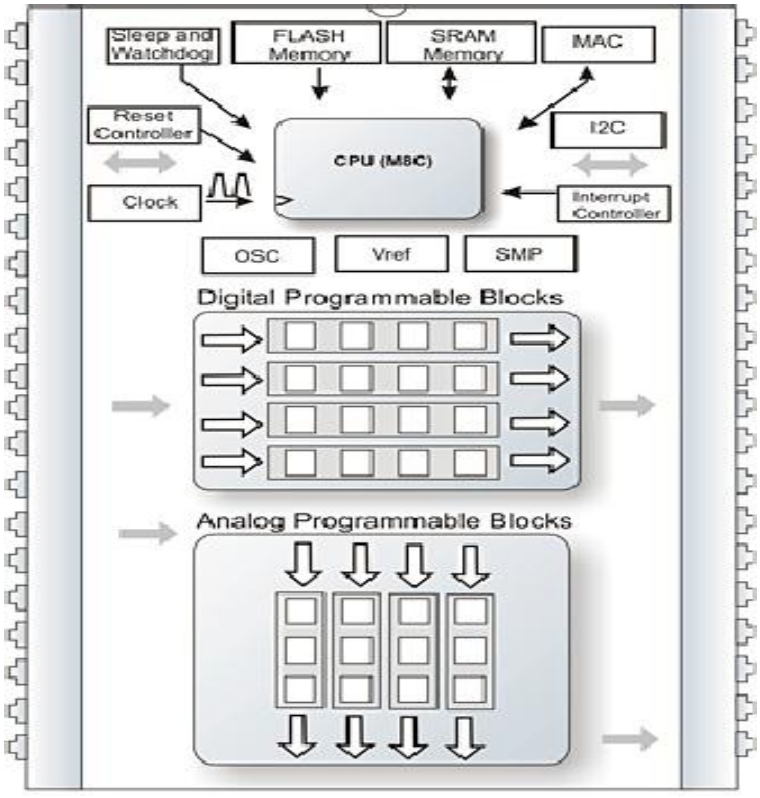

Figure 3: PSoC architecture
Reprinted from Scribd (2012) [2]

Figure 3 shows the internal architecture of PSoC microcontrollers. The central processing unit (CPU) is the brain of the microcontroller which controls the program execution. The frequency generator provides the required frequency to the CPU to work.

The reset controller enables the start of MCU and brings back the MCU to the right track if something unusual happens. The watch dog timer is used to detect software dead loops. Sleep timer can wake up the MCU periodically from power saving modes. [2]

The input/output pins enable communication between the CPU, digital blocks, analog blocks and the real world. The digital and analog blocks enable users to configure digital and analog programmable components added by users. The interrupt controller handles the interrupt action. The I2C (inter integrated circuit) controller enables the hardware realization of I2C communication. [2]

In figure 2, Vref refers to the voltage reference which acts as the frame of the reference for the voltage measurement in the analog operation. The MAC unit is responsible for multiplication and accumulation operation and the SMP is the symmetric multiprocessor which can be used as a part of the voltage regulator. For instance, it is possible to supply the voltage to the PSoC microcontroller from single 1.5 V battery. [2]

2.1.2 Central Processing Unit

The instructions during programming are stored in the flash memory. The CPU fetches each instruction from the flash memory, decodes it and executes accordingly. The CPU has internal registers such as a program counter, stack pointer, index register, flag register and arithmetic and logic unit, which are involved in a complete instruction execution [2].

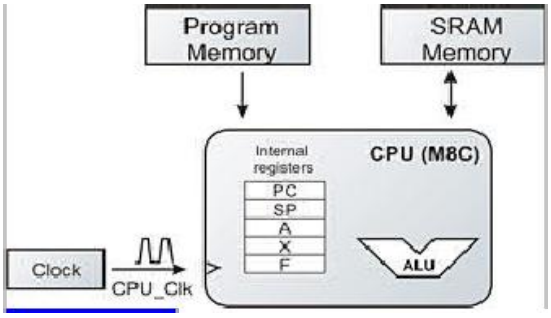

Figure 4: CPU of PSoC
Reprinted from Scribd (2012) [2]

The program counter keeps track of which instruction is executed and which one is on the way. The stack pointer points to the address of the static random access memory (SRAM) where data is written to or read from in case of any PUSH or POP operation. The accumulator handles all the arithmetic, logical and data transfer operations and the index registers act as an accumulator when large instructions are handled. The result of previously executed instruction is stored in flag register as a bit value. [2] The arithmetic and logical unit is used for the arithmetic and logical operation such as addition, subtraction or shifting.

2.1.3 Frequency Generator

A frequency generator is important to the CPU unit functioning and programmable blocks. Each programmable component has a certain operating speed. The fed in frequency signal should match the desired operating speed for the efficient result. PSoC microcontrollers have a system for generation of different frequency signals, which is done by selecting different parameters in the global resources window. Timers, counters or PWMs can be used to synchronize the clock further.

The system clock is the main internal clock signal with a speed of 24 MHz. It is used as a reference clock for most other signals. The global resources' parameters such as VC1, VC2 and VC3 further divide the system clock. The allowed dividing value for VC1 and VC2 is 16 or less, while that of VC3 is 256 or less. Even having done that, if desired clock frequency is not achieved, one can further divide it using timers or PWMs.

Apart from the internal system clock, the CPU has its own synchronizing clock for the instruction execution. It is used as a CPU unit frequency which has a direct impact on the instruction execution speed. The CPU clock can have any of eight frequencies that are in the range from 93.75 MHz to 24 MHz, which can be done by setting appropriate parameters in the Device Editor or during program runtime by selecting three lower bits of OSCCR0 register. [2]

2.1.4 PSoC Power Consumption

The CPU clock in the PSoC is responsible for the instruction execution speed. Doubling the frequency, the program executes twice faster. However, a higher frequency does not necessarily mean a better overall performance. A higher frequency could bring in an undesired effect on microcontroller power consumption. This results in problems with battery supply. Moreover, generation of electromagnetic interference increases with a higher frequency and that intrudes the surrounding devices.

It is imperative to bring power consumption to its lowest satisfactory functional level in battery-operated devices to maintain the longest possible uninterrupted work time. Microcontrollers often operate periodically. Power saving can be done by switching the microcontroller to the sleep mode while it has no important role. The microcontroller could be woken up from sleep mode only by reset or an interrupt.

2.1.5 Reset

Power on Reset (POR) can be done to avoid any unpredictable actions. If the supply voltage variation occurs, it is likely that the voltage drops beneath a certain limit. The microcontroller is switched into the POR mode and stays there until the voltage stabilizes above a critical limit defined by the Trip Voltage parameter.

The external reset (XRES) allows the user to switch the microcontroller to the start state by pressing a button. Reset is activated when the XRES pin reads the logic true. The simple reset circuit can be made with a pull-down resistor and one switch. The watch dog reset (WDR) is used for avoiding infinite loops or other irregularities by switching the system to the start state.

2.1.6 Digital Inputs and Outputs

Digital inputs and outputs pins connect the real world with the PSoC microcontroller. Each port has eight pins of which one is the power supply and the other is the ground. Regardless of the number of ports in microcontrollers, the read and write operations are done in the same fashion. Port access registers are stored inside the register address space noted as PRT0DR, PRT1DR, PRT2DR, PRT3DR, PRT4DR or PRT5DR.

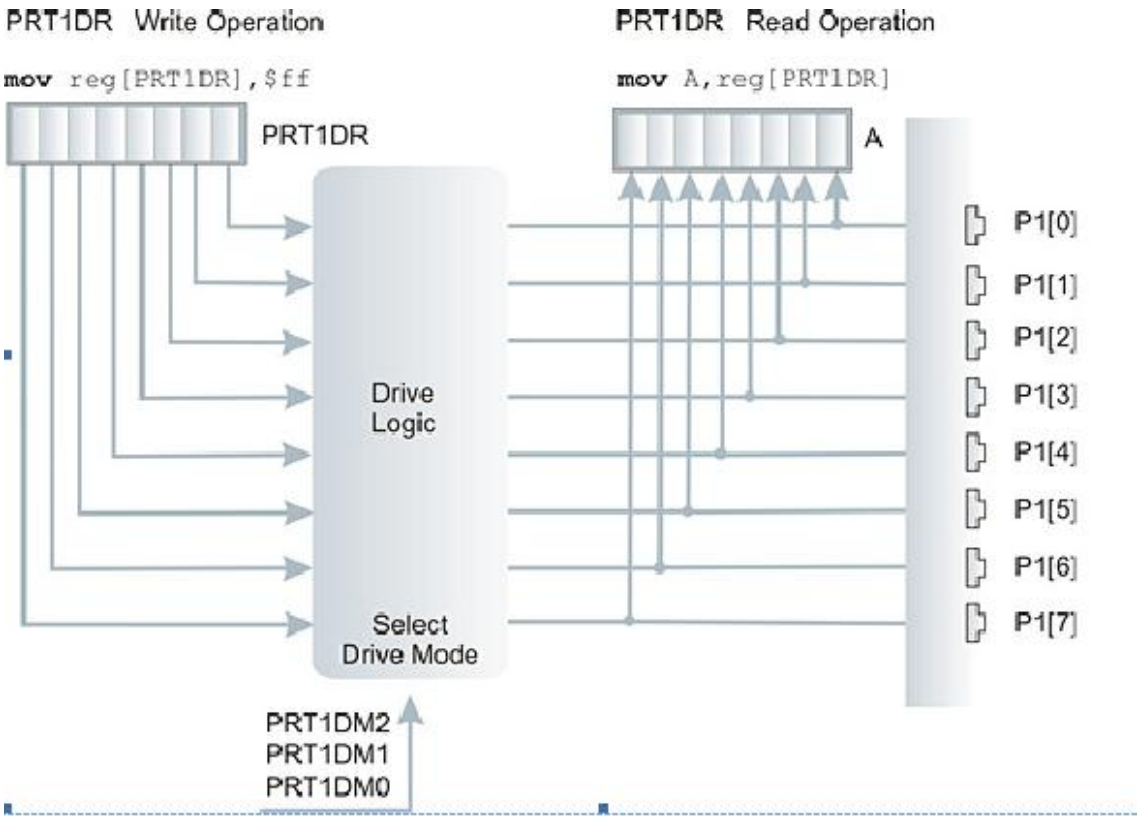

Figure 5: Write and read operation through PSoC port
Reprinted from Scribd (2012) [2]

The PSoC port pins can be accessed by directly writing to or reading from the port data (PRTxDR) registers. To write to a particular port pin, the corresponding mask and bit-wise AND or OR operation is done. For example, to set and clear pin 4 of port 1, the C code implementation is done in the following way:

```
PRT1DR |= 0x10;
PRT1DR &= ~0x10;
```

To read from a port pin, read the PRTxDR register and corresponding bit mask is used. For example, the C code implementation is done in the following way which reads the third pin of port one and processes the pin set state provided the condition is true.

```
if (PRT1DR & 0x08)
{
   // Code to process Pin Set state
}
```

The pin configuration can be done in two ways. One way to configure the pins is to define the configuration as part of the initialization in the Device Editor. It is easy to configure the pins if the pin configuration is the same at all times. The other way to configure is to use API. The pin configuration can be changed through API any times.

The drive mode for pins can be selected in the Device Editor or can be implemented through APIs. Setting up the drive mode during runtime can be done initializing the drive mode registers PRTxDM2, PRTxDM1 and PRTxDM0. The different drive modes available for PSoC pins are:

- **Strong** mode is used when it is needed to connect the state on the PRTxDR register directly to pins. This mode is used for outputs.
- **Analog Hi-Z** mode is used to connect analog signals like an ADC input. All internal connections between PRTxDR and pins are disconnected in this mode to ensure non-interference with the value of the brought voltage.
- **Pull-up** or **pull-down** resistors are used when input is fed in through some buttons. These resistors ensure the correct state when the button is not pushed.
- **Open drain** mode is used to maintain several devices to the same line when an external pull-up or pull-down resistor is added. [4]

2.1.7 Analog Inputs and Outputs

Some of input-output pins, besides their standard use, can perform an analog input or output operation. Any pin of port P0 as well as lower four pins of port P2 can be used as an analog input. Inputs of port P0 are connected to analog blocks over analog multiplexers, while in the case of port P2 they are connected directly to programmable switched capacitor blocks. Pins P2 [4] and P2 [6] can serve as external referent voltage inputs. [2] Outputs from analog blocks can be connected to 4 output buffers, which are connected to P0 [2], P0 [3], P0 [4] and P0 [5] pins.

2.1.8 Digital Programmable Blocks

Digital components such as timers, counters, PWM or Pseudo-Random Sequence Generator (PRS) can be configured. The digital communication protocol such as serial peripheral interface (SPI), I2C or a universal asynchronous receiver and transmitter (UART) are responsible for establishing communication among hardware compo-

nents. Most components such as counters or PRS can be stored inside any free block, while communication components such as UART or SPI can be set on the right side of the programmable blocks [2]. A digital programmable block is shown in figure 2 above.

The frequency signal is imperative for the digital components to work. Depending on the desired speed, there is a range of several built-in frequencies to choose from. The faster the frequency, the faster the speed is. However, it does not ensure overall better performance. A higher frequency means higher power consumption on the one hand and on the other, it results in higher interference with the surrounding devices which is often undesired. Therefore a suitable frequency is the one that satisfies the power consumption and interference issues.

2.1.9 Analog Programmable Blocks

Analog programmable blocks are grouped into columns of three programmable blocks. Depending on the family of the microcontroller, there could be 1, 2 or 4 analog columns. Each column has an input multiplexer, one frequency line, output analog and a comparator line [2]. The analog programmable block is shown in figure 2 above.

Some of the analog components such as AD converters or filters require frequency signal to determine the sampling rate of the analog signals that affect the component's speed. For each of the columns it is possible to select:

- internal frequency VC1
- internal frequency VC2
- output from the digital blocks such as PWM, counters or timers selected through a multiplexer.

2.2 Radio Transceivers

A transceiver is a device that comprises of a transmitter and a receiver with a common circuitry. In the past, there used to be a separate transmitter and a receiver device for any radio transmission, but at present most of the radio device is a transceiver. A walkie-talkie is an example of a radio transceiver.

Radio frequency (RF) is the rate of oscillation in the range of about 3 KHz to 300 GHz, which is the frequency range of radio waves [5]. RF is the acronym commonly used to represent radio. An RF transceiver uses radio frequency, which is responsible for transmitting and receiving data wirelessly. RF modules have an antenna that receives thousands of radio waves at a time. A resonator circuit amplifies the radio signals in the frequency band but attenuates the oscillation outside the frequency band and hence used to tune into a particular frequency band.

nRF24L01 Chip

The nRF24L01 is a highly integrated, ultra low power RF transceiver. It can have up to 2 Mbps on air data rate. It can operate at very low voltages. The supply voltages can vary from 1.9 to 3.6 volts. However they have an input tolerance of 5 volts. The transceiver nRF24L01 is marked by its key features which include Enhanced ShockBurst, Automatic packet handling, Auto packet transaction handling. [6] The nRF24L01 transceivers have a wide range of applications. They can be used as wireless computer peripherals, mice, keyboards and remotes, VoIP headsets, game controllers, sports watches and sensors, RF remote controls for consumer electronics, home and commercial automation, ultra low power sensor networks, active RFID, asset tracking systems or toys.

The nRF24L01 has a built-in state machine that controls the transitions between different operating modes of the chip. The state machine takes input from the user defined signals and register values. The nRF24L01 can be configured in four main modes of operation.

The nRF24L01 is disabled with minimal current consumption in the *power down mode*. The register values available from the SPI are maintained and the SPI can be activated in power down mode. The *power down* mode is entered by setting the PWR_UP bit in the CONFIG register low. The nRF24L01 should be in *power down* mode for at least 1.5 milliseconds to ensure settling time. [6]

The device enters *standby-I mode* by setting the PWR_UP bit in CONFIG register to 1. The nRF24L01 returns to this mode from the transmit (TX) or receive (RX) mode when the chip enable (CE) is set low. The device enters the *RX mode* and behaves as a re-

ceiver when the PWR_UP bit in CONFIG register is set high, the PRIM_RX bit is set high and the CE pin is set high. The transceiver must have the PWR_UP bit set high, the PRIM_RX bit set low, a payload in the TX FIFO (first-in-first-out) and the CE pin set high for more than 10 microseconds to enter *TX mode*. [6]

Enhanced ShockBurst uses ShockBurst for automatic packet handling and timing. The key features are:

- 1 to 32-byte dynamic payload length
- Automatic packet handling
- Auto-packet transaction handling
    - Auto-acknowledgement
    - Auto-retransmit. [6]

ShockBurst assembles the packet and clocks the bits in the data packet during transmit and during receive, it searches for a valid address in the demodulated signal. When it finds the valid address, it processes the rest of the packet and validates it by the CRC. If the packet is valid, the payload is moved into the RX FIFO. The high speed bit handling and timing is controlled by ShockBurst. [6]

The *data and control interface* allows access to all the characteristics in nRF24L01. The data and control interface consist of the following digital signals:

- IRQ (this signal is active low and is controlled by three maskable interrupt resources)
- CE (this signal is active high and is used to activate the chip in RX or TX mode)
- CSN (chip select SPI signal)
- SCK (serial clock SPI signal)
- MOSI (master out slave in SPI)
- MISO (master in slave out SPI). [6]

2.3 Electric Motors

An electric motor is an electromechanical device that converts electrical energy into mechanical energy. Electric motors work on the principle of electromagnetic induction. Electric motors are found in applications such as industrial fans, blowers and pumps, household appliances or disk drive. DC motors and AC motors are by far the most

common electric motors. On the other hand generators convert a mechanical energy into electric energy. [7] For example, generators use benzene to produce electric energy.

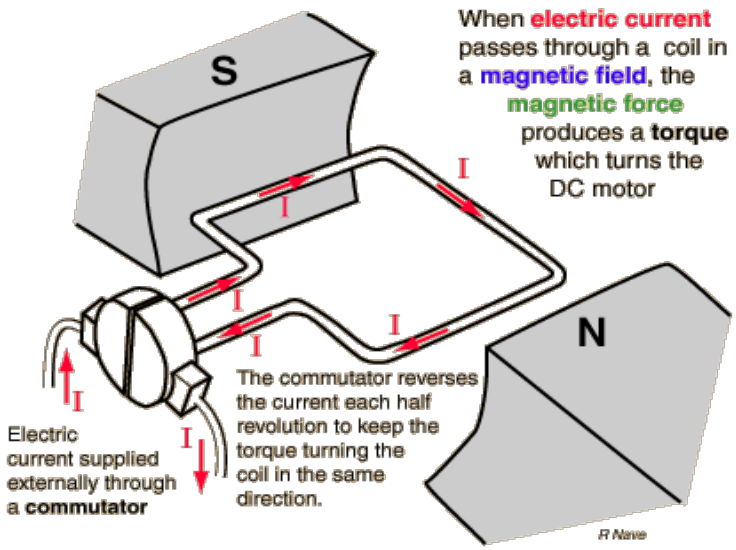

Figure 6: Basic operating principle for an electric motor
Reprinted from GSU (2008) [7]

Figure 6 shows how an electric motor works. There are two permanent magnets with opposite poles facing each other and a current conducting coil between those magnets. This coil is connected to a commutator through which the electric current is supplied to the coil. The commutator reverses the direction of the current so that the coil experiences a torque in the same direction as long as the electric current is passed through. A magnetic field is produced around a magnet. The direction of the magnetic field is from north to south. In figure 6, a magnetic field from north to south is set up. When a current carrying a conductor is brought into this field, it experiences a torque which is responsible for rotating the armature. [7]

The direction of the torque can be explained using Fleming's left hand rule. If the middle finger points in the direction of the magnetic field and the index finger points in the direction of the current then the thumb points to the direction of torque experienced in the coil. That means that the coil rotates in clockwise direction as illustrated in figure 6. The electric current supplied to the system as shown in figure 6, can either be direct current (DC) or an alternating current (AC). If the motor is designed to run with a DC power source, it is called a DC motor, and if it is designed to run with an AC power source, then it is called AC motor. The most common DC motors are brushed DC motor and brushless DC motors. [7]

2.4 Brushed DC Motors

A brushed DC motor has an armature. An armature is a set of wound coils which acts as an electromagnet with two poles. A commutator reverses the direction of the electric current flowing through the coil when the poles of the armature electromagnet pass the poles of the permanent magnet in the electric motor.

The advantages of the brushed DC motors are:

- Two-wire control (with the help of commutator)
- Replaceable brushes for extended life
- Low cost of construction
- Simple and inexpensive tool
- No controller is required for fixed speeds. [9]

The disadvantages are:

- Periodic maintenance is required
- Torque is moderately flat; at higher speeds, brush friction increases, thus reducing torque
- Poor heat dissipation due to internal rotor construction
- Higher rotor inertia, which limits the dynamic characteristics
- Lower speed range due to mechanical limitations of the brushes
- Brush arcing will generate noise. [9]

2.5 Brushless DC Motors

Brushless DC motors are synchronous motors powered by a DC source but an integrated inverter converts this DC power to AC to drive the motor. There are additional sensors and electronics to control the inverter output. Unlike brushed DC motors, brushless motors have rotating permanent magnets and a fixed armature. This eliminates the problem of connecting the current to the rotating armature. An electronic controller replaces the commutator and hence the brushes are not used.

The advantages of brushless motors are:

- Less required maintenance due to absence of brushes
- Torque is relatively higher

- High efficiency, no voltage drop across brushes
- Higher speed range
- Low electric noise generation. [9]

The disadvantages of the brushless motors are:

- Higher cost of construction
- Control is complex and expensive
- Electric controller is required to keep the motor turning. [9]

2.6 Servo Motors and Servo winch

Servo motors are the electric motors that work on the principle of magnetic induction. However the working mechanism is different. A servo motor is an assembly of a DC motor, a gear reduction unit, a position-sensing device and a control circuit. [10]

Servo has three terminals for the input. Two terminals are used to feed in voltage and ground while the third one receives a digital control signal from a MCU and causes the servo to produce torque accordingly. The control signal represents the desired output position of the servo shaft. The servo receives this control signal that represents the desired position and applies power to the embedded DC motor until its shaft turns to that position.

A position sensing device is used to determine the rotational position of the shaft. The degree of rotation of the shaft in is confined to certain angle. It turns back and forth approximately 200 degrees. The control signal is pulse width modulated. The positive pulse determines the position of the shaft. For instance, 1.52 milliseconds is the center position for the Futaba S148 servo [10]. A longer pulse turns the shaft clockwise from center and a shorter turns it counter clockwise.

The basic principle of servo motors and winch servos is the same. The external interface looks similar: one power signal, one ground signal and one digital control signal. Unlike a usual servo motors, a winch servo is not capable of detecting the position of the rotating shaft. A separate algorithm should be implemented to know the position of the shaft. The application of the servo motor can be seen widely where positional accuracy plays a role. Sometimes positional accuracy is not the only aim. The torque produced by the servo should be large enough to turn the load to the desired location.

The usual servo can rotate around 200 degrees. If the load is to be rotated more than 200 degrees like in a sail in a yacht, then the winch servo is useful.

Winch servos are more powerful than usual servos. The winch servo model VSD-22YMB can rotate around 2160 degrees that is 6 rotations around the central axis. As mentioned before, the positive pulse going through the control signal cable controls the direction of the servo motion. The neutral position for this type of servo is 1500 micro-seconds. The pulse travelling time is between 800 and 2200 microseconds and has a dead band width of 2 microseconds [11]. The downside of a winch servo is that positional precision is unknown. Some external means should be applied to detect the position of the rotating shaft.

2.7 Trolling Motor

A trolling motor is the marine propulsion system that contains an electric motor and propeller and steers the boat in water. By and large trolling motors are used for fishing. Moreover, they are also used as a primary source of propulsion for a smaller water craft such as canoes and kayaks. It is also known as an electric outboard motor. [13] Modern electric trolling motors are designed to operate at 12 volts, 24 volts or 36 volts. The electric motor is sealed inside a watertight compartment at the end of the shaft. This part is submerged during use, which prevents overheating. Depending on the price, trolling motors differ from one another. Hand controlled, foot controlled and remote controlled are the ones currently available on the market. Figure7 shows how a hand-controlled trolling motor looks like.

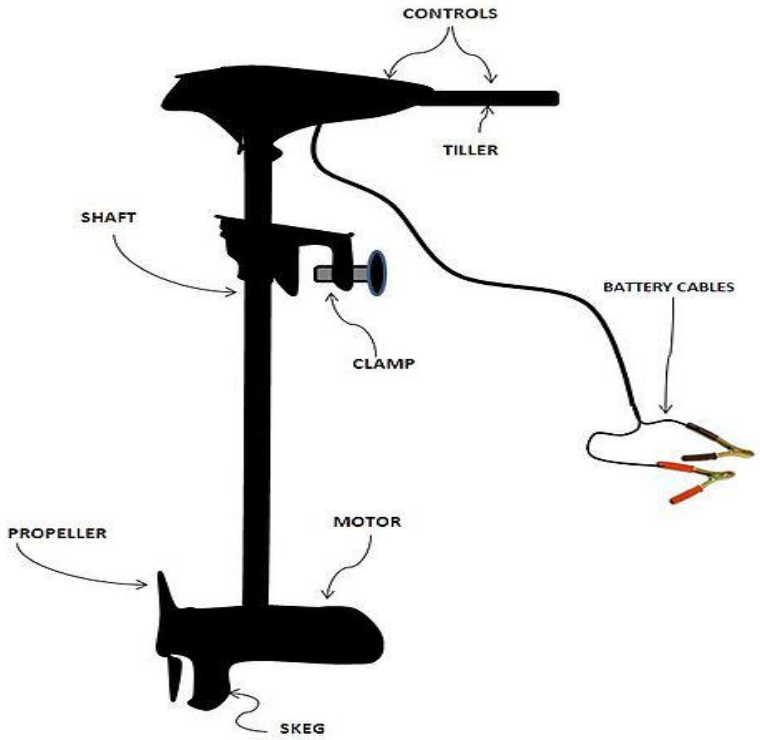

Figure 7: Hand controlled trolling motor
Reprinted from Aol Answers (2012) [12]

In figure 7, “Controls” shows the hand-control system. Rotating the tiller changes the speed and also causes the boat to move forward or backward. Moving the controls around the shaft causes the boat to move to the right or left. The trolling motors can be attached either to the bow or stern of the boat and voltage is supplied through battery cables.

The trolling motor has a DC motor inside it and there are three different cables connected to different resistors. The different possible combination of these resistors in turn changes the voltage drop across the load, which makes the load rotate at a different speed. The trolling motor used in this project was 36 pound Shakespeare trolling motor and it has five forward speeds and two reverse speeds and operates in 12 volt power supply [13].

3 Interfacing Microcontroller

Radio transceivers are physically connected with PSoC microcontrollers with some connecting wires but that does not ensure the communication between PSoC and the RF transceivers. There has to have some communication interface between them. The

digital communication protocols could be UART, I2C and SPI etc. The nRF24L01 is interfaced with PSoC through SPI.

3.1 Serial Peripheral Interface (SPI)

The SPI is a synchronous serial data link that operates in a full duplex mode shifting a bit at time [14]. Devices communicate in a master/slave mode where master device initiates the data frame. The microcontroller is the master and the RF transceiver operates as a slave. The SPI is depicted in figure 6. The master in figure 6 represents the microcontroller and the slave is the RF transceiver. SPI communication is used to communicate with a serial peripheral device or with another microcontroller with an SPI interface.

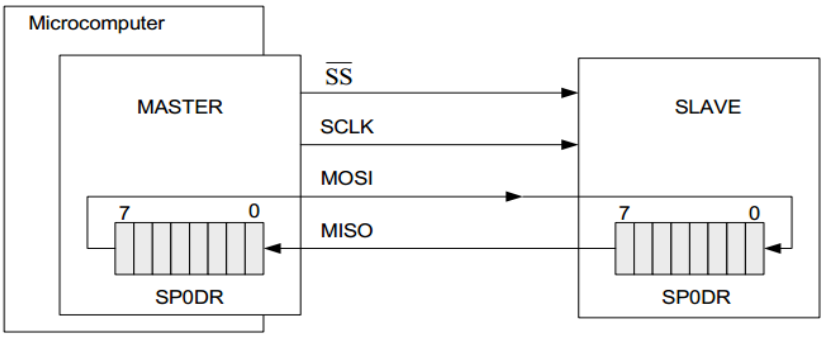

Figure 8: Serial peripheral interface
Reprinted from RPI (2012) [Serial Peripheral Interface, SPI]

In figure 8, shifting of one bit in or out at a time takes place. The bits are sent out from the master out slave in (MOSI) pin and received through the master in slave out (MISO) pin in the master SPI. The bits to be shifted are stored in the SPI data register and are sent out either the most significant bit (7^{th} bit) first or the least significant bit (0^{th} bit) first. This is done by writing to the 7^{th} bit of the SPIM control register CR0. The MSB first specifies that the MSB bit should be transmitted first. The 7^{th} bit of the slave shifts into bit 0 of the master through the MISO pin when the 7^{th} bit of the master is shifted out through the MOSI pin. This bit will eventually end up in bit 7 of the master after eight clock pulses. [14]

3.2 Serial Peripheral Interface in PSoC

Two digital communication modules are available in PSoC for SPI communication, SPI master and SPI slave. When configured for SPI, the data register DR0 acts as a shift register. All data is shifted in or out of this register. After a complete byte is received, the data is transferred to another data register DR2 that acts as one byte RX register. DR1 acts as one byte TX register in a similar fashion. The data to be transmitted is loaded in DR1 and shifted to DR0 during transmission. [15]

SPIM is the master module. There are several parameters to be configured. The clock selects the input clock source. The MISO is the input to the master, and the MOSI is the input to the slave device. SCLK selects the destination for the output clock signal. The actual SCLK frequency is half of the clock frequency because of clock division that occurs in the module. A slave select signal is created in the chip level design by designating a port pin of choice as “StdCPU” with “strong” drive mode. The slave select signal connects and enables the slave device with the master. Several slave devices can be connected to the master but only one is active at a time.

3.4 Field Effect Transistors and their Application

The metal-oxide-semiconductor field-effect transistor, popularly known as MOSFET, is a transistor used for amplifying or switching electronic signals. The MOSFET is a four terminal device but body of the MOSFET is connected to the source leaving source as one terminal. Gate and drain are other two terminals. Gate intakes the control signal and the drain takes one terminal from a load, the other being connected either to ground or power source. An H bridge is an electronic circuit that enables voltage to be applied in either direction across a load. An electric motor is often required to move in either direction. An H bridge makes it possible to supply the voltage in either direction and achieve the desired movement through the motor. H bridges are available as integrated circuits or can be built using discrete components like MOSFETs and diodes. [17]

Figure 9 shows how an H bridge circuit can be built up using discrete components.

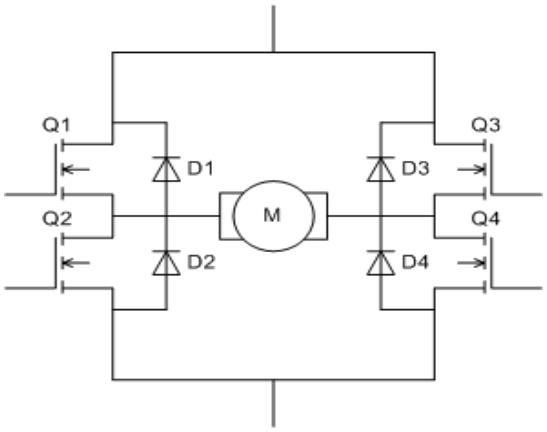

Figure 9: H bridge circuit
Reprinted from Modular Circuits (2011) [16]

In figure 9, Q1, Q2, Q3 and Q4 represent four different MOSFET switches. D1, D2, D3 and D4 represent four diodes. M represents electric motor. Q1 and Q3 are connected to voltage and Q2 AND Q4 are connected to ground. If Q1 and Q4 are closed, the motor rotates in clockwise direction and if Q3 and Q4 are closed the motor rotates in counter clockwise direction.The four catch diodes D1, D2, D3 and D4 seem to have no role while two or four switches are on. Once the switches are off, the induced field in the inductive loads looks for a way to collapse. These catch diodes provide a low resistive path for the induced field to collapse when the switches are off. [16]

There might be the case where unidirectional motion of the motor is desired. This complex H Bridge would only kill time and make the circuit complex then. A single MOSFET could be used to turn the motor on and off in that case. Either P channel MOSFET or N channel MOSFET could be used as a switch in such case. The working mechanism is the same however; the circuitry is different for N and P type MOSFET. Unlike in N channel MOSFET, source is connected to the power supply in P type MOSFET. The control signal from MCU goes to the gate in both the case. Load is connected between the drain and ground when P type MOSFET is used as a switch. Figure 10 shows how it is done using an N channel MOSFET.

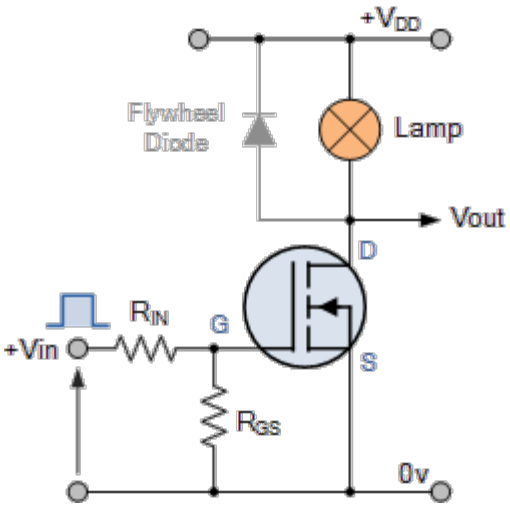

Figure 10: N channel MOSFET as a switch
Reprinted from MOSFET as a switch (2012) [17]

In figure 10, the PWM from MCU goes to the gate of the N channel MOSFET. The source is grounded and a load is connected between the drain and the voltage. A fly-wheel (catch) diode is connected across the load to provide a low resistive path for the collapsing field.

4 Implementation

Wireless control for the trolling motor involves two PSoC MCU, two RF transceivers (nRF24L01), trolling motor, push buttons and a winch servo (servo model number). The first step taken was to establish communication between a PSoC MCU and a RF transceiver. SPI interfacing protocol was used for the purpose. Three push buttons are used to toggle the input. Depending upon the output signal combination from three push buttons, MCU writes different values in the register which is read by the RF through SPI and sent to another RF. This read values are used to distinguish different speed conditions.

4.1 Push Button

Three push buttons named button_0, button_1 and button_2 were used. They have eight possible combinations. Depending upon the status of the push button, different values are written to the MCU register. These different values are sent to another MCU to generate different pulse widths. These different pulse widths drive the trolling motor

with different speeds. Push buttons were defined as directives. The pull down drive mode was used to ensure that the signal was pulled to the reference ground when switched off. This was implemented as follows:

```
#define button_0 ( Button_0_Data_ADDR & Button_0_MASK )
```

The above directive definition lets the program know that a button named button_0 is defined. A button with the same name is defined in the chip level design. The drive mode can be selected from the drop down list in the pin configuration window or by directly writing to the drive mode register. A pin status can be monitored through the button implementation as described above or can directly be read from the port data register. Reading and writing to a PSoC port is possible by directly accessing the port data register. Similar is the case for the drive mode implementation. Writing to or reading from a port is done using a corresponding bit mask and bitwise AND or OR operation. For example, to set and clear the fourth pin 4 of port 1, the C code implementation is as follows:

```
PRT1DR |= 0x10; //sets the forth pin port 1
PRT1DR &= ~0x10; //clears the forth pin port 1
```

Listing 1: Set and clear a pin of a port

To read from a particular port pin, the corresponding bit mask is used. For example to read the status of pin 3 of port 2, the C code implementation is as follows:

```
if (PRT2DR & 0x08)
{
    //Block of statment
}
```

Listing 2: Reading a pin status of a port

When three push buttons have been implemented, the possible combination would be eight. The following is the truth table for three push buttons.

Table 1: Push button conditions

Button_0	Button_1	Button_2	Output value
0	0	0	0x05
0	0	1	0x0A
0	1	0	0x0B
0	1	1	0x0C
1	0	0	0x0D
1	0	1	0x0E
1	1	0	0x0D
1	1	1	0x09

Table 1 shows eight possible push button combinations and hence values written to nRF24L01. If all three buttons are low, then the hex value 0x05 is written to nRF24L01 and sent to the other module of nRF24L01. The MCU reads this value and knows that three buttons are in the low state and hence accordingly generates a pulse width of 0, which is the case of the rest. Accordingly, when all buttons are pressed, 0x09 is sent and the MCU knows the condition and generates a PWM of the full duty cycle.

The drive mode configuration can be set either in the device editor or by using firmware. If the configuration is fixed all the time, it is a good idea to configure it in the device editor. The IO configuration can be changed at any time using firmware. For example if the drive mode strong is to be set for pin 4 of port 2, DM0=1, DM1=0, DM2=0, then the 5th bit of DM0 register is set and the same bit for the DM1 and DM2 registers are cleared [4].

Depending on the input read, the MCU prepares some output. For each button combination, the MCU assigns a unique value which is then written to the RF module (nRF24L01). This is done by using the SPI interface protocol.

4.2 PSoC and Radio Interface

The PSoC microcontroller uses the SPI master and slave protocol to interface RF transceivers [6]. The RF transceiver acts as a slave and the PSoC as a master. The SPI slave user module is added to the digital block. It performs full duplex synchronous

8-bit data transfers. The Devices communicate in the master/slave mode where the master device initiates the data frame. The microcontroller is the master and the RF transceiver operates as a slave.

Attention should be taken while doing the SPI read and write operation. The following timing diagram shows what should be taken into consideration for the SPI read and write operation. Every new command must be started by a high to low transition on the CSN. In parallel to the SPI command word applied on the MOSI pin, the STATUS register is shifted serially out on the MISO pin. The serial shifting SPI commands are in the following format:

```
<Command word: MSBit to LSBit (one byte)>
<Data bytes: LSByte to MSByte, MSBit in each byte first>. [6]
```

Listing 3: Command format for the SPI read and write operation

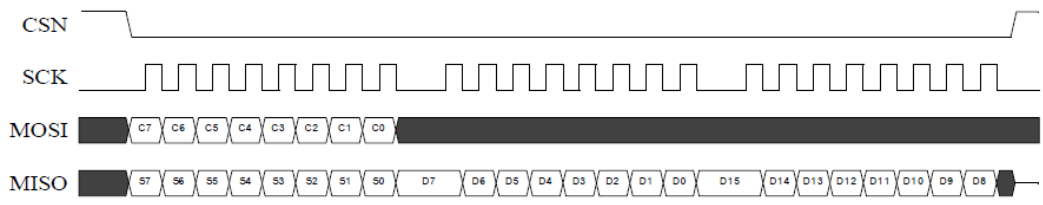

Figure 11: SPI read operation

Copied from nRF24l01 datasheet (2007) [6]

The read or write operation is done while the CSN is in a low state. In addition to the CSN signal, there are CE, SCK, MISO, MOSI, IRQ signals sharing the same port. Changing the pin status for the CSN could corrupt the pin configuration setting for other signals sharing the same port. It is therefore, the pin assigned for the CSN signal, should be controlled by the implementation of a shadow register.

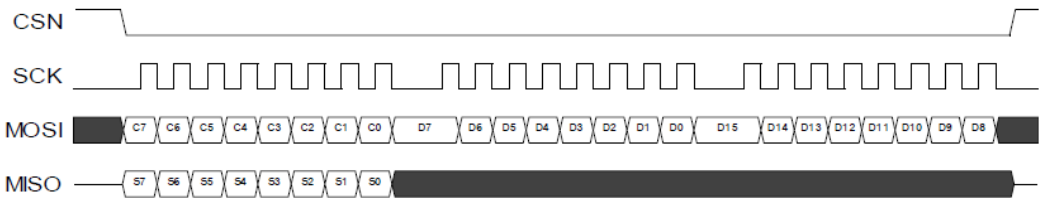

Figure 12: SPI write operation

Copied from nRF24l01 datasheet (2007) [6]

The SPI communication is initiated with the following function:

```
SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST)
```

A SPIM user module is added to the chip level design. The clock polarity (CPOL) and clock phase (CPHA) are two main parameters that define the SPI transfer modes of the clock to be used by the SPI. SPIM_SPIM_MODE_0 tells the MCU to start the SPIM in mode 0 that is CPOL and CPHA are both zero. In this mode, data sampling is done on the first edge of the clock and propagation occurs on the second edge.

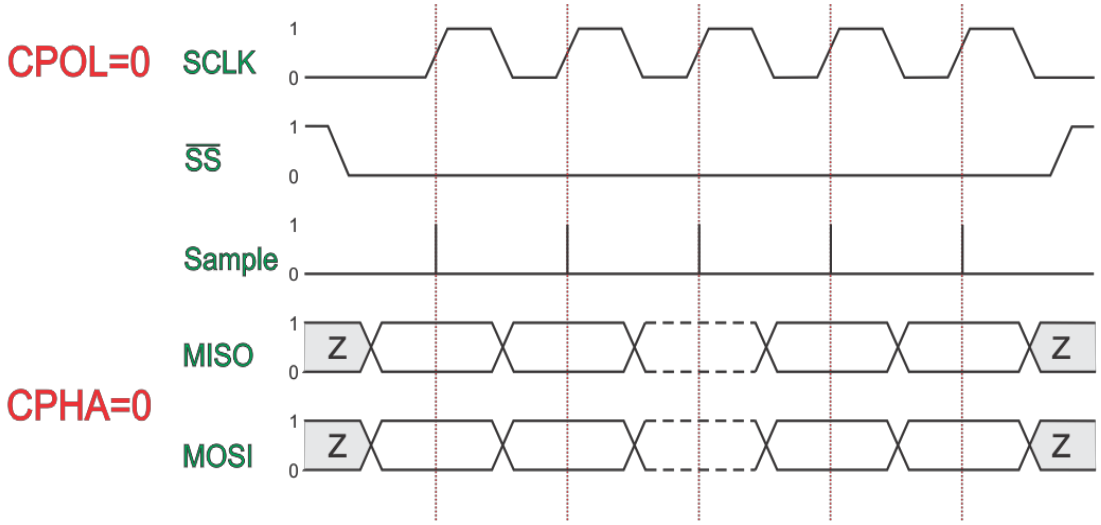

Figure 13: SPI transfer mode 0
Reprinted from SPI transfer modes (2012) [18]

Figure 7 shows how data sampling is done, when the SPI transfer mode 0 is used. Additionally, the SPI transfer mode 1, mode 2 and mode 3 are also available depending upon the status of CPOL and CPHA. When CPOL=0 and CPHA=1, it is mode 1; when CPOL=1 and CPHA=0, it is mode 2 and when CPOL and CPHA are both 1, it is mode 3. [18] SPIM_SPIM_MSB_FIRST tells the MCU to define SPI such that MSB data is read or written through SPI first.

4.3 Shadow Register

The ShadowRegs is a user module defined in PSoC. It creates a random access memory (RAM) variable (the shadow register) that caches values written to a port data register [19]. The shadow register enables the microcontroller's control of an individual GPIO pin without corrupting the settings of other GPIO pins sharing the same port. The ShadowRegs user module is software only and does not consume any PSoC blocks.

Once the shadow register is introduced, the data read or to be written is not directly read from or written to the port. The shadow register caches the information in a register and this register is copied to the required port. The C code implementation is as follows:

```
// nRF24_CE off
Port_0_Data_SHADE &= ~nRF24_CE_MASK;
PRT0DR = Port_0_Data_SHADE;

// nRF24_CE on
Port_0_Data_SHADE |= nRF24_CE_MASK;
PRT0DR = Port_0_Data_SHADE;
```

Listing 4: Shadow register code implementation

The nRF24_CE_MASK has a defined value and is written to the shadow register and copied to the port 0 of the microcontroller. The first code sets the CE signal low while the second code sets it high. The individual pin can be controlled in this way, which does not affect the settings of other pins sharing the same port. The concept of a shadow register can be implemented in the firmware without using a shadow register user module. A function which switches the state of the CSN signal and the CE signal is written and this is called whenever needed.

```
// Helper function for setting CSN off/on
void nRF24_CSN_state( BYTE state )
{
        if( state == CSN_ON) {
                PRT0DR |= nRF24_CSN_MASK;
        }
        else {
                //off
                PRT0DR &= ~nRF24_CSN_MASK;
        }
}

// Helper function for setting CE off/on
```

```
void nRF24_CE_state( BYTE state )
 {
          if( state == CE_ON) {
                    PRT0DR |= nRF24_CE_MASK;
          }
          else {
                    //off
                    PRT0DR &= ~nRF24_CE_MASK;
          }
}
```

Listing 5: CSN and CE signal state switching

The above code toggles the status of CSN and CE signals. Bitwise OR sets the state ON and bitwise AND sets the state OFF. Once these functions are written as a prototype, they can be called whenever the pin configuration of that particular pin has to be changed.

4.4 Radio Transmission

After interfacing nRF24L01 with the PSoC, the next task was to transmit values from this RF to the other. The nRF24L01 has a built-in state machine that controls the transition between different operating modes described in section 2.3. Figure 14 portrays the built-in state machine in nRF24L01 and how different modes can be accessed.

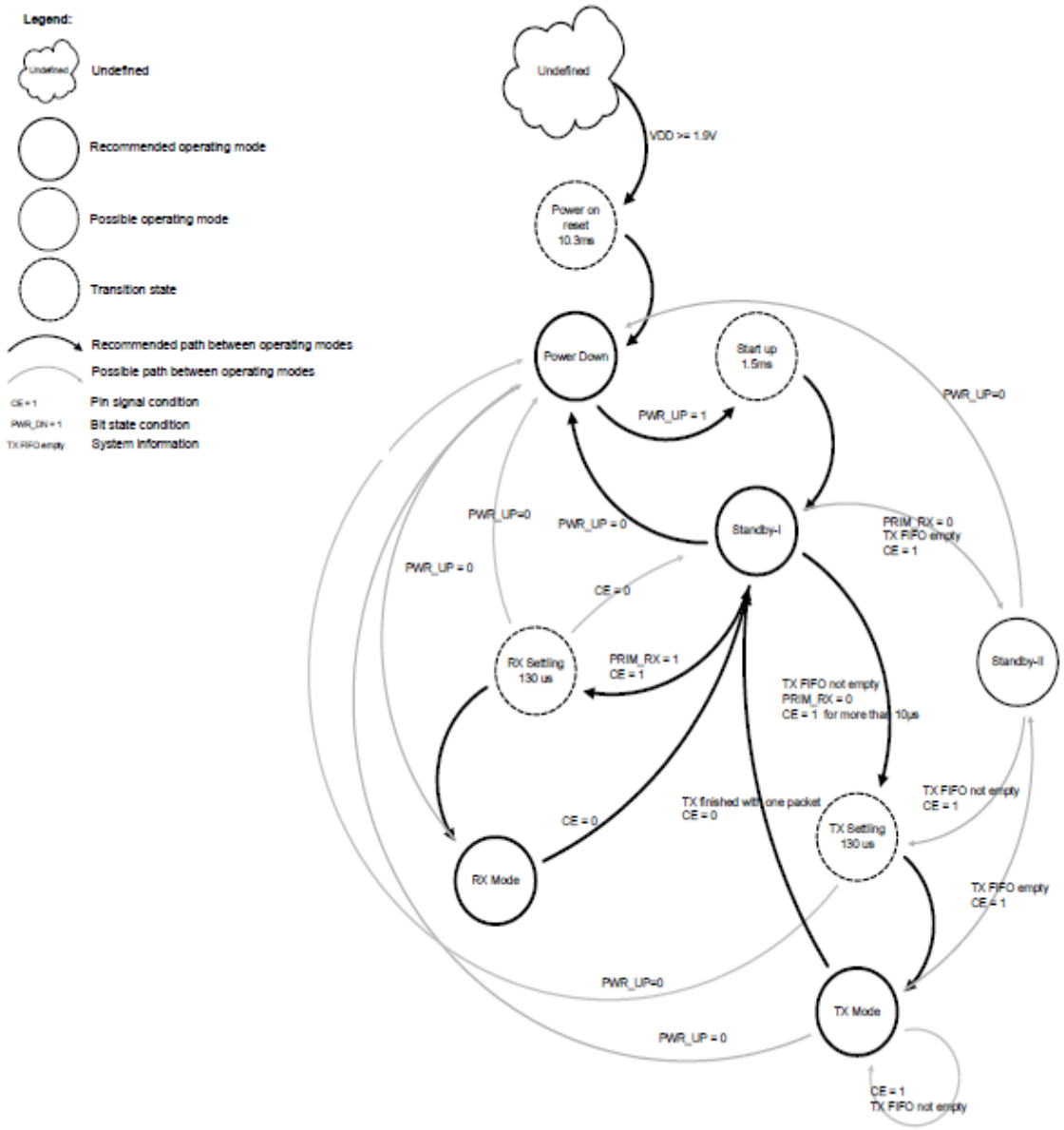

Figure 14: Operational modes state diagram
Reprinted from nRF24L01 datasheet (2007) [6]

As shown in figure 14, nRF24L01 is undefined until the power supply becomes 1.9 Volt or higher. It then enters in POR (power on reset) mode and remains so until 10.3 milliseconds or until it goes to the power down mode. The device goes to the Standby-I mode setting the PWR_UP bit in the CONFIG register. The Standby-II mode occurs if the CE is held high on a PTX device with an empty TX FIFO. Transmitting data is possible only if the device enters into the TX mode. Setting the CE signal high for 10 microseconds or more, setting the PRIM_RX low and a non empty TX FIFO make the device switch into the TX mode. The device stays in the TX mode until it finishes

transmitting the current payload. The CE low will switch the device to the Standby-I mode. A high CE signal means the successive action depends on the status of TX FIFO. According to the manufacturer, it is important to never keep the device in the TX mode for more than 4 milliseconds, so it is wise to enable auto transmit.

For a complete transmission, the other RF must be set as a receiver and should be able to receive what the transmitter transmits. It is done by setting the device in the RX mode. For this, the device must have the PWR_UP bit, the PRIM_RX bit and the CE pin set high. Sending data bytes from one RF module to the other is challenging. The address of the transmitter and the receiver must be the same. The communication fails unless the address is ensured to be the same, which is done in the code implementation. A brief example of how it is done is mentioned below:

```
// set both RX_ADDR_P0 and TX_ADDR addresses of the nRF24L01
// the default SETUP_AW, which is 5 bytes, is left unchanged

void nRF24L01_setRxTxAddr(void){

          BYTE i;
          BYTE RxAddrP0[5] = { 0x01, 0x01, 0x05, 0x01, 0x01};
          BYTE TxAddr[5] = { 0x01, 0x01, 0x05, 0x01, 0x01};

          // set the RX_ADDR_P0, 5 bytes
          nRF24L01_sendPayload(    W_REGISTER    |    RX_ADDR_P0,
RxAddrP0, 5);

          // set the TX_ADDR, 5 bytes
          nRF24L01_sendPayload(  W_REGISTER  |  TX_ADDR,  TxAddr,
5);

 }
```

Listing 6: Setting the same source and destination address

The number of registers should be defined in the correct way to establish the radio transmission. These registers occupy specific memory which was defined as directives

in a different header file. The different registers and their memory definitions are defined referring to the datasheet as below:

```
// Register addresses
#define CONFIG              0x00
#define EN_AA               0x01
#define EN_RXADDR           0x02
#define SETUP_AW            0x03
#define SETUP_RETR          0x04
#define RF_CH               0x05
#define RF_SETUP            0x06
#define STATUS              0x07
#define OBSERVE_TX          0x08
#define CD                  0x09
#define RX_ADDR_P0          0x0A
#define RX_ADDR_P1          0x0B
#define RX_ADDR_P2          0x0C
#define RX_ADDR_P3          0x0D
#define RX_ADDR_P4          0x0E
#define RX_ADDR_P5          0x0F
#define TX_ADDR             0x10
#define RX_PW_P0            0x11
#define RX_PW_P1            0x12
#define RX_PW_P2            0x13
#define RX_PW_P3            0x14
#define RX_PW_P4            0x15
#define RX_PW_P5            0x16
#define FIFO_STATUS         0x17
```

Listing 7: Register address

The reuse of the last transmitted data packet was done by setting the TX_REUSE bit high in the FIFO_STATUS register and the TX_REUSE is set by the SPI commands REUSE_TX_PL and reset by SPI commands W_TX_PAYLOAD or FLUSH_TX [6].

```
// SPI commands of the nRF24L01
#define R_REGISTER           0x00
#define W_REGISTER           0x20
#define R_RX_PAYLOAD         0x61
#define W_TX_PAYLOAD         0xA0
#define FLUSH_TX             0xE1
#define FLUSH_RX             0xE2
#define REUSE_TX_PL          0xE3
#define NOP                  0xFF
```

Listing 8: SPI commands of nRF24L01

Similarly, the interrupt mask was defined such that interrupts are reflected as active low on the IRQ pin.

```
// IRQ interrupt masks
#define MASK_RX_DR           0x40
#define MASK_TX_DS           0x20
#define MASK_MAX_RT          0x10
```

Listing 9: Interrupt masks

The status register mask for RX_DR and TX_DS was defined in the status register and whenever RX_DR is asserted that is new data arrives, RX_DR is cleared. When TX_DS is asserted referring that packet is transmitted on TX, TX_DS is also cleared.

```
// useful STATUS register masks
#define RX_DR_MASK           0x40
#define TX_DS_MASK           0x20
#define TX_FULL_MASK         0x01
```

Listing 10: Status register mask

Writing a dummy is required sometimes. For example, writing a dummy value to RF_CH register clears the packet loss counter and is hence defined as follow:

```
// general defines
#define DUMMYDATA              0x00
```

Listing 11: A dummy value directive definition

4.5 PSoC and Trolling motor Interface

The MCU reads in the input from push buttons, processes the output and this value is sent to the other MCU. Depending upon the value received, this MCU generates the PWM of a different duty cycle that generates a different speed for the trolling motor. A bridge circuit is required to feed in this PWM to the trolling motor. The nature of the bridge circuit depends on the desired functionality of the trolling motor. If the concern is a unidirectional motion with different speed, then a simple MOSFET circuit (where the MOSFET is used as a mere switch) is enough. Figure 10 showed how this circuit is connected. On the contrary, an H bridge circuit is required to get the motor turn in both directions. Naturally, the motor has a natural tendency to a convenient turn in one direction. Perceiving this fact, unidirectional rotation of the motor was taken into account. If the motor had to change the direction, the entire shaft holding the motor was rotated instead of changing the direction of rotor's motion. This was done using a winch servo. However, the movement was not smooth and as desired.

For the unidirectional motion with different speeds, the MCU can be interfaced with the trolling motor using either an N-type or a P-type MOSFET. N-type MOSFET was used in the project. A circuit as shown in figure 10 is connected between the MCU and the trolling motor. The PWM signal coming out of the MCU is fed into the gate terminal of the N-type MOSFET. The trolling motor is connected across the drain and the positive terminal of 12 V power supply. The source of the MOSFET is grounded. The PWM coming out of the MCU controls the power supply to the trolling motor. If the duty cycle of the PWM signal is 50%, the power supply is allowed for the half of the period. This brings the effect that the trolling motor rotates with half speed.

5 Results and Discussion

The goal set for the project was met partially. Each button input tells the MCU to write a unique value in the payload and send it to the other module. Based on the received value, the other MCU generates a PWM of a different duty cycle which rotates the trolling motor. The greater the generated duty cycle is, the higher the speed of the trolling motor is. Table 2 shows what happens in detail when an input is experienced.

Table 2: Result representation

BUTTON PRESSED	LCD MESSAGE TO IDENTIFY PRESSED BUTTON	DATA FED INTO THE PAYLOAD	DATA RECEIVED	GENERATED PWM DUTY CYCLE (%)
None	off	0x05	0x05	0
button_0	A	0x0A	0x0A	20
button_1	B	0x0B	0x0B	30
button_2	C	0x0C	0x0C	45
button_0 & button_1	AB	0x0D	0x0D	60
button_0 &button_2	AC	0x0E	0x0E	75
button_1 & button_2	BC	0x0F	0x0F	90
button_0 & button_1 & button_2	ABC	0x09	0x09	100

Table 2 shows the input button combination and the succeeding result. When button_0 is pressed, message “A” is printed in the LCD of the transmitting module. This will feed in the hex value 0x0A in the payload to be transmitted and consequently the value is transmitted to the receiver module. The receiver module will then generate a PWM signal with a 20% duty cycle. Table 2 in the same fashion shows different results for

the different input. The third column in table 2 shows the data fed into the payload to be transmitted. Technically this data is written through the SPI.

The designed prototype is advantageous because of its flexibility regarding distant use. One can control the speed from any part of the boat. The system is reliable and user friendly. It has three input buttons and eight possible input combinations. Each button combination has a unique speed situation. Since the input feeding is confined with button combination, there is no confusion for the user.

Two sets of MCUs and two sets of RF transceivers can convert a manual control trolling motor into a remote controlled trolling motor. This saves hundreds of euros from an economic perspective. Moreover it can be customized as per requirement, for example the speed situations and the degree of rotation of the rotor.

The goal set for the project was not achieved. However 80% of the job was done. Wireless control of the speed was achieved but the degree of rotation could not be controlled wirelessly. The motor used for the rotation of the trolling motor shaft is a winch servo motor model VSD-11YMB. Since this motor was used, it was not possible to locate the rotation degree and hence the rotation part was a failure. However, the servo aforementioned was able to rotate the shaft. The mechanical placement of the winch servo was also a problem. During testing, it was placed on the table and supported manually.

Implementation Issues

The first idea was to implement a knob that could be rotated by the user to control the speed. This has an underlying potentiometer which would produce a different voltage when the knob is rotated. This analog voltage is translated into a digital value by an inbuilt ADC in the MCU and these different digitized values are used to distinguish different speed situations. While implementing the above concept, the ADC did not function properly. The problem was with the code implementation. The program got stuck in the following FOR loop which was used in a function that defines the payload transmit.

```
for(i = 0 ; i < bytes ; i++)

{
```

```
            while(          !          (SPIM_bReadStatus()          &
            SPIM_SPIM_TX_BUFFER_EMPTY ) ){};

            SPIM_SendTxData( dataArr[i] );


            while(SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        SPIM_bReadRxData();

}
```

Listing 12: For loop defined within payload transmit function

It was disclosed that the program got stuck in the code implementation in listing 12 and listing 13. However, the reason why it occurred could not be disclosed. The same problem happened with the author's colleague using the same RF device and hence the concept was switched to the push button implementation.

The part in the code where data transmission was defined had a small delay.

```
void transmit_data(void){

            while( nRF24L01_is_TX_DS() != 1 ){

            nRF24L01_sendPayload(W_TX_PAYLOAD, payloadData, 2);

            nRF24L01_transmitPayload();


                        if( nRF24L01_is_MAX_RT() ){
                        nRF24L01_clear_MAX_RT();
                        break;  // This breaks out of while loop
                        }
            }
            // Clear the interrupt status
```

```
            nRF24L01_clear_TX_DS();
            // small delay
            for (i=0;i<60000;i++);

}
```

Listing 13: Function definition for data transmit

This delay was implemented to make sure that after a successful transmission of a data packet, the RF would wait for a small time before it transmits the other packet. After this delay loop was implemented, the transmission was not as expected. A single data packet was transmitted and the program was stopped. However, a reset on the receiver MCU would accept a succeeding data packet. The data packet transmission was successful only after having the reset button pressed. A simple solution was to check what happens when the delay loop in listing 12 is deleted and interestingly it worked perfectly.

The IRL530 bridge circuit was built to connect the control signal and the trolling motor. The 12 volt power supply was then synchronized by the control signal fed into the base terminal of the MOSFET. Figure 10 showed how the circuit was implemented. However, the catch diode was not used at the beginning, which brought in the massive heating of the MOSFET in a short interval of time. It was because the load was an inductive load and as the power goes off, the induced field looks for a collapse. The introduction of a catch diode in the circuit, as shown in figure 10, solved the problem as the diode provided a low resistive collapse path.

6 Conclusions

The goal of the project was to develop a wireless control mechanism for a trolling motor. A manually controlled trolling motor was used and customized to accept the user input. MCUs and RF transceivers were used to make it wireless. Controlling a trolling motor wirelessly means controlling its speed and degree of rotation. Initially the motor had five forward speeds and two reverse speeds. For the reverse motion, the idea was to rotate the motor shaft itself through 180, degrees which could not be met. However, a forward motion was achieved.

A different input button combination would send unique values to the receiving side and having based on the received value, the MCU would generate a PWM with a unique duty cycle. All three buttons at its “off” state would send a hex value 0x05 and the receiving side would generate a PWM with a 0% duty cycle and the system would be at off state. In the similar fashion 20, 30, 45, 60, 75, 90 and 100 % duty cycle was used for each different input button combination.

A wireless trolling motor in the market costs hundreds of Euros while a hand-control trolling motor is considerably cheaper. Buying a manual control trolling motor and implementing the designed prototype would save hundreds of euros. Not only the economic side but also the technical customization is equally flexible with this concept. The speed situation and degree of rotation can be customized. One can enjoy the unconfined movement within a boat while fishing by using design developed in this project, provided that the design is further developed to control the degree of rotation of the motor.

Technical customization and lower purchase expenditure are the main advantages of the design. However the RF transceivers used, operate at their best within a certain range. The range is not described in the datasheet but after the module was self-tested, it seemed that the module operates best within a 20 meters distance. The air data rate of the module is 2 Mbps at most, so it can never exceed this transmission rate unless the RF module is replaced. The technical customization is not confined within the speed conditions and the degree of rotation. One can implement “Auto Pilot” mechanism, the GPS (global positioning system) guided system and auto-obstacle detecting system if the project is to be further carried.

References

1. Kohlhard C. and Dickson C. Gliffy. [online]. California, USA: Gliffy Inc; 2012.
   URL: http://www.gliffy.com/gliffy/#templateId=blank&signup=1
   Accessed: August 29, 2012

2. Scribd Inc. Introduction to PSoC. [online]. San Francisco, USA: Scribd Inc; 2012.
   URL: http://www.scribd.com/doc/28429985/Chapter-1-Introduction-to-PSoC
   Accessed: September 5, 2012

3. Cypress Semiconductor Corp. Power Savings Using Sleep Mode. [online]. CA, USA: Cypress Semiconductor Corp.; 2012.
   URL: http://www.cypress.com/?docID=38262
   Accessed: November 20, 2012

4. Cypress Semiconductor Corp. PSoC IO Pin-Port Configuration. [online]. CA, USA: Cypress Semiconductor Corp.; 2007.
   URL: http://rtds.cs.tamu.edu/web_462/labs/port_IO.pdf
   Accessed: September 29, 2012

5. Stimac T. Definition of Frequency Band. [online]. Italy: Renato Romero; 2003.
   URL: http://www.vlf.it/frequency/bands.html
   Accessed: September 29, 2012

6. Nordic Semiconductor. Ultra Low Power 2.4 GHz RF Transceiver. [online]. Trondheim Norway: Nordic Semiconductor; July 2007.
   URL: http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01
   Accessed: June 15, 2012

7. Georgia State University. How does an electric motor work. [online]. Atlanta, Georgia: Georgia State University; May 17, 2008.
   URL: http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/mothow.html
   Accessed: September 16, 2012

8. Bailey F. B. The Induction Motor. New York, USA: McGraw-Hill Book Company; 1911.

9. Dynetic Systems. Brushless vs Brushed Motor. USA: Dynetic Systems; 2012.
   URL: http://www.dynetic.com/brushless%20vs%20brushed.htm
   Accessed: November 20, 2012

10. Sawicz D. Hobby Servo Fundamentals. [online]. New Jersey, USA: Princeton University; 2002.
    URL: http://www.princeton.edu/~mae412/TEXT/NTRAK2002/292-302.pdf
    Accessed: November 20, 2012

11. Vigor Precision Ltd. VSD-11 YMB Servo. [online]. Hong Kong: Vigor Precision Ltd; 2012.
    URL: http://www.vigorprecision.com.hk/uploadfile/20120530/20120530163258416.pdf
    Accessed: November 23, 2012

12. Yedda Inc. Aol Answers. [online]. USA: Aol Company; 2012.
URL: http://aolanswers.com/questions/
how_to_make_a_trolling_motor_bracket_for_a_gheenoe_p627801525375432
Accessed: September 13, 2012

13. Williams M. Fishing style, location determine best trolling motor choice. [online]. Kentucky, USA: FLW Outdoors; 2008.
URL: http://www.flwoutdoors.com/fishing-articles/tech-tackle reviews/148958/
trolling-motors-101/#.UMG9PeQUvyl
Accessed: June 25, 2012

14. Kraft R. Serial Peripheral Interface, SPI. [online]. USA: Rensselaer Polytechnic Institute; 2012.
URL: http://www.rpi.edu/dept/ecse/mps/SPI.pdf
Accessed: July 9, 2012

15. Cypress Semiconductor Corp. Getting started with SPI in PSoC. [online] . USA: Cypress Semiconductor; 2011.
URL: http://www.cypress.com/?docID=32340
Accessed: September 13, 2012

16. Tantos A. Modular Circuits. [online]. Hungary: The H-Storm Project; 2011.
URL: http://modularcircuits.tantosonline.com/blog/articles/old-h-bridge-secrets/
part-1/
Accessed: June 15, 2012

17. Storr W. MOSFET as a Swtich. [online]. Samoa: Electronic tutorials.ws; 2012.
URL: http://www.electronics-tutorials.ws/transistor/tran_7.html
Accessed: June 15, 2012

18. Diolan. SPI Transfer Modes. [online]. Israel: Diolan; 2012.
URL: https://www.diolan.com/dln_doc/spi-transfer-modes.html
Accessed: September 12, 2012

19. Cypress Semiconductor Corp. Shadow Registers. [online]. USA: Cypress Semiconductor; 2012.
URL: http://www.cypress.com/?docID=40322
Accessed: July 13, 2012

20. Ball B. Everything You Need to Know about the nRF24L01 and MiRF-v2. [online]. Brennen Ball; 2007
URL: http://www.diyembedded.com/tutorials/nrf24l01_0/nrf24l01_tutorial_0.pdf
Accessed: September 5, 2012

21. Comer D. Electronic Circuit Design. New Jersey, USA: Brigham Young University; 2003.

22. Saslow W. Electricity, Magnetism and Light. Canada: Transcontinental Gagne; 2002.

Appendix 1: Program code - Transmit side

```
//-------------------------------------------------------------------
// Hesinki Metropolia UAS
// Department of IT
// Embedded System Engineering (2012)
//-------------------------------------------------------------------

/*********************************************************************

This program transmits 2 bytes of payload through nRF24L01 module. SPI
protocol is used to interface the RF with MCU. The transmission here
is half duplex.

*********************************************************************/


#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "PSoCGPIOINT.h"
#include "nRF24L01.h"


#define button_0 ( Button_0_Data_ADDR & Button_0_MASK )
#define button_1 ( Button_1_Data_ADDR & Button_1_MASK )
#define button_2 ( Button_2_Data_ADDR & Button_2_MASK )



// Local prototypes

void feed_payload(BYTE data);
void displayPayload( BYTE *payload );
void transmit_data(void);

BYTE payloadData[2] ={0,255};
unsigned int i, loop, count;
char theStr[] = "transmit";
char blank[]="    ";



void main(void)
{

            M8C_EnableGInt;

            // initialization: SPIM is needed by RF-module
            SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);

//          PGA_SetGain(PGA_G1_00);
//          PGA_Start(PGA_MEDPOWER);
//
//          ADCINC_Start(ADCINC_HIGHPOWER); // Apply power to the SC
            Block
```

```
//          ADCINC_GetSamples(0); // Have ADC run continuously


    LCD_Start();
    LCD_Position(0,7);              // Place LCD cursor at row 0, col 5.
    LCD_PrString(theStr);           // Print "PSoC LCD" on the LCD

            // This is the transmitter side, but just to test mode
            changing we first make this receiver

//          // Initialize RF module
//          nRF24L01_commonConfiguration();
//          // Set as a trasmitter
//          nRF24L01_SetAsReciever();

            // Initialize RF module
            nRF24L01_commonConfiguration();
            // Set as a trasmitter
            nRF24L01_SetAsTransmiter();

//          LCD_Position(0,5);              // Place LCD cursor at row 0,
            col 5.
//          LCD_PrCString("DD");           // Print "PSoC LCD" on the LCD




            while (1) {

            //payloadData[1]--;

//**************************************************
            if (!button_0 && !button_1 && !button_2)

            {


                        feed_payload(0x05);
                        LCD_Position(0,0);//  Place LCD cursor at row 0,
                                              col 0.
                        LCD_PrCString("off");          // Print "PSoC LCD"
                                                          on the LCD


            }



            else if (button_0 && !button_1 && !button_2)

            {

                        feed_payload(0x0A);
                        LCD_Position(0,0);              // Place LCD cursor
                                                          at row 0, col 0.
                        LCD_PrCString("A");           // Print "PSoC LCD"
                                                          on the LCD
```

```
}



else if (button_1 && !button_0 && !button_2)

{

            feed_payload(0x0B);
            LCD_Position(0,0);              // Place LCD cursor
                                               at row 0, col 0.
            LCD_PrCString("B");           // Print "PSoC LCD"
                                               on the LCD
}



else if (button_2 && !button_1 && !button_0)

{

            feed_payload(0x0C);
            LCD_Position(0,0);              // Place LCD cursor
                                               at row 0, col 0.
            LCD_PrCString("C");           // Print "PSoC LCD"
                                               on the LCD
}


else if (button_0 && button_1 && !button_2 )

{


            feed_payload(0x0D);
            LCD_Position(0,0);              // Place LCD cursor
                                               at row 0, col 0.
            LCD_PrCString("AB");          // Print "PSoC LCD"
                                               on the LCD

}


else if (button_0 && button_2 && !button_1 )

{

            feed_payload(0x0E);
            LCD_Position(0,0);              // Place LCD cursor
                                               at row 0, col 0.
            LCD_PrCString("AC");          // Print "PSoC LCD"
                                               on the LCD

}

else if (button_1 && button_2 && !button_0 )

{
```

```
                        feed_payload(0x0F);
                        LCD_Position(0,0);              // Place LCD cursor
                                                           at row 0, col 0.
                        LCD_PrCString("BC");           // Print "PSoC LCD"
                                                           on the LCD

            }

//          if (button_0 && button_1 && button_2)
            else

            {

                        feed_payload(0x09);
                        LCD_Position(0,0);              // Place LCD cursor
                                                           at row 0, col 0.
                        LCD_PrCString("ABC");          // Print "PSoC LCD"
                                                            on the LCD

            }

//************************************************************




// The code below is just for testing fullduplex transmission.
// Do not try to make it work.

/*                      // Set as receiver
                        nRF24L01_SetAsReciever();

                        count = 0; // count receives

                        // this will end when data has been received 10
times
                        while( count < 10 ) {
                                    // check if something has been re-
ceived
                                    if( nRF24L01_is_RX_DR() == 1 ){
                                                // read payload
                                                nRF24L01_readPayload(
R_RX_PAYLOAD, payloadData, 2);
                                                // clear interrupt
                                                nRF24L01_clear_RX_DR();
                                                // Print it out
                                                displayPayload( payloadData
);
                                                count++; // count 10 re-
ceives and start transmitting
                                    }
                        }
*/

            }
```

```
}

void displayPayload( BYTE *payload )

{
            LCD_Position(1,0);
            LCD_PrCString("P1: " );
            LCD_Position(1,4);
            LCD_PrHexByte(payload[0]);
            LCD_Position(1,7);
            LCD_PrCString("P2: " );
            LCD_Position(1,10);
            LCD_PrHexByte(payload[1]);
}

void transmit_data(void)
{
            while( nRF24L01_is_TX_DS() != 1 ){


            nRF24L01_sendPayload(W_TX_PAYLOAD, payloadData, 2);

            nRF24L01_transmitPayload();

                        if( nRF24L01_is_MAX_RT() ){

                                    nRF24L01_clear_MAX_RT();
                                    break;  // This breaks out of while
                                                loop
                                    }
                        }

            nRF24L01_clear_TX_DS();// Clear the interrupt status


    }

void feed_payload (BYTE data)

{
            payloadData[0]=data;
            transmit_data();

            displayPayload( payloadData );
            LCD_Position(0,0);
            LCD_PrString(blank);

}
```

Appendix 2: Program Code - Receive Side

```
//-------------------------------------------------------------------
// C main line
//-------------------------------------------------------------------
/*********************************************************************

This program receives the data through nRF24L01. The RF is made to re-
ceive only 2 bytes of data in this case. However sendpayload function
can transmit 5 bytes at a time in this case. Literally, this RF can
handle 32 bytes of data.

*********************************************************************/


#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "PSoCGPIOINT.h"
#include "nRF24L01.h"
#include "PWM8.h"
#include "PWM16.h"


// Local prototypes

void displayPayload( BYTE *payload );

int j;

void main(void)

{
            BYTE payloadData[2] ={0,0};
            BYTE i;
            char theStr[] = "receiver";

            // initialization: SPIM is needed by RF-module
            SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);

            M8C_EnableGInt;

            LCD_Start();
            LCD_Position(0,5);            // Place LCD cursor at row 0,
                                             col 5.
            LCD_PrString(theStr);         // Print "PSoC LCD" on the
                                             LCD


            PWM8_Start();


            PWM8_DisableInt();
            PWM8_WritePeriod(200);
```

```
//PWM16_WritePeriod(1500);

// Initialize RF module
nRF24L01_commonConfiguration();

// Set as receiver
nRF24L01_SetAsReciever();


while(1)

{


// check if something has been received

if( nRF24L01_is_RX_DR() == 1 )

{

// read payload
nRF24L01_readPayload( R_RX_PAYLOAD, payloadData, 2);
nRF24L01_clear_RX_DR();
           switch (payloadData[0])

           {

           case 0x05:
                      displayPayload( payloadData );

                      PWM8_WritePulseWidth(0);

                      PWM16_WritePulseWidth(2800);

                      PWM16_DisableInt();

                      PWM16_Start();

                      break;

           case 0x0A:


                      displayPayload( payloadData );

                      PWM8_WritePulseWidth(40);

                      PWM16_WritePulseWidth(3000);

                      PWM16_DisableInt();

                      PWM16_Start();

                      break;

           case 0x0B:
```

```
                displayPayload( payloadData );
                PWM8_WritePulseWidth(60);
                PWM16_WritePulseWidth(3200);
                PWM16_DisableInt();
                PWM16_Start();
                break;
            case 0x0C:
                displayPayload( payloadData );
                PWM8_WritePulseWidth(90);
                PWM16_WritePulseWidth(3400);
                PWM16_DisableInt();
                PWM16_Start();
                break;
            case 0x0D:
                displayPayload( payloadData );
                PWM8_WritePulseWidth(120);
                PWM16_WritePulseWidth(3600);
                PWM16_DisableInt();
                PWM16_Start();
                break;
            case 0x0E:
                displayPayload( payloadData );
                PWM8_WritePulseWidth(150);
                PWM16_WritePulseWidth(3800);
                PWM16_DisableInt();
                PWM16_Start();
                break;
            case 0x0F:
```

```
                                displayPayload( payloadData );
                                PWM8_WritePulseWidth(180);
                                PWM16_WritePulseWidth(4000);
                                PWM16_DisableInt();
                                PWM16_Start();
                                break;
                        case 0x09:
                                displayPayload( payloadData );
                                PWM8_WritePulseWidth(250);
                                PWM16_WritePulseWidth(4200);
                                PWM16_DisableInt();
                                PWM16_Start();
                                break;
                        default:
                                displayPayload( payloadData );
                                PWM8_WritePulseWidth(0);
                                PWM16_WritePulseWidth(5000);
                                PWM16_DisableInt();
                                PWM16_Start();

                                }
                        }
                }
}

void displayPayload( BYTE *payload )

{
                LCD_Position(1,0);
                LCD_PrCString("P1: " );
                LCD_Position(1,4);
                LCD_PrHexByte(payload[0]);
                LCD_Position(1,7);
                LCD_PrCString("P2: " );
                LCD_Position(1,10);
                LCD_PrHexByte(payload[1]);
}
```

Appendixm 3: Function Prototypes Declaration

```
#include "nRF24L01.h"
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include <m8c.h>        // part specific constants and macros
#include "nRF24L01.h"
#include "PSoCGPIOINT.h"

// define the addresses of RX_ADDR_P0 and TX_ADDR, LSBs first

BYTE RxAddrP0[5] = { 0x01, 0x01, 0x05, 0x01, 0x01};
BYTE TxAddr[5] = { 0x01, 0x01, 0x05, 0x01, 0x01};

// Local prototypes
void nRF24_CSN_state( BYTE state );
void nRF24_CE_state( BYTE state );

void nRF24L01_commonConfiguration(void)

{

            // nRF24_CSN on
            nRF24_CSN_state(CSN_ON);

            // set CRC to 2 bytes in the CONFIG and disable interrupts
             on IRQ
            nRF24L01_sendInstruction( W_REGISTER | CONFIG, 0x7C );

            // disable EN_AA for all pipes
            //nRF24L01_sendInstruction( W_REGISTER | EN_AA, 0x00 );

            // ARC set to 15
            //nRF24L01_sendInstruction( W_REGISTER | SETUP_RETR, 0x0F
            );

            // set data rate to 2Mbps
            nRF24L01_sendInstruction( W_REGISTER | RF_SETUP, 0x0E );

            // set addresses for RX_ADDR_P0 and TX_ADDR, change values
            in function
            nRF24L01_setRxTxAddr();

            // set RX_PW_P0 to 2 bytes
            nRF24L01_sendInstruction( W_REGISTER | RX_PW_P0, 0x02 );

}

// set both RX_ADDR_P0 and TX_ADDR addresses of the nRF24L01
// the default SETUP_AW, which is 5 bytes, is left unchanged

void nRF24L01_setRxTxAddr(void)

{

    BYTE i;
```

```
        // set the RX_ADDR_P0, 5 bytes
        nRF24L01_sendPayload( W_REGISTER | RX_ADDR_P0, RxAddrP0, 5);
        // set the TX_ADDR, 5 bytes
        nRF24L01_sendPayload( W_REGISTER | TX_ADDR, TxAddr, 5);
}

// set as transmiter and power up the nRF24L01
void nRF24L01_SetAsTransmiter( void )
{
            BYTE presentConfig;
            // nRF24_CE off
            nRF24_CE_state( CE_OFF);
            presentConfig = nRF24L01_sendInstruction(R_REGISTER | CON
            FIG, DUMMYDATA);
            nRF24L01_sendInstruction( W_REGISTER | CONFIG, presentCon
            fig & ~0x01 );
            presentConfig = nRF24L01_sendInstruction(R_REGISTER | CON
            FIG, DUMMYDATA);
            nRF24L01_sendInstruction( W_REGISTER | CONFIG, presentCon
            fig | 0x02 );
            // nRF24_CE on
            nRF24_CE_state(CE_ON);
}
// set as reciever and power up the nRF24L01
void nRF24L01_SetAsReciever( void )
{
            BYTE presentConfig;
            // nRF24_CE off
            nRF24_CE_state( CE_OFF);
            presentConfig = nRF24L01_sendInstruction(R_REGISTER | CON
            FIG, DUMMYDATA);
            nRF24L01_sendInstruction( W_REGISTER | CONFIG, presentCon
            fig | 0x03 );
            // nRF24_CE on
            nRF24_CE_state(CE_ON);
```

```
}

// Helper function for setting CSN off/on

void nRF24_CSN_state( BYTE state )

 {
           if( state == CSN_ON)
           {
                     PRT0DR |= nRF24_CSN_MASK;
           }
           else {
                     //off
                     PRT0DR &= ~nRF24_CSN_MASK;
           }
}

// Helper function for setting CE off/on

void nRF24_CE_state( BYTE state )

 {
           if( state == CE_ON)
           {
                     PRT0DR |= nRF24_CE_MASK;
           }
           else
            {
                     //off
                     PRT0DR &= ~nRF24_CE_MASK;
           }
}



// Sends one command word or byte to the nRF24L01

BYTE nRF24L01_sendOneByte(BYTE oneByte){

           BYTE status = 0x00;

           // nRF24_CSN off
           nRF24_CSN_state( CSN_OFF);

           while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY )
           );
                     SPIM_SendTxData( oneByte );

           while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                     status = SPIM_bReadRxData();

           // nRF24_CSN on

           nRF24_CSN_state(CSN_ON);

           Delay50uTimes(20);

           return(status);
}
```

```
// Sends instruction commands to the nRF24L01

BYTE nRF24L01_sendInstruction(BYTE instructionWord, BYTE mapAddr){

            BYTE status;

            // nRF24_CSN off
            nRF24_CSN_state( CSN_OFF);

            while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY )
            );
                        SPIM_SendTxData( instructionWord );

            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        status = SPIM_bReadRxData();

                        while( ! (SPIM_bReadStatus() &
                        SPIM_SPIM_TX_BUFFER_EMPTY ) );
                        SPIM_SendTxData( mapAddr );

            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        status = SPIM_bReadRxData();

            // nRF24_CSN on
            nRF24_CSN_state(CSN_ON);

            Delay50uTimes(20);

            return(status);
}


// Sends multiple bytes, such as pipe addresses, to the nRF24L01
// dataArray[] must exist

void nRF24L01_sendPayload(BYTE instructionWord, BYTE* dataArr, BYTE
bytes) {

            BYTE i;

            // nRF24_CSN off
            nRF24_CSN_state( CSN_OFF);

            while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY )
            );

                        SPIM_SendTxData( instructionWord );

            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        SPIM_bReadRxData();

            for(i = 0 ; i < bytes ; i++) {
                        while( ! (SPIM_bReadStatus() &
            SPIM_SPIM_TX_BUFFER_EMPTY ) ){};

                        SPIM_SendTxData( dataArr[i] );
```

```
            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        SPIM_bReadRxData();

            }


            // nRF24_CSN on
            nRF24_CSN_state(CSN_ON);

            Delay50uTimes(20);
}


// Transmits payload via a nRF24L01 to a receiving nRF24L01
// this function must be called after sendPayload()

void nRF24L01_transmitPayload(void){

    // nRF24_CE on
             nRF24_CE_state( CE_ON);

             Delay50uTimes(1);

    // nRF24_CE off
            nRF24_CE_state( CE_OFF);

            Delay50uTimes(20);
}


// Reads multiple bytes. Note: first byte is STATUS register of the
nRF24L01

// dataArray[] must exist

void nRF24L01_readPayload( BYTE instructionWord, BYTE* dataArr, BYTE
bytes ) {

            BYTE i;

            // nRF24_CSN off
            nRF24_CSN_state( CSN_OFF);

                        while( ! (SPIM_bReadStatus() &
            SPIM_SPIM_TX_BUFFER_EMPTY ) );
                        SPIM_SendTxData( instructionWord );

            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
            dataArr[0] = SPIM_bReadRxData();

            for(i = 0 ; i < bytes ; i++) {

            while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY )
            );
                        SPIM_SendTxData( DUMMYDATA );

            while( SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL );
                        dataArr[i] = SPIM_bReadRxData();
```

```
            }

            // nRF24_CSN on
            nRF24_CSN_state(CSN_ON);

            Delay50uTimes(20);

}




// checks if payload's been received

BYTE nRF24L01_is_RX_DR( void ){

            BYTE tmp;

            // check the RX_DR of the STATUS register

            if( nRF24L01_sendOneByte( NOP ) & RX_DR_MASK )
            tmp = 1;

            else
                        tmp = 0;

            return tmp;
}

// checks if payload was successfully sent

BYTE nRF24L01_is_TX_DS( void ){

            BYTE tmp;

            // check the TX_DS of the STATUS register

            if( nRF24L01_sendOneByte( NOP ) & TX_DS_MASK )
                        tmp = 1;

            else
                        tmp = 0;

            return tmp;

}

// checks if MAX_RT is set

BYTE nRF24L01_is_MAX_RT( void ){

            BYTE tmp;

            // check the TX_DS of the STATUS register
            if( nRF24L01_sendOneByte( NOP ) & MASK_MAX_RT )
                        tmp = 1;
```

```
            else
                        tmp = 0;

            return tmp;

}


// clears RX_DR interrupt in STATUS

void nRF24L01_clear_RX_DR( void )

{

            BYTE presentStatus;
            presentStatus = nRF24L01_sendOneByte( NOP );
            nRF24L01_sendInstruction( W_REGISTER | STATUS, presentSta-
            tus |= MASK_RX_DR );

}

// clears TX_DS interrupt in STATUS

void nRF24L01_clear_TX_DS( void )
{

            BYTE presentStatus;
            presentStatus = nRF24L01_sendOneByte( NOP );
            nRF24L01_sendInstruction( W_REGISTER | STATUS, presentSta-
            tus |= MASK_TX_DS );

}

// clears MAX_RT interrupt in STATUS

void nRF24L01_clear_MAX_RT( void )
{

            BYTE presentStatus;
            presentStatus = nRF24L01_sendOneByte( NOP );
            nRF24L01_sendInstruction( W_REGISTER | STATUS, presentSta-
            tus |= MASK_MAX_RT );

}
```