LA TROBE UNIVERSITY

Embedded Bluetooth Stack

by

Dean Camera

Undergraduate Thesis

in the
Faculty of Science, Technology and Engineering
Department of Electronic Engineering

November 2011

Supervisor: Prof. John Devlin
Co-Supervisor: Robert Ross

LA TROBE UNIVERSITY

Abstract

Faculty of Science, Technology and Engineering
Department of Electronic Engineering

Bachelor of Computer Science/
Bachelor of Electronics Engineering

by Dean Camera

In modern electronic devices, both consumer and industrial, wireless technology is quickly becoming a must-have feature; wireless Bluetooth technology is now standard in almost all mobile phones, laptops and other devices. However, despite its prevalence, Bluetooth as a technology remains too expensive and/or impractical to integrate into embedded products and systems which lack large amounts of processing power and memory.

While some existing embedded Bluetooth stacks are available, these remain expensive, limited, and/or closed-source, which otherwise prevent their use in applications where Bluetooth technology would be both desired and applicable.

To combat this deficiency in the marketplace, the aim of this project is to designed and produce a free, open source, small footprint Bluetooth stack aimed to suit small embedded environments. This project will to allow for Bluetooth technology to be directly integrated into small scale products at a low cost, while remaining fully functional and extensible.

Acknowledgements

Special thanks to Robert Ross for his 3D modeling contributions to the project, without which the project's final robot design would be all the poorer. Also thanks to Ben Rampling for his assistance in reviewing the first revision of the project schematics.

Thank you to the wider AVR community for their interest and support for the project, and to Matt from *Opendous Inc.* for his contribution of the free *"Micropendous"* AVR microcontroller boards used in the final robot prototype.

Contents

Abstract **i**

Acknowledgements **ii**

List of Figures **vi**

List of Tables **vii**

1 Overview **1**
1.1 Project Background 1
1.2 Project Brief 2
1.3 Design Goals 4
1.3.1 Software Goals 4
1.3.2 Hardware Goals 5

2 Existing Implementations **6**
2.1 Classes of Existing Stacks 6
2.2 Existing Bluetooth Stacks 6
2.2.1 Operating System Stacks 7
2.2.2 Embedded Stacks 8

3 Bluetooth Stack Implementation **9**
3.1 Software Overview 10
3.2 Design Restrictions 10
3.3 Software Layers 10
3.3.1 Physical Transport 10
3.3.2 HCI Layer 10
3.3.3 L2CAP Layer 10
3.3.4 Bluetooth Services 10
3.3.4.1 SDP Service 10
3.3.4.2 HID Service 10
3.3.4.3 RFCOMM Service 10
3.4 Integration into User Applications 10
3.4.1 Events and Callbacks 10
3.4.2 Management Functions 10

4 Robot Hardware Implementation **11**

4.1 Hardware Overview 11
4.2 Hardware Modules 12
4.2.1 Microprocessor 12
4.2.2 Primary Power Supply 13
4.2.3 User Interface 14
4.2.3.1 RGB Status LED 14
4.2.3.2 LCD Display 15
4.2.3.3 Buttons 16
4.2.4 Headlights 16
4.2.5 Speaker 16
4.2.6 Motor Controller 17
4.2.7 Sensors 19
4.2.7.1 Sensor Power Supply 19
4.2.7.2 Level Converters 19
4.2.7.3 Atmel Sensor Boards 21

5 Robot Firmware Implementation 23
5.1 Build Dependencies 23
5.2 Firmware Overview 24
5.3 Firmware Modules 25
5.3.1 Main System Control and Configuration 25
5.3.1.1 System Initialization 25
5.3.1.2 Start-up Tasks 25
5.3.1.3 Main Program Loop 26
5.3.2 Hardware Drivers 27
5.3.2.1 Buttons Driver 27
5.3.2.2 External SRAM Driver 27
5.3.2.3 Headlights Driver 28
5.3.2.4 LCD Driver 28
5.3.2.5 Motor Driver 28
5.3.2.6 RGB LED Driver 29
5.3.2.7 Speaker Driver 30
5.3.3 Sensor Platform 30
5.3.3.1 Abstract Sensor Management 30
5.3.3.2 Individual Sensor Drivers 32
5.3.4 Third Party Modules 33
5.3.4.1 LUFA 33
5.3.4.2 FatFS 33
5.3.5 USB Management 34
5.3.5.1 Bluetooth Adapters 35
5.3.5.2 HID Devices 35
5.3.5.3 Mass Storage Devices 36
5.3.6 Bluetooth Management Layer 37

6 Project Results 40

7 Project Discussion 42

7.1 Issues Faced 42
7.1.1 Micropendous Pinout PCB Error 42
7.1.2 Motor Inrush Current 42
7.1.3 PCA9306 Physical Package 42
7.1.4 L298D Unavailabilty 42
7.1.5 Incorrect Transistor Pinout 42
7.1.6 Slow Drive Transistors 42
7.1.7 Unreliable Packet Buffering 42
7.1.8 PS3 Controller Compatibility 42
7.1.9 Wii Controller Compatibility 42
7.2 Project Significance 42

8 Conclusion **43**
8.1 Project Success 43
8.2 Future Work 44
8.2.1 Reliable Packet Buffering and Retransmissions 44
8.2.2 Packet Reception Fragmentation Support 44
8.2.3 Expansion of Bluetooth Services 45
8.2.4 Additional Physical Transports 45
8.3 Final Words 45

Abbreviations **47**

A Schematics **48**
A.1 Robot Schematics 48

B Source Code **54**
B.1 Obtaining the Source Code 54

C Stack API Overview **55**

D Robot User Guide **56**
D.1 Power Requirements 57
D.2 Supported USB Devices 57
D.2.1 HID Class Devices 57
D.2.2 Mass Storage Class Devices 58
D.2.3 Bluetooth Adapter Devices 58
D.3 Supported Bluetooth Services 58
D.3.1 HID Service 58
D.3.2 RFCOMM Service 58

Bibliography **59**

List of Figures

1.1 Diagram of a typical Bluetooth Stack 3
1.2 Diagram of the proposed robot 3

4.1 Hardware Block Diagram 12
4.2 Micropendous-A Board 13
4.3 Main Power Supply Schematic 14
4.4 Close-up of the RGB LED's Removable Wire Link 15
4.5 LCD Display Schematic 16
4.6 Speaker Schematic and Mounting Photo 17
4.7 Motor H-Bridge Schematic 18
4.8 Transistor Inverter Schematic 18
4.9 Motor Flyback Diodes 19
4.10 Bidirectional Level Translator Schematic 20
4.11 Bidirectional Level Translator Routing 20
4.12 I^2C Level Translator Schematic 21
4.13 Atmel Sensor Boards 22

5.1 Firmware Block Diagram 24
5.2 LCD Sensor Status Information 26
5.3 Sensor Platform Entry Structure Diagram 32

6.1 Photo of Completed Robot 40
6.2 Tested Working Controllers 41
6.3 Sensor Logging Host Application 41

D.1 Annotated diagram of the robot 56

List of Tables

2.1 Existing Operating System Bluetooth Stacks 7
2.2 Existing Embedded Bluetooth Stacks 8

4.1 RGB LED Colour Table 14

5.1 Mode Specific Button Roles 26
5.2 Sensor Configuration 32

D.1 USB HID Robot Button Mappings 57

Chapter 1

Overview

In almost all modern portable consumer devices, Bluetooth plays a large role; it is available in the vast majority of mobile phones and their associated accessories, in cars, in laptops and, most recently, in mobile tablet PCs. Bluetooth as a technology gives system designers a low power wireless communications standard from the baseband up to the higher level abstract services, allowing implementing devices to communicate with one another in a manufacturer-independant manner. This freeing of consumers from the proprietary short range wireless solutions (such as *ZigBee*) has helped make Bluetooth the wireless communication system of choice for many applications.

1.1 Project Background

Despite this ubiquity, Bluetooth remains firmly in the realm of systems containing large amounts of RAM and FLASH memories, processing power and—in many cases—full operating system stacks. For small-scale embedded devices with tiny 8-bit processors, clock speeds in the tens of MHz (or even less) and RAM measured in kilobytes, Bluetooth remains impractical; either due to its expense or the lack of suitable software.

However, existing solutions do exist. System designers can integrate off-the-shelf Bluetooth solutions in their products; small hardware modules containing the Bluetooth baseband and a fixed-function microprocessor. This microprocessor then handles the complex onion-like layers of the various Bluetooth stack components, off loading the computational load and implementation complexity from the main system processor. These modules are generally ridgid in their functionality however, making them unsuitable in applications where a specific or even custom Bluetooth service is required. In addition, such modules are generally significantly more expensive than the product's main processor, negating its cost/benefit ratio where a more powerful system processor could be substituted to manage the entire application including the Bluetooth component.

These turn-key modules are made all the less attractive when one considers the cost of a raw Bluetooth baseband IC module. Without an integrated processor to manage the software stack, these are generally available from multiple vendors at costs measured in the sub-US$5 range. This indicates that the main cost of the complete modular solutions lies not in the physical hardware, but the IP of the Bluetooth software stack. If such a stack could be made widely available for use in embedded systems, this fixed-cost vendor lock-in could be avoided and cheaper Bluetooth enabled systems developed for both hobbyist and commercial use.

1.2 Project Brief

To help fill the identified gap in the marketplace for a cheap, open source Bluetooth stack aimed at the low to mid-range embedded market, it is proposed that a new stack be designed from scratch specifically for this market segment. This stack would offer a base amount of functionality suitable for integration into new or existing embedded systems, to extend the system functionality to include wireless Bluetooth communications.

At a minimum, a functional Bluetooth stack needs to have at least four components:

1. A **Physical Data Transport layer** to and from a connected Bluetooth physical baseband transceiver IC
2. An implementation of the Bluetooth specification's **HCI layer** for the establishment and management of physical connections to and from remote devices
3. An implementation of the Bluetooth specification's **L2CAP layer** for the establishment and management of logical channels within an established connection
4. One of more **Bluetooth Services** on top of the L2CAP layer to implement functionality such as the Service Discovery Protocol (SDP)

These components, when put together, form the basis of a minimal Bluetooth stack *(see Figure 1.1)*. Additional services may or may not be added on top of the stack in parallel with the mandatory SDP protocol to expose local device functionality and interact with remote devices.

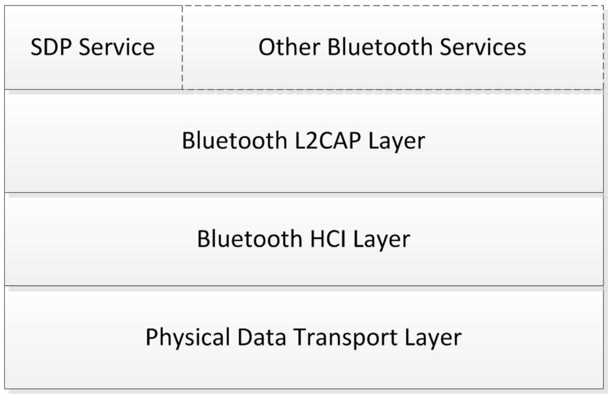


Figure 1.1: Diagram of the basic components which form a typical Bluetooth Stack

As the usefulness of an abstract piece of software is inherently low without a suitable demonstration platform, a second component of the project will be to design, develop and prototype a functional and practical device which uses the created stack. It is proposed that this hardware component be in the form of a small *"ExplorerBot"* robot (see Figure 1.2), able to stream local sensor data wirelessly to a remote PC for real-time graphing purposes, and to allow for remote wireless control over a Bluetooth link to a consumer Bluetooth control device, such as a current generation Game Console controller (*Wii* or *Playstation 3*). If possible, these two functions should be combined to allow for multiple simultaneous connections, allowing for remote control at the same time as the sensor data is logged remotely.

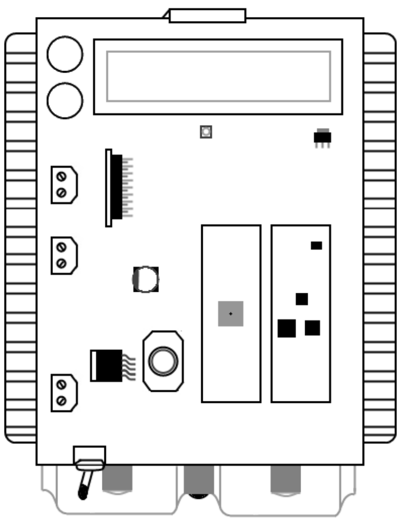


Figure 1.2: Diagram of the proposed robot platform.

With the combination of the software *(the stack)* and hardware *(the robot)* components of this project, the final design should offer a complete system and test platform which can be reproduced wholesale, modified to suit a particular application or used as a reference implementation for other projects.

1.3 Design Goals

The design goals of the project were split into two distinct parts, consisting of the software Bluetooth Stack implementation, and the physical robot hardware design. This separation of the testing platform hardware goals from those of the logical Bluetooth stack allowed for the clearest representation of the project as a whole.

1.3.1 Software Goals

For such a stack to be useful in an embedded environment, it must be able to conform to the restrictions such an environment imposes. Specifically, the completed stack must minimize its compiled and working set footprints, reduce or eliminate the need for dynamic memory allocation, and minimize its hardware dependencies to suit as wide a range of processors of differing capabilities as possible.

The design goals of the complete software stack were therefore set to:

- Use as little RAM as possible
- Compile to as small a binary as is practical
- Offer a framework upon which services can be added to suit a particular application
- Provide asynchronous events to which the user application can respond to
- Allow for a variable number of simultaneous connections and logical channels to/from remote devices
- Have no requirement for dynamic memory allocation on the heap
- Allow for integration into an optional RTOS, but support stand-alone operation
- Be fully decoupled from the physical transport to the Bluetooth Adapter
- Be endian-correct regardless of native processor endianness
- Maintain a level of compatibility with Bluetooth Version 2.1 specification as maintained by the Bluetooth *Special Interest Group* (SIG)

While at the point of the project's design and development a newer Bluetooth 3.0 version standard was available, a decision was made to implement the slightly older—and far more popular—version 2.1 of the Bluetooth specification. This decision was made due to the abundance of cheap Bluetooth 2.1 compatible hardware already of the market (and, conversely, the lack of cheap and widely available Bluetooth 3.0 hardware). In addition, the aim of the project was set to produce a *compatible* rather than fully *compliant* stack, due to the significantly increased development complexity of the latter for little additional benefit.

1.3.2 Hardware Goals

To make a useful, visual and functional testing platform, a decision was made to produce a small, battery powered robot. This robot would serve as both a testing platform for the completed stack to verify its correct operation in a real-world environment during development, and to function as a reference application of the completed stack.

In order to fully demonstrate the capabilities of the Bluetooth stack, the robot would have to include both locally initiated Bluetooth connections, as well as accept remotely initiated connections. In addition to this requirement, data would have to be both received and sent to and from a remote device to demonstrate full duplex communications.

A set of design goals was thus created for the robot:

- Allow the user to initiate a connection to a remote device via the robot
- Accept incoming connections from remote Bluetooth devices, including some level of authentication
- Consume data received from remote Bluetooth device(s) via one or more Bluetooth services
- Produce data to be transmitted to remote Bluetooth device(s) via one or more Bluetooth services
- Visually indicate status and debug messages via a display mechanism, for debugging
- Allow for the robot to be remotely driven via a set of PWM controlled DC motors

Chapter 2

Existing Implementations

Before work was started on the proposed software Bluetooth stack implementation, the existing field of Bluetooth stacks (both commercial and non-commercial) were evaluated to determine what capabilities are being offered.

2.1 Classes of Existing Stacks

During the course of the project background research into existing Bluetooth stacks, two distinct classes of stack were observed, each with distinct assumptions and capabilities:

- **Operating System based Stacks**, which assumed that they would be run on top of a complex full-featured OS, containing a kernel- and user-space, virtualized memory, synchronisation primitives, etc.
- **Embedded Stacks**, which assumed no OS was present, but nevertheless made assumptions as to the environment's capabilities for dynamic memory allocation

These two classes of stacks show two possible approaches to an implementation; one, the designer may write a stack around an existing Operating System API, or two, the designer can assume a "freestanding" or "bare metal" environment, with either no, or only a minimal, RTOS being present.

2.2 Existing Bluetooth Stacks

Below the discovered existing Bluetooth stacks are listed in parametric form for each class of stack for ease of reference.

2.2.1 Operating System Stacks

Stack Name	**Operating System**	**Commercial**
FreeBSD Stack	FreeBSD	No
Affix Stack	Linux	No
BlueZ Stack	Linux	No
Apple Stack	MacOS	Yes
BlueFritz! Stack	Windows	Yes
CSR Harmony Stack	Windows	Yes
FreeBT Stack	Windows	No
Microsoft Stack	Windows	Yes
Toshiba Stack	Windows	Yes
Widcomm Stack	Windows	Yes

Table 2.1: Parametric table of existing OS based Bluetooth stacks

As expected, the vast majority of existing OS based Bluetooth stacks are targeted towards the Microsoft Windows operating system, due to its large market share. In almost all cases, the existing stacks were found to support a rich number of Bluetooth services, in both device and server roles. While the majority of the Bluetooth stacks on the market are commercialized (i.e. require payment or hardware purchase for a license to use them) there are still several free and open source stacks available for the various Linux and BSD kernels.

2.2.2 Embedded Stacks

Stack Name	**Commercial**
BlueCode+ Stack	Yes
BlueLet Stack	Yes
BlueMagic Stack	Yes
Bluetopia Stack	Yes
BTStack Stack	No
ClarinoxBlue Stack	Yes
CSR Synergy Stack	Yes
EtherMind Stack	Yes
Jungo BTware Stack	Yes
lwBT Stack	No
Mecel Betula Stack	Yes
Symbian OS Stack	Yes

Table 2.2: Parametric table of existing embedded Bluetooth stacks

Contrasting with the OS based stacks discussed previously, almost all Bluetooth stacks aimed at the embedded market today were found to be exclusively commercialized, requiring large payments and license agreements before they could be integrated into existing designs. Of note in this area are the *BTStack* and *lwBT* embedded Bluetooth stacks, which were both found to be both free and open source, but suffered from the constraints of RTOS dependencies in the case of the former, and incompleteness in the case of the latter.

Chapter 3

Bluetooth Stack Implementation

3.1 Software Overview

3.2 Design Restrictions

3.3 Software Layers

3.3.1 Physical Transport

3.3.2 HCI Layer

3.3.3 L2CAP Layer

3.3.4 Bluetooth Services

3.3.4.1 SDP Service

3.3.4.2 HID Service

3.3.4.3 RFCOMM Service

3.4 Integration into User Applications

3.4.1 Events and Callbacks

3.4.2 Management Functions

Chapter 4

Robot Hardware Implementation

To help demonstrate the usefulness of the stack in a practical manner, a robot was designed and constructed. This robot, named the *ExplorerBot*, was then used to give a practical reference implementation of a full project utilizing the custom embedded Bluetooth stack in a real-world environment.

4.1 Hardware Overview

The completed robot design for the project contains many useful capabilities for both mobility and exploration. Built on top of a pre-fabricated (including raw DC motors and gearing) "Tank" style hobby robot base, the *ExplorerBot* robot implements the following features:

- Primary switch-mode based 5V power supply
- Secondary LDO based 3.3V power supply for attached sensors
- 2x16 Alphanumeric LCD Screen for feedback to the user
- Two momentary pushbuttons for user control
- One RGB status LED for basic status feedback
- Dual PWM motor control system, with variable speed and direction of DC motors
- Level converted I^2C bus for the attached sensor(s)
- Support for the Atmel *Inertial One* and *Pressure One* sensor boards
- High intensity LED based headlights for frontal illumination

- Piezo speaker for audio feedback and "horn" like functionality
- Atmel *AT90USB1287* 8-Bit Microcontroller
- External 128KB SRAM for temporary storage of packets to and from the Bluetooth adapter

The complete robot design was created in the *Altium Designer* software, including both the schematic design and board routing. Surface mount components were chosen where possible to reduce the board space required, and two board layers used as this proved to offer the lowest cost/time ratio. The final board design measured 10cm x 15cm, however much of this board space is relatively unused; with optimization, this board space could be reduced considerably.

To get the best results in the construction of the robot, the boards were manufactured commercially. This process ensured the manufactured board's quality while also provided solder mask and silk-screen to reduce the potential for error in the robot's construction.

4.2 Hardware Modules

In the section, the various hardware components of the constructed robot are detailed at the block level. Figure 4.1 below illustrates how the various hardware blocks that comprise the robot connect together to make the final design.

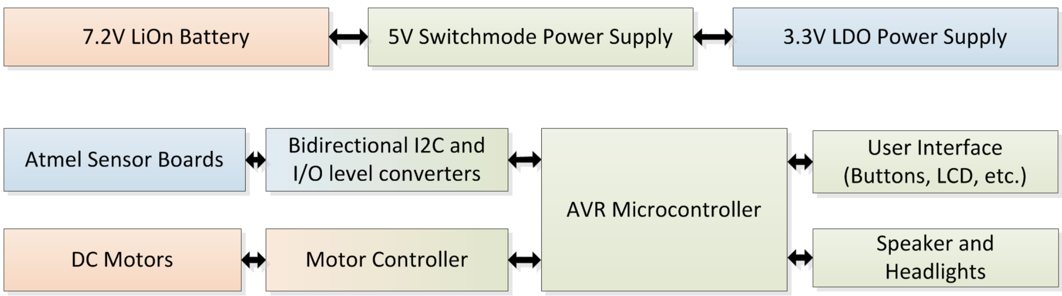


Figure 4.1: Robot Hardware Block Diagram

4.2.1 Microprocessor

Due to the author's familiarity with the Atmel line of *AVR* branded microcontrollers, one of the available models in this line-up was chosen to serve in the robot as the main processor, the AT90USB1287. This 8-bit microcontroller contains 128KB of non-volatile FLASH memory for program storage, 4KB of non-volatile EEPROM for user application parameter storage and 8KB of internal SRAM for scratch memory. A 16MHz clock (provided by an external crystal) was selected for the design as this offered the fastest possible speed the chip was capable of, while still allowing the hardware USB host controller inside the chip to function normally. As a trade-off, this higher clock speed put a constraint on the main logic level voltage; at 16MHz, the AVR microcontroller required 5V to be within the datasheet's specifications.

As the AT90USB1287 and associated USB components are difficult to source in single quantities at reasonable prices, the use of a commercial breakout module containing this chip was selected instead: the *Micropendous-A* board (see Figure 4.2). This board contains the surface mount AVR microcontroller and associated USB components, along with an external 128KB SRAM chip attached to the AVR's external memory bus interface. As the Bluetooth stack required a large temporary buffer for incoming fragments, the selection of this board proved ideal for the intended purpose.

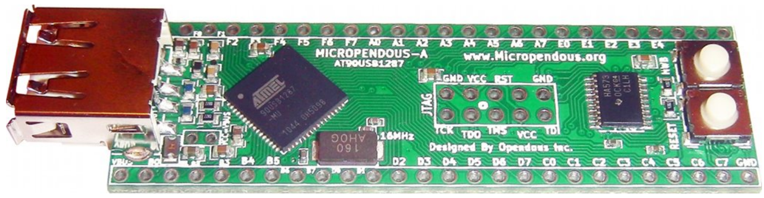


FIGURE 4.2: The Micropendous-A Board (Image courtesy *Opendous Inc.*)

4.2.2 Primary Power Supply

As the vast majority of the robot's hardware operated at a fixed 5V level, a power supply was required to reduce the Lithium Ion battery's raw 7.2V (nominal) voltage down to the 5V level needed to power the various components. Due to the use of battery power in the project, reducing power consumption where possible was a large concern; thus, a switch-mode design was chosen for maximum voltage conversion efficiency. A conventional linear regulator was considered for the design, but rejected due to the prohibitively large amount of power this would waste (approximately .45W, assuming an average 200mA operating current).

The regulator selected for the project was the M2595S-5.0, a fixed-function switch-mode regulator capable of outputting a fixed 5V rail at loads of up to 3A. While the robot design would not consume even a fifth of this power, the overhead in the specifications ensured that the power supply would remain robust and the output within the tolerances of the system components regardless of the load demanded. The exact schematic used in the final robot power supply design (see Figure 4.3) was taken from the regulator component's datasheet to ensure correct operation.

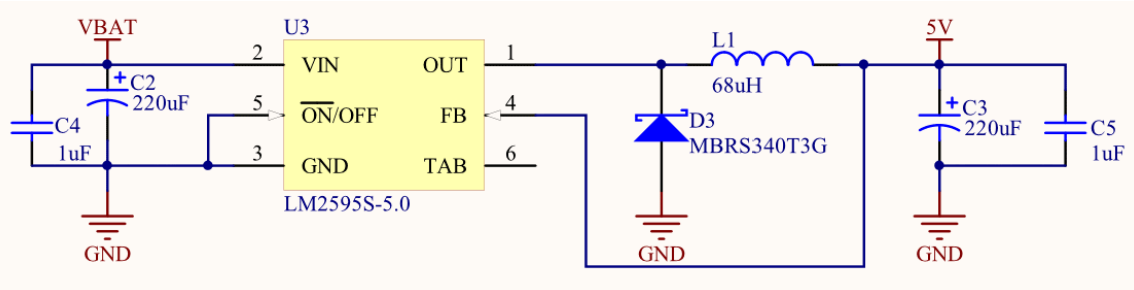


FIGURE 4.3: Schematic of the robot's main 5V switch-mode power supply.

4.2.3 User Interface

For interaction with the user, the robot contains several components, detailed below.

4.2.3.1 RGB Status LED

For primary status indication, a surface mount RGB LED indicates the current status of the robot. Due to a lack of free PWM channels on the AVR microcontroller, the three LED sub-components are wired directly to standard GPIO ports. While this design prevented PWM fading of the individual LED subcomponents to produce a many-bit custom colour from the LED, a three bit colour space is possible giving a total of 8 possible colour states (see Table 4.1). For the purposes of the created robot, this is in practice more than enough for basic status indication.

R	G	B	Output Colour
0	0	0	Off
0	0	1	Blue
0	1	1	Cyan
0	1	0	Green
1	1	0	Magenta
1	0	0	Red
1	0	1	Yellow
1	1	1	White

TABLE 4.1: Table of the possible output colours of the RGB LED, with binary inputs.

To achieve a somewhat uniform brightness, the three LEDs were adjusted with current limiting resistors to consume an equal amount of current (approx. 5mA) despite differing forward voltages. As the RGB status LED shares the same I/O pins as the microcontroller's JTAG port for programming and debugging, the RGB LED's common anode was connected via a removable wire link (see Figure 4.4) to ensure that it could be taken out-of-circuit if it proved to interfere with the external hardware debugger during development.

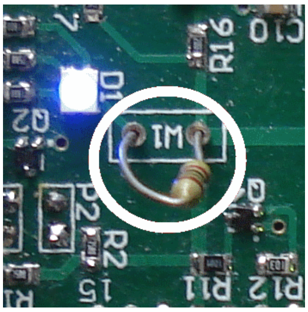


FIGURE 4.4: Close-up of the removable wire link used to disable the RGB status LED to prevent conflicts with the JTAG lines during programming/debugging.

4.2.3.2 LCD Display

For situations where more information needs to be communicated to the user than is possible via the RGB status LED, a 16x2 Alphanumeric LCD display—compatible with the well known Hitachi HD44780 chipset—was added to the design. Due to the limited number of GPIO pins available on the microcontroller, the LCD was wired in 4-bit mode, with the lower 4 data pins on the LCD being wired directly to ground (see Figure 4.5). While this doubled the time required to send a byte to the LCD (as bytes then need to be split into a pair of 4-bit nibbles) the high speed of the processor meant that in practice this had little or no effect to the overall speed of the system.

The LCD backlight was wired through a driver transistor to a spare PWM channel on the AVR microcontroller, allowing for 8-bit PWM brightness control to reduce power consumption of the backlight when not in use. As the LCD display's LED backlight had a nominal forward voltage of around 3V, a 10Ω resistor was used to limit the maximum drive current to around 200mA.

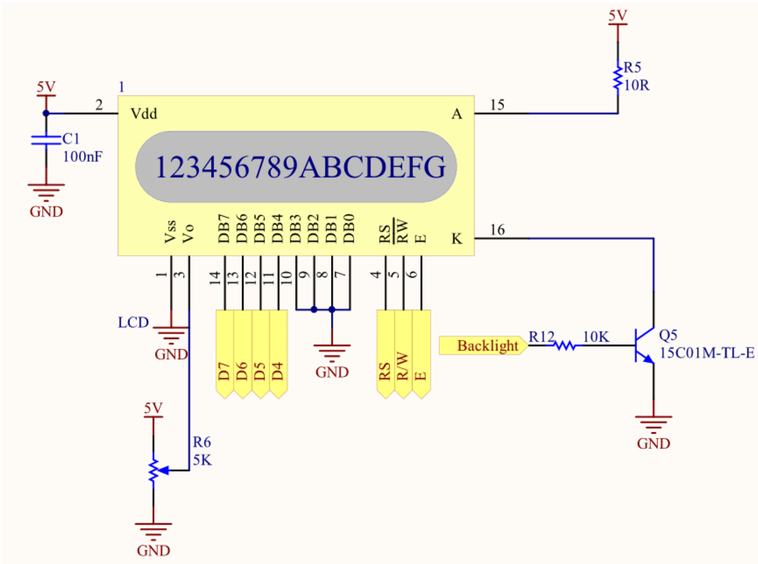


FIGURE 4.5: Schematic of the LCD connections, showing the 4-bit LCD data bus mode and backlight driver transistor.

4.2.3.3 Buttons

A pair of standard PCB round buttons were added to the design, for user input. These buttons were wired directly to the microcontroller's GPIO pins; internal pull-up resistors in the microcontroller takes care of maintaining a defined logic level on the pins when the buttons are released, while software handles the debouncing of the button signals.

4.2.4 Headlights

To provide illumination of the area immediately ahead of the robot, a pair of high intensity wide viewing angle white LEDs were added to the schematic, connected to a single common driver transistor and driven by a GPIO pin of the microcontroller. To ensure maximum illumination, the LEDs were driven at just under their full 20mA rating when turned on. These "headlights" were then mounted on the front of the robot chassis.

4.2.5 Speaker

A small PCB Piezo speaker was added to the robot, in order to provide both audio feedback for important events (such as Bluetooth connections and disconnections) as well as to act as a miniature horn to attract the attention of any organic obstacles to encourage then to move away from the robot's line of motion. Rather than mounting the speaker directly onto the PCB, it was determined that a better location was in between the two frontal headlight LEDs, with the speaker then connected back to the PCB via flyleads. This arrangement made the directional speaker point in the orientation most suited to a car horn, i.e. towards the front of the robot (see Figure 4.6).

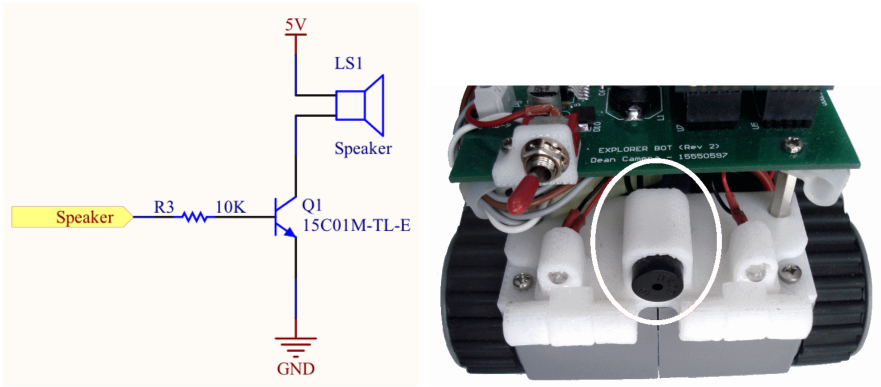


FIGURE 4.6: Schematic of the robot's piezo speaker (*left*) and photo showing the mounting in the robot prototype (*right*).

To drive the speaker, a standard NPN transistor was employed to provide sufficient current, driven from an 8-bit PWM timer output GPIO pin of the AVR microcontroller.

4.2.6 Motor Controller

While the plastic hobbyist "tank" style robot base selected for the prototype robot contained a pair of 6V DC motors stock from the factory, it did not contain a motor control system; implementation of a suitable motor control circuit was thus required as part of the project. To prevent motor noise from being injected back into the main 5V power supply used by the sensitive microcontroller, the motors were instead powered directly from the raw battery voltage. By directly powering the motors from the system battery, the main 5V logic power bus could remain relatively undisturbed by the potentially large current spikes caused by the switching on and off of the motors under load. This design had the additional benefit of a reduced the total power draw on the main power supply, reducing wasted power due to the supply's non-perfect efficiency and prolonging the operating time of the robot on a fresh battery.

As the raw battery voltage (7.2V nominal) was higher than the motor's 6V maximum, a PWM circuit was thus designed so that the average power delivered to each motor would prevent the motor from burning out during use. While not used in the final robot firmware, the use of variable duty cycle PWM drive signals to the motor would allow for additional speed control of the robot's motors without a corresponding loss of torque.

The motor controller design used in the project centered around the well-known conventional L298N Dual Channel Full H-Bridge Driver IC, notable for its high current drive capabilities and low-voltage logic level drive input support (see Figure 4.7). Originally the L298D variant was selected due to its convenient internal flyback diodes, however at the time of parts ordering a cheap source for the part could not be found. Unfortunately, the original PCB design did not allow for the possibility of adding external flyback diodes, resulting in the need for a second revised PCB manufacturing run to add in space for the missing components (see Figure 4.9). Due to space constraints, the L298N's current sensing capabilities for motor stall detection were not used.

To correctly drive the L298N's PWM inputs, it was necessary to construct a logically inverted version of the PWM signal from the main microprocessor, which would be fed into the L298's channel compliment pin in order to correctly switch on and off the correct portions of the internal H-Bridge circuit. As the chosen microcontroller did not contain enough free pins for this function, an pair of external inverters were constructed out of discrete parts (see Figure 4.8). This basic logic inverter was used to invert each of the two motor PWM signals for the H-Bridge IC.

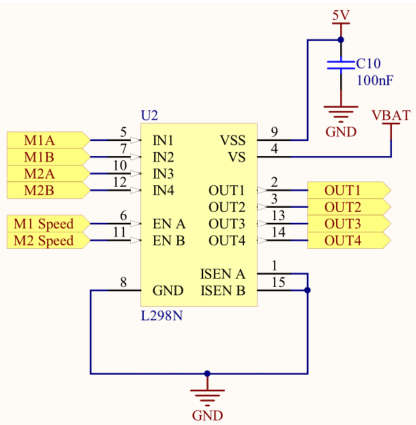


Figure 4.7: Schematic of the L298N H-Bridge circuit used.

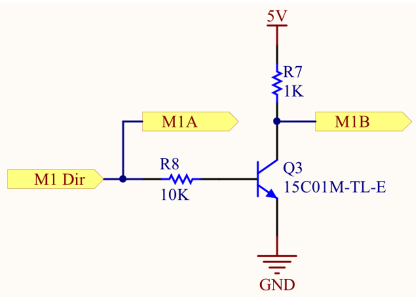


Figure 4.8: Schematic of one of the transistor inverters used to generate the compliment of the PWM signal used by the motor H-Bridge.

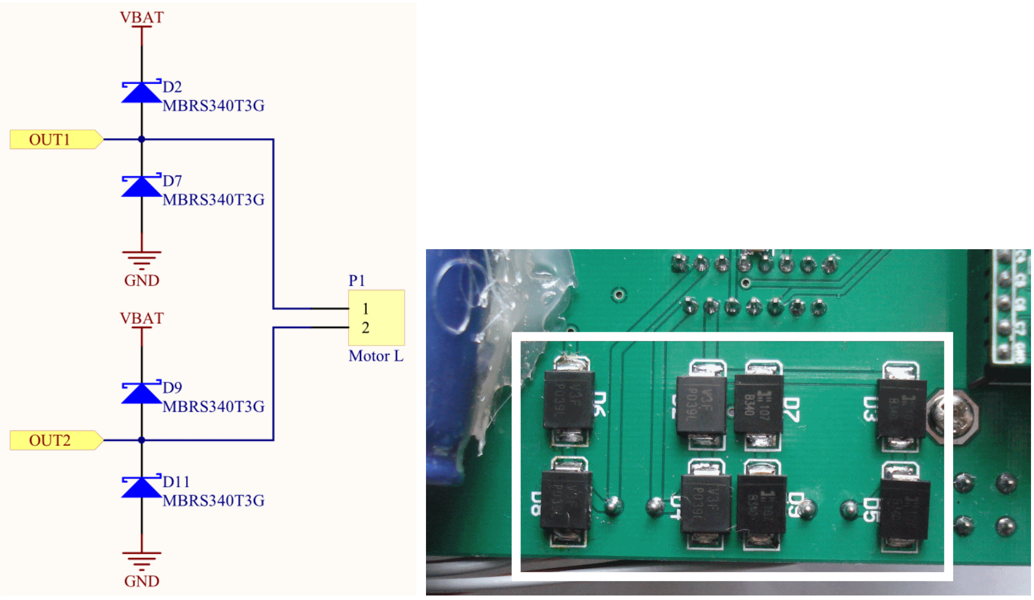


FIGURE 4.9: Partial schematic *(left)* and photo *(right)* of the external flyback diodes added in the second PCB revision to the robot's motors.

4.2.7 Sensors

To provide a measure of feedback from the robot, a number of sensors were added to the design. These sensors, when attached, would allow for the robot's environment to be logged and (potentially) reacted to.

4.2.7.1 Sensor Power Supply

While the main system logic and user interface components run from the main switch-mode 5V power supply, the sensor boards were required to run at a fixed 3.3V level, without the possibility of conversion to suit the higher rail voltage.

For this reason, and to reduce the amount of noise on the sensor power supply for maximum precision, a decision was made to add a secondary power supply, running from the 5V rail, to step down the voltage to the 3.3V required by the sensor boards. For best results, an ADP3308 Low Dropout (LDO) style regular was used as this provided both low output rail noise and minimal wasted power.

4.2.7.2 Level Converters

Due to the differing bus voltages between the sensor boards (3.3V) and the main processor (5V), level conversion of the I^2C bus and sensor interrupt/control lines was required.

While only a unidirectional buffer was strictly needed for each of the sensor interrupt/-control lines, it was decided to use a bidirectional converter to ease the board routing.

Initially, only an ADG3308 8-channel Bidirectional Level Converter IC was used, for both the sensor interrupt/control lines, as well as the I^2C bus. However, after further analysis it was discovered that the level translator would not meet the timing requirements of the I^2C bus, necessitating the addition of a secondary dedicated Texas Instruments PCA9306 fixed function I^2C bus level converter IC in the second revision of the board. As a bonus, the use of the later chip allowed the I^2C bus to be driven at the “Fast” I^2C speed of 200KHz for minimal latency and maximum throughput.

Unusually, the ADG3308 level converter IC required that its enable pin (located on the low voltage side of the translator) be connected to the higher logic level for the chip to become active (see Figure 4.10). This odd placement of the enable pin resulted in a non-optimal breaking of the ground plane underneath the chip to accommodate the required route, as the space between the chip package pins on the top layer was used to carry the 3.3V power bus to the sensor boards (see Figure 4.11).

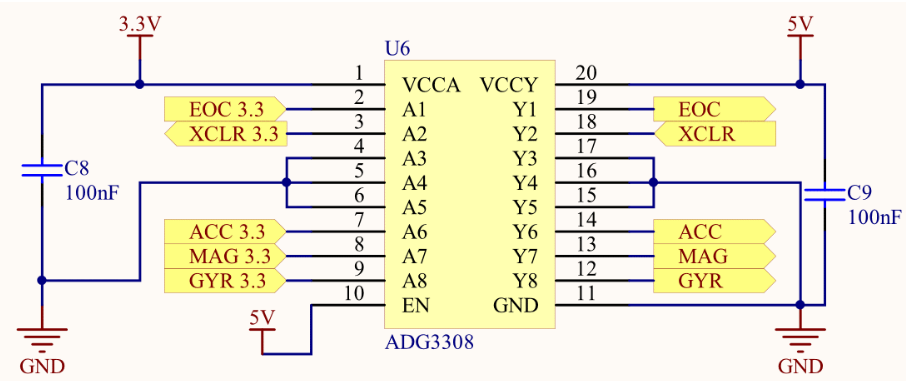


FIGURE 4.10: Schematic of the ADG3308, showing the unusual placement of the VCC-Y level active high enable pin.

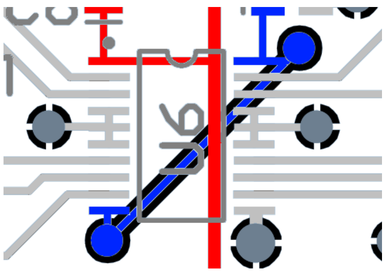


FIGURE 4.11: Routing of the ADG3308, showing the 3.3V bus (red) and 5V enable (blue) routes.

The board routing complexity was reduced slightly by swapping the functions of the PCA9306 bus level translator’s SDA and SCL pins (see Figure 4.12) on both sides of the IC; this modification (allowable as indicated in the device's datasheet) prevented the need to introduce additional board vias and longer trace routes.

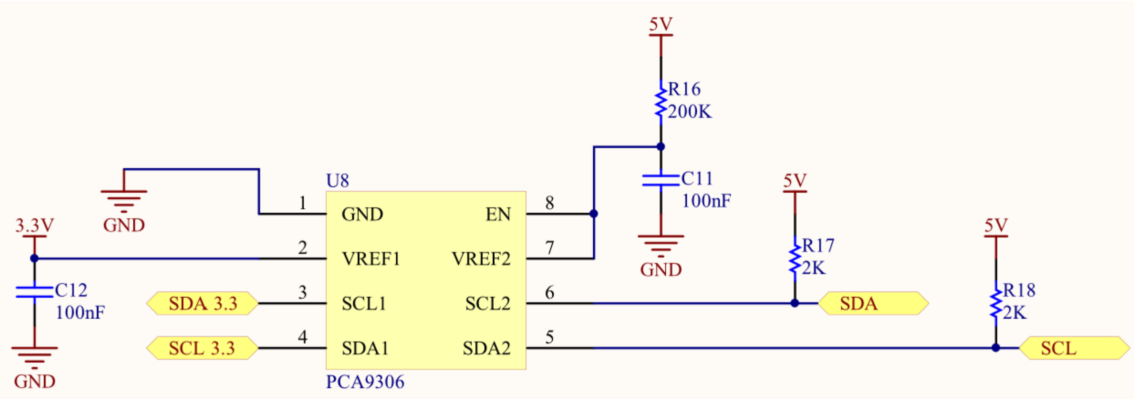


FIGURE 4.12: Schematic of the PCA9306, showing the swapped SDA and SCL pin functions.

4.2.7.3 Atmel Sensor Boards

By designing the robot around a pair of commercially available Atmel sensor boards for environmental feedback, the design of the robot was considerably simplified and the total unit cost lowered. The *Atmel Pressure One* board contains a Bosch BMP085 Pressure Sensor IC for air pressure sensing, while the Atmel *Inertial One* contains a 3-Axis ITG3200 Gyroscope, 3-Axis BMA150 Accelerometer and 3-Axis AK8975 Compass IC (see Figure 4.13). As several of the sensors also contain a digital temperature sensor in addition to the primary sensor (for calibration and stability feedback) this functionality was also used by the robot to measure the environmental temperature in real time.

Each sensor IC is driven by the main microcontroller of the robot over the level converted I^2C bus and one or more digital interrupt/control lines. The Atmel sensor board modules all use an identical form factor, with one standard .1" 2x5 female header located at one end of the board reserved for the mounted sensor's digital I/O pins, and another located at the opposite end of the board reserved for analogue sensor pinouts. As none of the sensor boards used contained analogue sensor outputs, the second female header consisted only of non-connected pins, and a matching male header was placed on the board for mechanical stability only.

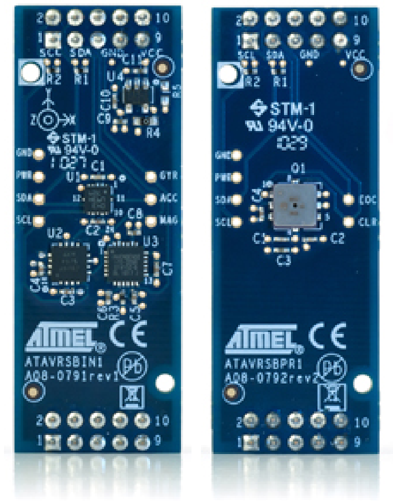


FIGURE 4.13: The Atmel *Inertial One* (left) and *Pressure One* (right) Sensor Boards (Image courtesy *Atmel Corporation*)

Chapter 5

Robot Firmware Implementation

With the creation of the software embedded Bluetooth stack and the *ExplorerBot* test robot platform hardware, it was necessary to integrate these two components into a functional prototype. By using the Bluetooth stack in a real-world, practical application while it was being developed, the quality, effectiveness and completeness of the stack could be evaluated.

5.1 Build Dependencies

To match the Bluetooth stack, each module was written in the C language, and targeted at the free open source AVR-GCC compiler and avr-libc library. A standard *makefile* included with the firmware allows for command line control over the building of the project files into a set of binaries which can then be programmed into the target microcontroller for use via the command `make all`. The following tools are required to build the firmware under Windows:

- The **WinAVR 20100101** release download, or Windows binaries of the **GNU Shell Utilities**
- The latest **AVR Toolchain** release from Atmel (Included with Atmel's free *AVRStudio 5* software)

Under Debian Linux environments, the following packages are required:

- **gcc-avr**
- **binutils-avr**

- **avr-libc**
- **avrdude**

Which can be installed via the command prompt using the command `sudo apt-get install gcc-avr binutils-avr avr-libc avrdude`.

5.2 Firmware Overview

The completed firmware of the *ExplorerBot* prototype was developed in a modular manner, to match the corresponding hardware components. This top-down methodology ensured that each portion of the firmware could be mocked up, tested and integrated as needed. Additionally, separating out the firmware components into logical modules gave the final firmware a level of flexibility which should allow for easy modification to suit any hardware changes made to those of the prototype. The completed set of modules (see Figure 5.1) served as the complete firmware for the robot.

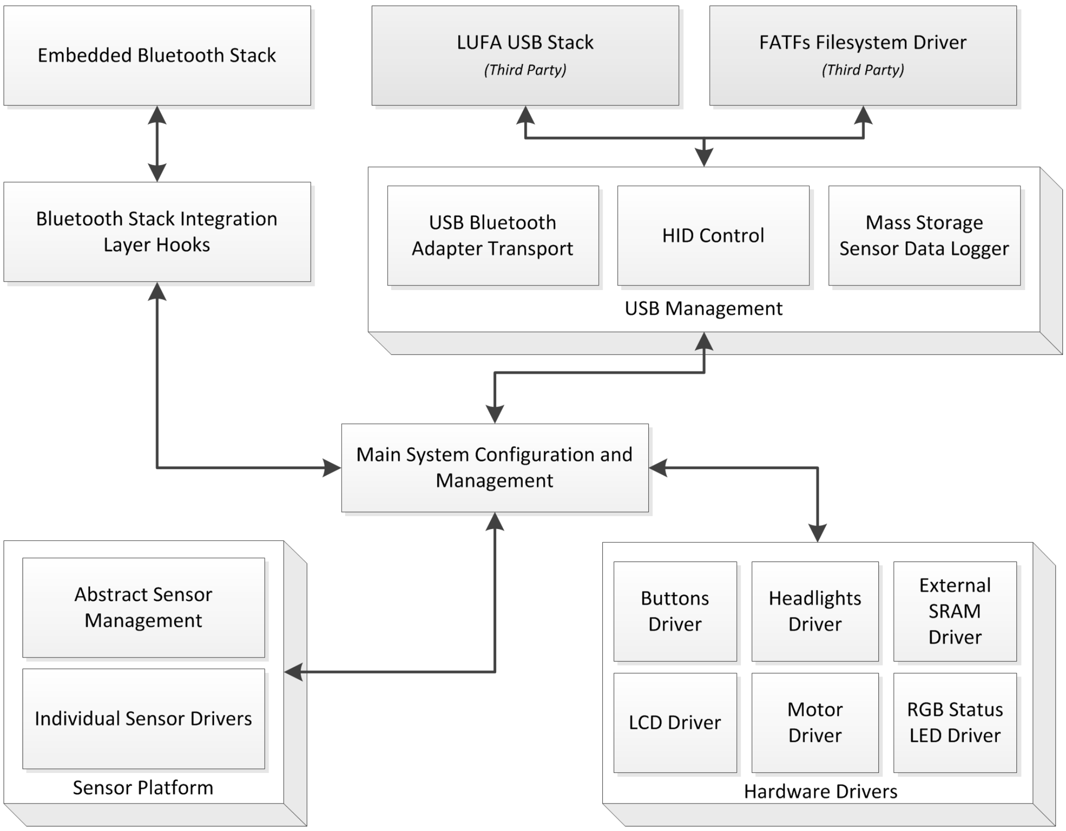


FIGURE 5.1: Robot Firmware Block Diagram

5.3 Firmware Modules

In this section, each of the robot firmware's main software modules are listed and described in additional detail so that the overall design and implementation of the firmware can be further understood.

5.3.1 Main System Control and Configuration

The main entry point and system loop of the firmware was contained into a single top level module. This module was then made responsible for the initial system hardware configuration, as well as the management of the main loop to dispatch the service task functions in each sub-module.

5.3.1.1 System Initialization

A series of initialization steps are followed during the hardware configuration step; first, the system watchdog (enabled if the chip was last reset through the expiry of the watchdog peripheral's timer) is disabled, the system CPU clock prescaler disabled to ensure the full 16MHz CPU clock speed is used, unused peripherals are powered down and the JTAG debug interface turned off so that the GPIO pins could be used for the RGB status LED. This latter procedure removes the ability to debug the firmware with an external JTAG debugger, however during development it was commented out.

Next, the setup routine calls each hardware driver module's `Init()` function, which serves to initialize each hardware module and configure the appropriate hardware ready for use. Finally, one of the robot's remaining 16-bit hardware timers is then configured to run at a 10ms period to serve as the master system tick for timeout management and time based events.

5.3.1.2 Start-up Tasks

Once all the system hardware is initialized, the main program flow then executes the start-up tasks; an informational message is displayed to the LCD while the RGB status LED sequences through all possible combinations, and the sensor platform is initialized to determine which sensors are currently connected. The state of each sensor is then displayed briefly onto the LCD using custom LCD character definitions before the main loop starts (see Figure 5.2).

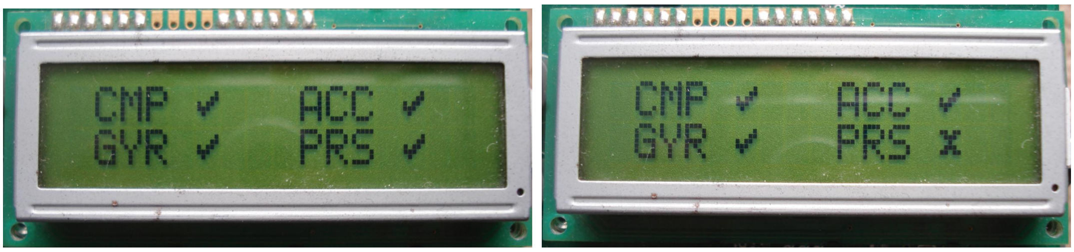


FIGURE 5.2: Photos of the LCD display showing successful (*left*) and failed (*right*) initialization.

5.3.1.3 Main Program Loop

As is the case with virtually all embedded systems, the main program execution was contained in an infinite loop; each iteration of the loop would dispatch to the various sub-components of the system to manage and react to various stimuli. In the case of the robot firmware, the main loop contained three main functions; one, it checked for presses of either of the two physical buttons on the robot, two, it would check for expiry of the system tick timer to dispatch timing-related events, and three, it would execute the various hardware and software module service tasks.

In the case of the two physical buttons, the first (top) button was assigned a soft-reset role, in the case of a communication failure which resulted in the robot's motors being left on and the system uncontrollable. This *emergency stop* style functionality was implemented using the microcontroller's internal watchdog system to reset the microcontroller approximately 15ms after the button press was detected. The second physical button was assigned a mode-specific role, according to Table 5.1.

Mode	**Function**
Bluetooth Mode	Initiates a connection to stored Bluetooth device address
HID Mode	*Unused*
Mass Storage Mode	Enables sensor logging to the attached flash drive

TABLE 5.1: Table showing the function of the mode-specific physical pushbutton.

Each time the main loop detected that the system update tick timer period had elapsed, it would notify all timing-dependant hardware modules of this fact; for example, the LCD driver relies on these updates to automatically fade the LCD backlight brightness after a given period of inactivity. Also performed in this section is the update of the sensor values via the sensor platform at regular intervals, and the logging of this data either to the attached Mass Storage USB disk (if logging is enabled) or an established virtual serial connection to a remote PC via Bluetooth.

5.3.2 Hardware Drivers

At the point at which the abstract software in the device needed to interact with the physical board hardware, a set of hardware abstraction drivers was created. These drivers served to encapsulate the functionality of the physical hardware and expose that functionality to the rest of the firmware via a set of basic control API functions. Not every driver sought to expose all the possible abilities of the hardware; due to time constraints only those features actually required by the prototype robot firmware were implemented in most cases.

5.3.2.1 Buttons Driver

The hardware for the board button driver was, as expected, very simple; no debouncing was implemented in the driver itself, as this was not found to be necessary in the firmware. Adequate debouncing for the button logic could be achieved elsewhere in the code instead, via the software flags the buttons controlled.

As a result, the completed button driver implementation was trivial, consisting only of a configuration routine to configure the appropriate GPIO lines as inputs with the microcontroller's internal pull-up resistors enabled, and a status routine to read and mask out the appropriate port lines.

5.3.2.2 External SRAM Driver

While the selected AT90USB1287 microcontroller contained 8KB of SRAM internally for stack, global variables and other working-set data, an external 128KB SRAM IC was mounted externally on the microcontroller board. This memory was attached to the AVR's external memory interface bus, and could then be used to extend the SRAM memory space at the cost of an extra CPU cycle for each external bus access.

Unfortunately, while this external SRAM memory uses a 17-bit address, the AVR's external memory bus interface is only 16-bits wide. As a result, a small software shim driver was required to perform manual bank swapping when required to select one of the two halves of memory. This total 128KB of external SRAM memory was thus divided into two 64KB memory banks, only one of which could be selected at one time.

5.3.2.3 Headlights Driver

Like the button driver, the headlight driver contained only a thin wrapper around the GPIO pin used to control the robot's headlights. Latching of the headlight state was achieved through the GPIO hardware itself; once set to a particular state, the headlights would remain in that state (illuminated or disabled) until changed by a subsequent call to the module's update routine.

5.3.2.4 LCD Driver

The LCD chosen for the robot contained a chipset compatible with the HD44780 display controller, common to many embedded systems where complex graphics are not required. As a result, there is already a plethora of LCD drivers available on the internet from hobbyists and from most microcontroller silicon vendors. Despite this, a simple custom LCD driver was written from scratch for the project, to ensure that as much of the project as was practical remained under the sole author's copyright and distribution control.

While the robot hardware contained a direct hardware connection to the LCD display's R/W pin (for read/write control) the final driver code used a more basic hard-coded busy-wait delay method to ensure the display's timing was met. This practice proved to be the easiest to implement however for better performance this would have to be re-written as a polling scheme of the LCD controller's logical busy flag to reduce the system latency.

To conserve battery life, the LCD driver implemented an optional auto-dimming feature for the LCD backlight; when enabled, the display backlight would remain at full brightness after any updates for several seconds, before being faded gradually down to half brightness. This feature ensured that the display remained visible even in low-light conditions, but the reduction in brightness conserved battery power from the relatively power hungry backlight.

5.3.2.5 Motor Driver

To drive the external H-Bridge circuit used in the robot's motor controller hardware, a software module had to be written to correctly generate the required direction and pulse train signals. A single 16-bit hardware timer was used for this purpose, with its dual PWM outputs connected to the motor control circuit hardware.

The function to control the motor output was significantly more complex than anticipated, due to the slow switching characteristics of the inverter hardware chosen (see Chapter 6). The resulting driver had to ensure that during a direction change of one or more motors, the PWM signal of the motor would be completely disabled, to prevent momentary shorts of the main battery during the switching period.

Through trial and error, the motor PWM timer period was set to `0x0FFF`, giving a frequency of around 8.1KHz at 16MHz due to the disabled prescaler and chosen *"Phase Correct"* timer mode. Values below this range moved the PWM frequency noticibly into the range of human hearing, while raising this frequency reduced the efficiency of the motors and reduced the motor drive torque.

5.3.2.6 RGB LED Driver

Like the robot headlights driver, the RGB status LED driver contained very little complexity. As a convenience, this driver exposed two sets of `enum` values which could be used to set the colour of the RGB LED on the board. The first enum contained the literal colour names, which could be used to set a particular colour, while the second enum contained logical aliases of these colours for the various system states (see Listing 5.1).

```
enum RGB_Colour_t
{
   RGB_COLOUR_Off         = 0,
   RGB_COLOUR_Red         = (1 << 4),
   RGB_COLOUR_Green       = (1 << 5),
   RGB_COLOUR_Blue        = (1 << 6),
   RGB_COLOUR_Yellow      = (RGB_COLOUR_Red  | RGB_COLOUR_Green),
   RGB_COLOUR_Cyan        = (RGB_COLOUR_Blue | RGB_COLOUR_Green),
   RGB_COLOUR_Magenta     = (RGB_COLOUR_Red  | RGB_COLOUR_Blue),
   RGB_COLOUR_White       = (RGB_COLOUR_Red  | RGB_COLOUR_Green | RGB_COLOUR_Blue
    ),
};

enum RGB_Colour_Aliases_t
{
   RGB_ALIAS_Disconnected = RGB_COLOUR_White,
   RGB_ALIAS_Enumerating  = RGB_COLOUR_Yellow,
   RGB_ALIAS_Error        = RGB_COLOUR_Red,
   RGB_ALIAS_Ready        = RGB_COLOUR_Green,
   RGB_ALIAS_Connected    = RGB_COLOUR_Blue,
   RGB_ALIAS_Busy         = RGB_COLOUR_Magenta,
};
```

Listing 5.1: RGB LED Driver's colour enums.

Either of these two enum's values could be passed into the RGB LED driver's update routine, however in most cases the second (logical alias) versions were used to allow for easy modifications to the status colours at a later stage if desired.

5.3.2.7 Speaker Driver

To drive the robot's small piezo speaker, an 8-bit hardware timer on the AVR microcontroller was used to generate an appropriate PWM square-wave of variable frequency to control the speaker driver transistor. Calling functions may supply either a raw timer count value to set the PWM frequency, or they may use the `SPEAKER_HZ()` macro exposed by the module to convert a desired frequency into the closest timer count value.

A secondary feature of the speaker driver is the ability to play back one of several predefined sequences of notes, which are embedded into the firmware. These sequences are then used to play audible status indications on request from an external module. Note sequences are encoded as arrays of 16-bit unsigned values; the upper byte of which contains the PWM timer value to load into the timer, and the lower byte contains the number of system ticks the note should play for. As a convenience, the module-internal `SPEAKER_NOTE()` macro performs the required encoding from a given note frequency and duration in milliseconds. A `0x0000` zero entry terminates each note sequence.

5.3.3 Sensor Platform

As the robot contained an (optional) set of physical environment sensors, a *"Sensor Platform"* module was created to logically encapsulate all aspects of the sensors—from initialization and updates, to data formatting of the retrieved values—into a single package that could be integrated into the rest of the project easily, but also remain extendable enough that it could also be re-used in other future projects. The sensor platform is comprised of two software layers; the abstract sensor management layer, and the physical sensor drivers.

5.3.3.1 Abstract Sensor Management

While the robot's auxiliary sensor boards (the Atmel *Inertial One* and *Pressure One*) contained several different sensor ICs with very different characteristics, the Sensor Platform module was designed to abstract these differences out from the rest of the firmware. This abstraction was achieved by providing a pair of simple initialization and update functions, and a consistent structure for the retrieved data from each sensor. An additional pair of functions were written to convert the retrieved sensor values into a Comma Separated Values (CSV) format. Using this method of encoding the data ensured that the retrieved data could be streamed out to one or more logical consumers in a standardized manner. Missing sensors (either not mounted or faulty) are automatically ignored by the sensor platform once the call to their initialization function has failed to complete.

Unfortunately, this abstraction led to one notable problem; as each sensor has a variety of configuration parameters which are specific to that particular device (or physical property it measures) an abstract interface for sensor configuration could not easily be written. While this could be solved with additional design and planning, for the purposes of the project each sensor's configuration was instead fixed to sane defaults inside the physical sensor drivers, and no interface provided to alter these parameters on the fly externally.

The C language structure used to encapsulate the state of a single sensor is shown in Listing 5.2. This structure definition is instantiated as an array inside the sensor platform, with one entry then being dedicated to each physical property being measured (as distinct from each physical sensor IC). In the case of the ITG3200 Gyroscope sensor IC, the internal temperature sensor was used in addition to the orientation data. In this particular case, the temperature sensor was assigned a second sensor structure entry in the sensor structure array.

```c
typedef struct
{
   /* Indicates if the current sensor is connected or not. */
   bool  Connected;

   /* Human-readable name of the sensor, stored in SRAM. */
   const char* Name;

   /* Indicates if the sensor data is represented as a single value, or a
    triplicate of three axis values. */
   bool SingleAxis;

   /* Last retrieved data from the sensor. */
   union
   {
      /* Sensor data if the sensor outputs a single axis value. */
      int32_t Single;

      /* Sensor data if the sensor outputs three axis values. */
      struct
      {
         /* X axis value of the triplicate. */
         int16_t X;

         /* Y axis value of the triplicate. */
         int16_t Y;

         /* Z axis value of the triplicate. */
         int16_t Z;
      } Triplicate;
   } Data;
} SensorData_t;
```

Listing 5.2: Sensor Platform's Abstract Sensor entry structure definition.

Of note is the use of a C *union* to contain the retrieved sensor data, as either a single `int32_t` signed 32-bit integer value, or a triplicate of three `int16_t` signed 16-bit integers. The use of this union minimises the amount of memory used by each sensor entry, as the two mutually exclusive styles of returned data can overlap physically in RAM (see Figure 5.3). For sensors returning only a single 32-bit value, the sensor initialization function sets the corresponding `SingleAxis` item in the structure so that the platform knows how to extract and format the retrieved data.

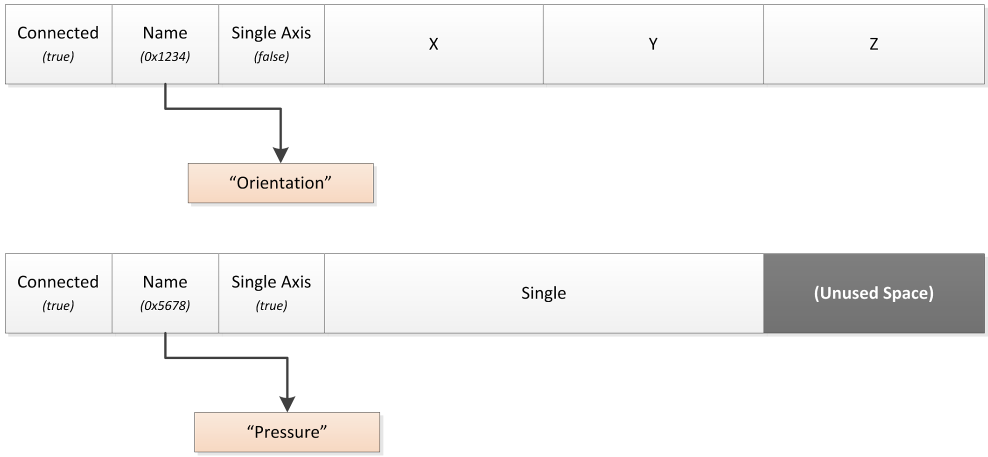


FIGURE 5.3: Diagram showing the layout of the Sensor Platform Entry structure in memory for triple axis (*top*) and single axis (*bottom*) sensors.

5.3.3.2 Individual Sensor Drivers

Each individual sensor connected to the board requires a custom sensor driver, specific to that make and model of sensor IC. Unique to each sensor is the sequence required to set up the sensor to a known set of default configuration parameters (see Table 5.2), as well as the exact command set, address of the I^2C bus, and optional use of control/interrupt GPIO lines used in the sensor's operation.

Sensor	**Type**	**I^2C Address**	**Settings**
AK8975	Direction	0x0C	N/A
BMA150	Acceleration	0x38	25Hz bandwidth, +/-2g range, Interrupt line enabled
BMP085	Pressure	0x77	N/A
ITG3200	Orientation	0x68	100Hz at an internal sampling rate of 1KHz, Low Pass Filter to use 20Hz bandwidth, Gyroscope X axis PLL as the clock source, Interrupt line enabled
ITG3200	Temperature	0x68	N/A *(Virtual Sensor)*

TABLE 5.2: Table showing the sensors used and their configuration properties.

To ensure the interface into each individual sensor driver was as uniform as possible, the exact implementation details was hidden from external modules, with each driver exposing just two functions; an `Init()` function to initialize the sensor, and an `Update()` function to pull the latest values from the sensor if it has completed a conversion. To prevent slow sensors from introducing unnecessary lag into the system, the update functions would abort if the next conversion was not ready at the point in time that the function was called, and the sensor entry structure would retain the previously retrieved sensor value. Each time a completed sensor data conversion was read from the sensor, a new conversion was started asynchronously.

5.3.4 Third Party Modules

While every effort was made to ensure that as much code as possible for the project was written from scratch, some allowances had to be made for third party libraries. Such libraries were used in places where the complexity of the module would hinder the project if a new implementation had to be implemented from scratch within the project's timeframe.

5.3.4.1 LUFA

A crucial part of the project was the ability to communicate with USB devices; the AVR's USB interface was to be used for both configuration and control of the robot, via a variety of USB devices. A full USB stack is a rather complicated affair; generally it takes an entire team of programmers working together and ample debugging time to produce a functional stack. Because of the short timeframe of the project, an existing stack was chosen instead.

The selected stack was LUFA, the *Lightweight USB Framework for AVR devices.* This stack, a personal side-project of the author, offers a rich device support in both USB host and device modes. The library contains many inbuilt drivers for the various classes of USB devices, some of which were used in the project for the HID joystick control and Mass Storage configuration and sensor logging.

For the robot firmware, LUFA version 20111009 was used.

5.3.4.2 FatFS

To read and write files onto an attached Mass Storage USB device, it was neccesary to add a filesystem driver into the system. Using a standard filesystem allowed the firmware to maintain compatibility with files stored onto the disk using other devices. As the de-facto standard for embedded system filesystem is Microsoft's FAT (due to its relative simplicity when compared to more modern filesystems) a FAT compatible filesystem driver was selected for the project.

The third-party FATFs library was chosen for this task, as it offers a free, tested, portable and light-weight C implementation of the FAT filesystem standard. The FATFs library is compatible with most variants of the FAT standard (including FAT16 and FAT32) making it an ideal choice for the project. Linking the FATFs library's read and write callback functions to the LUFA USB library's Mass Storage class driver section functions proved to be much easier than anticipated (see Listing 5.3).

```
DRESULT disk_read(BYTE drv, BYTE *buff, DWORD sector, BYTE count)
{
   uint8_t ErrorCode = RES_OK;

   if (USB_HostState != HOST_STATE_Configured)
   {
      ErrorCode = RES_NOTRDY;
   }
   else if (MS_Host_ReadDeviceBlocks(&Disk_MS_Interface, 0,
                                     sector, count, 512, buff))
   {
      MS_Host_ResetMSInterface(&Disk_MS_Interface);
      ErrorCode = RES_ERROR;
   }

   return ErrorCode;
}

DRESULT disk_write(BYTE drv, BYTE *buff, DWORD sector, BYTE count)
{
   uint8_t ErrorCode = RES_OK;

   if (USB_HostState != HOST_STATE_Configured)
   {
      ErrorCode = RES_NOTRDY;
   }
   else if (MS_Host_WriteDeviceBlocks(&Disk_MS_Interface, 0,
                                      sector, count, 512, buff))
   {
      MS_Host_ResetMSInterface(&Disk_MS_Interface);
      ErrorCode = RES_ERROR;
   }

   return ErrorCode;
}
```

Listing 5.3: FATFs shim layer, connecting it to the LUFA USB Stack Mass Storage class driver.

5.3.5 USB Management

In order to support the various types of USB devices needed by the robot firmware, a collection of USB class-specific management layers were written. These layers, sitting in parallel on top of the LUFA USB stack, provide the routines neccesary to manage each type of supported USB device.

While the LUFA USB stack provides support for several of these classes of USB devices internally, additional code was required to wrap the existing USB class driver, to extend the provided functionality and interface the driver with the rest of the system.

5.3.5.1 Bluetooth Adapters

The LUFA stack version used in the project did not contain a ready-made internal driver for USB Bluetooth adapters. A suitable driver was thus created specifically for the project using the USB transport specification outlined in the Bluetooth 2.1 specification document. In order to support all possible Bluetooth adapters from all silicon vendors, the driver was written to match generically on the defined `class`, `subclass` and `protocol` Device Descriptor values of `0xE0`, `0x01` and `0x01` respectively, as set by the Bluetooth specification for conformant Bluetooth adapters. By matching against these values instead of a particular Vendor ID and Product ID, the driver was able to support all devices conforming to the Bluetooth standard's USB transport interface specification without additional modifications being required.

During the enumeration process of an inserted USB device, the main function calls the module's `BluetoothAdapter_ConfigurePipes()` function to attempt to bind it to the inserted USB device. The module first validates the device descriptor to ensure the inserted device is reportedly a Bluetooth adapter. Next, the pipe configuration routine attempts to configure the USB controller's logical data pipes so that the *Data In*, *Data Out* and *Event In* pipes are correctly connected to their matching logical endpoints within the adapter. If the driver is unable to bind to the device for any reason, the enumeration process is aborted for the Bluetooth transport driver.

Periodically, the main firmware loop will call the Bluetooth transport driver's `BluetoothAdapter_USBTask()` service task if the transport driver is currently active. This service task is responsible for checking the logical data pipes for new data, and (if data is availiable) reading in the data packet before dispatching it to the Bluetooth stack. A `CALLBACK_Bluetooth_SendPacket()` callback function from the Bluetooth stack takes care of sending packets generated from the Bluetooth stack to the Bluetooth adapter.

5.3.5.2 HID Devices

For local diagnostics of the system before the Bluetooth stack was completed, a local Human Interface Device (HID) driver was implemented into the firmware. This driver builds on top of the HID class driver included in the LUFA distribution used, and binds supported HID devices (such as game controllers) to the robot's physical functions. Using this driver, a USB joystick or game controller can be inserted into the robot and used to control the motors, headlights and horn.

As most HID devices carry a unique physical and logical report layout, it is important to include a mechanism to correctly bind the appropriate buttons on the attached device to the correct logical function. Two seemingly identical USB joysticks can output very different report data structures to the host, requiring the use of a *HID Descriptor Report Parser* to correctly parse the reports into a standard format. The HID Report Parser included in the LUFA USB stack was used for this purpose, and linked into the rest of the system. This allows the robot to maintain compatibility with virtually all HID devices containing an appropriate number of buttons.

5.3.5.3 Mass Storage Devices

As a means of system configuration and monitoring, a Mass Storage Device (MSD) driver was also added into the device firmware. This proved useful for both fault-finding and data logging, as data produced by the firmware could be stored onto an inserted USB flash drive. During the initial firmware development, this functionality was used to store logs of the on-board sensor outputs, to ensure the correctness of the sensor platform before the completion of the wireless serial port functionality. At a later stage, this mode was extended so that it could be used to configure the target remote Bluetooth Devive Address used in the robot's Bluetooth Mode when a remote connection was initiated by the user.

To be able to read and write files on an attached USB flash disk drive, the FATFs library was linked to the LUFA Mass Storage class driver. This abstracted out the physical medium, giving a higher level file-centric view of the attached storage medium, as opposed to the direct physical sectors.

When a disk is inserted, the firmware will first attempt to open an existing file called `SENSLOG.CSV` on the disk, for sensor logging in CSV format (see Listing 5.4). If such a file does not exist, a new one is created and the sensor log header (constructed by the Sensor Platform module) is written to the start of the new file.

Additionally, the firware will attempt to open and process a second file names `REMADDR.TXT`, which holds the Bluetooth device address of the remote Bluetooth device the robot should attempt a connection to upon demand. This file is expected to contain an address of the format `XX:XX:XX:XX:XX:XX`, containing the six consecutive octets of the

```
static bool MassStorage_OpenSensorLogFile(void)
{
   uint8_t ErrorCode;

   /* Create a new sensor log file on the disk, fail if one already exists */
   ErrorCode = f_open(&MassStorage_DiskLogFile, DATALOG_FILENAME,
                      (FA_CREATE_NEW | FA_WRITE));

   /* See if the existing log was created sucessfully */
   if (ErrorCode == FR_OK)
   {
      /* Construct the sensor CSV header line(s) */
      char     LineBuffer[200];
      uint8_t  LineLength = Sensors_WriteSensorCSVHeader(LineBuffer);

      /* Write constructed CSV header to the attached mass storage disk */
      uint16_t BytesWritten = 0;
      f_write(&MassStorage_DiskLogFile, LineBuffer, LineLength, &BytesWritten);
   }
   else if (ErrorCode == FR_EXIST)
   {
      /* Open the already existing file on the disk */
      f_open(&MassStorage_DiskLogFile, DATALOG_FILENAME,
             (FA_OPEN_EXISTING | FA_WRITE));

      /* Seek to the end of the existing log file */
      f_lseek(&MassStorage_DiskLogFile, MassStorage_DiskLogFile.fsize);
   }
   else
   {
      /* Return disk error */
      return false;
   }

   return true;
}
```

Listing 5.4: Example of the Mass Storage manager code to open and write to a file on an attached USB flash disk using the FATFs library.

remote device's address, in hexadecimal format. This address is then stored into the robot's non-volatile EEPROM memory for later use when a Bluetooth adapter is subsequently inserted. If no such file exists on the attached disk, one is created and the currently stored remote device address written to it.

5.3.6 Bluetooth Management Layer

Connecting the project's Bluetooth Stack to the rest of the robot firmware was the Bluetooth Management layer, responsible for providing the appropriate callback function implementations required by the stack. These callback functions are fired by the Bluetooth stack in response to defined events, such as connection requests, channel establishment and packet reception.

A secondary duty performed by the management software layer is the dispatch of events and data to and from the various Bluetooth services. These services, such as RFCOMM, SDP and HID, are optional in all implementations, and thus require manual integration into each project on top of the base Bluetooth stack if desired. These services must be linked to the various Bluetooth stack callback events so that they may function correctly (see Listing 5.5).

```
void EVENT_Bluetooth_DataReceived(BT_StackConfig_t* const StackState,
                                  BT_L2CAP_Channel_t* const Channel,
                                  uint16_t Length,
                                  uint8_t* Data)
{
   /* Dispatch packets with a known protocol to the integrated services - for
    unknown packets, display a message to the LCD */
   switch (Channel->PSM)
   {
      case CHANNEL_PSM_SDP:
      case CHANNEL_PSM_HIDCTL:
      case CHANNEL_PSM_HIDINT:
      case CHANNEL_PSM_RFCOMM:
         SDP_ProcessPacket(StackState, Channel, Length, Data);
         HID_ProcessPacket(StackState, Channel, Length, Data);
         RFCOMM_ProcessPacket(StackState, Channel, Length, Data);
         break;
      default:
         LCD_WriteFormattedString_P(PSTR("\fL2CAP Recv:%04X\n"
                                         "PSM:%04X C:%04X"),
                                    Length, Channel->PSM, Channel->LocalNumber);
         break;
   }
}
```

Listing 5.5: One callback from the Bluetooth Stack, dispatching received packets to the integrated Bluetooth services.

In some cases, a Bluetooth enabled application may peform specific filtering on incoming connection and channel requests; an embedded device may be programmed to reject all but one specfic remote device, or only accept specific channel protocols. However, as the *ExplorerBot* was designed to be controlled publically via any supported Bluetooth enabled device, no such filtering was performed in the request callback routines. All HCI connection and L2CAP channel event callbacks were implemented, however, to display their relevant event data onto the robot's LCD display for debugging purposes.

Inside this management layer, the RFCOMM service channel open and close events were hooked, so that the opened RFCOMM channel handles could be captured. This captured handle is then stored temporarily by the robot firmware and used for later streaming of the sensor data. As only one connection handle is stored by the device at any one time for this purpose, a limitation of the system is imposed; only one wireless serial port can receive sensor data at the one time.

The HID service callback for packet reception was also hooked in this layer, and implemented to process incoming HID reports from Bluetooth devices. These processed reports—from game controllers, mobile phones, or other HID compliant devices—were then fed back into the main firmware module to control the robot's motors, headlights and horn hardware.

Chapter 6

Project Results

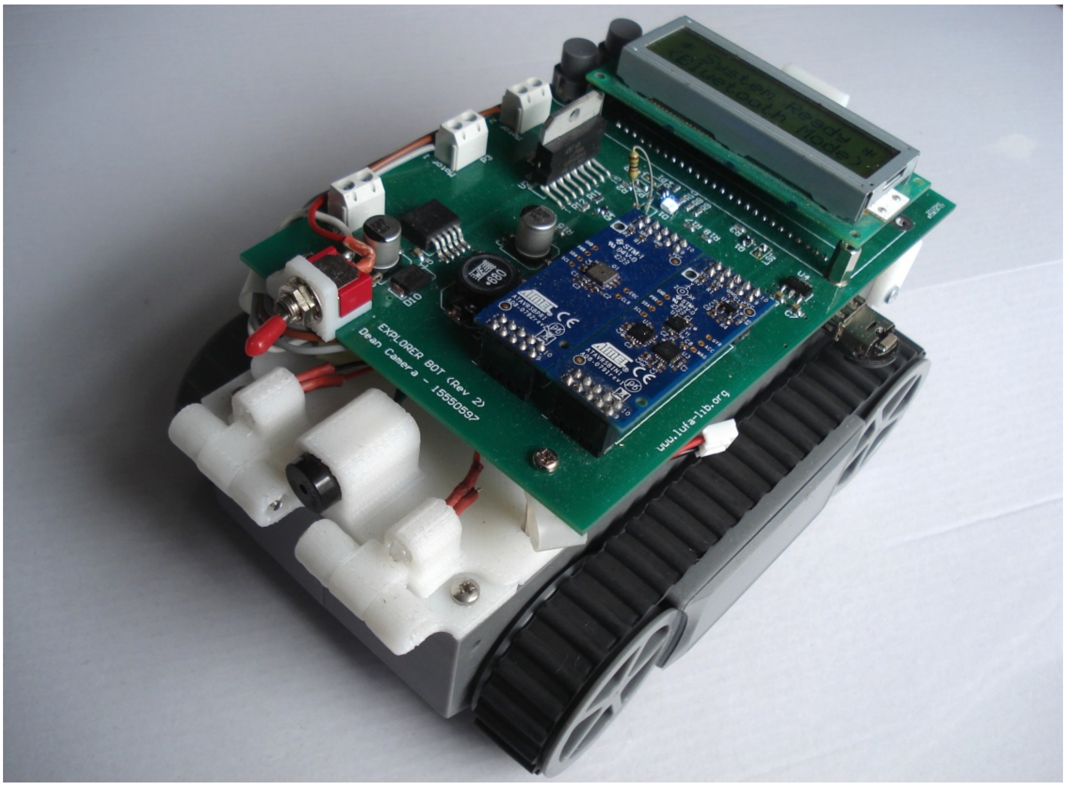


FIGURE 6.1: Photo of the completed *ExplorerBot* prototype robot

FIGURE 6.2: Tested consumer grade Bluetooth enabled devices: Playstation 3 controller *(left)*, Nintendo Wii controller *(middle)*, Sony-Ericcson z550i mobile phone *(right)*

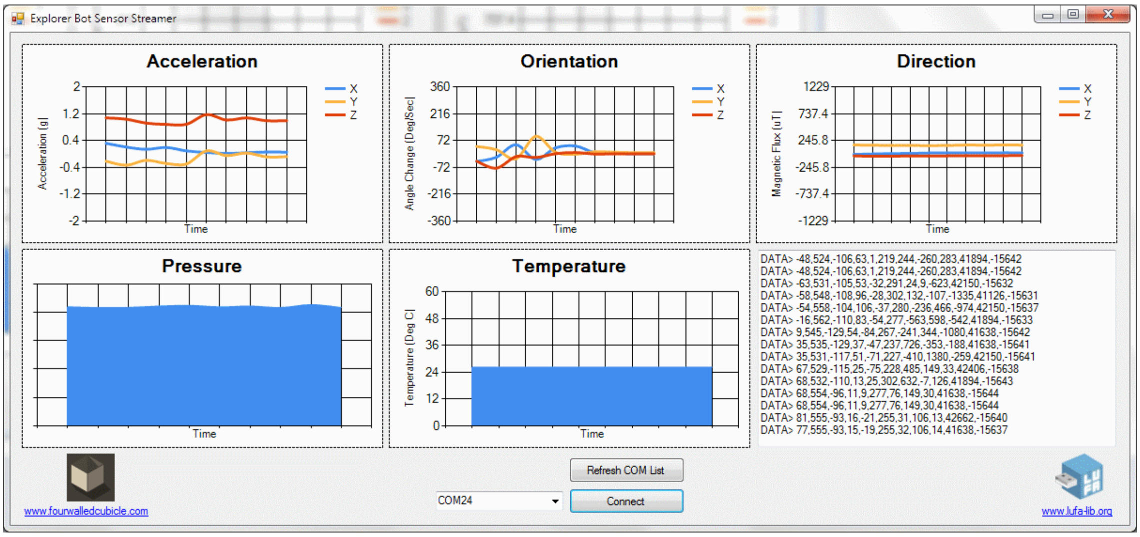


FIGURE 6.3: Sensor logging application, showing data streaming wirelessly from the connected robot via a Bluetooth virtual serial port.

Chapter 7

Project Discussion

7.1 Issues Faced

7.1.1 Micropendous Pinout PCB Error

7.1.2 Motor Inrush Current

7.1.3 PCA9306 Physical Package

7.1.4 L298D Unavailabilty

7.1.5 Incorrect Transistor Pinout

7.1.6 Slow Drive Transistors

7.1.7 Unreliable Packet Buffering

7.1.8 PS3 Controller Compatibility

7.1.9 Wii Controller Compatibility

7.2 Project Significance

Chapter 8

Conclusion

The purpose of this project was to research, design and implement a functional Embedded Bluetooth Stack for inclusion in small-scale embedded environments. A secondary goal of the project was to design, manufacture and test a practical implementation of an embedded project using this stack to demonstrate the stack's completeness and praticality in a real-world environment.

To date, both of these goals have been achieved to some degree; the Bluetooth stack was completed to a functional state suitable for use in some environments, and a working Bluetooth controlled robot was completed to a point where it could interact with consumer Bluetooth devices wirelessly over several protocols. Several areas of the Bluetooth stack warrant future expansion and refinement, but the code as-is demonstrates a solid platform onto which a rich variety of low-cost Bluetooth enabled devices can be implemented.

8.1 Project Success

Measuring the success of a project is a difficult task; one has to take into account the original goals of the project, the timeframes set for each goal, and make allowances for changes to these goals and project specifications in response to technical and physical challenges faced in the course of completing the project. Several unexpected delays in obtaining a correct PCB from the board manufacturer threatened to push out the timeframes alloted for the physical robot construction by several weeks, and supply problems in obtaining several parts caused delays due to the need to revise the PCB design around alternative parts.

Despite these setbacks, the project can objectively be shown as a success overall — the objectives of the project were met, and a completed working robot was produced running from the Bluetooth stack within the alloted project timeframe.

8.2 Future Work

While functional, the Bluetooth stack at the point of the project's end was not complete. Several areas have been identified for improvement, which at the point of writing prevent the stack from being used in some environments. These areas are listed below.

8.2.1 Reliable Packet Buffering and Retransmissions

The L2CAP layer of Bluetooth has several similarities to the TCP/IP network stack of a modern computer network; messages transmitted through an established link are considered to be reliable, that is, messages sent will attempt retransmission automatically if they are not correctly processed by the remote device. For this mechanism to function, a retransmission timer must be implemented, and an outgoing packet buffer added into the system to buffer packets and re-send them in the event that the remote end does not respond in the correct manner within the the specified timeout period.

For the purposes of the project, such a reliable buffer and timeout system was not implemented due to time constraints. A special reliable event system was included in the L2CAP management layer to ensure that channels are correctly negotiated regardless of the presence or lack of such a buffering layer, however the Bluetooth services implemented on top of the L2CAP layer do not contain such a mechanism. This can result in messages sent from the upper Bluetooth services to be lost and the service put into a lock-up state until the connection to the non-responsive remote device is reset.

Future stack development would implement a system where either the services are expanded to reliably manage their state internally, or the lower layers extended to require the addition of a reliable packet buffer and timeout service.

8.2.2 Packet Reception Fragmentation Support

A large component of modern communication systems in the concept of data packeting and fragmentation, the act of splitting up a larger communication message into a set of smaller discrete units for individual transfer. Bluetooth is no exception to this; each logical layer within the Bluetooth specification may specifiy a maximum fragment size to a remote device, above which large messages are split into smaller chunks. These fragments must then be received individually and reconstructed to obtain the original message.

In the Bluetooth stack written for the project, outgoing packets are correctly fragmented into sizes appropriate for the receiving device, however received packets do not currently run through a reassembly layer. In lieu of this feature, the stack requires that the local device is capable of receiving packets in their entirety up to the maximum single packet size as defined by the Bluetooth 2.1 specifications, that of 64KB. For the stack to be considered feature complete, fragmentation reassembly support should be added at a later stage.

8.2.3 Expansion of Bluetooth Services

Only a small subset of the standardized Bluetooth services were implemented in the stack for the project, and of those implemented, only the specific roles (device or server) required for the robot's functionality were completed. Future expansion of the project would require the completion of the existing services for both roles, as well as the addition of other Bluetooth classes such as the Bluetooth Personal Networking (PAN) service for wireless networking.

8.2.4 Additional Physical Transports

The Bluetooth 2.1 specification defines additional physical transport mediums, for the transport of HCI packets between the Bluetooth Host Controller and the application Microcontroller. Callback hooks from the stack are supplied for the user-supplied implementation of any desired transport mechanism, however for the project only the USB transport was implemented. Future development of the stack should implement the most widely used transport medium—i.e. the UART serial HCI packet tranport—so that Bluetooth HCI controllers may be installed directly on the PCB alongside the main microcontroller in an embedded device.

8.3 Final Words

While the formal portion of this project is now complete, I look forward to the continuing development and discussion of the project in my spare time; now that a solid base has been written, future work may be performed at my leisure to move the project out of the academic envionment and into real-world use. The next critical step in this project is the construction of a community around the stack's development, so as to gain new ideas, bug reports and community contributions.

From the development of this project I was able to gain valuable experience in building real-world projects. I was able to practice proper project time mangement and goal-based development, and experience the realities of PCB manufacturing delays, component shortages and tight deadlines. This experience has enriched my engineering abilities, and I now feel confident in moving towards my imminent professional future with these new skills and knowledge.

Abbreviations

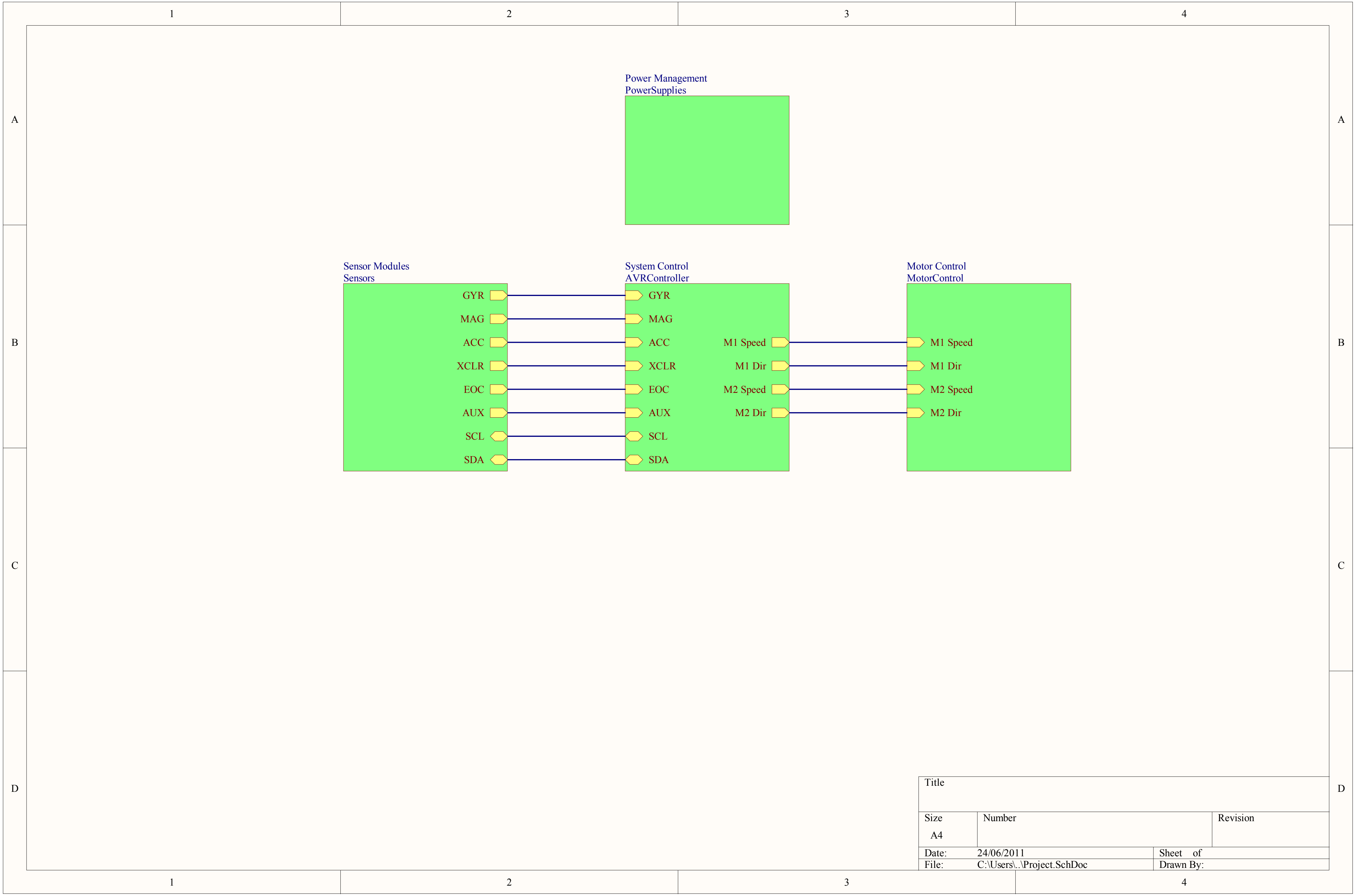
API	**A**pplication **P**rogramming **I**nterface
AVR	This isn't an acronym. Officially, at least.
CPU	**C**entral **P**rocessing **U**nit
EEPROM	**E**lectronically **E**rasable **R**ead **O**nly **M**emory
FAT	**F**ile **A**llocation **T**able
GPIO	**G**eneral **P**urpose **I**nput/**O**utput
HCI	Bluetooth **H**ost **C**ontroller **I**nterface
HID	**H**uman **I**nterface **D**evice
IC	**I**ntergrated **C**ircuit
I^{2}C	**I**nter **I**ntegrated **C**ircuit Bus
JTAG	**J**oint **T**est **A**ction **G**roup
L2CAP	Bluetooth **L**ogical **L**ink **C**ontrol and **A**daption **P**rotocol
LCD	**L**iquid **C**rystal **D**isplay
LED	**L**ight **E**mitting **D**iode
LUFA	**L**ightweight **U**SB **F**ramework for **A**VRs
PC	**P**ersonal **C**omputer
PCB	**P**rinted **C**ircuit **B**oard
PLL	**P**hase **L**ocked **L**oop
PWM	**P**ulse **W**idth **M**odulation
RFCOMM	**R**adio **F**requency **Comm**unication
RTOS	**R**eal **T**ime **O**perating **S**ystem
SDP	Bluetooth **S**ervice **D**iscovery **P**rotocol
SRAM	**S**tatic **R**andom **A**ccess **M**emory
USB	**U**niversal **S**erial **B**us

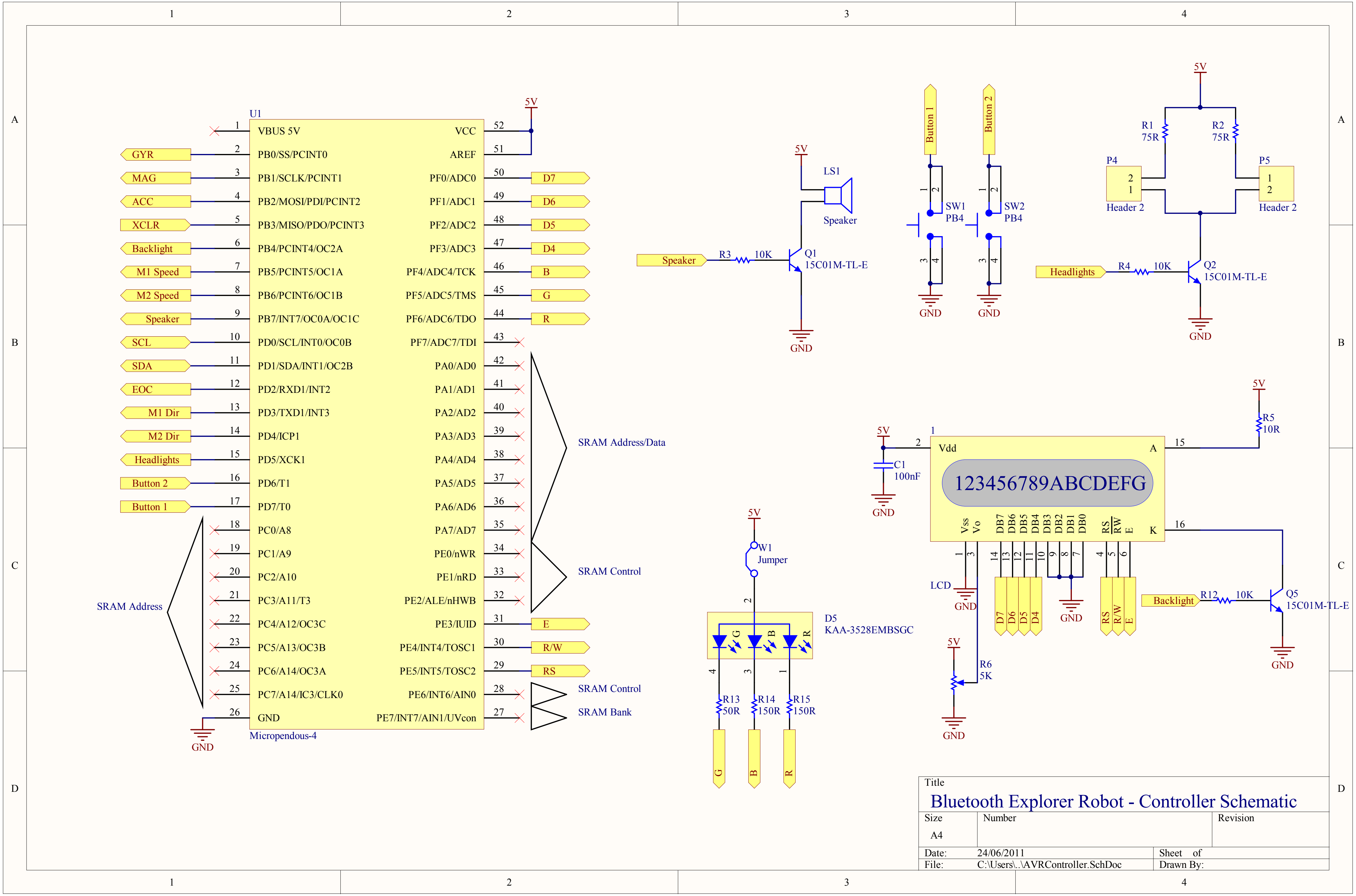
Appendix A

Schematics

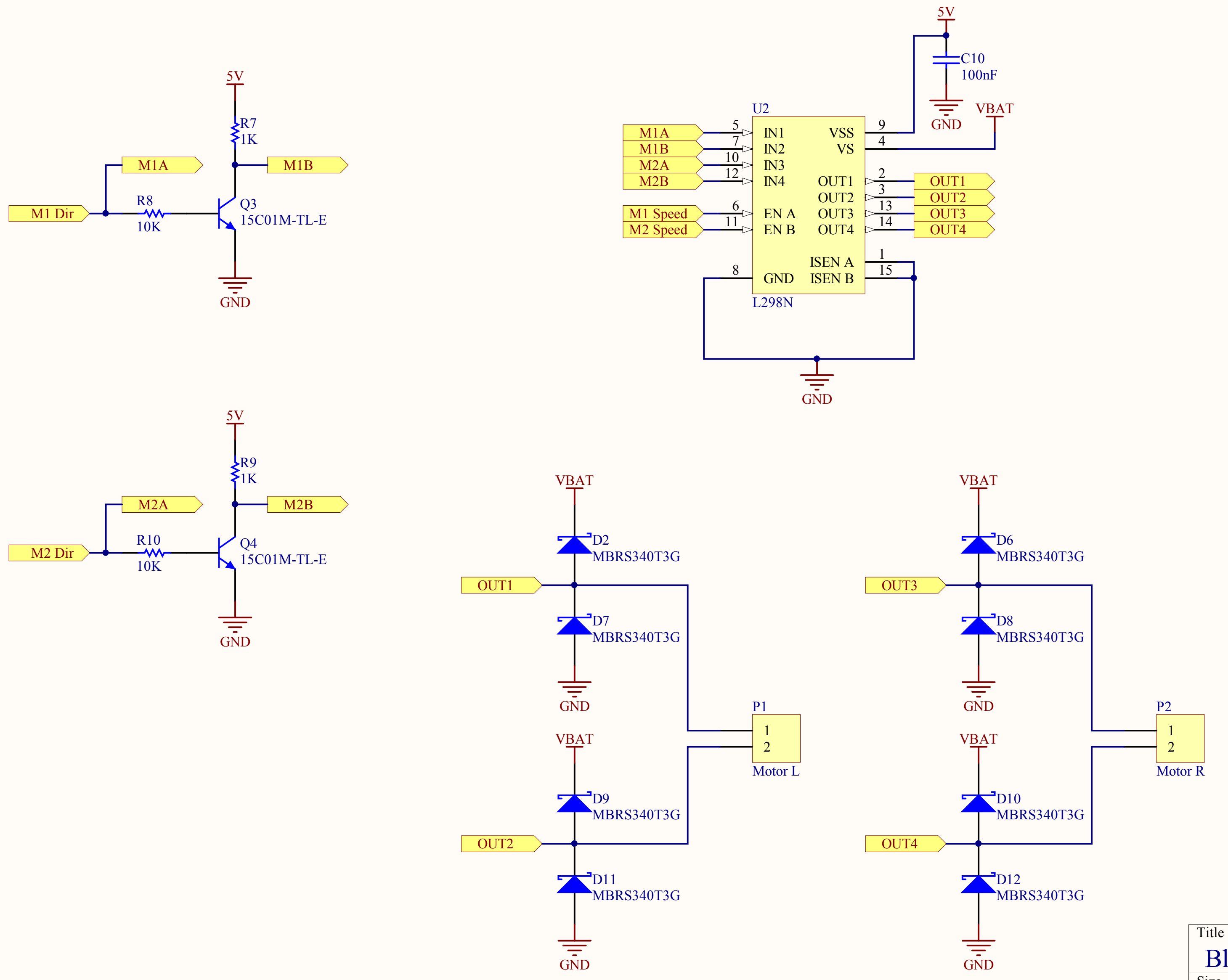
A.1 Robot Schematics

The following pages illustrate the complete schematics of the *ExplorerBot* robot hardware created to demonstrate a practical application of the Bluetooth stack. These are reproduced in logical block form, for each main functionality block of the finished robot hardware.

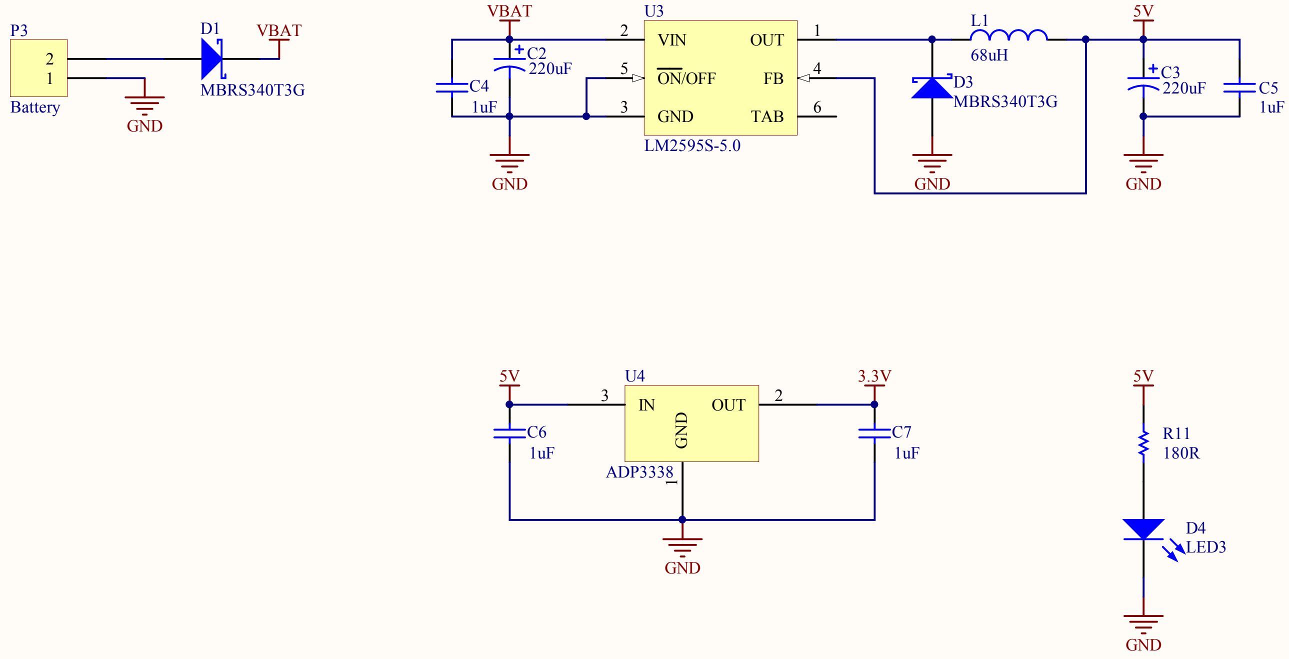

1
2
3
4
A
B
C
D
Power Management
PowerSupplies
Sensor Modules
Sensors
GYR
MAG
ACC
XCLR
EOC
AUX
SCL
SDA
System Control
AVRController
GYR
MAG
ACC
XCLR
EOC
AUX
SCL
SDA
M1 Speed
M1 Dir
M2 Speed
M2 Dir
Motor Control
MotorControl
M1 Speed
M1 Dir
M2 Speed
M2 Dir
Title
Size
A4
Number
Revision
Date: 24/06/2011
Sheet of
File: C:\Users\..\Project.SchDoc
Drawn By:



U1
Micropendous-4
VBUS 5V
PB0/SS/PCINT0
PB1/SCLK/PCINT1
PB2/MOSI/PDI/PCINT2
PB3/MISO/PDO/PCINT3
PB4/PCINT4/OC2A
PB5/PCINT5/OC1A
PB6/PCINT6/OC1B
PB7/INT7/OC0A/OC1C
PD0/SCL/INT0/OC0B
PD1/SDA/INT1/OC2B
PD2/RXD1/INT2
PD3/TXD1/INT3
PD4/ICP1
PD5/XCK1
PD6/T1
PD7/T0
PC0/A8
PC1/A9
PC2/A10
PC3/A11/T3
PC4/A12/OC3C
PC5/A13/OC3B
PC6/A14/OC3A
PC7/A14/IC3/CLK0
GND
VCC
AREF
PF0/ADC0
PF1/ADC1
PF2/ADC2
PF3/ADC3
PF4/ADC4/TCK
PF5/ADC5/TMS
PF6/ADC6/TDO
PF7/ADC7/TDI
PA0/AD0
PA1/AD1
PA2/AD2
PA3/AD3
PA4/AD4
PA5/AD5
PA6/AD6
PA7/AD7
PE0/nWR
PE1/nRD
PE2/ALE/nHWB
PE3/IUID
PE4/INT4/TOSC1
PE5/INT5/TOSC2
PE6/INT6/AIN0
PE7/INT7/AIN1/UVcon
GYR
MAG
ACC
XCLR
Backlight
M1 Speed
M2 Speed
Speaker
SCL
SDA
EOC
M1 Dir
M2 Dir
Headlights
Button 2
Button 1
SRAM Address
D7
D6
D5
D4
B
G
R
SRAM Address/Data
SRAM Control
E
R/W
RS
SRAM Control
SRAM Bank
5V
GND
LS1
Speaker
R3 10K
Q1 15C01M-TL-E
Button 1
Button 2
SW1 PB4
SW2 PB4
R1 75R
R2 75R
P4 Header 2
P5 Header 2
R4 10K
Q2 15C01M-TL-E
W1 Jumper
D5 KAA-3528EMBSGC
R13 50R
R14 150R
R15 150R
C1 100nF
LCD
Vdd
123456789ABCDEFG
Vss Vo DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0 RS RW E
A
K
R5 10R
R6 5K
R12 10K
Q5 15C01M-TL-E
Title
Bluetooth Explorer Robot - Controller Schematic
Size
A4
Number
Revision
Date: 24/06/2011
Sheet of
File: C:\Users\..\AVRController.SchDoc
Drawn By:



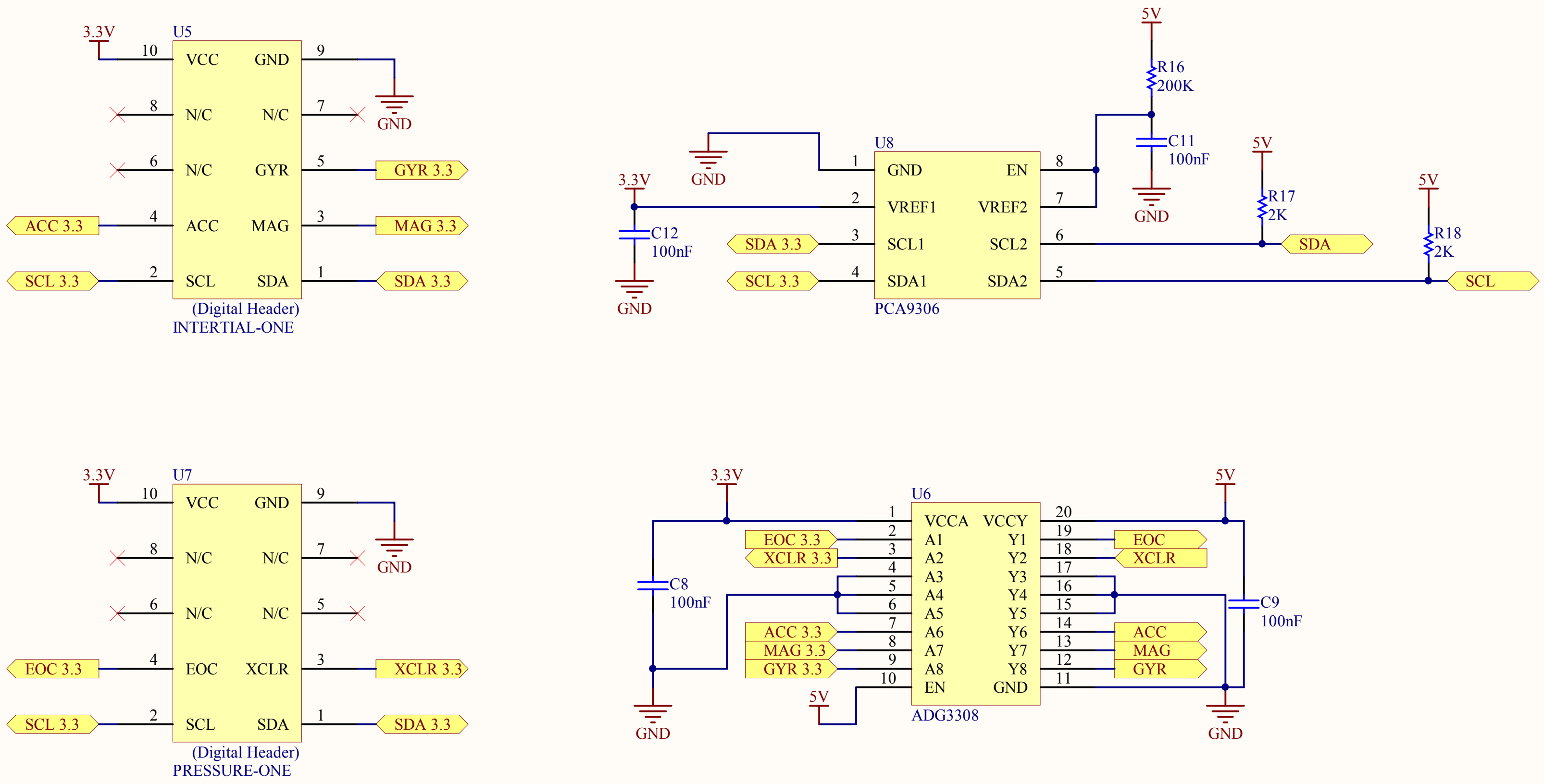
Title		
Bluetooth Explorer Robot - Motor Control Schematic		
Size: A4	Number	Revision
Date: 24/06/2011		Sheet of
File: C:\Users\..\MotorControl.SchDoc		Drawn By:



Title
Bluetooth Explorer Robot - Power Supply Schematic

Size	Number	Revision
A4		

Date:	24/06/2011	Sheet of
File:	C:\Users\..\PowerSupplies.SchDoc	Drawn By:



Title		
Bluetooth Explorer Robot - Sensors Schematic		
Size	Number	Revision
A4		
Date: 24/06/2011		Sheet of
File: C:\Users\..\Sensors.SchDoc		Drawn By:

Appendix B

Source Code

The complete project source code is available for download online, due to its significant size. Both the robot firmware and the embedded Bluetooth stack may be viewed and modified according to the licence agreements included at the top of each source file in the download package.

B.1 Obtaining the Source Code

All relevant material relating to this project (including source code, schematics, and this document) may be obtained from the official project page, located at *http://www.fourwalledcubicle.com/ExplorerBot.php*.

Appendix C

Stack API Overview

Appendix D

Robot User Guide

In this section, the basic operation of the completed *ExplorerBot* robot hardware is outlined. An annotated diagram of the completed robot design is shown in Figure D.1 below.

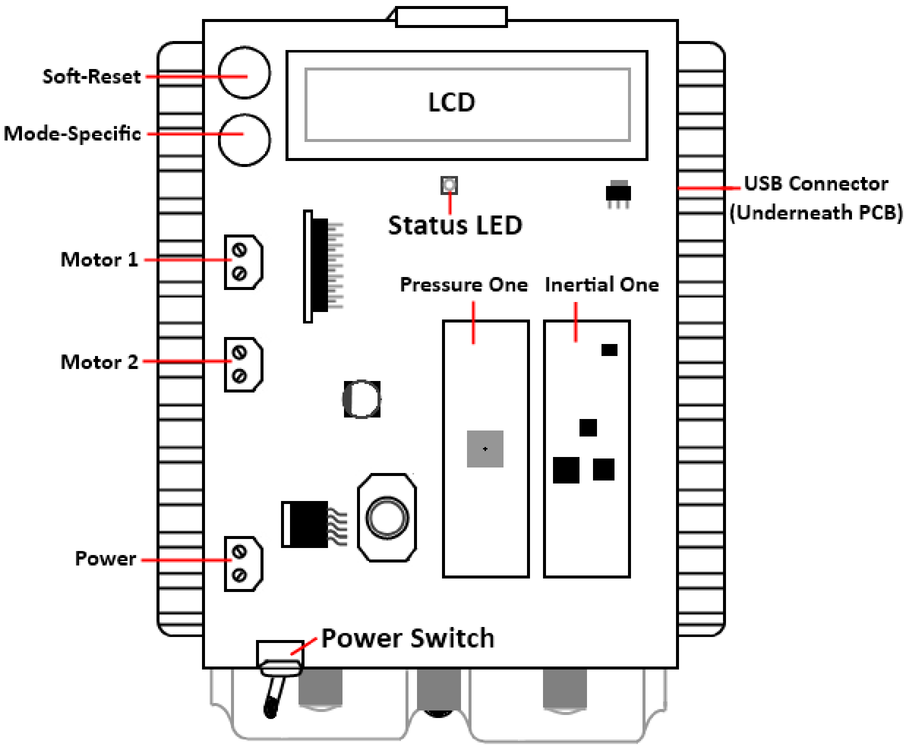


Figure D.1: Annotated diagram of the robot.

D.1 Power Requirements

The robot requires a 6-9V DC input power supply, capable of supplying up to 1A of current. Ideally, a 7.2V Lithium Ion battery should be used for this purpose for maximum operating time (due to its abilitiy to supply large amount of instataneous current, used by the motors when switched on).

D.2 Supported USB Devices

Several classes of USB devices are supported by the robot firmware. Where possible, each class is supported in a generic manner, so that any device compliant with the relevant USB class specification can be used.

D.2.1 HID Class Devices

For robot control over a wired interface, USB HID class devices may be inserted into the robot's USB recepticle. Compatible HID devices must contain at least one four-way directional pad and four buttons for all robot features to be operational. The button mappings are listed in Table D.1 below.

Function	**PS3 Controller**	**Other HID Device**
Left	D-Pad Left	Button 1
Forward	D-Pad Up	Button 2
Backward	D-Pad Down	Button 3
Right	D-Pad Right	Button 4
Headlights (Momentary)	R2	Button 5
Horn	L2	Button 6
Headlights (Toggle)	R1	Button 7
Novelty Horn	L1	Button 8

Table D.1: Button mappings between various USB HID devices and the robot functions.

In the special case of a Playstation 3 controller being inserted into the robot, the controller will be automatically configured to pair over Bluetooth with the address of the last Bluetooth USB adapter inserted into the robot. Once paired, the controller may then establish a connection with the robot over Bluetooth by pressing the PS3 button located in the center of the controller.

D.2.2 Mass Storage Class Devices

Flash drives suitably formatted with a FAT32 filesystem may be used for sensor logging and system configuration of the robot. Compatible memory sticks must be no more than 4GB in size.

On insertion, the robot will attempt to read a file names `REMADDR.TXT` from the disk. This file should contain the Bluetooth Device Address of the remote device the robot should attempt to establish a connection with while in Bluetooth mode.

D.2.3 Bluetooth Adapter Devices

D.3 Supported Bluetooth Services

D.3.1 HID Service

D.3.2 RFCOMM Service

Bibliography