TURUN AMMATTIKORKEAKOULU
ÅBO YRKESHÖGSKOLA

Thesis

Building a SumoBot Robot with AVR Butterfly Microcontroller

Nuno Miguel Nunes

Information Technology

2010

TURKU UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Degree Program: Information Technology	
Author: Nuno Miguel da Silva Nunes	
Title: Building a SumoBot robot with a AVR Butterfly Microcontroller	
Specialization line: Embedded Systems	Instructor: Jari-Pekka Paalassalo, Lic.Sc. (Tech.), Principal Lecturer
Date: 14.05.2010	Total number of pages: 67

Summary:

This thesis is a good documentation where tells how to design and develop a system for a Sumo robot using a Avr Butterfly Microcontroller. The goal is to give an idea of what is necessary to do to understand and develop embedded drivers for each component/sensor.

The first part is the background chapter, which transmit basic background information to have a better comprehension and a global view of the later parts of this project.

The main content of the report are the chapters describing implementation of the system, being this divided in three major topics: Analysis and Design, Implementation and Testing.

In the analysis and design chapter a studied of the hardware limitations, suitable system architectures and testing plan to select the best hardware and software designs for the system.

Most of the descriptions in the implementation-chapters are in the form of instructions to make it easier to follow and understand the different phases involved. There you can find the steps to build and program the Sumobot.

Finally the test chapter gives exemplifications of some ad hoc tests that are essential for the robot components' analysis.

Keywords: Embedded, AVR Butterfly, ATmega169, SumoBot, Parallax

For my Father that never gives up...

ACKNOWLEDGEMENTS

This thesis would not have been possible without teacher Jari-Pekka Paalassalo, that has been the ideal thesis supervisor. His effort to guide and lecture this work was amazingly needed and really appreciated. I also would like thank Tino M. which I am indebted for his perfect team work capabilities and great sense of humour.
Lastly, I offer my greatest regards and gratitude to Vasco Conde that supported me structuring all thesis' documentation.

Contents

List of Figures

Glossary

Atmega169: It's the microcontroller model from AVR company.

IR: Acronym for Infra-red.

Olimex - Microcontroller built by OLIMEX Ltd. that is company responsible for Electronic design and PCB sub-contract assembly.

Protocol - Set of rules used by different devices to be able to communicate.

Sumobot - Robot design to enter in sumo competitions.

Led - light-emitting diode.

Ohm - International system unit of electrical impedance.

Chassis - Supporting frame of a structure.

QTI - Close-proximity infra-red emitter and receiver pair.

Polarity - It's the flow of the electrons from the negative pole to the positive pole.

Volts - International System derived unit of electromotive force.

PWM - Pulse Wide Modulation is a very efficient way of providing intermediate amounts of electrical power between fully on and fully off.

ADC - Analogue to digital converter.

1 Introduction

The main focus of this thesis is to provide a good documentation how to develop and build a Sumobot using an Avr Butterfly controller. The first part of the thesis is the background chapter which tries to supply the necessary basic background information to have a better comprehension and a global view of the later parts of this project. The main content is divided in three topics: Analysis and Design, Implementation and Testing.

In the analysis and design chapter a studied is made by going through the hardware limitations, suitable system architectures and testing plans so that the hardware and software designs chosen can be justified and better understood. The most significant part of the thesis is the implementation. It consists on mechanical, electronic and software descriptions in form of instructions to make it easier to follow and comprehend the different phases involved. And the last part is a small exemplifications of some ad hoc tests that are essential for the robot components' analysis.

This thesis will mainly focus on the system's development for the Avr Butterfly, consequently some parts won't be so worked out as the Software topic. The chapters expect that the reader has basic knowledge of electronics and embedded systems programming. This includes some C programming and IDEs/plugins installation skills, which are basic skills for any computer science student.

The robot's code is all in the appendix section, it's well commented and easy to understand. For each code module there is a general description of it's logic, just some specific definitions like registers' configurations and timers/counters calculations and values are described. Nevertheless, following the Avr Butterfly's official documentation is crucial for the software development.

Once the thesis' fundamental work was finished, it started to evolve to another level. An Olimex Board was added and Infra-red communication between robots and control stations became the new goal. Each successful step brought more ideas and the main theme started to be wider, so it was decided by the author and the teacher Jari-Pekka Paalassalo that some things would be discarded. The Olimex board working as a Sumobot control station will be documented by Constantino Miguélez. Nevertheless, it will be made some reference to the infra-red communication's driver for the Avr Butterfly controller.

At the end of the thesis, there are some tools' evaluation, explanations of problems and best practices found during the work. Most of these explanations are subjective and based only on the authors experience on Embedded systems and electronics.

2 Background

This chapter provides necessary background information for understanding the later parts of the document and also introduces some concepts used. Most of the information is on a very general sources level and the same information can be found from many sources.

2.1 The Parallax SumoBot Robot Kit

The SumoBot robot objective is to perform official sumo matches according to the rules and regulations.

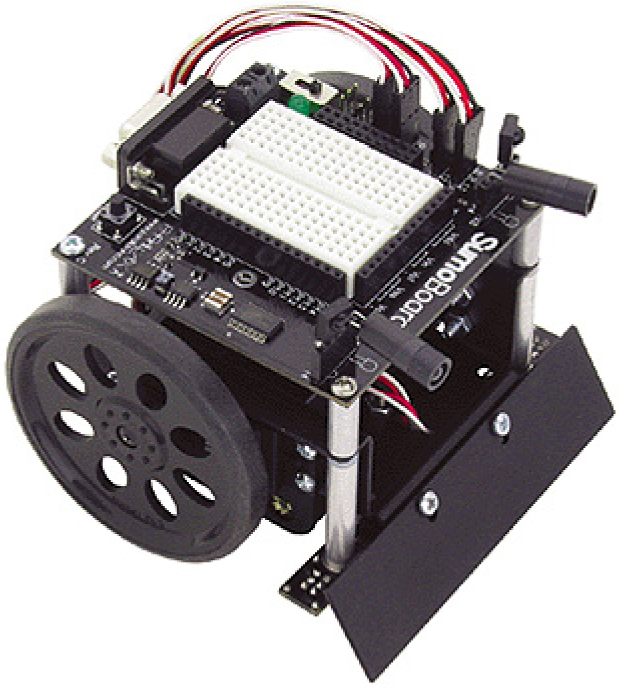

Figure 1: Parallax SumoBot Robot [11]

The main goal of this Sumobot is to locate and push the opponent out of the ring and, at the same time, avoid going out from it by detecting the outside white line and moving away from it.The electronics supplied in the Sumobot Kit are a surface-mounted BASIC Stamp 2 module, infra-red sensors and QTIs to detect the opponent and the edge of the Sumo Ring. The hardware package includes the black anodized aluminium chassis and scoop, servo motors, wheels, 4AA power pack, mounting stand-offs and screws.

Quantity and Description of Parallax SumoBot contents: [1]

- (1)SumoBot Board with surface-mounted BS2;
- (2)QTI Sensor;

- (1)Parallax Screwdriver;
- (1)Chassis, SumoBot;
- (1)Front Scoop, SumoBot;
- (2)Wheel, Plastic, 2.58 Diameter , 3 Width;
- (3)Rubber Band Tire;
- (1)Battery holder, 4 cell, AA, leads;
- (1)SumoBot Manual;
- (2)Res, CF, 5%, 1/4W, 470 Ohm;
- (1)CD ROM, Parallax software and documentation;
- (1)LED-GREEN-T 3/4;
- (2)LED-Infra-red - T1 3/4;
- (1)LED-Red - T1 3/4;
- (2)IR Receiver;
- (3)LED Stand-off;
- (3)LED Light Shield;
- (1)Serial Cable;
- (1)3 inch Jumper Wires (1 Bag of 10);
- (2)Servo Extension Cable (10 inches);
- (1)Piezoelectric Sound Generators;
- (2)Continuous Rotation Servo (Futaba);
- Assortment of screws, washers, and stand-offs.

The kit comes with a good guide that teaches how to assemble and program the robot in BASIC Stamp language. The manual gives good examples and explanations how to start the Basic Sumobot locomotion, edge avoidance and opponent detection based on the sensors inputs and some tips for better performance and efficiency of the programming code.

The Parallax website also supplies an advanced guide so that, when the robot is finished and ready to run, it's possible to give a more competitive behaviour to the Sumobot by adding some artificial intelligence, tuning the sensors and the algorithm.

This Kit is very enjoyable and easy to follow allowing the programmer to expand its knowledge and general idea about embedded systems. Due to the programming language BASIC Stamp being not much used in embedded systems besides Parallax, the original microcontroller won't be used. It will be replaced by an Avr Butterfly microcontroller.

2.2 The Competition

Robot-sumo is considered as a sport where two robots attempt to push each other out of a circle shaped Ring. This Type of robots are called Sumobots.
The most challenging part is to make the robot able to find it's opponent, normally using Infra-red sensors or even sonar devices, and push the rival out from the ring. The robot must be able to recognize the border of the ring and avoid to trespass it with the help of more IR Sensors called QTI.

The most common tool used in a Sumobot competition is an angled blade at the front of the robot. Regularly the blades are tilted at about a 45 degree angle towards the back of the robot. This blade may contribute for different tactics has an adjustable height.

The Sumobot competition is divided into classes and fought on progressively smaller arenas. The classes are further divided into two types of control systems, remote-controlled and autonomous robots. Robot fights are only valid is both belong to the same class and same type of control system.

The robots class specifications: [13]

Class	Height	Width	Length	Weight
Mega Sumo	unlimited	20 cm	20 cm	3,000 g
Mini Sumo	unlimited	10 cm	10 cm	500 g
Micro Sumo	5 cm	5 cm	5 cm	100 g
Nano Sumo	2.5 cm	2.5 cm	2.5 cm	25 g
Lego/Vex Sumo	unlimited	15.2 cm	15.2 cm	1,000 g
Humanoid Sumo	50 cm	20 cm	20 cm	3,000 g

Definition of a Sumo Match:
"A match is fought between two teams, each team having one or more contestants. Only one team member may approach the ring while the rest of the team members must watch from the audience. In accordance with the game rules each team competes on a Dohyo (the sumo ring) with a robot that they have built by themselves respecting the Sumobot's competition specifications. The match starts at the judge's command and continuous until a contestant earns two Yuhkoh points. At the end of the match a judge determines the winner of the match."[13]

Requirements for the Dohyo (Sumo ring):
"A Dohyo is an cylinder and of the appropriate dimensions for the given size Sumobot class. The starting lines, called "Sikiri-Sen", are indicated as two brown lines. The border line is indicated as a white circle. It is defined as being within the interior of the Dohyo, anywhere outside this area is called the Dohyo exterior."[13]

The Dohyo specifications: [13]

Dohyo Construction & Painting				**Shikiri Lines**	
Class	**Height**	**Diameter**	**Width**	**Length**	**Border W.**
Mega\Humanoid	5 cm	154 cm	2 cm	20 cm	5 cm
Mini\Lego\Vex	2.5 cm	77 cm	1 cm	10 cm	2.5 cm
Micro	1.25 cm	38.5 cm	0.5 cm	5 cm	1.25 cm
Nano	0.625 cm	19.25 cm	0.25 cm	2.5 cm	0.625 cm

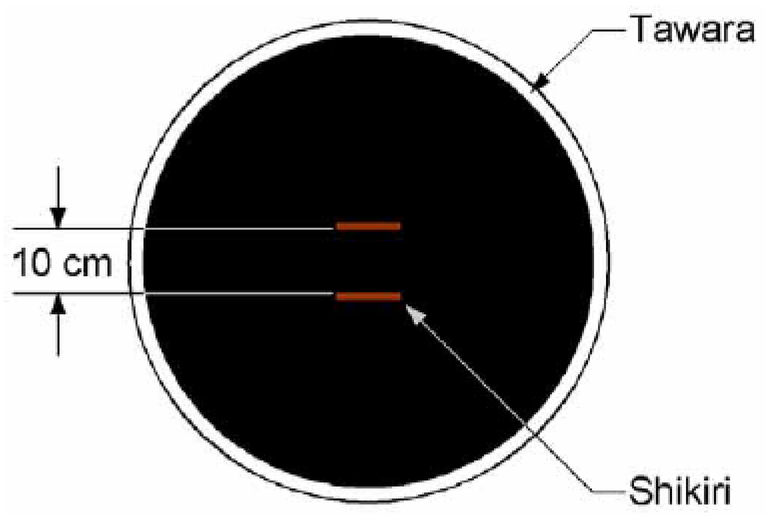

Figure 2: The Dohyo - Sumobot Ring [13]

Requirements for Robots - Specifications and Restrictions

”A robot is allowed to expand in size and change is shape after the match begins, although it must remain a single centralised robot. Robots violating these restrictions shall lose the match. Screws, nuts, and other robot parts with a total mass of less than 5 g falling off from a robot’s body shall not cause the loss of match.

All the Sumobots must be autonomous except the 3 kg and humanoid robots. They can be either autonomous or remote control. Any control mechanisms can be employed, as long as all components are contained within the robot and the mechanism does not interact with an external control system.

Autonomous class robots must not start operating for a minimum of five seconds after initiation by the user.

The robot must have some identification tags like a name or number for registration purposes. This ID should be displayed on the robot to allow spectators and officials to identify your robot.

Jamming devices, such as IR LEDs intended to saturate the opponents IR sensors, are not allowed, also parts that could break or damage the ring are prohibited. Do not use parts that are intended to damage the opponents robot or it’s operator. Normal pushes and bangs are not considered intent to damage.

The Sumobot can’t carry any devices that can store liquid, powder, gas or other substances for throwing at the opponent, any flaming or throwing things devices are not accepted. Devices to increase down force, such as a vacuum pump or magnets, are only allowed in the 3 kg class. They are not allowed in all other classes.

Any modifications like adding sticky substances to improve traction are not allowed. All edges, including but not limited to the front scoop, must not be sharp enough to scratch or damage the ring, other robots, or players."[13]

How to carry Sumo Matches

"The game consists of three matches of three minutes each, unless extended by the judges. The first contestant to win two Yuhkoh points is the winner of the game. The contestant who wins a Yuhkoh point at the end of the game is judged as the winner. When neither contestant receives any Yuhkoh points, the winner is decided by judgement. However, if no obvious superiority exists and a winner can still not be determined, an extra three-minute match can be played."[13]

Start, Stop, Resume, End a Match

Starting: "The match starts upon the judge's instructions, the two teams approach the ring and place a robot within their half of the ring on or behind the Shikiri line. When the judge announces the start of the round, the teams start their robots, and after a five second pause the robots may start operating. During these five seconds, players must clear out of the ring area. Note that is not required that a robot be placed directly behind the Shikiri line. It may be offset to the side, as long as it is behind an imaginary line collinear with the Shikiri line."

Stop and Resume: "The match stops and resumes only under judge orders."

End: "The match ends when the judge makes the final announces and the two teams have to retrieve the robots from the ring area."[13]

Yuhkoh

The Yuhkoh point is won when:

- When a robot successfully puts the opponent out of the ring.
- When the opponent's robot falls out of the ring by himself.
- When the opponent's robot is disqualified due to excess of warning or penalties.

More detailed information can be found in the Sumobot Guide from parallax. There are still several important rules that can't be ignored. Before entering in any competition it's also good to know the violations and penalties.

2.3 Avr Butterfly Microcontroller

he Avr Butterfly is a little controller from Atmel with Low power design, runs the ATMEL AVR ATmega169PV Microcontroller and includes several peripherals. Originally comes with a preloaded code where all it's peripherals can be tested.

The AVR Butterfly Includes:[2]

- ATmega169 AVR microcontroller
- 100 segment LCD Display
- 4Mbit Dataflash
- 32kHz oscillator for RTC
- 4-way directional button
- Light sensor (LDR)
- Temperature Sensor (NTC)
- Speaker for Sound Generation
- Access to peripherals through header connectors
- RS-232 Level Converter
- Voltage Reading 0-5V

In the next picture are displayed the components and respective physical location of the pin entrances:

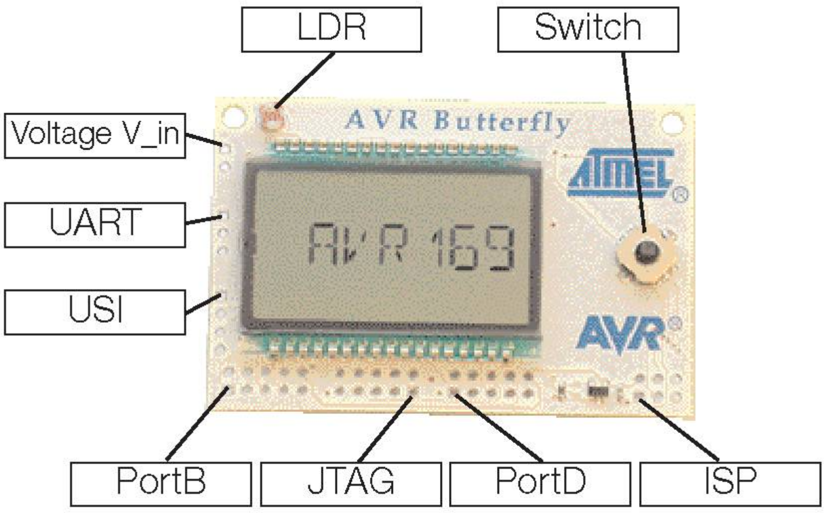

Figure 3: AVR Board Description - Front side [4]

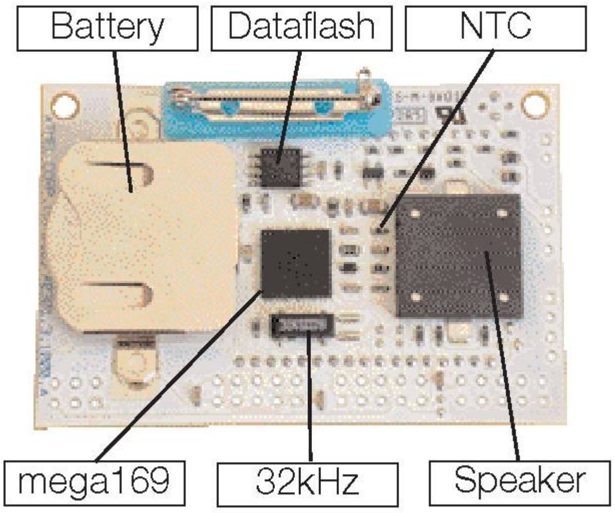

Figure 4: AVR Board Description - Back side [4]

The following image is a generalized diagram with the main components' pin connection to the processor ATmega169:

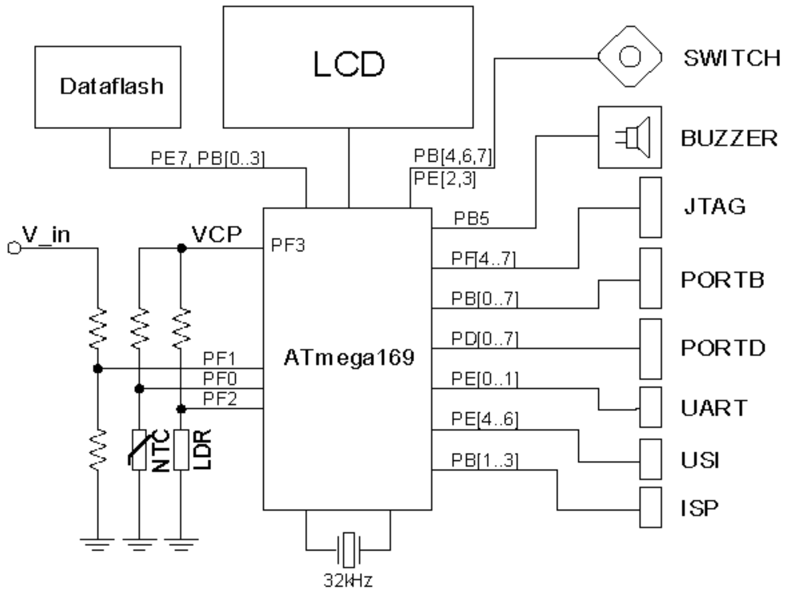

Figure 5: AVR Butterfly simplified block diagram [4]

2.4 Programming Environment Tools

In this topic it will be mentioned the tools that are required to program the Avr Butterfly microcontroller. They may vary depending on the programmer's style, usability and necessary features. All the software used for the thesis' project is free and it can be easily downloaded from the internet.

There are many different IDE's that you can use for embedded programming. During the thesis work, it was tested two integrated development environments, one for Windows (AVR Studio 4 + WinAVR) and other for Linux operating system (Eclipse + AVR Eclipse plugin).

2.4.1 AVR Studio 4 and WinAVR

AVR Studio is a Integrated Development Environment for developing AVR applications, but only supported in windows. It includes a good and complete set of features:

- Debugger supporting run control including source and instruction-level stepping and breakpoints;
- Registers, memory and I/O views;
- Target configuration and management;
- Integrated assembler;
- Integrated simulator;
- Integrates with GCC compiler plug-in.

"The AVR Studio 4 was the latest by this time and can be found on the Atmels website: $http : //www.atmel.com/dyn/products/tools_card.asp?tool_id = 2725$. This IDE requires a software developing tools for Atmel AVR boards called WinAVR.

WinAVR is a suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. It includes the GNU GCC compiler for C and C++. The suite itself and more develop information may be found on Sourceforge at $http : //sourceforge.net/projects/winavr/$." [16]

2.4.2 Eclipse and AVR Eclipse Plugin

"Eclipse is a multi-language software development environment composed by an integrated development environment and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others." [14]

To use Eclipse for programming AVR boards is required to install an AVR plugin. This Eclipse Plugin provides the necessary tools and settings for developing C programs for the ATMEL AVR boards.
Eclipse is a very modern IDE with many features. Unlike some other IDEs, due to being Open Source and well supported/documented, many features are available to extend it even further.
Since Studio IDE from Atmel doesn't support subversion integration or code completion in source editor it might be preferred to use Eclipse. With the AVR plugin is possible to generate the makefiles for building the application automatically and use Eclipse IDE' due to its main benefits. [14]

3 The Project Scope and Requirements

3.1 Scope definition

The main scope of this thesis is to document and implement a functional embedded system using an AVR Butterfly board. The system, the board and the robot were fully prepared to give life to a small sumo robot. The mechanical and electronic tools, as most of the requirements of the project were defined before it started.The system was specifically developed to run on a Avr butterfly Atmega169 board and with the Parallax Sumobot kit.

Since all requirements were completed and mastered, it was decided to improve the robot and take it to another level by adding a new board interfacing with the robot via infra red. The new board, Olimex, with a Atmega128 microcontroller is equipped with six IR leds and is programmed to work as a control station.

The communication is between the Olimex and the robot. A protocol with network characteristics was developed so that a unicast, broadcast and multicast would be possible to perform.
The control station is composed by a menu where it's possible to select different modes such as Radio controlled robot and Sumobot Fight.

This project work can be separated in two parts:

- Sumobot Robot with Avr Butterfly Microcontroller;
- Olimex Control Station with IR communication.

The robot directive is to respect all the official Sumobot rules and perform successful combats during the competition. It's final system is organized by drivers, which have to control perfectly each different components, and be logic enough to not affect the efficiency of the robot during the combat.

With later improvements, the robot, was prepared to interpret IR signals and read them as a message. This signals are stored and tested, if correct they are interpreted as a command.
The Olimex Control Station is a board connected with a couple of IR Leds and programmed with an organized menu. The IR communication protocols share network characteristics and are known by the station and the target robot.

3.2 Requirements of the SumoBot system

The requirements can be divided in Hardware, Software, Competition:

Hardware related requirements	
H1	The robot must use the Avr Butterfly board and the parallax Sumobot kit.
H2	The Avr microcontroller's pin and timers limitations have to be handled on a good manner.
H3	All the components must be fully operational with the system.
H4	The PCB created for the robot has to be organized, functional and connect all the components to the Avr board.

Software related requirements	
S1	The robot must have a code prepared to win a Sumo match - Sumo Mode.
S2	Infra-red control mode has to be stable and usable - IRC Mode on a good manner.
S3	All the components' drivers for the Avr must be stable and efficient.

Sumo Competition related requirements	
C1	The Robot must respect all regulations.
C2	It's code has to be operational and logical enough to perform a successful match.

4 Analysis and Design

This section is a full requirement and hardware analysis so that it will be easier to understand the software architecture and hardware design chosen for the robot. Most of the design decisions were made very intuitive at the beginning due to lack of time or because it was the first logical one to come up. It's recommended to have a planned, organized and structured project ideas, otherwise there will be many time consuming changes that can be avoided from the beginning.

4.1 Hardware Limitations

It is a bit confusing to separate and have a good perspective of all Butterfly board port pins because there are some with more than one functionality. Most of the pins have alternate uses, for instance, the Port D pins can be used for either general I/O or to operate the LCD. This information can be found in several places but sometimes is to specific that it's not possible to see it instantaneously. This is a important hardware limitation to consider, so here will be discussed the most important alternate functions of the available pins and it's utilities for the robot.

Port D and the LCD

When using Port D for output and the LCD at the same time you can notice that some problems may occur, like the LCD outputting lot of strange characters and changing very inconstantly. Well, that is because the LCD also uses the Port D pins. If it's necessary to use the LCD must be avoid the use of the Port D, or vice versa.

Port B and the Piezo, Joystick, DataFlash, and ISP

There are also a few pins that are shared between Port B and other components. If the port B pin 5 is used for I/O it will be heard some noise on the Piezo. Port B pins 1,2 and 3 are used to program DataFlash and to program the Atmega169 via the ISP connector. The DataFlash will ignore the ISP bus as long as the Reset Flash pin is help in the proper state.
The Joystick uses Port B pins 4, 6, and 7, these pins can be used with no problem.

JTAG

The JTAG connector is connected on the bottom of the Butterfly because, differently from Ports B and D, the first JTAG's pin is on the right. If JTAG port is disable in software it's possible to use Port F pins 4, 5, 6, and 7 for their normal functions, which include the Analog-to-Digital Converters. To disable the JTAG can be either by changing the JTAG enable fuse via ISP or by setting the JTD bit in the MCUSCR register in software at application startup.

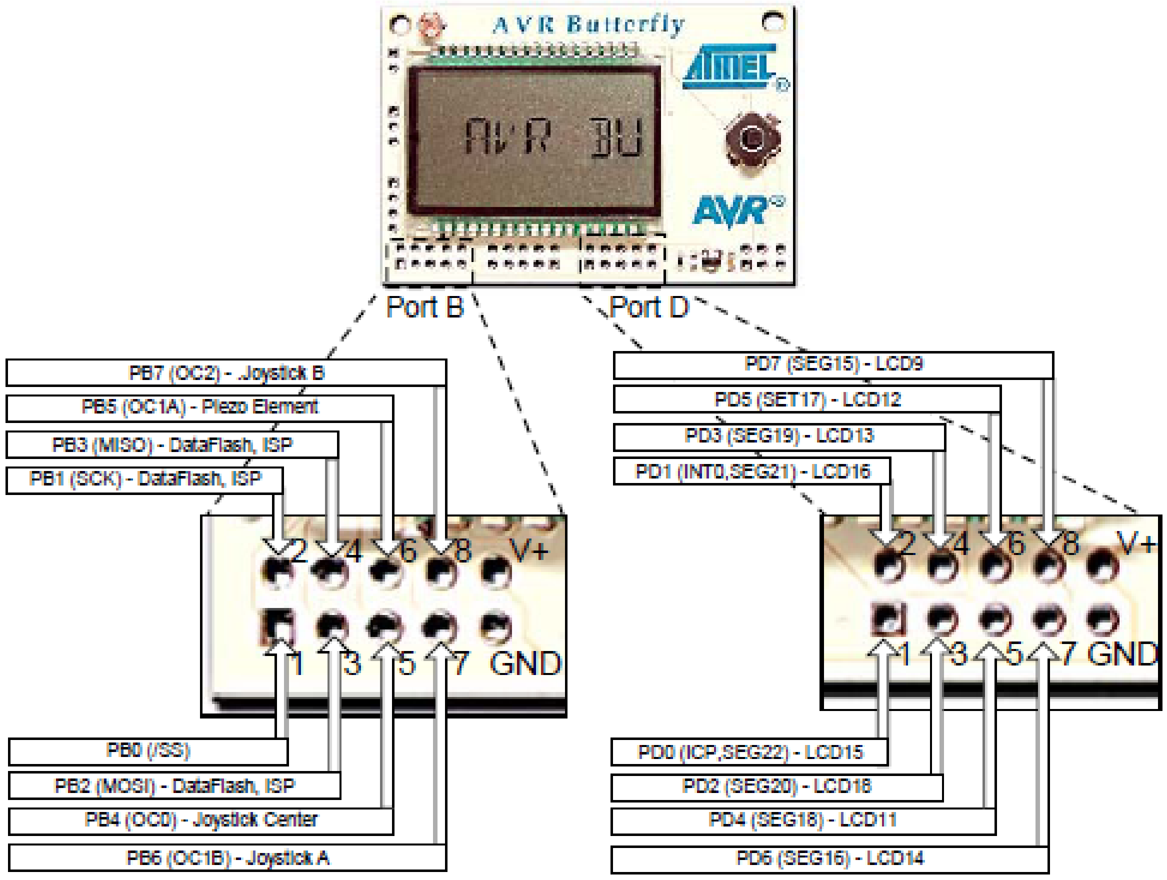

Figure 6: AVR Butterfly Pins - Port B and D [12]

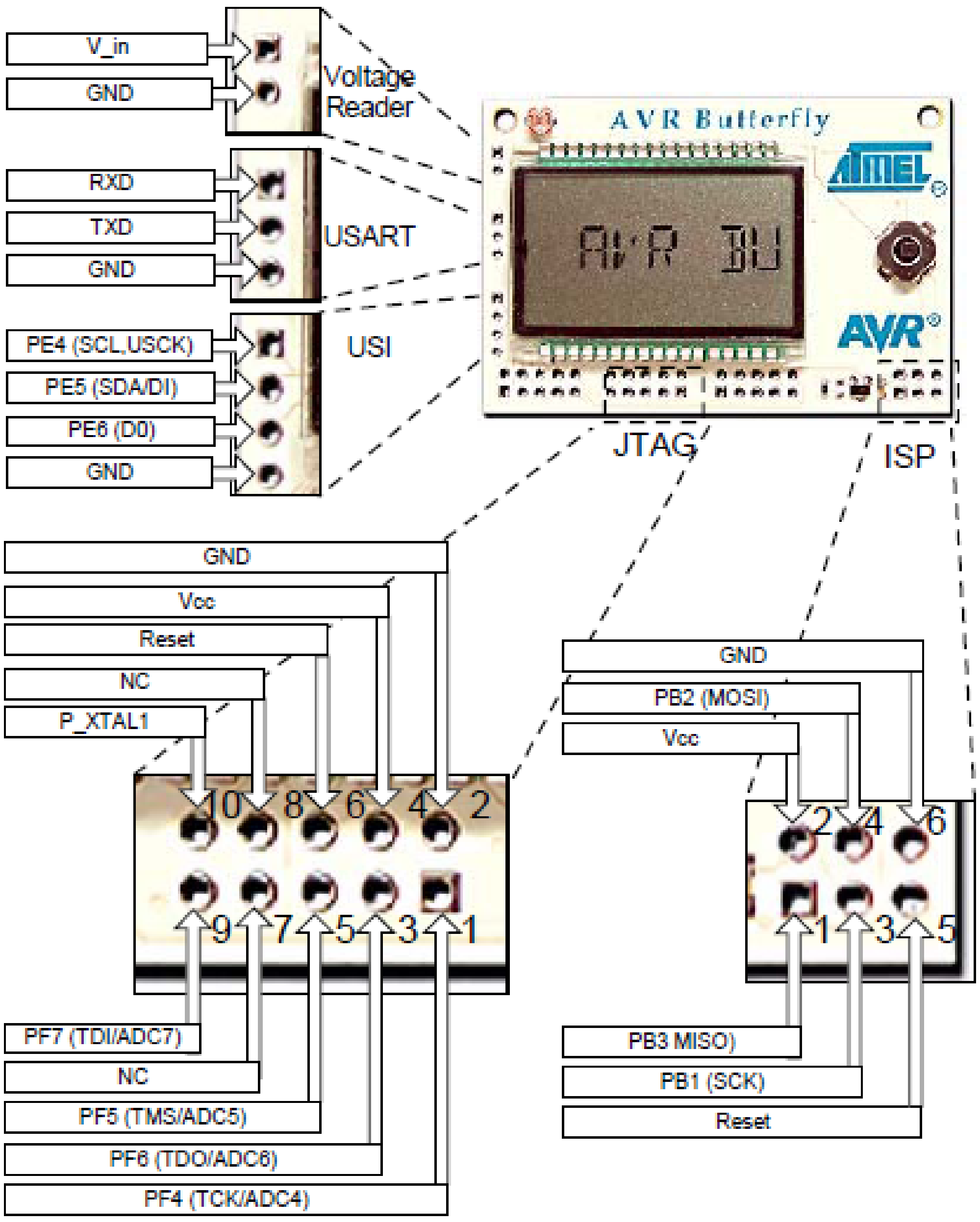

Figure 7: AVR Butterfly Pins - Other Pins and components [12]

AVR Butterfly Pad and Pins' Table

The next table shows Avr's Pads (physical number of the hardware pin), registers and their respective uses in Port B and Port D. Based on it, is easier to understand and notice the common ports and specific pin uses. This way the analysis and decision of each pins' functionality is less complicated and more direct.

Port B			**Port D**		
Pads	**Port Pin**	**use**	**Pads**	**Port Pin**	**use**
Pad 1	PB0 (/SS)		Pad 1	PD0 (ICP,SEG22)	LCD15
Pad 2	PB1(SCK)	DataFlash, ISP	Pad 2	PD1(INT0,SEG21)	LCD16
Pad 3	PB2(MOSI)	DataFlash, ISP	Pad 3	PD2(SEG20)	LCD18
Pad 4	PB3(MISO)	DataFlash, ISP	Pad 4	PD3(SEG19)	LCD13
Pad 5	PB4 (OC0)	Joystick Center	Pad 5	PD4 (SEG18)	LCD11
Pad 6	PB5(OC1A)	Piezo Element	Pad 6	PD5(SEG17)	LCD12
Pad 7	PB6(OC1B)	Joystick A	Pad 7	PD6(SEG16)	LCD14
Pad 8	PB7(OC2)	Joystick B	Pad 8	PD7(SEG15)	LCD9
Pad 9		GND	Pad 9		GND
Pad 10		V+ External	Pad 10		V+ External

Avr's Pin and Port attribution

Since there are two servos, two IR emitters, five IR sensors and two QTI, it's necessary to have four timers/counters as output, two ADC ports and five other input ports. The Sumobot's Avr Board will use the following pins, timers and ports:

Port B					
Pads	**Port Pin**	**use**	**Pads**	**Port Pin**	**use**
Pad 1	PB0	Left QTI input	Pad 2	PB1	Top Left IR Sensor input
Pad 3	PB2	Right QTI input	Pad 4	PB3	Top Right IR Sensor input
Pad 5	PB4 (OC0)	Left IR LED timer	Pad 6	PB5(OC1A)	Left Servo Timer
Pad 7	PB6(OC1B)	Right Servo Timer	Pad 8	PB7(OC2)	Right IR LED timer
Pad 9		GND	Pad 10		V+ External
Port F					
Pads	**Port Pin**	**use**	**Pads**	**Port Pin**	**use**
Pad 5	PF5 (ADC 5)	Left IR Sensor	Pad 1	PF4 (ADC 4)	Top center IR Sensor
Pad 3	PF6 (ADC 6)	Right IR Sensor			

4.2 System Architecture

There are several different types of software architecture that can be used in embedded systems. It's wise to study the project's requirements and hardware limitations before deciding which architecture to use.

Embedded software architectures:

- Simple control loop - Background;
- Interrupt controlled system - Background/Foreground;
- Cooperative multitasking;
- Preemptive multitasking or multi-threading;

After the hardware analysis it's easy to discard a few types of architectures for this project. Due to lack of memory it's impossible to include an operating system with pre-emptive multitasking nor multi-threading. Since the goal is to get a simplified, reduced and organized code, the best options are a simple control loop and interrupt controlled system.

The most suitable options for the developing system are foreground/background or background type. The foreground/background systems is based on interrupt-only systems where the polled loop is replaced by code that performs useful processing.This system type is the most used for embedded applications. They involve a set of interrupt-driven or real-time processes called foreground and a collection of non interrupt-driven processes called background. During all time, the project was aiming for a foreground/background system, but there were no special needs to use interrupts in the code. The necessary system interrupts are used by the timers to obtain continuous signals for the servos and IR emitters. So it can't be considered as a foreground/Background system because the interrupts' do not affect the time sequence of the system and the it does not depend exclusively on them.

The software architecture used is a background system. It consists on a simple control loop with a main file where it calls subroutines, each of which manages a part of the hardware or software.
For the hardware management the code was divided in drivers, each one for a different component.

4.3 Testing plan

The testing strategy is based on ad hoc testing. This is a commonly used term for software testing performed without planning and documentation. The tests are intended to be run only once, unless defect is discovered. Ad hoc testing is a part of exploratory testing, being the least formal of test methods. Supported on that, ad hoc testing has been criticized because it isn't structured, but this can also be a strength: important things can be detected quickly. It is performed with improvisation, the tester seeks to find bugs with any means that seem appropriate.

5 Implementation

The implementation is divided in three parts: Mechanics, Electronics and Software. The mechanics topic is a simple assembly guide of the main structure and chassis of the robot; the electronics consists on the connections and board cable mapping; finally, software sustains some descriptions for the drivers' code and main algorithm of the robot.

5.1 Mechanics

The mechanic part of the robot is very simple and can be followed on the Parallax guide book. Although there are slight differences from the original architecture and design of the robot, because the original PCB is not used, it's assembly is quite the same. There are also several photos of the robot to support the description of some steps.

5.1.1 Assemble the Sumobot

The chassis assembling is very easy to follow in the Parallax instruction book. Everything is similar except the board and infra-red emitters/sensors construction. The board, as above referred, is a 8 cm x 8 cm PCB with several melted pin connections for all connections and all components. It contains 4 screw holes in the corners and a free space here the AVR Butterfly board is fixed.

Step # 1 - Install the Battery box

Flip the Sumobot chassis upside down and install the plastic battery pack using two flat-head screws and nuts. The screws will be countersunk into the battery pack when tightened and will be out of the way of the batteries.

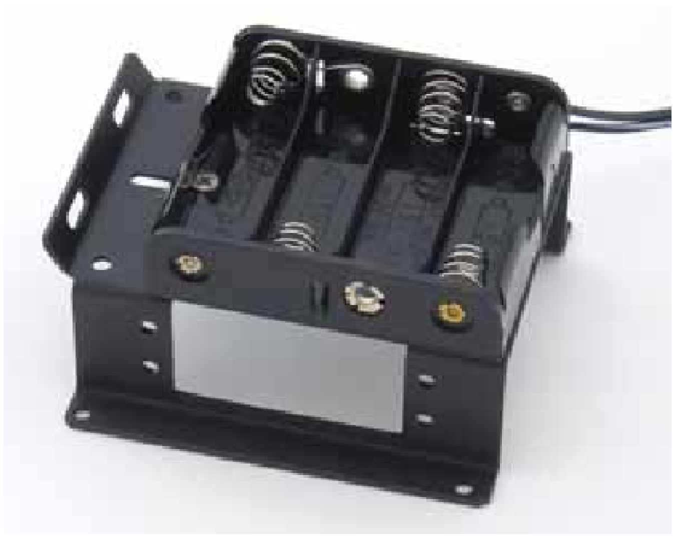

Figure 8: Step 1 - Battery installation [11]

Step # 2 - Install the Servo Motors

Using four pan-head machine screws and nuts, attach each servo motor to the chassis.

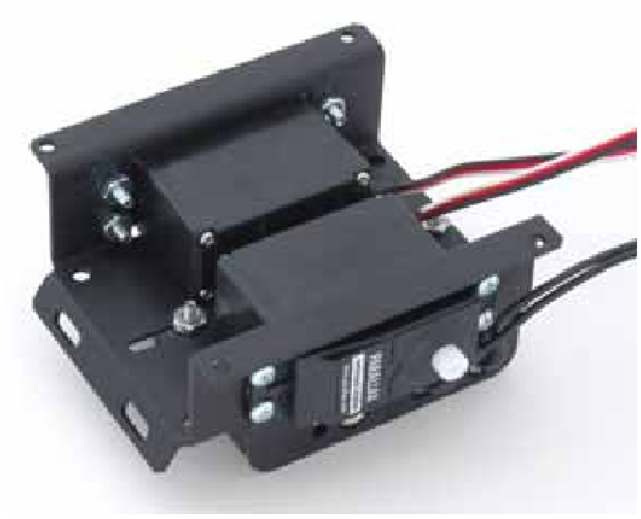

Figure 9: Step 2 - Servos' installation [11]

Step # 3 - Install the Circuit Board, Stand-offs and IR sensors

Attach each standoff to the front and back holes of the Circuit board. Then put the IR sensors and secure them with pan-head screws, on the rear side, might be necessary to add some Circuit board material or nylon washer to level the structure.

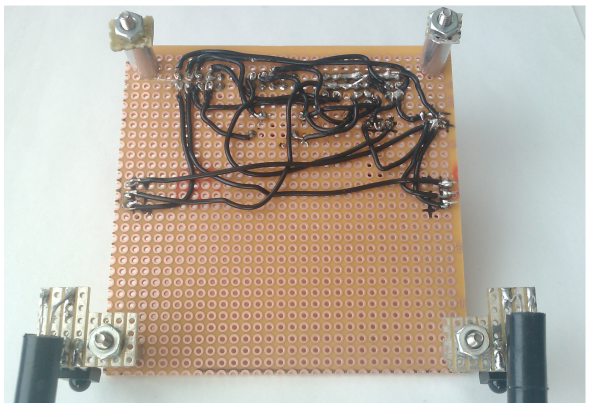

Figure 10: Step 3 - Circuit Board's Stand-off bottom side

Step # 4 - Mounting the support board

Fit the Circuit board and it's screws on each hole of the robot's chassis. The rear screws are fixed by two pan-head machine screws, while in the front, two of the bigger stand-offs will secure the board.

Step # 5 - Prepare the Wheels
Stretch a "tire" of each wheel and adjust so that the "tire" is centred across the wheel.

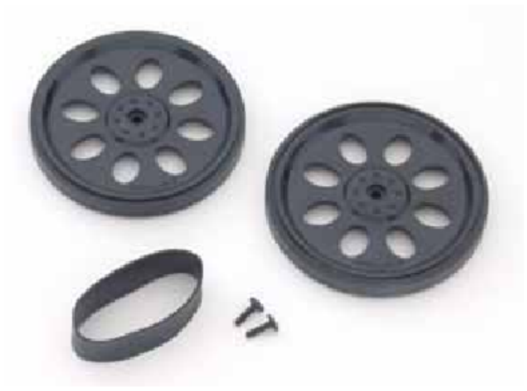

Figure 11: Step 5 - Wheels' preparation [11]

Step # 6 - Mount the Wheels
Press each prepared wheel onto the servo splines. Secure each wheel with the small black Phillips head screw.

Step # 7 - Mount the Scoop
Attach the scoop into the SumoBot chassis. Carefully center the scoop before tightening the screws and nuts.

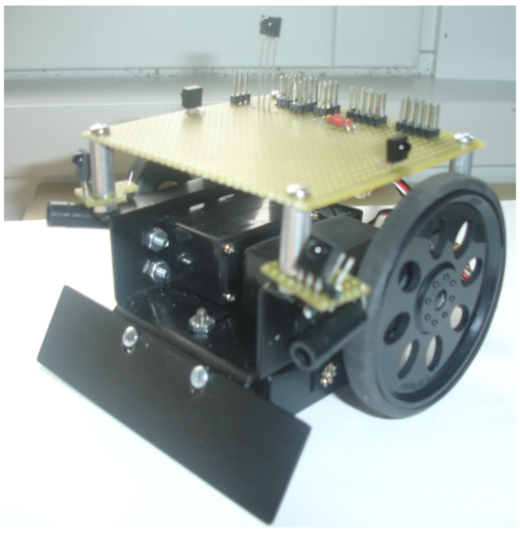

Figure 12: Step 7 - Scoop attachment

Step # 8 - Install Line Sensor Wires
Feed each 3-pin extension cable through the center chassis slot from the bottom to the top.

Step # 9 - Install the QTI Line Sensors
Using two smaller pan-head machine screws, attach the QTI line sensors to the bigger round stand-offs on the bottom. Connect the ends of the 3-pin extension cables to the

QTI line sensors, with care about the polarity identification letters(B[lack]-R[ed]-W[hite]) on the QTI sensors.

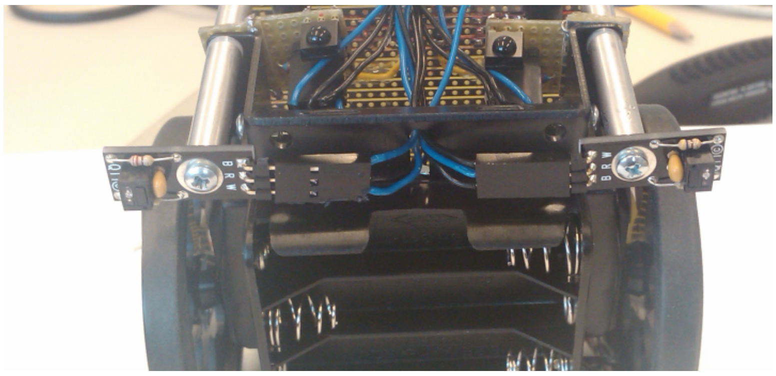

Figure 13: Step 9 - QTI connections and supports

Step # 10 - Make the Connections

Plug the servo motors and QTI sensors into the SumoBot support board connectors.Note that the ground pin on each connector in the front of the Sumobot board. Connect the battery pack wires to the SumoBot support board. From the circuit board two large cables will connect to the Jtag and Port B of the Avr Butterfly board.

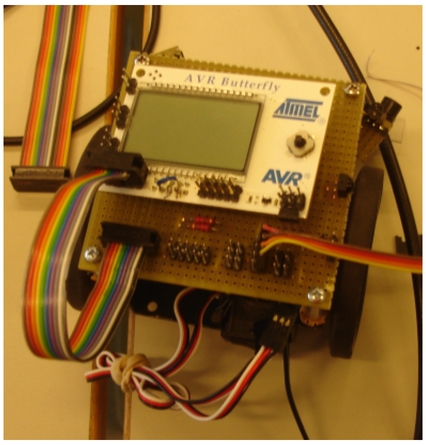

Figure 14: Step 10 - Wide and small cable connections

Step # 11 - Powering the Sumobot

To simplify the turning on and off of the AVR and all components, a switch was built between the power supply cable and the support board. Just need to connect the connectors in the pins and it's ready to be used.

5.2 Electronics

In this topic each electronic component used for the robot will have a description and an analyse to better understand how the drivers were developed and how they work. Starts with the most simple electronic devices (QTI, IR Sensors/Emitters and Servo motors) and finishes with a specific Expansion Board and Cable Structure analysis.

5.2.1 QTI Sensors

The Parallax QTI sensor uses a QRD1114 infra-red reflective sensor to capture the reflectivity of the surface. When the QTI sensor is over a dark surface, the reflectivity is very low; when the QTI is over a light surface, the reflectivity is very high and will cause a different reading from the sensor. Since the sensors only need to detect dark or white surfaces, in the driver development is just necessary to check if the bit is set or clear. If the bit is clear it's on a dark surface, if it's set to one the sensor is above a white surface.

"The QTI sensor is activated by placing 5 Volts (Vdd) on the W pin. This will cause current to flow through the 470 ohm resistor to the LED side of the QRD1114. IR light reflecting of the surface below will cause a change in the ability for the current to flow through the phototransistor side of the QRD1114. The transistor behaves like an IR controlled resistance." [1]

Features:

- Photo-transistor Output;
- Unfocused for sensing diffused surfaces;
- Daylight filter on sensor;

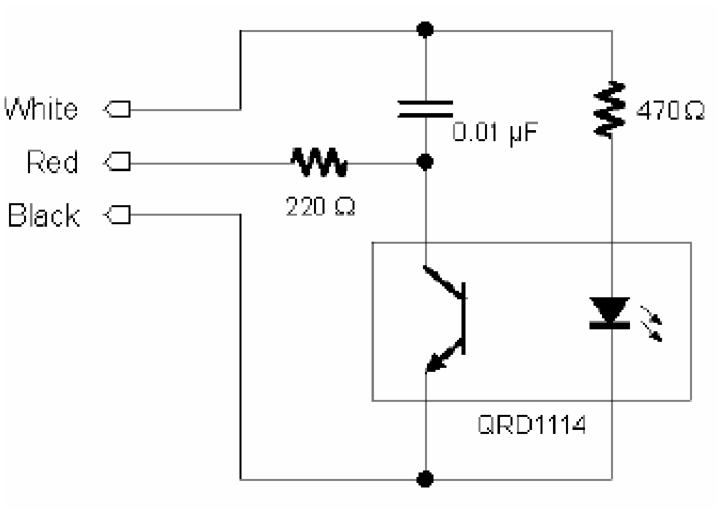

Figure 15: QTI Sensor Circuit [10]

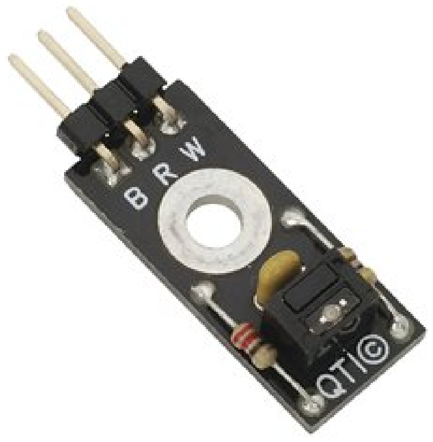

Figure 16: QTI Sensor Component [1]

5.2.2 IR Emitters/Detectors Implementation

These are the "eyes" of the Sumobot, they detect the bouncing IR beam from the IR LED after it hits an obstacle or opponent, allowing the Sumobot to know in which direction the opponent is situated.

IR LEDs are fed with 5V and a 39.5 khz frequency signal from timer0 and timer2, the LEDs are in series with a respective 220 Ohm resistor to prevent them from burning. From several tests, this frequency was the one that suited the best for the actual LED brightness and the desired range to detect the opponent within.

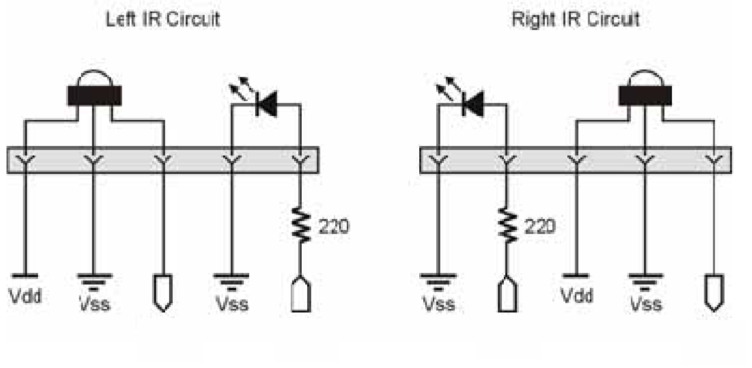

Figure 17: IR object detector Schematics [1]

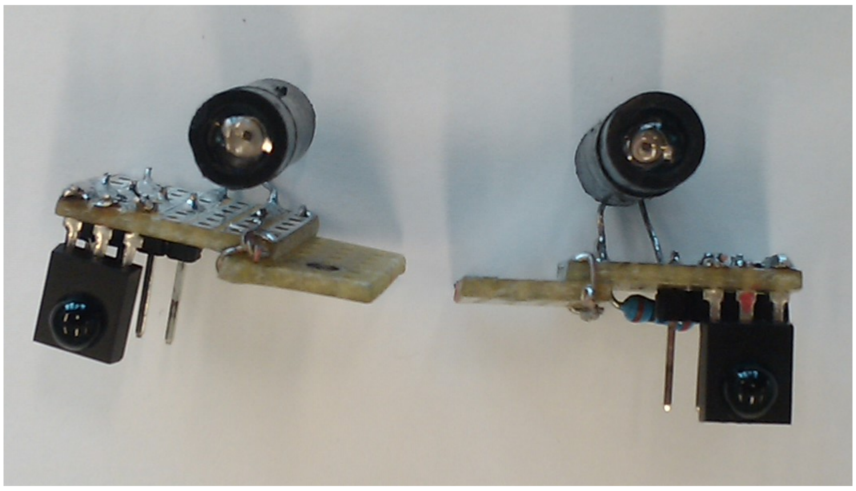

Figure 18: IR object detector

5.2.3 Servo Motors

The Parallax continuous rotation servo is a Futaba S148 servo that has been modified for continuous rotation. The servo receives and converts standard RC servo position pulses into continuous rotation speed. It can be controlled directly by a microcontroller without any additional electronics, which makes it a great actuator for robotics projects. The servo includes an adjustable potentiometer that can be used to center the servo and comes with a star-shaped servo horn and an 11” (270 mm) lead.

The Sumobot motion is controlled by two servo motors using a process called differential drive. When both motors are turning in the same direction, the Sumobot will move in that direction, when they turn in different directions, the chassis will rotate and the rate of movement or rotation is determined by motor speeds. The control signal that the AVR Butterfly sends to the servos' control line is called a ”pulse train”.

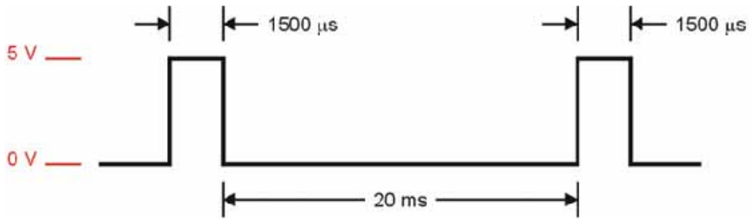

Figure 19: Servo Pulse Train signal [9]

The AVR Butterfly can be programmed to produce this waveform using PB5 and PB6 pins for left and right servo. In the previous image the pulse train has a 1500 μs high time and a 20 ms low time. The high time is the main characteristic for controlling a servos' motion, and it is most commonly referred to as the pulse width. Since these pulses go from low to high (0V to 5V) for a certain amount of time, they are called positive pulses. Negative pulses would involve a resting state that's high with pulses that drop low.

The ideal pause between servo pulses is 20 ms, but can be anything between 10 and 40 ms without adversely affecting the servos' performance. A pulse width of 1500 μs will cause the modified servo to stop. To make the servo turn we must give change the pulse width toward either end of the standard control range of 1000 to 2000 μs. Since the right side servo motor is physically mirrored from the left, its control signals are as well.

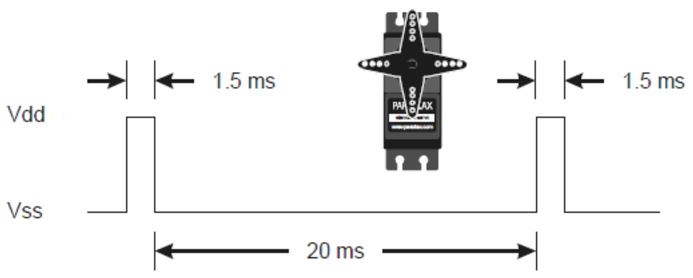

Figure 20: Servo Stop Pulse train [9]

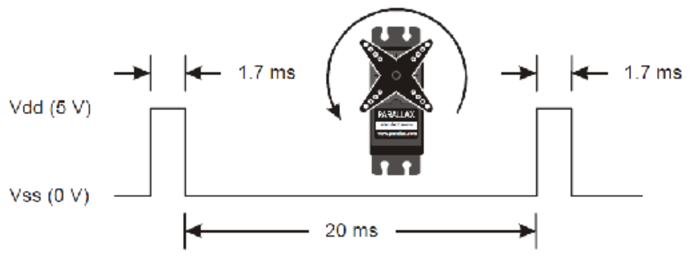

Figure 21: Servo Left Pulse train [9]

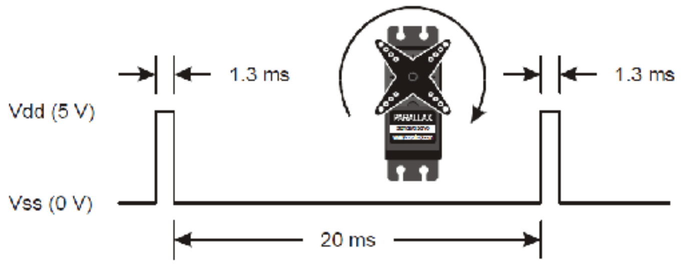

Figure 22: Servo Right Pulse train [9]

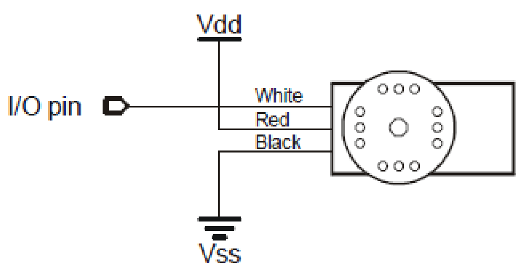

Figure 23: Servos' connections: [9]

Adjusting the set point for the servo

The most easy way to adjust the set point for the servos is to program the timers and the pulses so that the will stop. Once thy are generated, the servos are powered and connected, it's just necessary to insert a screwdriver on the adjustment port and find the correct position of the potentiometer. The servos direction will turn depending the side that the screw turns. When the servos stops, the right adjustments were completed.

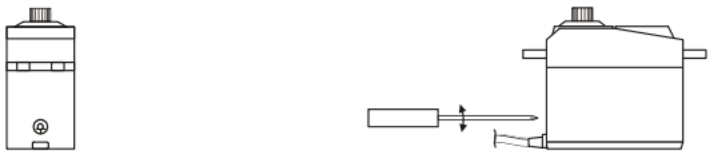

Figure 24: Servos' set point adjustment screw [9]

5.2.4 The Circuit Board and Cable distribution

The circuit board is a very important component in the robot, not only is responsible for supporting the Avr Butterfly board and the components on the chassis, but also helps to organize and structure the connectors and the cables.

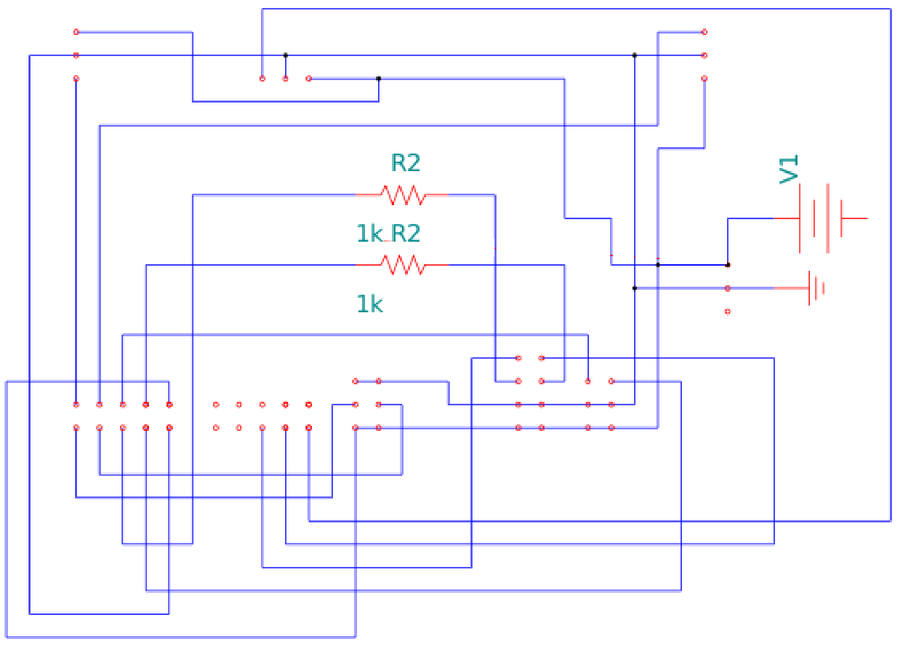

Figure 25: Circuit Board Schematic

The pins in the schematic are like an extension from the ones in the board, therefore, the pads organization are is the same as in Avr's front side. From the left to the right, the set of 10 pins on the left are from Port B and the one next to it from the JTAG's, followed by the left and right Servo, left and right IR object detectors and the left and right QTIs. There are alse three pins on the right where two of them are for the power supply.

The following pictures are the latest implementation of circuit board. All the pin connections were made below and on the top there are the pins for the servos, QTIs and IR object detectors.

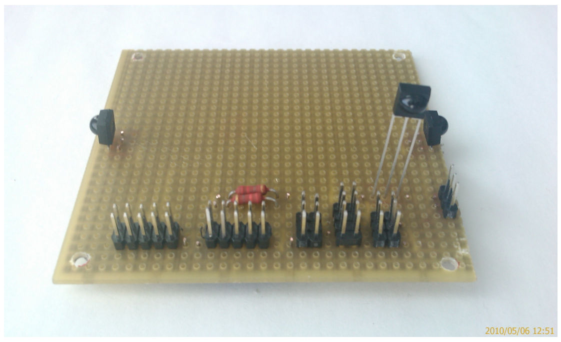

Figure 26: Circuit Board's Top

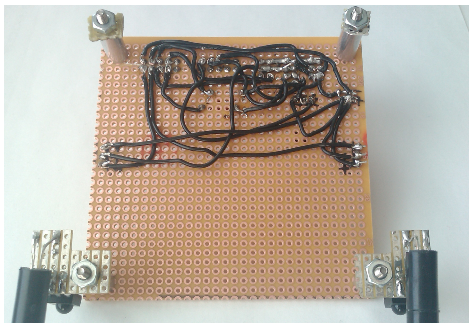

Figure 27: Circuit Board's Bottom

5.3 Software

Developing for embedded systems is quite demanding, it's required a good understandability about the target hardware, it's limitations, functionalities and structure. Therefore this section will be focusing on the development of the Sumobot's main code and drivers. Each section has a little explanation of the code and refers the significant for it's good understandability.
All the Code can be found on the appendix section, only some fragments of it may appear to support the respective description.

5.3.1 ADC Driver

The Analogue-to-digital converter code can be found in many documentation about the Avr Butterfly. This piece of code was based on the AVR Butterfly - Application Rev07 supplied on the Atmel website. The driver was originally coded to work for a light sensor, temperature sensor and voltage sensor. It was possible to reduce it significantly and to adapt it to our needs.
The ADC code is composed by two functions, one to initialize the adc depending on the sensor or the port pin that we want to read from, and other to read, do a analogue to digital conversion and calculate an average of 16 reading. Before calling the read function, it's necessary to initialize the correct pin (5 for left Infra-red sensor and 6 for the right one).

Functions:

- void ADC_ init(char sensor) - Initialize the ADC with the selected ADC-channel;
- int ADC_ read(void) - Do a Analogue-to-Digital Conversion.

5.3.2 LCD Driver

This is a basic driver for the Avr Butterfly LCD. It offers the ability to change the contrast and display strings (scrolling or static) from flash or SRAM memory only. This has been completely rewritten from the Atmel code by Dean Camera. In this version, the code performs with as much processing as possible by the string display routines rather than the interrupt so that the interrupt executes as fast as possible.

Functions:

- void LCD_ puts_ f(const char FlashData) - Rather than create a new buffer here and waste RAM, the TextBuffer global is re-used as a temp buffer. Once the ASCII data is loaded in to TextBuffer, LCD_ puts is called with it to post-process it into the correct format for the LCD interrupt.
- void LCD_ puts(const char Data) - Displays a string from SRAM onto the Butterfly's LCD;
- void LCD_ Init(void) - Initializes the Butterfly's LCD for correct operation, ready to display data;
- void LCD_ ShowColons(const uint8_ t ColonsOn) - Routine to turn on or off the LCD's colons.

5.3.3 IR Emitters/Detectors Driver

The IR emitters driver was more demanding to write due to the necessary calculations for the timers values. It's known that the IR Detectors optical filters that only allows to read signals close to 38.5 kHz. After a few tests, the optimal frequency was concluded to be 39.5 kHz with 50% duty cycle.

The following calculations will show how to get the number of ticks for the timer:
$\#Ticks = Time(ms) \div (Prescaler \times (1000 \div ClockFreq(Hz)))$
$ClockFreq = 8000000Hz$
$Prescaler = 8$
$IRFreq = 39.5kHz$
$Time = \frac{1}{IRFreq(Hz)}$

$Time = \frac{1}{39500Hz} = 25.316 \times 10^{-6}s$

$\#Ticks = 0.025316 \div (8 \times 1000 \div 8000000) = 25.316Ticks$

The TOP value conclusion: Using 50% duty cycles and toggle bit on compare match, it's only necessary half of the 25 ticks. With this calculations the top values of the counters, OCR2A and OCR0A, will have the value of 12 because it's only accepted integer numbers.

The timer0 is required to run on CTC mode to update the counter immediately with the TOP as OCR0A, toggle the bit on compare match and work with a prescaler of 8.

Following the Atmega169 data sheet is very easy to understand and select the needed bit configurations.

The 8-bit Timer0/Counter Register Description:

- **WGM01** = 1;
- **WGM00** = 0;
- **COM0A1** = 0;
- **COM0A0** = 1;
- **CS02** = 0;
- **CS01** = 1;
- **CS00** = 0;

For timer2 it's demanded to behave exactly as timer0, so the bit configurations will be the same. The register names are very similar and the TOP is OCR2O.

The 8-bit Timer2/Counter Register Description:

- **WGM21** = 1;
- **WGM20** = 0;
- **COM2A1** = 0;
- **COM2A0** = 1;
- **CS22** = 0;
- **CS21** = 1;
- **CS20** = 0;

Functions:

- void IR_ Init(void) - Timer0 (Left LED), Timer2 (Right LED) initialization frequency: 39.5 khz, duty cycle 50%.

5.3.4 QTI Driver

The QTI driver is the most simple to write and understand because is only composed by the port definitions, small delay of 1 ms for QTI charge, and then just reading the digital value of the bits. Depending on the dark or white surface, the bit can be checked if is clear or set.

Functions:

- void QTI_ Init(void) - Initializes QTIs by setting butterfly pins where QTI signal pin is connected, to output, charging their capacitors for 1 ms, and after setting butterfly pins as input for reading them;
- int Left_ QTI() - Returns 1 when left QTI sees white and 0 when black;
- int Right_ QTI() - Returns 1 when right QTI sees white and 0 when black.

5.3.5 Servo Driver

The most complex part on servo motor's drivers are the timers' configuration. It's necessary a good comprehension on the registers and on the ticks calculation time.
As the timers on the IR emitters, the same formula can be applied but not the same register configurations. The timers' need to run on fast PWM mode, with a prescaler of 8, toggle on compare match mode and supply two outputs registers.The 16-bit Timer1/Counter Register Descriptions are as following:

On TCCR1A register:

- **WGM11** = 1;
- **WGM10** = 0;
- **COM1A0** = 1;
- **COM1A1** = 1;
- **COM1B0** = 1;
- **COM1B1** = 1;

On TCCR1B register:

- **CS12** = 0;
- **CS11** = 1;
- **CS10** = 0;

The following calculations will show how to get the number of necessary ticks for the timer:
$\#Ticks = Time(ms) \div (Prescaler \times (1000 \div ClockFreq(Hz)))$
$ClockFreq = 8000000Hz$
$Prescaler = 8$

It's necessary to calculate the number of ticks for 20, 1, 1.5 and 2 ms:

$Time = 20ms$
$\#Ticks = 20 \div (8 \times 1000 \div 8000000) = 20000Ticks\ Time = 1ms$

$\#Ticks = 1000\ Time = 1.5ms$

$\#Ticks = 1500\ Time = 2ms$

$\#Ticks = 2000$

The TOP values conclusion: To obtain 20 ms signal the Maximum top should be 20000, but the desired it to have a high signal for 1, 1.5 or 2 ms and the rest of the time, until 20 ms, with a low signal. So the easiest way to do this is from the 20000 counts subtract the number of counts from the respective time of the signal, therefore it will be 20 000 less the ticks' number of desired time for low signal. When that value is reached, the bit is toggled and there will be high signal until the maximum top is reached, that is 20 ms. After this the counter is reset and starts over again.

Functions:

- void Motor_ Init(void) - Timer1 initialization;
- void Stop_ Bot(void) - Stop both Servo motors with 1.5 ms pulse output;
- void Go_ Forward(void) - Left Servo motor with 1 ms pulse output, right servo motor 2 ms pulse output;
- void Go_ Back(void) - Right Servo motor with 1 ms pulse output, left Servo motor 2 ms pulse output;
- void Go_ Left(void) - Left Servo motor with 2 ms pulse output, right Servo motor 1 ms pulse output;
- void Go_ Right(void) - Right Servo motor with 1 ms pulse output, Left Servo motor 2 ms pulse output.

5.3.6 Competition Code

The main competition's code is separated in two important parts: Initialization of all ports/timers and the main loop with the IR sensor and the QTI line sensor analysis. Inside the initialization function are invoked the prototypes of each component driver where it prepare the ports functionality and timers values.

The main system loop starts after the IR sensor values calibration. It can be executed automatically by doing a couple of readings and store the value as a reference, or give a premeditated value as reference. The chosen values where 250 due to after many tests (changing the values) and analysing on the oscilloscope, it was noticeable big electric noise interferences and a big jump of the values by approaching one object. With this meddle value, the lower electric noise won't affect the reading so much and the sensors actuate more with sensibly to any Infra-red response.

First thing to perform is the initialization of all components. Since they are organized in different source files, each has it's own initialization function. The function Initialization, called only once in the beginning just before the main loop, is responsible to execute the servo's, LCD, IR sensors and QTI initialization by defining the ports input/outputs and timers' registers.
The main code sequence is first to check the QTI values and then check the infra-red sensors. All the decisions are influenced by those readings:

- If something is detected (the white surface) in the QTI, depending on which side, is executed a set of movements to back off, turn for while and continue the search or the opponent;
- For the IR sensors it has to have a better logic sequence. First read the ADC values, shutting down, simultaneously, the opposite infra red led signal to avoid interference. After both values were stored in memory, they are compared with the references and verified in which sensor the value is greater. If both are in the same interval of values, something in front.

This way, depending on the IR sensors' and QTIs' readings, the timers' values of the servos are changed to obtain the desired movements.

Funtions:

- void Initialization(void) - Call the initialization functions of each driver
- int Left_ Ir_ Scan(void) - Initializes the ADC pin for the left IR sensor and returns the digital converted value;
- int Right_ Ir_ Scan(void) - Initializes the ADC pin for the right IR sensor and returns the digital converted value;
- void FallBack_ Right_ Move(void) - Set of movements to make the robot move back and right from the border;
- void FallBack_ Left_ Move(void) - Set of movements to make the robot move back and left from the border;
- void Delay(double millisec) - Normal Delay function.

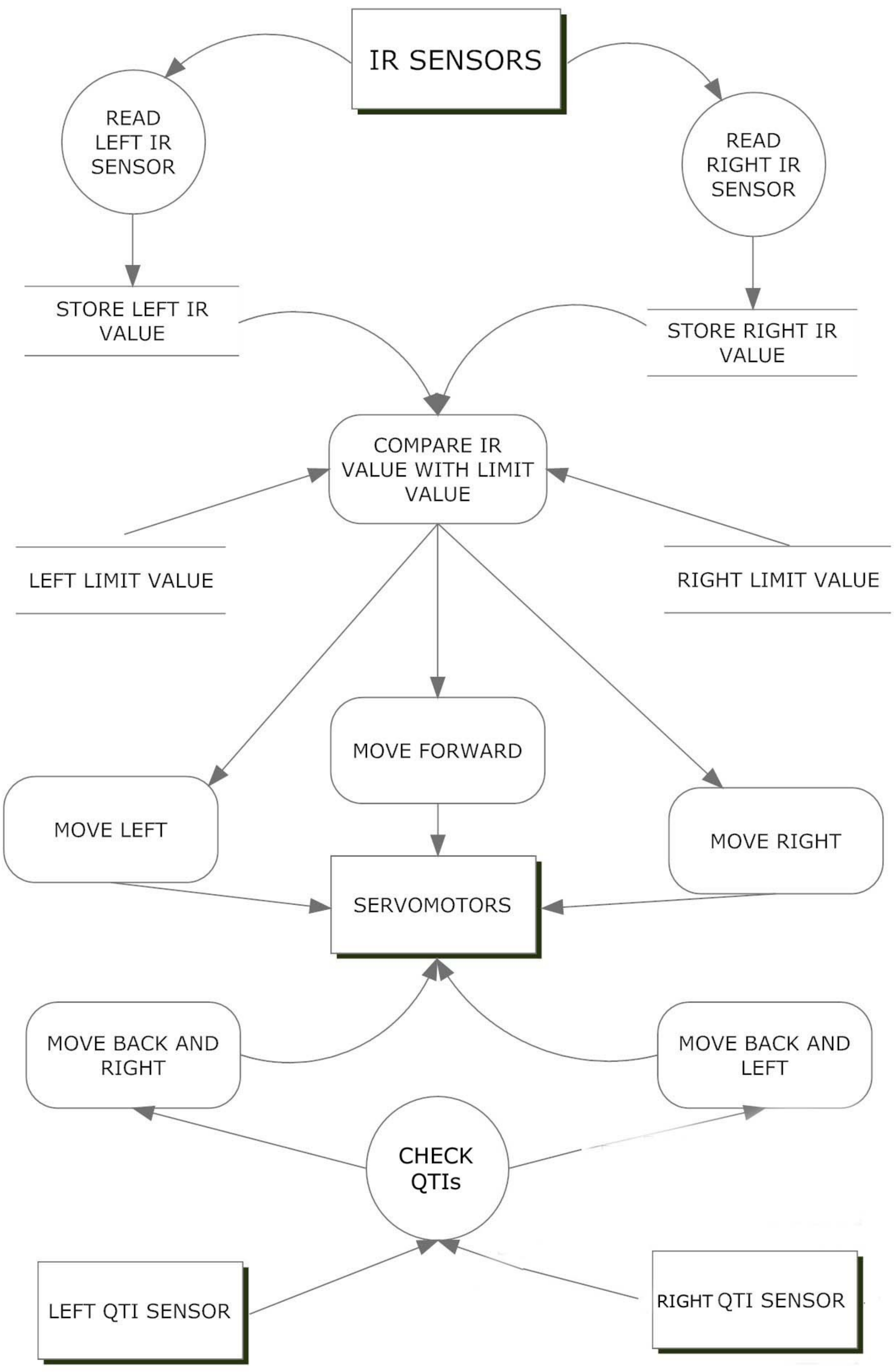

Figure 28: Context Diagram

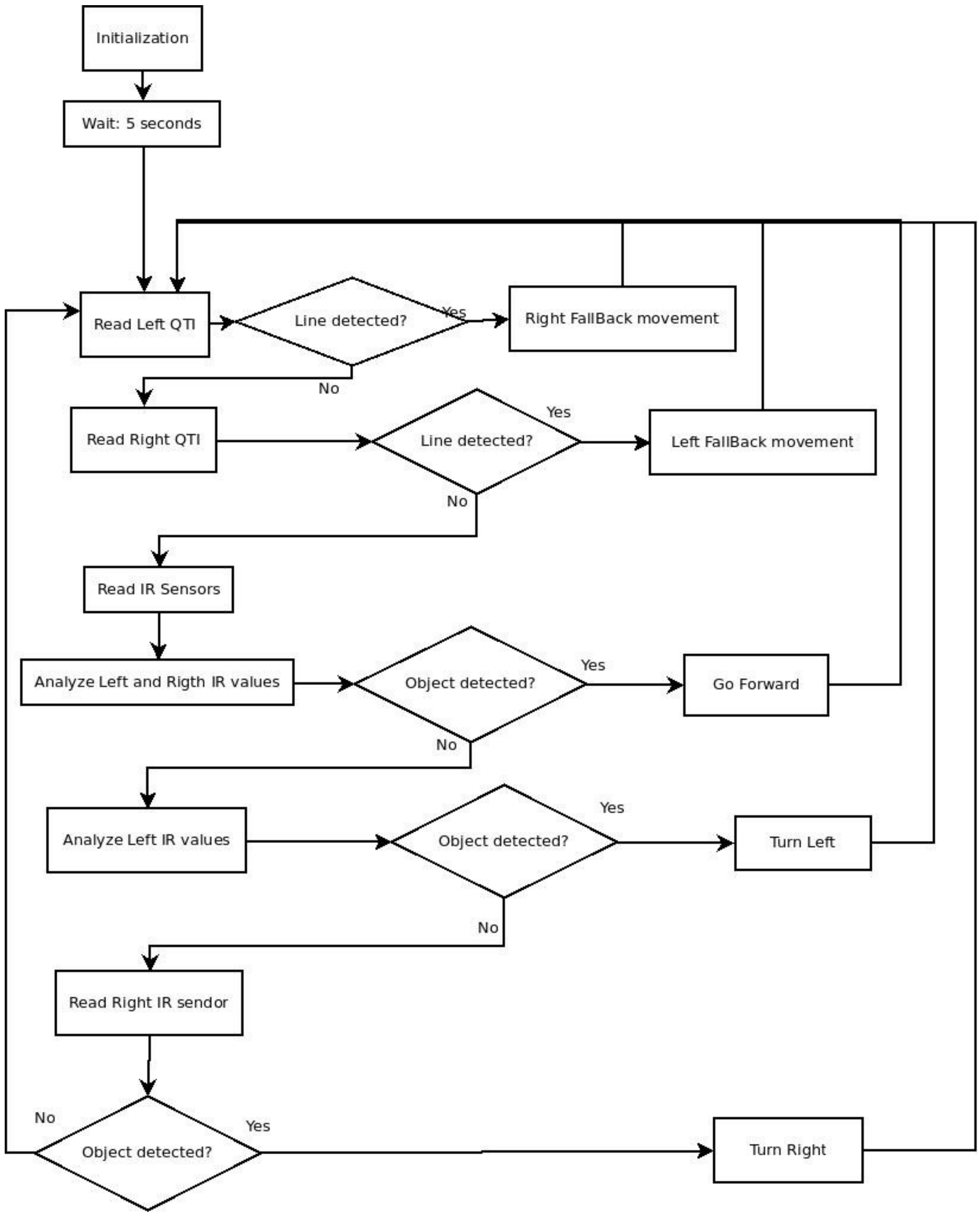

Figure 29: Data Flow Diagram

5.3.7 IR communication Driver

This is one drivers that is not essential for the Robots main goal, the Sumo fight. It was developed with the objective of improving the robot's project itself and with it more ideas came. This drivers uses one IR sensor at a time, it's basically one loop that is interrupted whenever some abnormal action happens, in other words, there are a few verifications and counters during the reception to make sure that the message is well received.

Funtions:

- void IR_ Com_ Init() - Initializes the pins where the sensors are connected as input;
- int Read_ Info(int sensor) - Main loop for reading and decode the messages;
- void send(int frame[]) - Blinks the IR led according to the frame, this way sending a message;
- void sendburst(void) - Sends a predefined frame 3 times for testing purpose;
- void one(void) - Delay of 2 ms with the IR led turned on;
- void zero(void) - Delay of 1 ms with the IR led turned on;
- void lowlevel(void) - Delay of 1 ms with the IR led turned off between each '0' and '1' sent;
- int bin2dec(char *bin) - Converts 4 bits number in the array to decimal;
- int Sensor_ bit_ set(int sensor) - checks if bit is set from the desired sensor;
- int Sensor_ bit_ clear(int sensor) - checks if bit is clear from the desired sensor;

The message: $|Header|GID|UID|Message|Parity|$
It's composed by 14 bits being the first four the Header, with the combination of "0101"; followed by a 3 bit for group ID; next a 2 bit unique ID for the robot; finally the message containing 4 bits; and for last but not least, the parity bit.

The code:
The message it's very similar to Morse code, so longer IR signals (2 ms) correspond to '1' and the short ones (1 ms) indicates the '0'. Each bit has 1 ms of break with the led off and between each frame there are 3 ms.

The driver consists on a infinite loop that breaks when some error occurred or the message is fully interpreted. Once in the Read_ Info function, it starts with a cycle that counts the time that there is no signal detected, if no IR frequency is detected for three count attempts, an time-out error value is returned. This cycle is responsible for synchronizing with the sender, since between each sent frame there is a 3 ms pause, if that pause is found, the bit counter is reset and the message is stored.

Next comes the signal counter, when something is detected on the sensor the time is counted and if it's longer than 25 and less than 50 counts a '1' is stored in the message array, otherwise '0'. In the first four bit's each time one is received, it is compared to the header, if wrong the the bit counter is reset and the loop prepares to start over from the beginning.

After a message passed the Frame Header verification and all 14 bits were stored in an array, the parity bit is calculated and tested, if incorrect a another error value will be returned from the function. Then comes the Group ID and the unique ID that represent the team and the Robot number, this may be all ones if a multicast or broadcast message is sent. Each robot has it's own group ID (team) and the unique ID, if the message does not have broadcast, multicast or unicast ID's for that team or robot, it will be returned an error integer value.

If all the frame received passes all tests, it's assured that it is destined to that robot and it's completely correct, the message bit's are converted to decimal and returned from the function.

In the following image is visible the signals from the IR led (yellow line) and sensor (blue line). When something is being sent the yellow signal goes from high to low, while when something is received, the blue signal, goes from low to high.

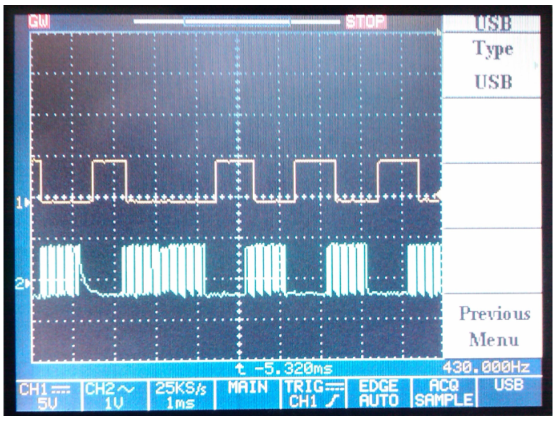

Figure 30: IR message on the Oscillator

5.3.8 IR Remote Control Mode Code

This is a very simple code that, once specified the messages and it's meanings, just a switch/case the robot can be controlled. The three top sensors are scanned and if any message is received, it's decoded and returned the respective integer. The value from the message will enter in the case and change the robots movements.

Funtions:

- int RCmode(void) - Controls the Robot according to the received message/command.

6 Testing

There are many contributing factors that influence the robot's functionalities and finding them is very time consuming. This phase has to be taken carefully and patiently, the problems may be from damaged hardware, bad cable connections, wrong software logic, etc..

There are three main components that need to be tested: Servo Motors, QTI Sensors and Infra-red Sensors. To analyse the signals of the timers outputs it was used a Digital Storage Oscilloscope.

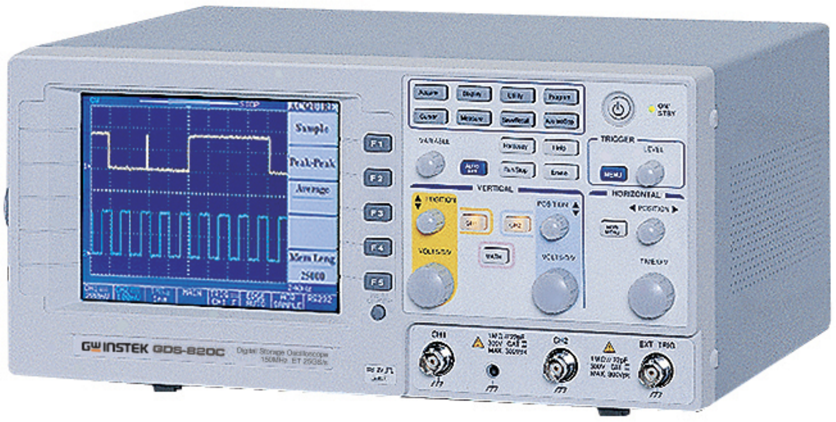

Figure 31: Digital Oscilloscope [6]

6.1 Servo Motors

The servo motors are dependent of the train pulse modulation that is generated by the timers. To test the servos' code, the oscillator must be connected the to each timer output pin and to the ground pin. In it is visible the pulse with 1.5 ms, 2 ms or 1 ms high and the rest low voltage.
If the timers are corrected the servos will start moving immediately, although it might be on the wrong direction because the set point of the potentiometer needs to be adjusted. After that is simple to understand in which direction the robot will turn by observing the servos' movements.

6.2 QTI Sensors

Since the QTIs don't need any timer's signal, they are easily tested. It's just necessary to connect the output pin and the ground pin to the oscillator and expose the sensor to a black and to a white object or surface. In the oscillator the response can be seen on the voltage level passing from high to low values. When over a black surface the phototransistor current flow decreases and when on a white surface the current raises.

6.3 IR Led and Sensors

There are two tests required to analyse the functionality of the sensor:

- First is necessary to check the frequency that the timers output to the led with the oscilloscope. It as to be close to 39 kHz, depending on how sensitive to close or far object detection the IR sensors need to be, it may be between 37 kHz and 39.5 kHz. If passed the test, it's for sure that the good condition sensor will detect something.
- Finally the voltage of the sensors may be measured verified. By approaching an object or with the hand in front of the sensor, it will react and change the voltage in the output pin from high to low.

To check if the both IR object detectors sensors were working harmoniously together, a code was developed to print the direction where an object was. Positioning an object in front of the robot, it would be printed "Front". If moved to the right, the "Right" word would appear in the LCD and similar if something detected on the left.

7 Conclusion

This chapter describes the conclusions made while analysing and building the Sumobot system. The first part of the chapter is a light written evaluation how the project fill the requirements set to it. The second part describes some of the problems, workarounds and best practices that were identified during the work.

7.1 Basic evaluation of the system

Building an Avr Butterfly embedded system for a Sumobot can be very tricky when you don't have good knowledge for that. In my particular case I had the basic familiarity with the similar Atmega boards, although with this one the things done with it reached greater level of difficulty. It was required to read and understand entirely the microcontrollers functionalities, however the project ran very smoothly and well. The main goal was reached and there was still time for improvements. The result of this thesis is a documentation where anyone with some training and understanding can follow and reproduce simple SumoBot system for the Avr Butterfly microcontroller.

The Sumobot surpassed the expectations, it is able to compete on a sumo tournament and the system is working harmoniously with components. During the competition the robot detected the opponent and pushed it out from the ring several times and it was capable of maintaining itself on the ring without going out by mistake. The servos movements and speed was quite good and fast, although not very fluent. Also there wasn't any visible delays on the response to object detections neither to line detection.

Analysing the requirements it's noticeable that all requirements were conquered. The imposed hardware and tools were all used, the Avr's pins were carefully selected depending on each functionality, all drivers work it the respective components and the latest circuit board is much more organized than the first design.

The software allowed the Sumobot to win several matches, proving that is stable and efficient. The Remote control Mode has an acceptable performance, even though the distance sensitivity of the sensors to capture the signals could be stronger.
The Sumobot design and structure stability respected the regulations and rules of the official Sumobots' Competition.

7.2 Problems and possible solutions

During the project development there were several problems that needed to be confronted and they weren't easy neither fast to solve.

Hardware: In the beginning there was a few components lost because of distracted handling. An IR led was burned because the resistors were forgotten, a few IR sensors were destroyed due to their heat sensitivity when soldering them in the circuits.
The biggest issue was the electric noise. It was found that it was significant only when the Sumobot was running out of batteries, this caused the sensor's signals to have less

amplitude and therefore the noise could disrupt them. A low pass filter on the IR sensors output was implemented but without success. The solution consisted on powering up the Sumobot either with the power supply when testing or with new batteries for the matches.

Software: The servo motors' timers/counters were complicated to get right because the wrong register were selected or the formula wasn't the correct one. With the IR led signal was the same, but once understood how the counters worked, no more problems were found.
In the until the very end, the robot suffered of inaccurate delay times because the CPU clock in the Eclipse IDE or AVR studio were wrongly defined, the best solution was to redefine it in the code.
The latest software issue was for the IR communication. Since to count the signal's time, a while condition incrementing a variable was used, it was quite complicated to find how many increments the microcontroller could do in 1 ms. The best solution was putting the Avr Butterfly to count and display on the LCD the counter value at end of the cycle. Knowing that the code lines between the counter wasted time, most of the time had to be by trial and error until a good level of accuracy was reached.

7.3 Summary

Designing and implementing an embedded system for a Sumobot using an Avr Butterfly is quite a enjoyable and challenging job, which involves many tasks from building the robot, developing the system and testing it step by step. The Avr butterfly is a very good microcontroller for this type of works, because it has a good variety of peripheral, is compact and has low power design, also the Sumobot kit from Parallax was a great set of tools to build the little robot.

Before starting anything is advisable to have a good look to the projects requirements, available hardware and necessary goals to archive. After that is possible to decide the type of system do be developed, what components will be needed to be built and what tools will be used.

Once the necessary components like the circuit board and the IR emitters/sensors are completed, the cables can be done and the robot assembled. The next phase is start programming for the easiest components first so that the work can be moralizing in the beginning and to start showing faster results. During the project each component was worked each at the time, it started with the QTI's, then Servos and finally the IR emitters and sensors.

With all drivers completed and tested individually the main code can be easily done. Remaining just the final task that is to test the system working together with the components and putting the robot to push some opponents from the ring.

8 Appendix

ADC Driver

Listing 1: ADC Driver Header file

```c
#define L_IR                    5
#define R_IR                    6

#ifndef ADC_H_
#define ADC_H_

void ADC_init(char);
int ADC_read(void);

#endif /* ADC_H_ */
```

Listing 2: ADC Driver source file

```c

//***************************************************************************
//
//  File........: ADC.c
//
//  Author(s)...: ATMEL Norway
//
//  Target(s)...: ATmega169
//
//  Compiler....: AVR-GCC 3.3.1; avr-libc 1.0
//
//  Description.: AVR Butterfly ADC routines
//
//  Revisions...: 1.0
//
//  YYYYMMDD - VER. - COMMENT                                       - SIGN.
//
//  20030116 - 1.0  - Created                                       - LHM
//  20031009          port to avr-gcc/avr-libc                      - M.Thomas
//
//***************************************************************************

#include <avr/io.h>
#include <avr/pgmspace.h>
#include "ADC.h"
#include "main.h"

/****************************************************************************
 *
 *   Function name : ADC_init
 *
 *   Returns :        None
 *
 *   Parameters :     char input
 *
 *   Purpose :        Initialize the ADC with the selected ADC-channel
 *
 ****************************************************************************/
void ADC_init(char input) {

  ADMUX = input; // external AREF and ADCx

  // set ADC prescaler to , 1MHz / 8 = 125kHz
  ADCSRA = (1 << ADEN) | (1 << ADPS1) | (1 << ADPS0);

  input = ADC_read(); // dummy
}

/****************************************************************************
 *
 *   Function name : ADC_read
 *
 *   Returns :        int ADC
 *
 *   Parameters :     None
 *
 *   Purpose :        Do a Analog to Digital Conversion
 *
 ***************************************************************************/
int ADC_read(void) {
  char i;
  int ADC_temp;
  int ADCr = 0;

  // To save power, the voltage over the LDR and the NTC is turned off when not used
  // This is done by controlling the voltage from a I/O-pin (PORTF3)
  sbiBF(PORTF, PF3);   // Enable the VCP (VC-peripheral)
  sbiBF(DDRF, DDF3);

  sbiBF(ADCSRA, ADEN);   // Enable the ADC

  //do a dummy readout first
  ADCSRA |= (1 << ADSC); // do single conversion
  while (!(ADCSRA & 0x10))
    ; // wait for conversion done, ADIF flag active

  // do the ADC conversion 16 times for better accuracy
  for (i = 0; i < 16; i++)
  {
    ADCSRA |= (1 << ADSC);   // do single conversion
    while (!(ADCSRA & 0x10))
      ;  // wait for conversion done, ADIF flag active

    ADC_temp = ADCL;       // read out ADCL register
    ADC_temp += (ADCH << 8); // read out ADCH register

    ADCr += ADC_temp; // accumulate result (16 samples) for later averaging
  }

  ADCr = ADCr >> 3; // average the 16 samples

  cbiBF(PORTF,PF3); // disable the VCP
  cbiBF(DDRF,DDF3);

  cbiBF(ADCSRA, ADEN); // disable the ADC

  return ADCr;
}
```

LCD Driver

Listing 3: LCD Driver Header file

```
/***************************************************************
 BUTTLCD -- Butterfly LCD Driver

 Copyright (C) Dean Camera, 2008

 dean [at] fourwalledcubicle [dot] com
 www.fourwalledcubicle.com
 ***************************************************************/

#ifndef LCDDRIVER_H
#define LCDDRIVER_H

// INCLUDES:
#include <avr/io.h>
#include <avr/pgmspace.h>
```

```
#include <avr/interrupt.h>
#include <stdbool.h>

// EXTERNAL VARIABLES:
extern volatile uint8_t ScrollFlags;

// DEFINES:
#define LCD_LCDREGS_START          ((uint8_t*)&LCDDR0)
#define LCD_SPACE_OR_INVALID_CHAR  0xFF

#define LCD_CONTRAST_LEVEL(level)  do{ LCDCCR = (0x0F & level); }while(0)
#define LCD_WAIT_FOR_SCROLL_DONE() do{ while (!(ScrollFlags & LCD_FLAG_SCROLL_DONE)) {} }while(0)

#define LCD_SCROLLCOUNT_DEFAULT    6
#define LCD_DELAYCOUNT_DEFAULT     20
#define LCD_TEXTBUFFER_SIZE        20
#define LCD_SEGBUFFER_SIZE         19
#define LCD_DISPLAY_SIZE           6

#define LCD_FLAG_SCROLL            (1 << 0)
#define LCD_FLAG_SCROLL_DONE       (1 << 1)

// PROTOTYPES:
void LCD_puts_f(const char *FlashData);
void LCD_puts(const char *Data);
void LCD_Init(void);
void LCD_ShowColons(const uint8_t ColonsOn);

#if defined(INC_FROM_DRIVER)
static inline void LCD_WriteChar(const uint8_t Byte, const uint8_t Digit);
#endif
#endif
```

Listing 4: LCD Driver Source file

```
/***********************************************************
 BUTTLCD -- Butterfly LCD Driver

 Copyright (C) Dean Camera, 2008

 dean [at] fourwalledcubicle [dot] com
 www.fourwalledcubicle.com
 ***********************************************************/

// Include files.
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include "main.h"

#include "LCD_driver.h"

//LCD Text+ Nulls for scrolling + Null Termination
static volatile char
    TextBuffer[LCD_TEXTBUFFER_SIZE + LCD_DISPLAY_SIZE + 1] = { };
static volatile uint8_t StrStart = 0;
static volatile uint8_t StrEnd = 0;
static volatile uint8_t ScrollCount = 0;
static volatile uint8_t UpdateDisplay = false;
static volatile uint8_t ShowColons = false;
volatile uint8_t ScrollFlags = 0;

const uint16_t LCD_SegTable[] PROGMEM =
{     0xEAA8, // '*'
    0x2A80, // '+'
    0x4000, // ','
    0x0A00, // '-'
    0x0A51, // '.' Degree sign
    0x4008, // '/'
```

```c
    0x5559, // '0'
    0x0118, // '1'
    0x1e11, // '2
    0x1b11, // '3
    0x0b50, // '4
    0x1b41, // '5
    0x1f41, // '6
    0x0111, // '7
    0x1f51, // '8
    0x1b51, // '9'
    0x0000, // ':' (Not defined)
    0x0000, // ';' (Not defined)
    0x8008, // '<'
    0x1A00, // '='
    0x4020, // '>'
    0x0000, // '?' (Not defined)
    0x0000, // '@' (Not defined)
    0x0f51, // 'A' (+ 'a')
    0x3991, // 'B' (+ 'b')
    0x1441, // 'C' (+ 'c')
    0x3191, // 'D' (+ 'd')
    0x1e41, // 'E' (+ 'e')
    0x0e41, // 'F' (+ 'f')
    0x1d41, // 'G' (+ 'g')
    0x0f50, // 'H' (+ 'h')
    0x2080, // 'I' (+ 'i')
    0x1510, // 'J' (+ 'j')
    0x8648, // 'K' (+ 'k')
    0x1440, // 'L' (+ 'l')
    0x0578, // 'M' (+ 'm')
    0x8570, // 'N' (+ 'n')
    0x1551, // 'O' (+ 'o')
    0x0e51, // 'P' (+ 'p')
    0x9551, // 'Q' (+ 'q')
    0x8e51, // 'R' (+ 'r')
    0x9021, // 'S' (+ 's')
    0x2081, // 'T' (+ 't')
    0x1550, // 'U' (+ 'u')
    0x4448, // 'V' (+ 'v')
    0xc550, // 'W' (+ 'w')
    0xc028, // 'X' (+ 'x')
    0x2028, // 'Y' (+ 'y')
    0x5009, // 'Z' (+ 'z')
    0x1441, // '['
    0x8020, // '\'
    0x1111, // ']'
    0x0000, // '^' (Not defined)
    0x1000 // '_'
    };

// ========================================================================================

/*************************************************************
 NAME:       | LCD_Init
 PURPOSE:    | Initializes the Butterfly's LCD for correct operation, ready to display data
 ARGUMENTS: | None
 RETURNS:    | None
 *************************************************************/
void LCD_Init(void) {
  // Set the initial contrast level to maximum:
  LCD_CONTRAST_LEVEL(0x0F);

  // Select asynchronous clock source, enable all COM pins and enable all segment pins:
  LCDCRB = (1 << LCDCS) | (3 << LCDMUX0) | (7 << LCDPM0);

  // Set LCD prescaler to give a framerate of 64Hz:
  LCDFRR = (0 << LCDPS0) | (3 << LCDCD0);

  // Enable LCD and set low power waveform, enable start of frame interrupt:
  LCDCRA = (1 << LCDEN) | (1 << LCDAB) | (1 << LCDIE);
```

```c
}

/*************************************************************
 NAME:       | LCD_puts
 PURPOSE:    | Displays a string from flash onto the Butterfly's LCD
 ARGUMENTS:  | Pointer to the start of the flash string
 RETURNS:    | None
 *************************************************************/
void LCD_puts_f(const char *FlashData) {
  /* Rather than create a new buffer here (wasting RAM), the TextBuffer global
   is re-used as a temp buffer. Once the ASCII data is loaded in to TextBuffer,
   LCD_puts is called with it to post-process it into the correct format for the
   LCD interrupt.                                                            */

  strcpy_P((char*) &TextBuffer[0], FlashData);
  LCD_puts((char*) &TextBuffer[0]);
}

/*************************************************************
 NAME:       | LCD_puts
 PURPOSE:    | Displays a string from SRAM onto the Butterfly's LCD
 ARGUMENTS:  | Pointer to the start of the SRAM string
 RETURNS:    | None
 *************************************************************/
void LCD_puts(const char *Data) {
  uint8_t LoadB = 0;
  uint8_t CurrByte;
  uint8_t Nulls;

  do {
    CurrByte = *(Data++);

    switch (CurrByte) {
    case 'a' ... 'z':
      CurrByte &= ~(1 << 5); // Translate to upper-case character
    case '*' ... '_': // Valid character, load it into the array
      TextBuffer[LoadB++] = (CurrByte - '*');
      break;
    case 0x00:
    //Null termination of the string - ignore for now so the nulls can be appended below
      break;
    default: // Space or invalid character, use 0xFF to display a blank
      TextBuffer[LoadB++] = LCD_SPACE_OR_INVALID_CHAR;
    }
  } while (CurrByte && (LoadB < LCD_TEXTBUFFER_SIZE));

  ScrollFlags = ((LoadB > LCD_DISPLAY_SIZE) ? LCD_FLAG_SCROLL : 0x00);

  for (Nulls = 0; Nulls < 7; Nulls++) {
   // Load in nulls to ensure that when scrolling, the display clears before wrapping
    TextBuffer[LoadB++] = LCD_SPACE_OR_INVALID_CHAR;
  }
  TextBuffer[LoadB] = 0x00; // Null-terminate string

  StrStart = 0;
  StrEnd = LoadB;
  ScrollCount = LCD_SCROLLCOUNT_DEFAULT + LCD_DELAYCOUNT_DEFAULT;
  UpdateDisplay = true;
}

/*************************************************************
 NAME:       | LCD_WriteChar (static, inline)
 PURPOSE:    | Routine to write a character to the correct LCD registers for display
 ARGUMENTS:  | Character to display, LCD character number to display character on
 RETURNS:    | None
 *************************************************************/
static inline void LCD_WriteChar(const uint8_t Byte, const uint8_t Digit) {
  uint8_t* BuffPtr = (uint8_t*) (LCD_LCDREGS_START + (Digit >> 1));
  uint16_t SegData = 0x0000;
  uint8_t BNib;

  if (Byte != LCD_SPACE_OR_INVALID_CHAR) // Null indicates invalid character or space
    SegData = pgm_read_word(&LCD_SegTable[Byte]);

  for (BNib = 0; BNib < 4; BNib++) {
    uint8_t MaskedSegData = (SegData & 0x0000F);

    if (Digit & 0x01)
      *BuffPtr = ((*BuffPtr & 0x0F) | (MaskedSegData << 4));
    else
      *BuffPtr = ((*BuffPtr & 0xF0) | MaskedSegData);

    BuffPtr += 5;
    SegData >>= 4;
  }
}

/**********************************************************
 NAME:       | LCD_ShowColons
 PURPOSE:    | Routine to turn on or off the LCD's colons
 ARGUMENTS: | Boolean - true to turn on colons
 RETURNS:    | None
 **********************************************************/
void LCD_ShowColons(const uint8_t ColonsOn) {
  ShowColons = ColonsOn;
  UpdateDisplay = true;
}
/**********************************************************
 NAME:       | LCD_vect (ISR, blocking)
 PURPOSE:    | ISR to handle the display and scrolling of the current
       | display string onto the LCD
 ARGUMENTS: | None
 RETURNS:    | None
 *********************************************************/
ISR(LCD_vect, ISR_NOBLOCK)
{
  uint8_t Character;
  if (ScrollFlags & LCD_FLAG_SCROLL) {
    if (!(ScrollCount--)) {
      UpdateDisplay = true;
      ScrollCount = LCD_SCROLLCOUNT_DEFAULT;
    }
  }

  if (UpdateDisplay) {
    for (Character = 0; Character < LCD_DISPLAY_SIZE; Character++) {
      uint8_t Byte = (StrStart + Character);

      if (Byte >= StrEnd)
        Byte -= StrEnd;

      LCD_WriteChar(TextBuffer[Byte], Character);
    }

    if ((StrStart + LCD_DISPLAY_SIZE) == StrEnd) // Done scrolling message on LCD once
      ScrollFlags |= LCD_FLAG_SCROLL_DONE;

    if (StrStart++ == StrEnd)
      StrStart = 1;

    if (ShowColons)
      *((uint8_t*) (LCD_LCDREGS_START + 8)) = 0x01;
    else
      *((uint8_t*) (LCD_LCDREGS_START + 8)) = 0x00;

    UpdateDisplay = false; // Clear LCD management flags, LCD update is complete
  }
}
```

IR Driver

Listing 5: IR Driver Header file

```c
#ifndef IR_SENSORS_H_
#define IR_SENSORS_H_

void IR_Init(void);
#endif /* IR_SENSORS_H_ */
```

Listing 6: IR Driver Source file

```c

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>

#include "IR_sensors.h"
#include "main.h"

/* Fast PMW timer mode function: #ticks= Ms/(prescaler*1000/Clock Freq(Hz))
 *
 *Freq = 39.5
 *time = 1/freq
 *
 * MS   #ticks
 * 0,25 - 25
 *
 *
 * with half dutty cycle we have half ticks for toggling the bit
 * it's needed half time:
 *
 * #ticks = 12;
 */

void IR_Init() {

  SREG |= 1 << SREG_I;   //Global interrupt enable

  /*8-bit Timer/Counter0*/
  /* TIMER 0 WGM: CTC , CS0x:Preescaler 8*/
  TCCR0A |= 1 << WGM01 | 1 << COM0A0 | 1 << CS01;
  TCCR0A &= ~(1 << COM0A1);
  TCCR0A &= ~(1 << WGM00);
  TCCR0A &= ~(1 << CS02);
  TCCR0A &= ~(1 << CS00);

  TIMSK0 |= 1 << TOIE0;   //Timer0 interrupt overflow enable

  /*TIMER0 OUTPUTS*/
  DDRB |= 1 << DD4; //OC0A Data direction register
  PORTB &= ~(1 << DD4); //OC0A

  /*8-bit Timer/Counter2*/
  /* TIMER 2 WGM: CTC, CS0x:Preescaler 8*/
  TCCR2A |= 1 << WGM21 | 1 << COM2A0 | 1 << CS21;
  TCCR2A &= ~(1 << COM2A1);
  TCCR2A &= ~(1 << WGM20);
  TCCR2A &= ~(1 << CS22);
  TCCR2A &= ~(1 << CS20);

  TIMSK2 |= 1 << TOIE2;   //Timer2 interrupt overflow enable

  /*TIMER2 OUTPUTS*/
  DDRB |= 1 << DD7;      //OC2A Data direction register
  PORTB &= ~(1 << DD7);   //OC2A

  OCR0A = 12; //TOP L
  OCR2A = 12; //TOP R
}
```

QTI Driver

Listing 7: QTI Driver Header file

```c
#ifndef QTI_DRIVER_H_
#define QTI_DRIVER_H_

void QTI_Init(void);
int Left_QTI();
int Right_QTI();

#endif /* QTI_DRIVER_H_ */
```

Listing 8: QTI Driver Source file

```c

#include <avr/io.h>
#include "main.h"
#include "QTI_driver.h"

/**********************************************************
 *
 *   Function name : QTI_Init
 *
 *   Purpose :  Initializes PORTD 0 and 2 for the QTI sensor
 *
 ***********************************************************/
void QTI_Init(void) {

  //Data direction register -> out
  DDRB |= 1 << DD0;
  DDRB |= 1 << DD2;

  //Set bit to high to initialize
  PORTB |= 1 << DD0;
  PORTB |= 1 << DD2;

  //Time the capacitors need to charge
  Delay(1);

  //Set bit to LOW to start the normal reading
  DDRB &= ~(1 << DD0);
  DDRB &= ~(1 << DD2);

}

/**********************************************************
 *
 *   Function name : Left_QTI
 *
 *   Purpose :        Check if the QTI bit is set or clear.
 *             1 - Black surface
 *             0 - White line
 *
 *   Return:      integer 1 or 0
 *
 **********************************************************/
int Left_QTI() {

  if (bit_is_set(PINB,PINB0)) {
    //Black surface
    return 1;

  } else {
    //White line
    return 0;
  }
}

/**********************************************************
 *
 *   Function name : Right_QTI
 *
 *   Purpose :        Check if the QTI bit is set or clear.
 *             1 - Black surface
 *             0 - White line
 *
 *   Return:      integer 1 or 0
 *
 **********************************************************/
int Right_QTI() {

  if (bit_is_set(PINB,PINB2)) {
    //Black surface
    return 1;

  } else {
    //White line
    return 0;
  }
}
```

Servo Driver

Listing 9: Servo Driver Header file

```
#ifndef MOTOR_DRIVER_H_
#define MOTOR_DRIVER_H

void Motor_Init();
void Stop_Bot();
void Go_Forward();
void Go_Back();
void Go_Left();
void Go_Right();

#endif /* MOTOR_DRIVER_H_ */
```

Listing 10: Servo Driver Source file

```
#include "motor_driver.h"
#include "main.h"

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include "math.h"
#include <string.h>

/* Fast PMW timer mode function: #ticks= Ms/(prescaler*1000/Clock Freq(Hz))
 *
 *
 * MS   #ticks
 * 20   ms - 20000
 * 1    ms -  2000 -> L reverse/ R Forward
 * 1.5 ms -  1500 -> L Stop/ R stop
 * 2    ms -  1000 -> L Forward/ R reverse
 *
 *Forward:
 *  OCR1A = 20000 - 316456 2000;
 *  OCR1B = 20000 - 1000;
 *
 *Left:
 *  OCR1A = 20000 - 1000;
 *  OCR1B = 20000 - 1000;
 *
 *Right:
 *  OCR1A = 20000 - 2000;
 *  OCR1B = 20000 - 2000;
 *
 *Back:
 *  OCR1A = 20000 - 1000;
 *  OCR1B = 20000 - 2000;
 *
 *Stop:
 *  OCR1A = 20000 - 1500;
 *  OCR1B = 20000 - 1500;

 * */

void Motor_Init() {

  DDRB |= 1 << DD5; // Data direction register -> out
  PORTB |= ~(1 << DD5);

  DDRB |= 1 << DD6; // Data direction register -> out
  PORTB |= ~(1 << DD6);

  //Timer 8 prescaler
  //Fast PWM mode - 15
  //Toggle on compare match

  TCCR1A |= 1 << COM1A0 | 1 << COM1A1 | 1 << COM1B0 | 1 << COM1B1 | 1 << WGM11;
  TCCR1A &= ~(1 << WGM10);

  TCCR1B |= 1 << CS11 | 1 << WGM13 | 1 << WGM12;
  TCCR1B &= ~(1 << CS10);
  TCCR1B &= ~(1 << CS12);

  //TOP Value
  ICR1 = 20000;

}

/****************************************************************************
 *   Function name : Go_Forward
 *
 *   Purpose :       Change timer values so that the servos' move forward
 ****************************************************************************/
void Go_Forward() {

  OCR1A = 18000;
  OCR1B = 19000;
}

/****************************************************************************
 *   Function name : Go_Left
 *
 *   Purpose :       Change timer values so that the robot turn Left
 ****************************************************************************/
void Go_Left() {

  OCR1A = 19000;
  OCR1B = 19000;

}

/****************************************************************************
 *   Function name : Go_Right
 *
 *   Purpose :        Change timer values so that the robot turn right
 ****************************************************************************/
void Go_Right() {

  OCR1A = 18000;
  OCR1B = 18000;

}

/****************************************************************************
 *   Function name : Go_Back
 *
 *   Purpose :        Change timer values so that the robot reverse
 ****************************************************************************/
void Go_Back() {

  OCR1A = 19000;
  OCR1B = 18000;
}
```

Main code

Listing 11: Main Header file

```
#ifndef MAIN_H_
#define MAIN_H_


#define F_CPU 8000000

#define BOOL    char
#define FALSE   0
#define TRUE    (!FALSE)
#define NULL ((void *)0)

// Macro definitions
// sbi and cbi are not longer supported by the avr-libc
// to avoid version-conflicts the macro-names have been
// changed to sbiBF/cbiBF "everywhere"

//set bit in port
#define sbiBF(port,bit)  (port |= (1<<bit))
//clear bit in port
#define cbiBF(port,bit)  (port &= ~(1<<bit))

//functions' definition
void Initialization(void);
int Left_Ir_Scan(void);
int Right_Ir_Scan(void);
void FallBack_Right_Move(void);
void FallBack_Left_Move(void);
void Wait(void);
void Delay(double millisec);

#endif /* MAIN_H_ */
```

Listing 12: Main Source file

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <avr/wdt.h>
#include <avr/version.h>
#include <string.h>
#include <stdlib.h>

```

```c
#include "main.h"
#include "ADC.h"

#include "LCD_driver.h"
#include "IR_sensors.h"
#include "QTI_driver.h"
#include "motor_driver.h"

int main(void) {

  int L_IR_temp, L_IR_REF;
  int R_IR_temp, R_IR_REF;


  // Program initialization
  Initialization();

  //Enable interrupts
  sei();


  //IR Manual Calibration
  L_IR_REF = 250;
  R_IR_REF = 250;

  //5 second wait phase before match
  Wait();

  LCD_puts("Start");

  while (1) // Main loop
  {

    /*Verify if the white line is detected by the QTI sensors
     * if so perform the fallback movement to the right or left
     * depending on the sensor side */

    //if white line is detected on the left
    if (!Left_QTI()) {
      FallBack_Right_Move();
    }

    //if white line is detected on the right
    if (!Right_QTI()) {
      FallBack_Left_Move();
    }


    /*Read the values of the IR sensor thru the ADC and keep
     * them in a variable*/

    L_IR_temp = Left_Ir_Scan();
    R_IR_temp = Right_Ir_Scan();



    /*Analyze the values received from each sensor and
     * compare them to a chosen value that is the reference value.
     * This value is defined in the calevince %.pdfibration reference where it can be
     * done automatically or manually */

    /*If the values are both lower than the references means that
     * Something is detected in front so Move the robot forward*/

    if (L_IR_temp < L_IR_REF && R_IR_temp < R_IR_REF) {
      Go_Forward();
      LCD_puts("Forward");
      Delay(500);


    }

    /*If the values are greator than the references the sensors
     * are not data flowseeing anything besides so the robor will move forward*/

    if (L_IR_temp >= L_IR_REF && R_IR_temp >= R_IR_REF) {
      LCD_puts("BLIND");
      Go_Forward();
      Delay(500);

    }


    /*If the value on the Left is lower and on the right is
     * bigger than the references, there is something on the Left */

    if (L_IR_temp < L_IR_REF && R_IR_temp >= R_IR_REF) {

        Go_Left();
        LCD_puts("LEFT");
        Delay(500);


    }


    /*If the value on the Right is lower and on the Left is
     * bigger than the references, there is something on the Right */

    if (L_IR_temp >= L_IR_REF && R_IR_temp < R_IR_REF) {

        Go_Right();
        LCD_puts("RIGHT");
        Delay(500);


    }

  }
  return 0;

}

/*****************************************************************************
 *   Function name : Initialization
 *
 *   Purpose :       Initializate the different modules
 *****************************************************************************/
void Initialization(void) {

  Motor_Init(); //Initialize Motor

  QTI_Init(); //Initialize QTI sensors

  IR_Init(); //initialize IR sensors

  LCD_Init(); // initialize the LCD

}

/*****************************************************************************
 *   Function name : FallBack_Left_Move
 *
 *   Purpose :       Fallback Moviment and left turn due to line detection
 *****************************************************************************/
void FallBack_Left_Move(void) {

  Go_Back();
  Delay(700);
  Go_Left();
```

```c
  Delay(700);
  Go_Forward();

}

/*****************************************************************************
 *   Function name : FallBack_Right_Move
 *
 *   Purpose :        Fallback Moviment and right turn due to line detection
 *****************************************************************************/
void FallBack_Right_Move(void) {

  Go_Back();
  Delay(700);
  Go_Right();
  Delay(700);
  Go_Forward();
}

/*****************************************************************************
 *   Function name : Left_Ir_Scan
 *
 *   Purpose :        Turn off Right led, turn on the left Led,
 *           initiate the ADC for the Left IR and
 *           the value from the sensor is read
 *
 *   Return :     The Value from ADC_read();
 *****************************************************************************/
int Left_Ir_Scan(void) {

  //Check IR LED
  /*turn off Right*/
  PORTB &= ~(1 << DD7); //OC2A

  /*TIMER0 OUTPUTS turn on LEFT*/
  PORTB |= (1 << DD4); //OC0A

  ADC_init(L_IR);

  return ADC_read();
}

/*****************************************************************************
 *   Function name : Right_Ir_Scan
 *
 *   Purpose :        Turn off Left led, turn on the right Led,
 *           initiate the ADC for the Right IR and
 *           the value from the sensor is read.
 *
 *   Return :     The Value from ADC_read().
 *****************************************************************************/
int Right_Ir_Scan(void) {

  /*turn on Right led*/
  PORTB |= (1 << DD7); //OC2A

  /*turn off Left led*/
  PORTB &= ~(1 << DD4); //OC0A

  ADC_init(R_IR);
  return ADC_read();

}

/*****************************************************************************
 *   Function name : Wait
 *
 *   Purpose :        5 Seconds delay; time required before the match.
 *****************************************************************************/
void Wait() {
  LCD_puts("5 SEC");
  Delay(5000);

}

/***************************************************************************
 *
 *   Function name : Delay
 *
 *   Parameters :     double millisec
 *
 *   Purpose :        Delay-loop
 *
 ***************************************************************************/
void Delay(double millisec) {
  uint8_t i;

  millisec = 4 * millisec;

  while (millisec--)

    for (i = 0; i < 125; i++)
      asm volatile ("nop"::);
}
```

IR Communication Driver

Listing 13: IR communication Header file

```
#ifndef IR_COM_H_
#define IR_COM_H_

void IR_Com_Init();
int Read_Info(int *sensor);
void send(int frame[]);
void sendburst(void)
void one(void);
void zero(void);
void lowlevel(void);
int bin2dec(char *bin);
int Sensor_bit_set(int *sensor);
int Sensor_bit_clear(int *sensor);

#endif /* IR_COM_H_ */
```

Listing 14: IR communication Source file

```
#include <avr/io.h>
#include "stdio.h"
#include <string.h>
#include <math.h>

#include "main.h"
#include "IR_sensors.h"
#include "ADC.h"
#include "LCD_driver.h"
#include "IR_Com.h"

//combination on the start of each message
const char Frame_Header[4] = "0101";
//Broadcast and Multicast ID
const char Broad_Multi_Unique_ID[3] = "111";
const char Broadcast_Global_ID[2] = "11";

//Bot Unique ID and Team ID
const char Bot_ID[3] = "001"; //--> Bot ID 001
const char Team_GID[2] = "00"; // --> Bot's Team 02

/***************************************************
 *  UID  GID
 * 111  11  --> Broadcast (ALL BOTS)
 * 111  01  --> Multicast (All bots of team 1)
 * 001  01  --> Singlecast (Bot 001 in team 01)
 * ************************************************/

/*****************************************************************************
 *   Function name :  void IR_Com_Init()
 *
 *   Purpose : Initialize the IR communication components.
 *
 *   Return : void
 *****************************************************************************/
void IR_Com_Init() {

  //set the Sensor pins to input mode
  //...............................
  //Set bit to LOW to start the normal reading
  //DDRE &= ~(1 << DD6);

  //pb1
  DDRB &= ~(1 << DD1);

  //pb3
  DDRB &= ~(1 << DD3);

  //pf4
  DDRF &= ~(1 << DD4);

}

/*****************************************************************************
 *   Function name : int Read_Info(void)
 *
 *   Purpose : Receive and decode the message via Infra-red.
 *
 *   Return : returns integer depending on the error or successful  command
 *
 *         n success -> n number of the conversion
 *         0 end of function (impossible)
 *         -1 time-out error
 *         -2 Wrong Frame Header error
 *         -3 Wrong Unique ID -> message is not for this robot
 *         -4 Wrong Group ID -> message is not for this group
 *         -5 Parity Error check
 *
 *****************************************************************************/

int Read_Info(int *sensor) {

  int count = 0;
  char FrameM[14];
  int bit = 0;
  int times = 0;

  while (1) {

    count = 0;

    while (Sensor_bit_set(&sensor) && count <= 150) {
      LCD_puts("Timer");
      count++;
    }

    if (count >= 100) {
      bit = 0;
      times++;

    }

    if (times >= 3) {
      return -1;
    }

    //Reset counter
    count = 0;

    while (Sensor_bit_clear(&sensor) && count <= 50) {
      LCD_puts("COUNTING");
      count++;
    }

    if (count >= 25) {
      FrameM[bit] = '1';
    } else {
      FrameM[bit] = '0';
    }

    //Reset counter
    count = 0;

    //check frame header
    if (bit <= 3 && FrameM[bit] != Frame_Header[bit]) {
      bit = 0;
      //LCD_puts("Bad FRAME");

    } else {
      bit++;

    }

    //check the information
    if (bit == 14) {
      int aux_bit, parity_numb = 0;

      //check parity
      for (aux_bit = 0; aux_bit <= 12; aux_bit++) {
        if (FrameM[aux_bit] == '1') {
          parity_numb++;
        }
      }

      /*parity number == 1 --> odd
       *parity number == 0 --> even
       */

      if ((parity_numb % 2 == 0 && FrameM[13] == '0') || (parity_numb % 2
          != 0 && FrameM[13] == '1')) {

      } else {
        return -5;
      }

      //Robot's Unique  ID verification
      for (aux_bit = 4; aux_bit <= 6; aux_bit++) {

        if (FrameM[aux_bit] == Bot_ID[aux_bit - 4] || FrameM[aux_bit]
            == Broad_Multi_Unique_ID[aux_bit - 4]) {

        } else {
          return -3;
        }

      }

      //Robot's Group ID verification

      for (aux_bit = 7; aux_bit <= 8; aux_bit++) {

        if (FrameM[aux_bit] == Broadcast_Global_ID[aux_bit - 7]
            || FrameM[aux_bit] == Team_GID[aux_bit - 7]) {

        } else {
          return -4;
        }

      }

      /*convert the binary message to decimal */

      char bin[5];
      int CMD_number = 0;
      strncpy(bin, &FrameM[9], 4);
      bin[4] = '\0';

      //LCD_pNumuts(FrameM);
      //Delay(5000);

      CMD_number = bin2dec(bin);
      return CMD_number;
      //return number of the conversion and command
    }

  }
  return 0;
}

/*****************************************************************************
 *   Function name : int bin2dec(char *bin)
 *
 *   Purpose : Receives an array with a binary number and converts to decimal
 *
 *   Return : returns integer converted
 *
 *****************************************************************************/
int bin2dec(char *bin) {
  int b, k, n;
  int len, sum = 0;

  len = strlen(bin) - 1;
  for (k = 0; k <= len; k++) {
    b = 1;
    n = (bin[k] - '0');
    if ((n > 1) || (n < 0)) {

      return (0);
    }
    b = b << (len - k);

    sum = sum + n * b;

  }
  return (sum);
}

/*****************************************************************************
 *   Function name : int Sensor_bit_clear(int *SensorNumber)
 *
 *   Purpose : Allows to make different bit verifications according with the desired
 *         sensor number
 *
 *   Return : returns '0' or '1'
 *
 *****************************************************************************/
int Sensor_bit_clear(int *SensorNumber) {

  switch (*SensorNumber) {
```

```

  case 1:
    if (bit_is_clear(PINF,PINF4))
      return 1;
    break;

  case 2:
    if (bit_is_clear(PINB,PINB1))
      return 1;
    break;

  case 3:
    if (bit_is_clear(PINB,PINB3))
      return 1;
    break;

  default:
    //LCD_puts("Erro");
    return 0;


  }

  return 0;
}

/*****************************************************************************
 *   Function name : int Sensor_bit_set(int *SensorNumber)
 *
 *   Purpose : Allows to make different bit verifications according with the desired
 *          sensor number
 *
 *   Return : returns '0' or '1'
 *
 *****************************************************************************/
int Sensor_bit_set(int *SensorNumber) {

  switch (*SensorNumber) {

  case 1:
    if (bit_is_set(PINF,PINF4))
      return 1;
    break;

  case 2:
    if (bit_is_set(PINB,PINB1))
      return 1;
    break;

  case 3:
    if (bit_is_set(PINB,PINB3))
      return 1;
    break;

  default:
    //LCD_puts("Erro");
    return 0;

  }

  return 0;
}

void sendburst(void) {

  int i;
  int frame[14] = { 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0 };

  LCD_puts("Sending Frame");
  for (i = 0; i <= 3; i++) { //send the frame 3 times
```

```c

    i++;
    send(frame);

  }

  LCD_puts("FRAME SENT");
  Delay(1000);
}

void send(int frame[]) { // Sends an array of bits
  int i = 0;

  for (i = 0; i <= 13; i++) {

    if (frame[i] == 1)
      one();

    else
      zero();

    lowlevel();
  }


}

void one(void) { // If there is a 1 is the array, enables the LED to
  DDRB |= 1 << DD4; // output a 39.5khz signal during 2 miliseconds which will
  // be interpreted as a 1 by the receiver
  Delay(2);
}

void zero(void) { // If there is a 0 is the array, enables the LED to

  DDRB |= 1 << DD4; // output a 39.5kHz signal during 1 milisecond which will
  // be interpreted as a 0 by the receiver
  Delay(1);
}

void lowlevel(void) {

  DDRB &= ~(1 << DD4); //LOW LEVEL no output on the LED by changing the data direction
  //register
  Delay(1);
}
```

Listing 15: IR Remote control mode Header file

```c
#ifndef IRC_MODE_H_
#define IRC_MODE_H_

int RCmode(void);

#endif /* IRC_MODE_H_ */
```

Listing 16: IR Remote control mode Source file

```c

#include "LCD_driver.h"
#include "motor_driver.h"
#include "IR_Com.h"
#include "IRC_Mode.h"
#include "main.h"

int RCmode() {

  int n = 0;
  LCD_puts("wait 3 sec");
  Delay(3000);

  while (1) {

  //scan the 3 IR sensors
  for(i=0; i>=3; i++ ){
    n = Read_Info(i);
  }
    switch (n) {

    case 1:
      Go_Forward();
      LCD_puts("Forward");
      Delay(2000);
      Stop_Bot();
      break;
    case 2:
      Go_Left();
      LCD_puts("left");
      Delay(2000);
      Stop_Bot();
      break;
    case 3:
      Go_Right();
      LCD_puts("right");
      Delay(2000);
      Stop_Bot();
      break;
    case 4:
      Go_Back();
      LCD_puts("back");
      Delay(2000);
      Stop_Bot();
      break;

    case 6:
      LCD_puts("Good Bye");
      Delay(2000);
      return 0;

    default:
      LCD_puts("Unknown CMD");
      Stop_Bot();

    }
  }

  return 0;
}
```

References

[1] *SumoBot - Mini-Sumo Robotis.* Parallax Inc., 2010.

[2] Atmel. Avr butterfly. `http://www.atmel.com/dyn/products/tools_card.asp`, May 2010.

[3] Atmel Corporation. Avr butterfly evaluation kit - user guide. Technical report, Atmel Corporation, 2005.

[4] Atmel Corporation. Avr butterfly quick start user guide. Technical report, Atmel Corporation, 2005.

[5] Atmel Corporation. 8-bit avr microcontroller with 16k bytes in-system programmable flash. Technical report, Atmel Corporation, 2006.

[6] SJ Electronics. Digital oscilloscope. `www.sjelectronics.co.uk/acatalog/GDS-800C.jpg`.

[7] Carryll Hsu. Technical data sheet infrared remote-control receiver module. Technical report, EVERLIGHT ELECTRONICS CO.,LTD., 2004.

[8] Parallax Inc. Applied robotics with the sumobot - student guide. Technical report, Parallax Inc., 2004.

[9] Parallax Inc. Continuous rotation servo. Technical report, Parallax Inc., 2004.

[10] Parallax Inc. Qti line sensor. Technical report, Parallax Inc., 2004.

[11] Parallax inc. Sumobot - mini-sumo robotics. Technical report, Parallax inc., 2005.

[12] Joe Pardue. *Butterfly Alternate Pin Uses.* `www.smileymicros.com`, 12 06.

[13] RoboGames. Unified sumo robot rules. `http://robogames.net/rules/all-sumo.php`, May 2010.

[14] Wikipedia. Eclipse (software). `http://en.wikipedia.org/wiki/Eclipse_(software)`, May 2010.

[15] Wikipedia. Embedded system. `http://en.wikipedia.org/wiki/Embedded_system`, May 2010.

[16] Winavr. Winavr. `http://winavr.sourceforge.net/`, May 2010.