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 design and produce a free, open source, small footprint Bluetooth stack aimed to suit small embedded environments. This project will 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 expert 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 generous contribution of the free *"Micropendous"* AVR microcontroller boards used in the final robot prototype.

Contents

List of Figures

List of Tables

List of Code Listings

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.

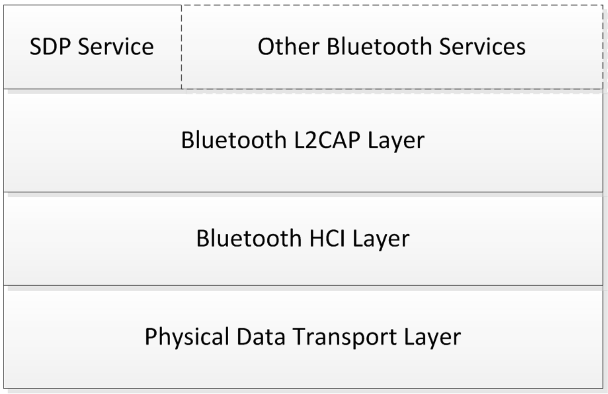


Figure 1.1: Diagram of the basic components which form a typical Bluetooth Stack

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.

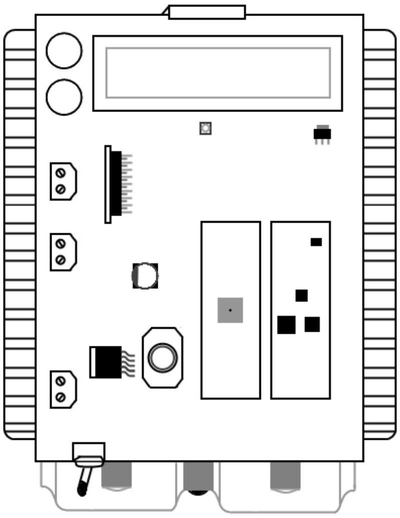


Figure 1.2: Diagram of the proposed robot platform.

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 the market currently offers.

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

The development of the Bluetooth stack followed a bottom-up development methodology, designing and writing each software layer in sequence from the lowest layer first, to the highest and most abstracted layers last. This sequence ensured that each layer was functionally correct and could be verified before the higher layers were implemented. This suited the development of the Bluetooth stack—and, indeed, most software stacks—as each higher software layer is dependant solely on lower logical layers in the stack.

3.1 Software Overview

The software layers implemented in the completed Bluetooth stack are shown in Figure 3.1.

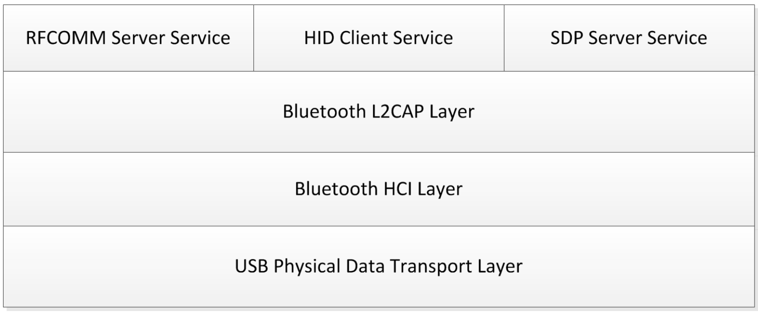


Figure 3.1: Diagram showing the completed layers of the Bluetooth stack.

Each software layer was implemented as a separate pair of source code module files, written in the C language and targeted towards the C99 language specification. Configuration for the stack is located in the file `BluetoothCommon.h`, which must be included in the user application to operate the stack.

By default, only the Bluetooth stack core (giving the HCI and L2CAP layers) is integrated; services must be linked into the stack manually via the provided event and callback routines (see later in this chapter) to provide service-level functionality on top of the base stack functionality. This includes the SDP server service; while generally a core requirement of most Bluetooth devices, it may be omitted in applications not exposing server role services to save space in the resulting application binary.

3.2 Design Considerations

A number of design restrictions were placed on the development of the Bluetooth stack; these ensured that the completed stack remained modular, extensible and suitable for integration in a wide range of embedded systems with various capabilities and, conversely, limitations. As an unfortunate but expected side effect, a number of these restrictions complicated the software development process significantly.

3.2.1 No Heap Memory Allocations

The largest restriction placed on the stack design was the requirement of static and stack allocated memory allocation only — no dynamic heap allocated memory was to be used via the usual `malloc()` and `free()` functions from the `libc` library. Most embedded environments contain three types of memory allocations:

1. **Static Allocations**, including global variables and `static` qualified variables.
2. **Stack Allocations**, automatically populated to store local automatic variables, function parameters, return addresses, etc.
3. **Heap Allocations**, persistent dynamic allocations manually managed by the user application.

However, for significantly memory constrained systems, heap allocations are generally avoided due to the possibility of memory fragmentation; large numbers of differently-sized allocations and deallocations may cause fragmented holes to appear within the heap environment, causing allocation exhaustion to occur even if sufficient (raw) memory is available for an allocation request. In addition, dynamic memory complicates the formal analysis of a program, making memory requirements harder to judge, and thus harder to determine whether a given set of firmware will operate correctly within a specified environment without further in-depth analysis. In large embedded environments (ones with virtual memory management, and/or a large heap space) these problems are less of an issue and dynamic memory may be preferable for its simplicity.

The design of embedded stacks based on a no-heap-allocation strategy is a complex one, requiring certain trade-offs to be made. In the case of the Bluetooth stack presented here, a major side-effect of this decision was the need to fix the maximum number of simultaneous HCI device connections, L2CAP channel connections, and other queue- and list-like data structures within each instance of the stack. A major downside to this approach is the increased complexity of packet buffering and re-assembly, as fixed-size buffers must be used.

3.2.2 Endianness Correction

A second complication based on the physical constraints from the execution environment is the native endianness of the processor hardware, i.e., the order in which multiple-byte data values are stored and interpreted inside the processor registers. Depending on the execution architecture, data values may be represented in Big Endian (most significant byte at the lowest address) or Little Endian (least significant byte at the lowest address) format within the CPU. Data transmitted and received at the local device to remote device boundary must undergo a encoding conversion if the endianness of the source and sink do not match.

In the case of the Bluetooth specification, this was further complicated by the various layers; some layers used Little Endian encoding for exchanged data, while others selected a Big Endian format. To perform the correct encoding/decoding on each platform, a set of conversion macros were implemented at points where multi-byte values were exchanged. This ensured that regardless of the native endian format of the architecture the stack is compiled to, multi-byte values will be interpreted correctly.

3.2.3 Physical Transport Independence

As the Bluetooth 2.1 specification outlines a several possible physical transports of HCI packets between the Bluetooth transceiver silicon and the application microcontroller, it was important that the physical transport layer be made independent from the rest of the stack. By making no assumptions about the form of transport protocol used to transfer packets to and from the attached Bluetooth silicon, the user application is free to implement their own transport mechanism without having to modify the internals of the stack. One instance of the stack may run across a USB connection to an attached

Bluetooth module, while another may simultaneously connect to the microcontroller over a UART link.

3.2.4 Bluetooth Specification Compatibility

For a software stack to be suitable for general use, it must be designed around, and undergo a series of tests to ensure compatibility with, a particular version of the protocol specification. Without a fixed version of a specification, correctness of the finished stack and compatibility with other implementations of the same technology cannot be assured.

As a *compliant* stack would require extensive development and expensive certification procedures, the stack was instead aimed to be *compatible* with version 2.1 of the Bluetooth specification. This subtle difference in the terminology indicates a large difference in the restrictions around the exact implementation. While a *compliant* stack must be verified against the entire specification and rigorously tested to ensure exact conformance, a *compatible* must only work with (most) other devices implementing their own compliant or compatible stacks of the same protocol version.

In the future the stack could be made conformant if desired, however for the purposes of the project only protocol compatibility was tested.

3.2.5 Multiple Stack Instances

For maximum utility, the stack was designed to allow for multiple simultaneous stack instances. This was implemented to allow for unusual Bluetooth usage scenarios where more than one transceiver is desirable in a single system, either for security, wireless signal coverage or bandwidth reasons. Each physical transceiver is allocated a logical stack instance, capable of sustaining one or more simultaneous connections, and one or more logical channels between devices. This flexibility ensures that the majority of usage scenarios are covered by the stack's implementation.

Code-wise, the multiple stack functionality is implemented as a common first parameter to all functions within the stack, `BT_StackConfig_t* const StackState`. This parameter is used to pass around a reference to the stack instance being operated upon, and is somewhat analogous to the `this` property in C++ classes, if the entire stack was wrapped in a single class. As the stack was written in the C language which lacks Object Orientated language capabilities, this form of basic polymorphism was implemented manually. For each physical transceiver, the user application is expected to declare a new instance of the `BT_StackConfig_t` structure within their code, and pass this to the main Bluetooth stack management routines (see Listing 3.1).

```
static BT_StackConfig_t BluetoothAdapter_Stack =
   {
      .Config =
         {
            .Class               = (DEVICE_CLASS_SERVICE_CAPTURING    |
                                    DEVICE_CLASS_MAJOR_COMPUTER       |
                                    DEVICE_CLASS_MINOR_COMPUTER_PALM),
            .Name                = "Bluetooth Robot",
            .PINCode             = "0000",
            .Discoverable        = true,
            .PacketBuffer        = EXTERNAL_MEMORY_START,
         }
   };
```

Listing 3.1: Configuration example of a Bluetooth stack instance.

3.2.6 Multiple Simultaneous Connections and Channels

The HCI connections and L2CAP channels for each stack instance are implemented as an pair of object pools within the `BT_StackConfig_t` structure; L2CAP channel objects within the pool are shared amongst all established HCI connections — for example, one HCI connection may use all the available L2CAP channels, or two connections may share the pool equally. This design decision allows the full pool of L2CAP channel objects to be used by the stack regardless of the HCI connection requesting them. Figure 3.2 illustrates a single stack instance with a typical usage scenario, with two of the stack's HCI objects containing established connections, each with several L2CAP channel objects associated with the connection from the channel object pool.

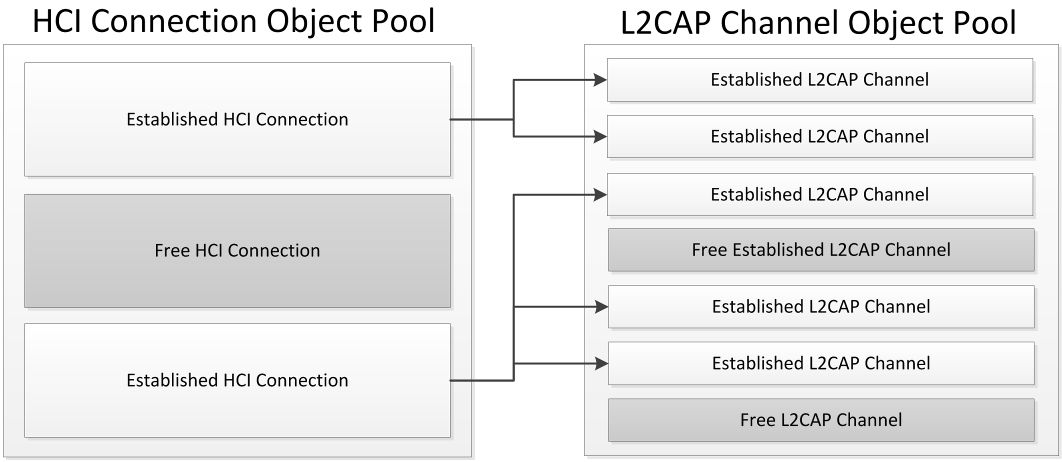

HCI Connection Object Pool
L2CAP Channel Object Pool
Established HCI Connection
Free HCI Connection
Established HCI Connection
Established L2CAP Channel
Established L2CAP Channel
Established L2CAP Channel
Free Established L2CAP Channel
Established L2CAP Channel
Established L2CAP Channel
Free L2CAP Channel

Figure 3.2: Diagram showing the HCI Connection and L2CAP object pools within a single Bluetooth stack instance.

To positively link one used L2CAP channel instance to its parent HCI connection object, the HCI connection's `handle` property is stored in the L2CAP channel object as a primary key. This handle value, allocated uniquely by the external Bluetooth HCI controller silicon when a new HCI connection is made, is then used to locate the HCI connection object within a stack instance from an established channel instance. While a pointer to the parent HCI object would offer a faster method of locating the associated HCI connection, this would tightly couple the HCI layer with the L2CAP layer as the former would need to alter the latter's objects if and when HCI connections are terminated.

As the stack does not use heap-based dynamic memory allocation, this approach ensures that the stack contains similar levels of flexibility to a dynamically allocated object approach, with only a minimal overhead. The maximum size of each object pool is set via the Bluetooth Stack's `BT_MAX_DEVICE_CONNECTIONS` and `BT_MAX_LOGICAL_CHANNELS` configuration defines.

3.2.7 Minimal Memory Usage

While large embedded systems may run the completed Bluetooth stack, the design decisions made during its development were primarily aimed at ensuring the best performance and smallest footprint in severely resource-constrained microcontrollers. This resulted in the stack being optimised for small systems; only the minimal attributes required to establish and maintain connections are stored by the stack. One example of these optimizations is in the event system (described in detail later in this chapter) implementation: rather than implementing a callback registration system for each event, named callbacks are used instead. Various portions of the Bluetooth stack expect the user application to provide callback and event functions using names and prototypes defined in the stack header files.

This named callback function system reduces the memory footprint of the stack, as the callback addresses do not need to be stored in RAM at runtime, and the compiler does not need to inflate the binary with additional functions to manage the callback function registrations.

3.3 Software Layer Implementation

In this section, the design and implementation of each individual software layer that comprises the complete Bluetooth stack is described in detail.

3.3.1 Physical Transport

While not part of the completed stack proper, a functional USB transport layer was completed for the project as an sample implementation in the user application, and hooked into the stack via the provided APIs. The Bluetooth USB transport requires the use of four logical USB communication channels ("pipes" in USB terminology) to the USB Bluetooth adapter; the mandatory control pipe for HCI commands, a pair of *Bulk* type pipes for data packets to and from the controller, and an *Interrupt* pipe for HCI event notifications.

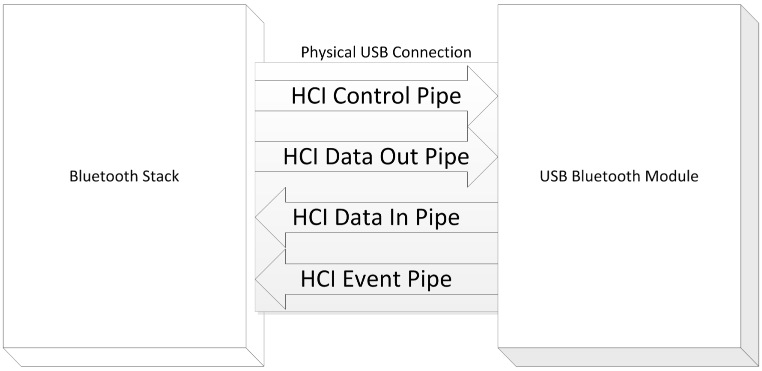


Figure 3.3: Diagram showing the logical USB pipes within the Bluetooth USB physical transport.

Data and event packets received from the USB Bluetooth adapter's logical pipes are written into the packet buffer indicated in the stack instance configuration, and dispatched to the stack via the `Bluetooth_ProcessPacket()` function. There they are processed internally via the Bluetooth stack, moving up the HCI, L2CAP and user-provided service layers as appropriate. The matching `CALLBACK_Bluetooth_SendPacket()` callback function from the stack implements the data and HCI control packet transmission code back to the adapter, sourcing the generated packets from the same packet buffer location.

3.3.2 HCI Layer

Sitting at the lowest level of the core stack, the HCI layer performs the main role of managing connections between Bluetooth devices, as well as processing events from the Bluetooth controller, and data encapsulation/decapsulation for transport over the Bluetooth link. To perform these tasks, the HCI layer is invoked each time a data packet is received or ready to be sent, an event is received from the controller, or during idle periods of the user application for general connection management.

The current HCI layer state is implemented as a state machine, which is automatically advanced through the appropriate states on stack initialization to configure the attached Bluetooth device, according to the configuration parameters set in the stack instance configuration structure. These states, shown in Listing 3.2, correspond to the asynchronous initialization steps performed by the stack when the HCI layer is reset.

```
enum BT_HCIStates_t
{
   HCISTATE_Init_Reset           = 0,
   HCISTATE_Init_ReadBufferSize = 1,
   HCISTATE_Init_GetBDADDR      = 2,
   HCISTATE_Init_SetLocalName   = 3,
   HCISTATE_Init_SetDeviceClass = 4,
   HCISTATE_Init_SetScanEnable  = 5,
   HCISTATE_Idle                = 6,
};
```

Listing 3.2: State machine states for the HCI management layer.

The HCI baseband controller management in the HCI layer is split into two distinct parts; an event processing routine to process commands and manage transitions between HCI states, and the state machine processing routine to execute commands based on the current HCI state. As commands must only be sent once on the transition between HCI state machine states, a flag `StateTransition` in the HCI state structure of each Bluetooth stack instance indicates if a state transition has occurred within the event processing function, so that the state machine processor can issue the appropriate responses on transition edges only.

State timeouts within each of the HCI states is managed via a `TicksElapsed` counter in the HCI state structure of each Bluetooth stack instance. This counter, incremented each time the HCI layer is notified by the user application of a tick period elapsing, forces a resend of the current HCI state's command by forcing the state transition flag `StateTransition` high when the timeout period has expired. This system ensures that HCI commands dropped during the initialisation of the module are properly re-issued.

If a connection is requested by a remote device, the HCI layer will issue a `CALLBACK_Bluetooth_ConnectionRequest()` callback into the user application, if space is available within the HCI connection object pool. If no space is available (i.e. all HCI channel objects are currently allocated for established connections) the connection request is silently discarded with the Bluetooth error code `HCI_ERROR_LIMITED_RESOURCES`. If the user application accepts the connection request, the stack will issue an acceptance response to the remote device. If rejected, a suitable rejection notification is issued by the stack's HCI layer.

Authentication is currently implemented in a slave connection role only, as a PIN based authentication. All sent connection requests and acceptance packets indicate that the device requires a slave role in the connection establishment, indicating that the remote device is responsible for requesting the authentication PIN code from the user, which the local stack then authenticates against the PIN code configured in each stack instance.

Inside the HCI layer, received data packets are stripped of their HCI layer headers and passed upwards to the L2CAP layer for further processing. Packets sent from the L2CAP layer to the HCI layer undergo the opposite process; a HCI layer packet header is prefixed onto the packet contents and passed to the user application for transmission to the Bluetooth controller.

As an implementation-specific detail, HCI connection destruction events (when an established HCI connection is disconnected) are notified to the L2CAP layer via the internal function `Bluetooth_L2CAP_NotifyHCIDisconnection()`. This ensures that all associated L2CAP channels are correctly destructed once their host HCI connection is destroyed, to prevent synchronisation issues and to ensure that the L2CAP object pool is freed of any dead channel objects as soon as possible to prevent unnecessary pool exhaustion.

3.3.3 L2CAP Layer

The L2CAP layer, responsible for managing logical channels between devices for an established connection, was implemented in an unusual manner; due to uncertainties of the user application packet buffering scheme, the stack is not able to guarantee that packets to be sent are buffered correctly. As the L2CAP layer is used to establish communication channels between devices, it is essential that any issued channel configuration commands are delivered correctly to the remote device; delays in the channel negotiation process may cause communications to and from a remote device to fail. As a result, the L2CAP layer implemented a special event buffering scheme to ensure the reliable issuing of responses to remote devices even if a packet fails to send due to the controller's internal buffers becoming full.

As L2CAP signalling packets (identifiable via their destination channel ID of `0x0001`, the defined ID for `BT_CHANNEL_SIGNALING` packets) are received by the L2CAP layer, the request is processed and an event object created in the L2CAP Event Queue. This queue, a fixed length queue of a size configured by the stack's `BT_MAX_QUEUED_L2CAP_EVENTS` configuration parameter, stores the events in a First-In, First-Out ordered manner within each stack instance. The implementation of this queue is a simple flat array of event structure elements, where inserted items are appended to the end of the last allocated object in the array (see Figure 3.4). A separate `PendingEvents` property in the L2CAP state structure to track the number of queued events.

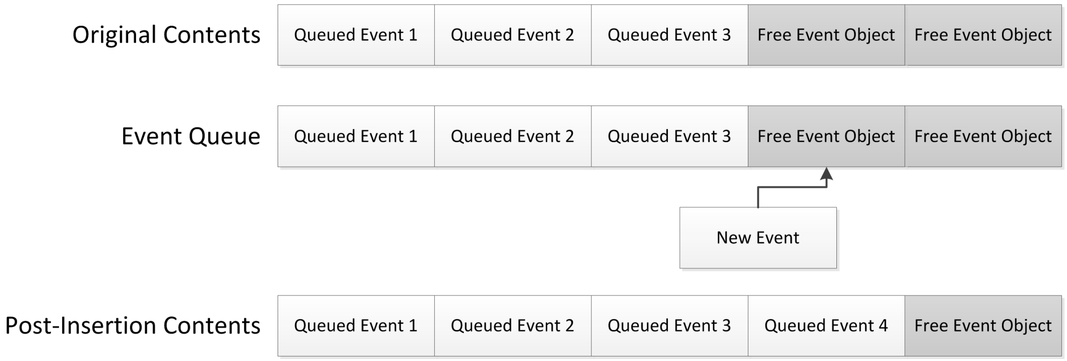


Figure 3.4: Diagram showing the insertion of a new L2CAP event into the tail of the queue.

Event queue removal is performed by the moving of all queue items above the head of the list leftwards in the array, to shuffle all entries one place towards the head of the queue (see Figure 3.5). L2CAP event objects are only dequeued once the transport layer confirms the transmission of the signalling packet request to the remote device. In this manner, L2CAP events maintain a reliable transport regardless of the physical transport's abilitiy to buffer excess packets.

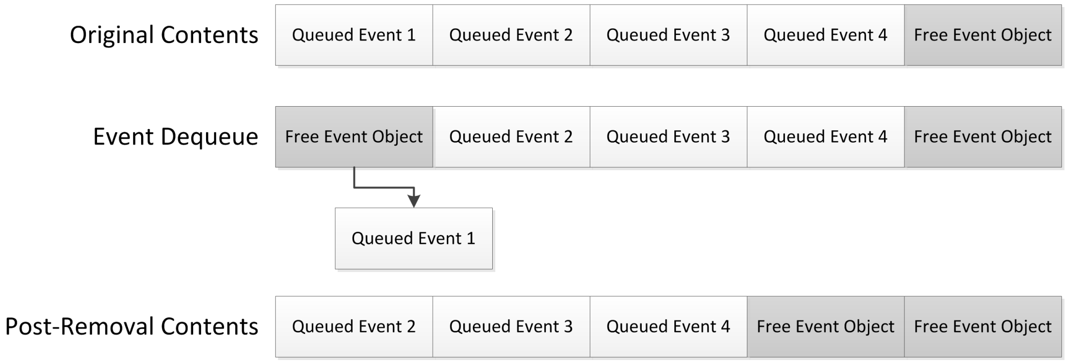


Figure 3.5: Diagram showing the removal of the L2CAP event queue head item.

If too many L2CAP events are attempted to be added to the queue, the new event allocation request will fail. This scheme will preserve existing events within the queue at the expense of dropping newer events until space becomes available. Despite this limitation, with an adequate queue size (of approximately 10 items for most applications) the queue limit should never be exceeded under normal circumstances.

3.3.4 Bluetooth Services

The Bluetooth services designed as part of the project were implemented separately to the stack core; they are designed to hook into the stack APIs if desired by the user application, however they are entirely separate entities which are not required to be built with the stack core if not needed.

Bluetooth services can implement two possible roles; a *client* role, or a *server* role. It is possible for services to implement both roles, however this is not a requirement of the Bluetooth specification. Server role services (such as a Virtual Serial Port service server) provide a function to remote Bluetooth devices on demand, while client role services control remote service server instances (such as a termination application connecting to a remote Virtual Serial Port service server.)

Each of the services designed for the stack use a common set of interface APIs; functions are provided to the user application to notify the service of channel open/close events, data packet reception, and other such common events. In addition, each service may or may not expose a set of service-specific functions and callbacks, which can be used to interact with the service's functionality from the user application.

3.3.4.1 SDP Service

For service discovery to take place between devices, each device must implement the Service Discovery Protocol service. The SDP service exposes the capabilities and characteristics of other services within the device, allowing remote systems to query the server to determine what services a device implements, and their associated properties. Other services may register themselves with the SDP service within a device, so that remote Bluetooth devices may detect the existence of the second service and make use of it if desired.

On large scale systems, SDP records are generally registered independently, and appropriate SDP records are built dynamically from the properties given to the SDP service by other services within a device. This approach allows for attributes to be dynamically added, removed and updated as required, however requires a variable amount of memory to store the generated SDP records.

By contrast, the project's Bluetooth stack uses a novel static record initialization approach for SDP service attributes; services are required to contain pre-built SDP attribute records, which are then registered wholesale with the SDP server. While this requires complex SDP attribute tables to be built inside each service manually, this eliminates the need for costly run-time dynamic service record management and dynamic memory allocation.

An example of such a static SDP attribute record is shown in Listing 3.3. Here, a single UUID attribute is encoded in the correct format structure for the SDP server.

```
static const struct
{
   SDP_ItemSequence8Bit_t UUIDList_Header;
   struct
   {
      SDP_ItemUUID_t     SerialPortUUID;
   } UUIDList;
} ATTR_PACKED PROGMEM SerialPort_Attr_ServiceClassIDs =
   {
      SDP_ITEMSEQUENCE8BIT(sizeof(SerialPort_Attr_ServiceClassIDs.UUIDList)),
      {
         SDP_ITEMUUID(SP_CLASS_UUID),
      }
   };
```

Listing 3.3: Example of a statically initialized SDP attribute.

In addition to each individual SDP attribute structure, a single Service Attribute Table must be built for each service requiring SDP registration. Listing 3.4 shows an example of a service's complete attribute table. When queried, the SDP server will example the attributes contained in this table to determine what (if any) attributes need to be returned to the requesting remote Bluetooth device.

```
static const SDP_ServiceAttributeTable_t PROGMEM SerialPort_Attribute_Table[] =
   {
      { SDP_ATTRIBUTE_ID_SERVICERECORDHANDLE,
        &SerialPort_Attr_ServiceHandle        },
      { SDP_ATTRIBUTE_ID_SERVICECLASSIDS,
        &SerialPort_Attr_ServiceClassIDs      },
      { SDP_ATTRIBUTE_ID_PROTOCOLDESCRIPTORLIST,
        &SerialPort_Attr_ProtocolDescriptor   },
      { SDP_ATTRIBUTE_ID_BROWSEGROUPLIST,
        &SerialPort_Attr_BrowseGroupList      },
      { SDP_ATTRIBUTE_ID_LANGUAGEBASEATTROFFSET,
        &SerialPort_Attr_LanguageBaseIDOffset },
      { SDP_ATTRIBUTE_ID_SERVICENAME,
        &SerialPort_Attr_ServiceName          },
      { SDP_ATTRIBUTE_ID_SERVICEDESCRIPTION,
        &SerialPort_Attr_ServiceDescription   },
   };
```

Listing 3.4: Example of a statically initialized SDP attribute table.

Finally for each service, a SDP entry node must be created. This master node is used to register the entire service and associated SDP attribute table with the SDP server. Listing 3.5 contains a sample SDP node.

```
SDP_ServiceEntry_t ServiceEntry_RFCOMMSerialPort =
   {
      .Stack                 = NULL,
      .TotalTableAttributes  = (sizeof(SerialPort_Attribute_Table) /
                                sizeof(SerialPort_Attribute_Table[0])),
      .AttributeTable        = SerialPort_Attribute_Table,
   };
```

Listing 3.5: Example of a statically initialized SDP service entry node.

The SDP server maintains a linked list of registered services internally, using the externally created SDP node entry instances. As services are registered and removed, these nodes are inserted and removed from the master SDP service list. When the SDP server is queried by a remote device, the server will transverse the list, looking for matching attributes to the request to return back to the initiating device.

3.3.4.2 RFCOMM Service

For virtual serial port functionality, the RFCOMM service was implemented as a server role; remote devices can then request logical channels to be opened in the service, so that data can be exchanged. The RFCOMM service requires the creation of a logical channel multiplexer; this allows a single physical RFCOMM channel to multiplex a collection of one or more data channels simultaneously. In the implementation of this service, the multiplexer was created using a static array of `RFCOMM_Channel_t` structure instances, each of which represented a virtual channel on the multiplexer.

As the RFCOMM service is not tied directly into the core Bluetooth stack, the channel object array is not associated with a specific Bluetooth stack, rather, the channel object pool is shared amongst all Bluetooth stack instances in the system (see Figure 3.6). The first time the service is initialised, the entire channel object pool is reset; in subsequent initializations, only channel objects currently associated to the stack instance being reset will be deallocated. This allows the RFCOMM service resources to be shared with all Bluetooth instances, without having to have multiple per-instance channel pools. The maximum number of simultaneously open RFCOMM channels is set via the `RFCOMM_MAX_OPEN_CHANNELS` configuration parameter.

While newer versions of the RFCOMM service specification (i.e. versions 1.1 onwards) add a mandatory "credit based" flow control scheme to control the flow of data in and out of each logical channel, versions 1.0B and earlier do not contain these additions. As most RFCOMM clients are compatible with either version (due to the prevalence of devices based on the older specification) the implemented service was based on the earlier protocol version for simplicity.

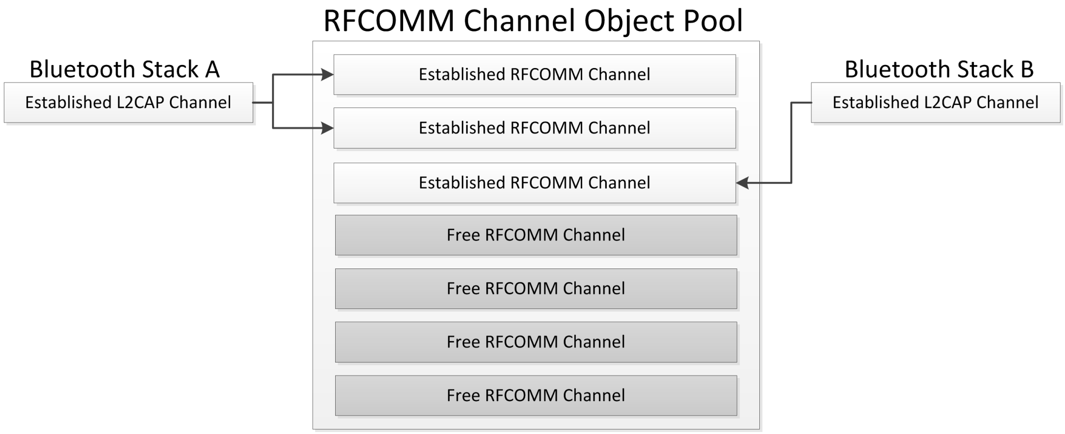

RFCOMM Channel Object Pool
Bluetooth Stack A
Established L2CAP Channel
Established RFCOMM Channel
Established RFCOMM Channel
Established RFCOMM Channel
Free RFCOMM Channel
Free RFCOMM Channel
Free RFCOMM Channel
Free RFCOMM Channel
Bluetooth Stack B
Established L2CAP Channel

FIGURE 3.6: Diagram showing the RFCOMM channel object pool, shared amongst two Bluetooth stack instances.

3.3.4.3 HID Service

Due to time limitations of the project, the HID service implementation was extremely minimal; only a basic implementation of the HID client role was designed and coded, so that received packets could be decoded and used to control the robot. As a client role version of the SDP server was not attempted, the HID service was unable to query the remote HID device's service tables for the HID descriptor, used to determine how to decode received HID data packets. This unfortunately resulted in the final robot firmware needing to hard-code the packet interpretation for each device type based on the length of the received data packet.

3.4 Integration into User Applications

In order to implement the Bluetooth stack into a user application, the entire base stack (excluding the services) must be added to the project for compilation. The user configuration and interface to the stack is made via the header file `BluetoothCommon.h`. User applications attempting to use the stack must be compiled using the C99 language specification, and a GCC compiler derivative.

3.4.1 Management Functions

The following functions must be executed by the user application to operate the stack correctly.

```
void Bluetooth_Init(BT_StackConfig_t* const StackState);
```

The stack initialization function must be called once on system startup, to initialize a given instance of the stack ready for use.

```
bool Bluetooth_ManageConnections(BT_StackConfig_t* const StackState);
```

This function must be executed periodically to manage existing connections and reply to any pending L2CAP or HCI events. The exact frequency this function must be called at is not a concrete value, rather, it should be called as fast as the user application will allow. Failure to call this function fast enough will introduce latency when establishing and maintaining new connections and logical channels or, in extreme cases, will cause connection and channel negotiations to fail.

```
void Bluetooth_ProcessPacket(BT_StackConfig_t* const StackState,
                             const uint8_t Type);
```

When a new packet is received from the Bluetooth controller, this function must be invoked. Note that the received packet data should be stored into the packet buffer set in the Bluetooth stack instance's configuration structure prior to invoking the packet processing routine.

```
void Bluetooth_TickElapsed(BT_StackConfig_t* const StackState);
```

To manage timeouts, the stack must be updated regularly of the passage of time using this function. Each time the configured `BT_TICK_MS` tick period elapses, this function should be invoked to determine if any timeout conditions must be generated internally.

3.4.2 Events and Callbacks

In response to conditions set within the stack internals, one or more event or callback functions may be invoked. These functions, implemented by the user application, allow the system to respond to events in a synchronous manner as they occur within the stack, and/or indicate to the stack the appropriate action to take when a decision is required. The user-implemented functions are split into two categories:

1. **Event Functions**, which allow the user application to respond to events. This class of function has no return value, and the user application may choose to not

provide a functional implementation for one or more events, if the event condition is of no interest to the user application.

2. **Callback Functions**, invoked when the stack reaches a condition where a decision must be made by the user application—such as whether to accept or reject a particular incoming connection or channel request—before stack operations can continue. These functions require a decision to be give as the function's return value, which is then used by the stack internally.

Event and callback invocations are generated by the stack layers as required. Figure 3.7 shows the general message, callback and response sequence path a typical software stack will follow throughout its normal processing activities.

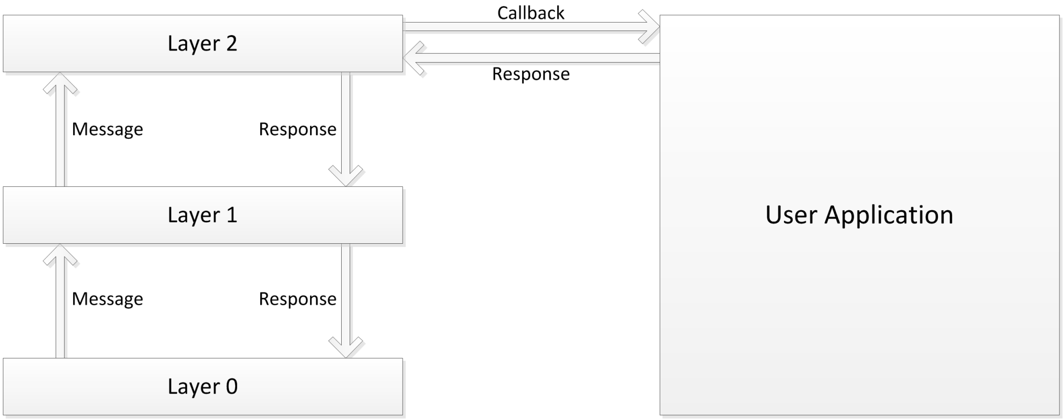


FIGURE 3.7: Diagram showing the Message, Response and Callback sequence of a typical software stack when passing a message and response through several intermediate layers.

In the project's Bluetooth stack, the names of the event and callback functions are fixed, and cannot be changed. User applications are required to implement all callback and event functions (identified via their `CALLBACK_` and `EVENT_` prefixes) using the exact prototypes given in the module headers. This design was preferred over a traditional registration system to reduce the stack's binary size and RAM footprint.

A minimal set of callback and event functions (excluding the physical packet transport callback) required to link in the SDP and RFCOMM services to the core Bluetooth stack are shown in Listing 3.6.

```
void EVENT_Bluetooth_InitServices(BT_StackConfig_t* const StackState)
{
   SDP_Init(StackState);
   RFCOMM_Init(StackState);
}

void EVENT_Bluetooth_ManageServices(BT_StackConfig_t* const StackState)
{
   SDP_Manage(StackState);
   RFCOMM_Manage(StackState);
}

bool CALLBACK_Bluetooth_ConnectionRequest(BT_StackConfig_t* const StackState,
                                          BT_HCI_Connection_t* const Connection)
{
   /* Accept all requests from all devices regardless of BDADDR */
   return true;
}

void EVENT_Bluetooth_ConnectionStateChange(BT_StackConfig_t* const StackState,
                                           BT_HCI_Connection_t* const Connection)
{
   /* Ignore state changes to the HCI connections */
}

bool CALLBACK_Bluetooth_ChannelRequest(BT_StackConfig_t* const StackState,
                                       BT_HCI_Connection_t* const Connection,
                                       BT_L2CAP_Channel_t* const Channel)
{
   /* Accept all channel requests from all devices */
   return true;
}

void EVENT_Bluetooth_ChannelStateChange(BT_StackConfig_t* const StackState,
                                        BT_L2CAP_Channel_t* const Channel)
{
   /* Determine the channel event that has occurred */
   if (Channel->State == L2CAP_CHANSTATE_Open)
   {
      SDP_ChannelOpened(StackState, Channel);
      RFCOMM_ChannelOpened(StackState, Channel);
   }
   else if (Channel->State == L2CAP_CHANSTATE_Closed)
   {
      SDP_ChannelClosed(StackState, Channel);
      RFCOMM_ChannelClosed(StackState, Channel);
   }
}

void EVENT_Bluetooth_DataReceived(BT_StackConfig_t* const StackState,
                                  BT_L2CAP_Channel_t* const Channel,
                                  uint16_t Length,
                                  uint8_t* Data)
{
   SDP_ProcessPacket(StackState, Channel, Length, Data);
   RFCOMM_ProcessPacket(StackState, Channel, Length, Data);
}

void CALLBACK_RFCOMM_DataReceived(BT_StackConfig_t* const StackState,
                                  RFCOMM_Channel_t* const Channel,
                                  uint16_t Length,
                                  uint8_t* Data)
{
   /* Process received RFCOMM data on the virtual serial port here */
}
```

Listing 3.6: Minimal set of event callbacks for SDP and RFCOMM service use.

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 best cost/benefit 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.

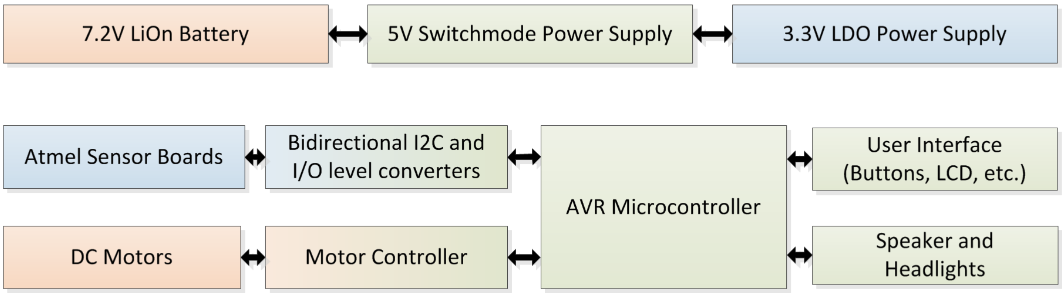


FIGURE 4.1: Robot Hardware Block Diagram

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.

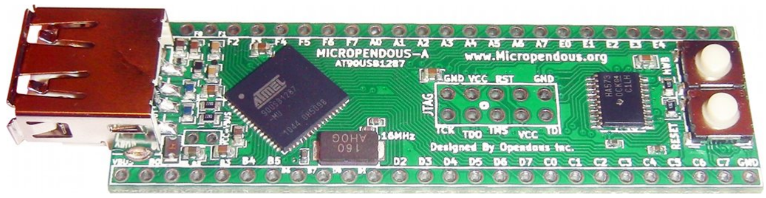


FIGURE 4.2: The Micropendous-A Board (Image courtesy *Opendous Inc.*)

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 LM2595S-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.

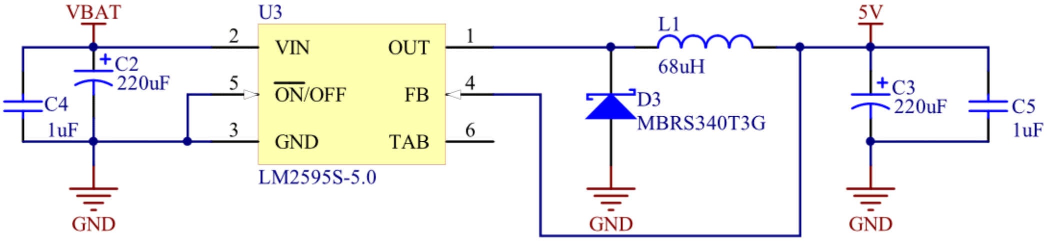


FIGURE 4.3: Schematic of the robot's main 5V switch-mode power supply.

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: List of the possible output colours of the RGB LED, from the possible binary channel 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.

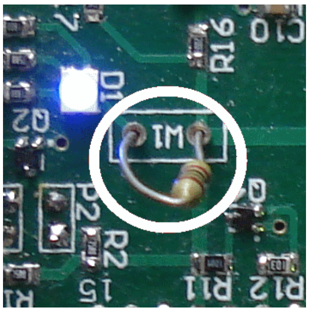


FIGURE 4.4: Close-up of the removable wire link used to disable the RGB status LED to prevent conflicts with the JTAG lines during programming/debugging.

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.

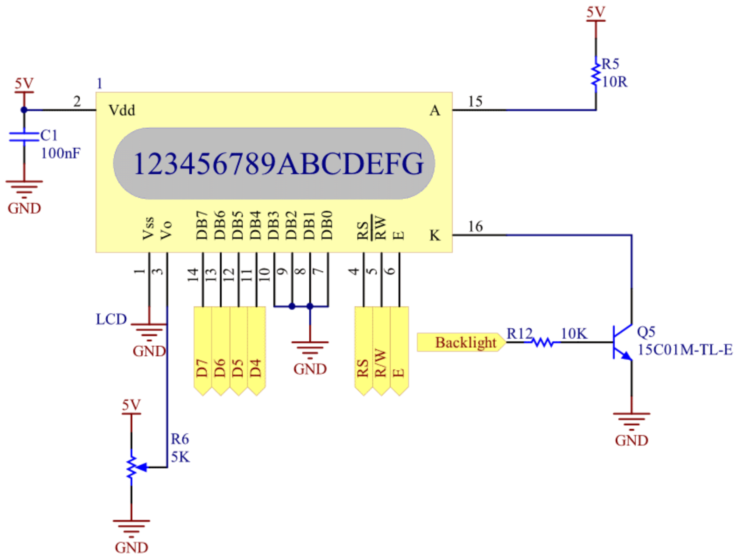


FIGURE 4.5: Schematic of the LCD connections, showing the 4-bit LCD data bus mode and backlight driver transistor.

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 them 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).

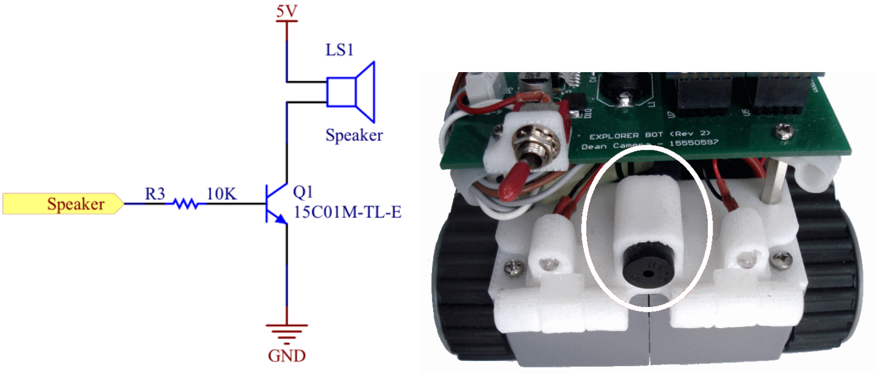


FIGURE 4.6: Schematic of the robot's piezo speaker (*left*) and photo showing the mounting in the robot prototype (*right*).

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 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, a 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.

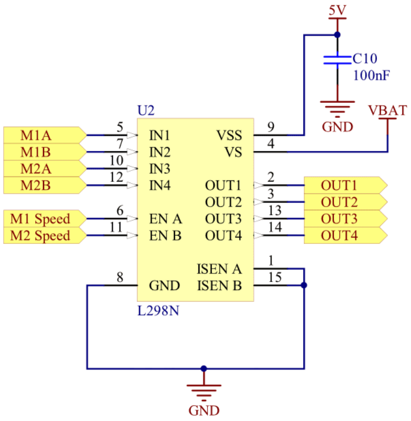


FIGURE 4.7: Schematic of the L298N H-Bridge circuit used.

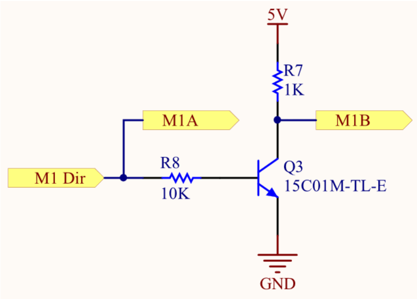


FIGURE 4.8: Schematic of one of the transistor inverters used to generate the compliment of the PWM signal used by the motor H-Bridge.

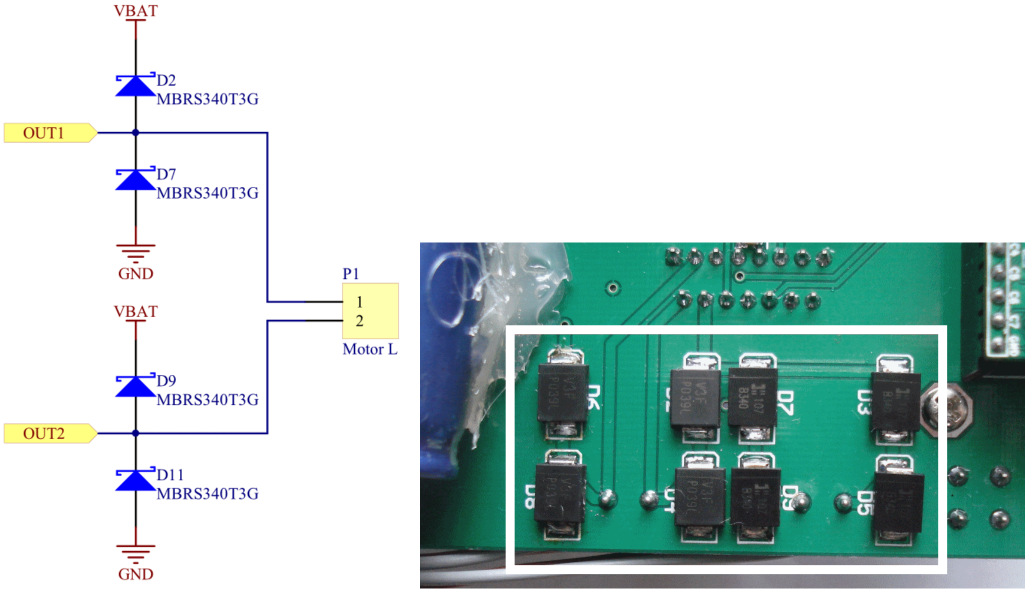


Figure 4.9: Partial schematic *(left)* and photo *(right)* of the external flyback diodes added in the second PCB revision to the robot's motors.

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^{2}C 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^{2}C bus. However, after further analysis it was discovered that the level translator would not meet the timing requirements of the I^{2}C bus, necessitating the addition of a secondary dedicated Texas Instruments PCA9306 fixed function I^{2}C bus level converter IC in the second revision of the board. As a bonus, the use of the later chip allowed the I^{2}C bus to be driven at the "Fast" I^{2}C 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).

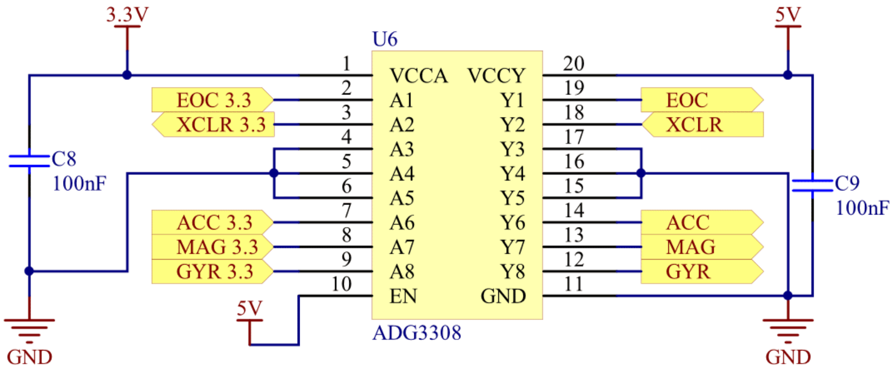


FIGURE 4.10: Schematic of the ADG3308, showing the unusual placement of the VCC-Y level active high enable pin.

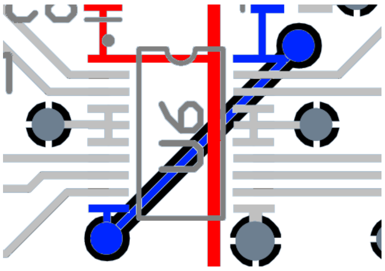


FIGURE 4.11: Routing of the ADG3308, showing the 3.3V bus (red) and 5V enable (blue) routes.

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.

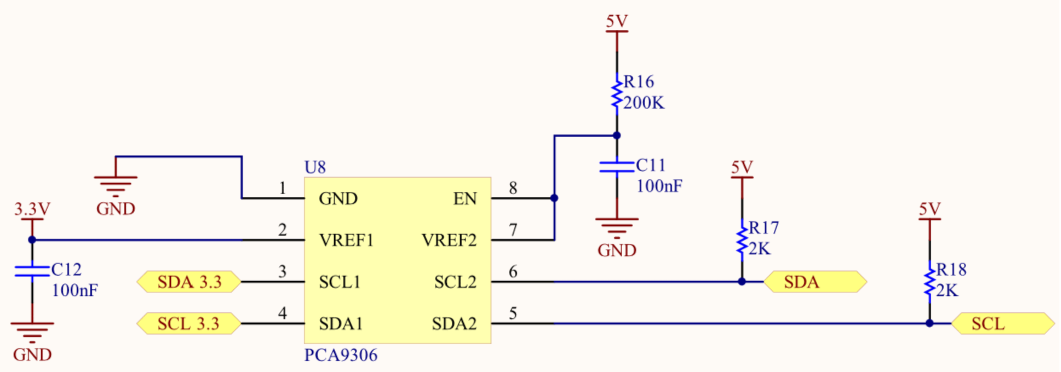


FIGURE 4.12: Schematic of the PCA9306, showing the swapped SDA and SCL pin functions.

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^{2}C 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.

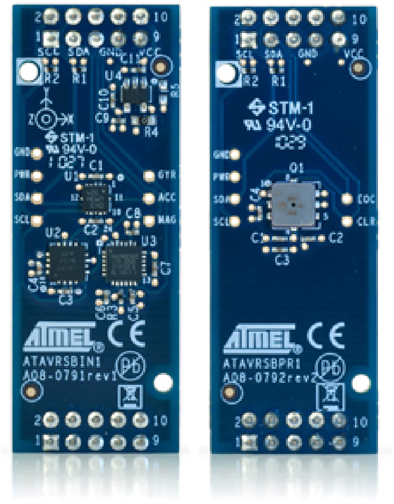


FIGURE 4.13: The Atmel *Inertial One* (left) and *Pressure One* (right) Sensor Boards (Image courtesy *Atmel Corporation*)

4.3 Final PCB Design

The final revision of the robot PCB is shown in Figure 4.14. The layout of the various hardware sections on the PCB was kept as modular as possible, and confined to the top layer where practical to ensure a good ground plane on the bottom PCB layer.

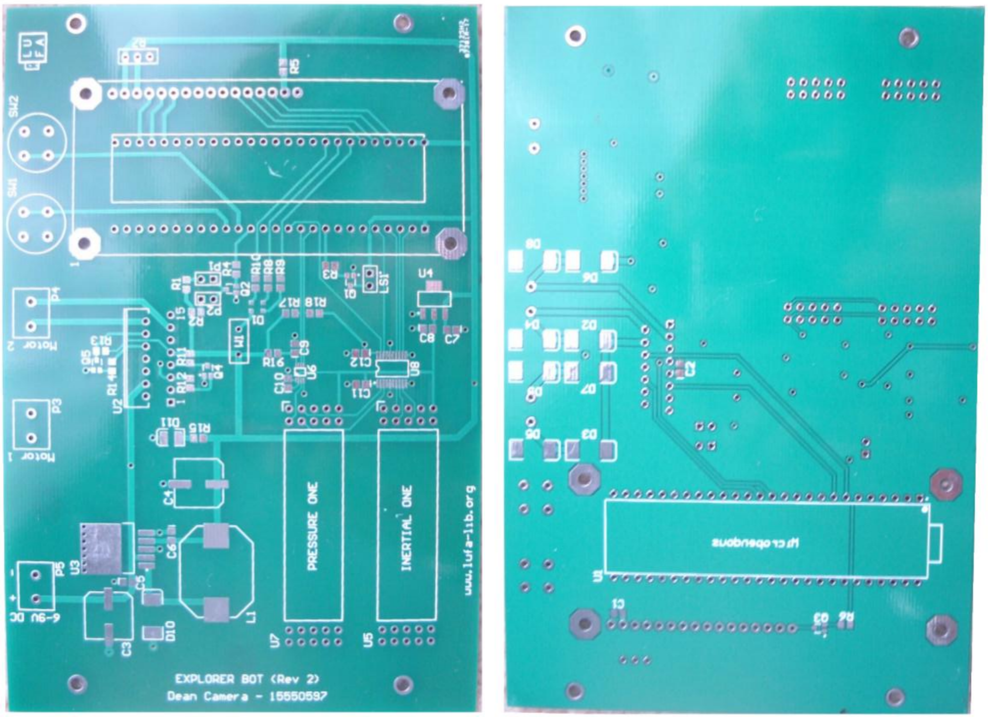


FIGURE 4.14: Photos of the final revision PCB top side (*left*) and bottom side (*right*) used in the robot prototype.

The PCB artwork is shown in Figure 4.15. Note that the ground layer fill has been removed from the bottom layer for clarity.

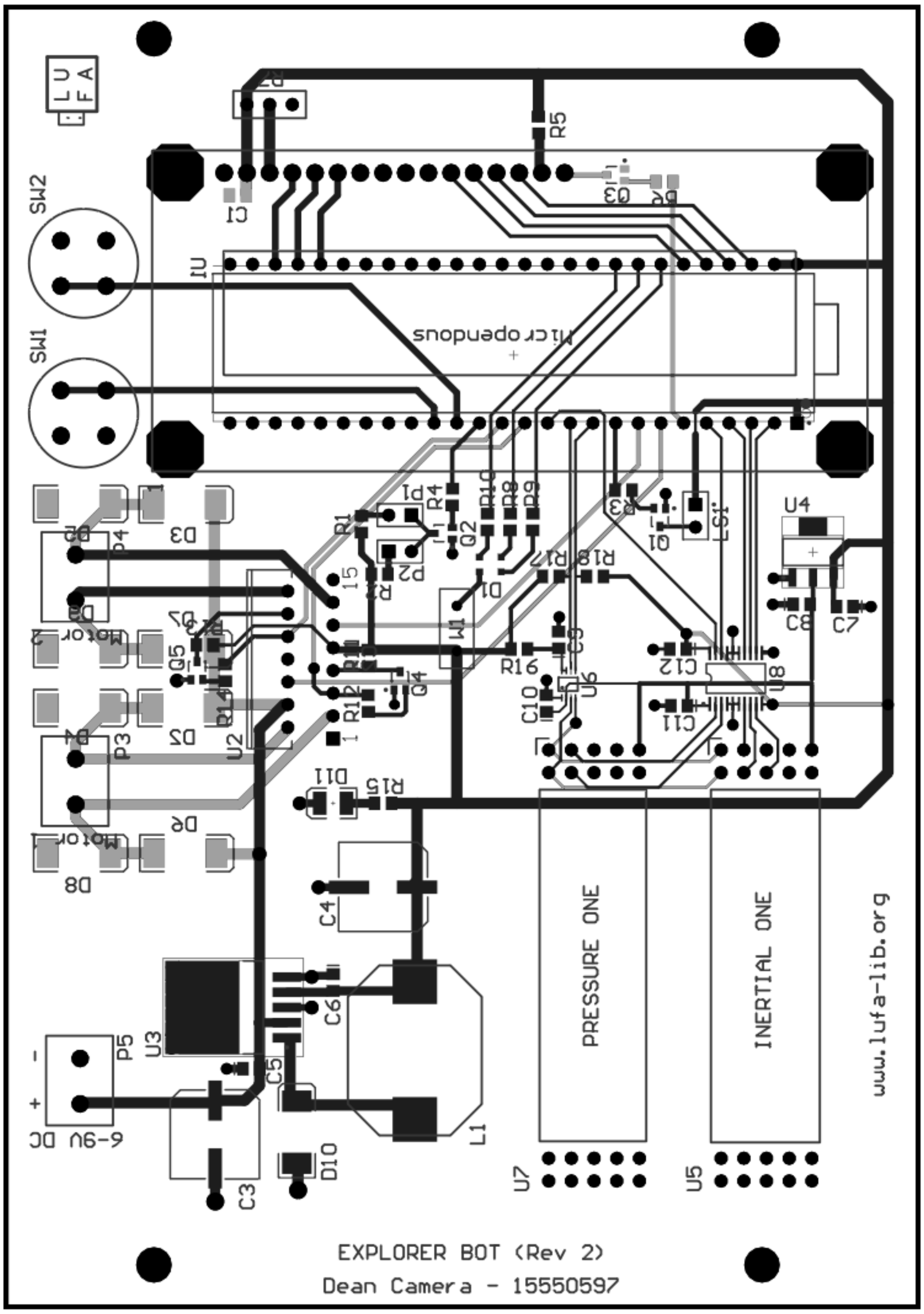


FIGURE 4.15: Artwork of the final revision PCB, showing both the top (*black*) and bottom (*gray*) layers.

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 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 shown in Figure 5.1 served as the complete firmware for the robot.

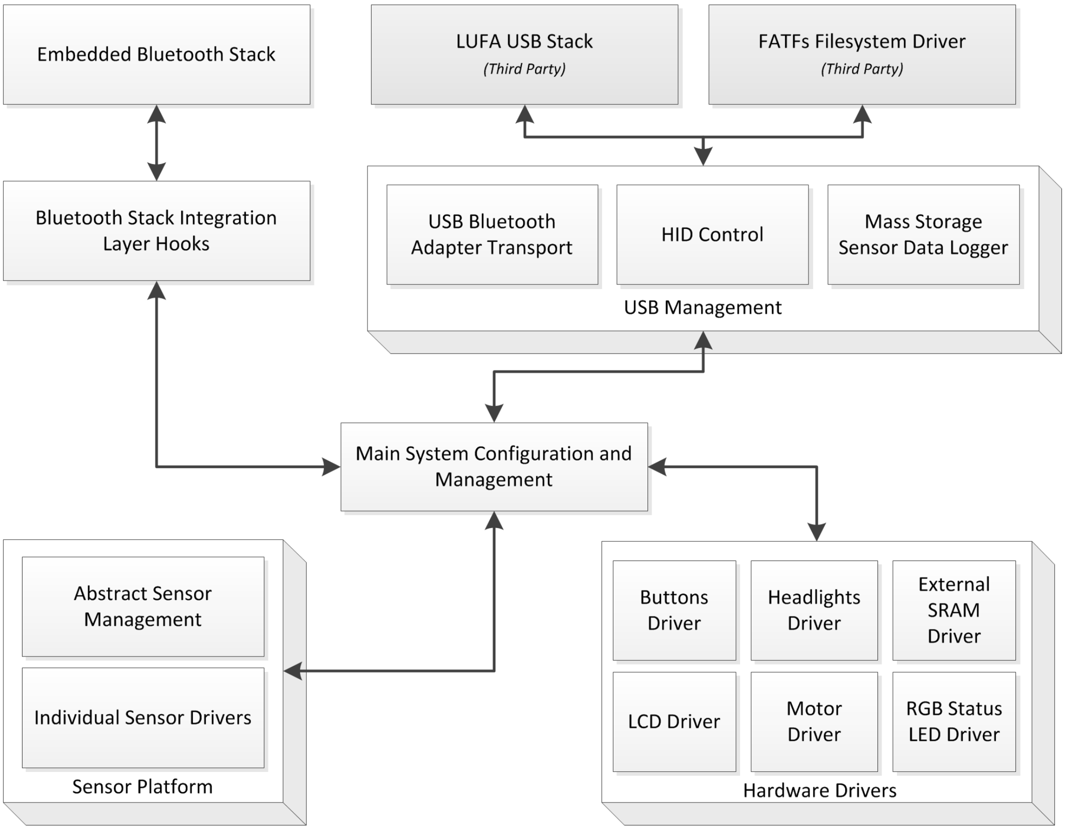


Figure 5.1: Robot Firmware Block Diagram

5.2 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.

5.2.1 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.

5.2.1.1 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.

5.2.1.2 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).

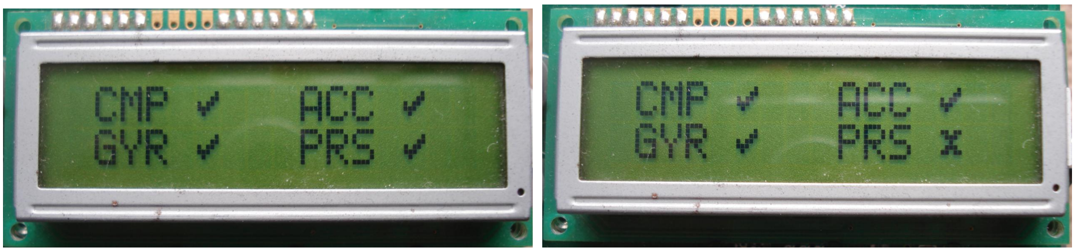


Figure 5.2: Photos of the LCD display showing successful (*left*) and failed (*right*) initialization.

5.2.1.3 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.

5.2.2 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.

5.2.2.1 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.

5.2.2.2 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.

5.2.2.3 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.

5.2.2.4 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.

5.2.2.5 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 H-Bridge and inverter hardware chosen (see Chapter 7). 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 1.9KHz at 16MHz due to the disabled prescaler and chosen *"Phase Correct"* timer mode. Values below this range moved the PWM frequency harmonics to an irritating portion the human range of hearing, while raising this frequency reduced the efficiency of the motors and reduced the motor drive torque.

5.2.2.6 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 enumeration aliases.

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.

5.2.2.7 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.

5.2.3 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.

5.2.3.1 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.

```
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.

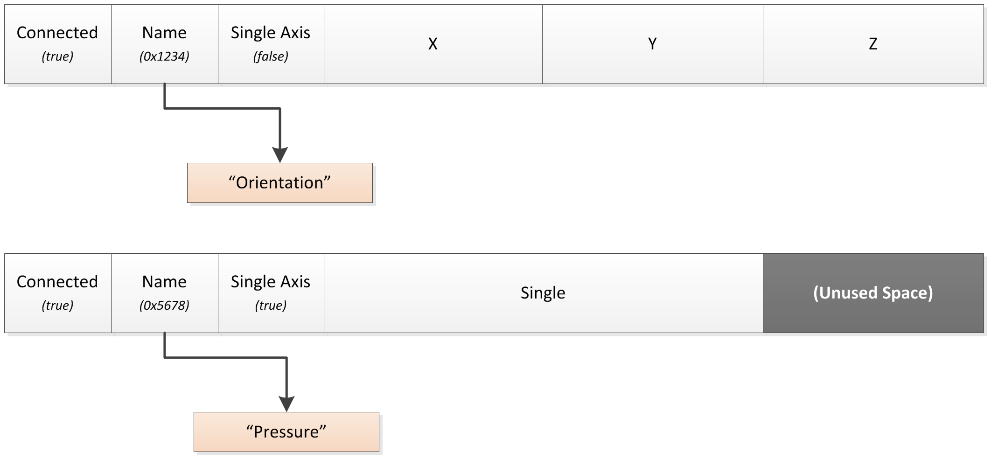


Figure 5.3: Diagram showing the layout of the Sensor Platform Entry structure in memory for triple axis (*top*) and single axis (*bottom*) sensors.

5.2.3.2 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**	**Address**	**Settings**
AK8975	Direction	0x0C	Power-on default settings
BMA150	Acceleration	0x38	25Hz bandwidth, +/-2g range, Interrupt line enabled
BMP085	Pressure	0x77	Power-on default settings
ITG3200	Orientation	0x68	100Hz at a 1KHz internal sampling rate, 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.

5.2.4 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 time frame.

5.2.4.1 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 time frame 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.

5.2.4.2 FatFS

To read and write files onto an attached Mass Storage USB device, it was necessary 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 code to connect the library to the LUFA Mass Storage class driver.

5.2.5 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 necessary 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.

5.2.5.1 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 available) 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.

5.2.5.2 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.

5.2.5.3 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 Device 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 firmware will attempt to open and process a second file on the attached storage disk named `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 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.

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

5.2.6 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: Bluetooth Stack event callback example, dispatching received packets to the integrated Bluetooth services.

In some cases, a Bluetooth enabled application may perform specific filtering on incoming connection and channel requests; an embedded device may be programmed to reject all but one specific 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

In this section, the results obtained and testing procedures used in the verification of the robot are detailed.

6.1 System Testing

An important part of any project is an appropriate level of testing, performed at both the hardware and software levels. Without adequate testing, undiagnosed latent issues such as hardware faults and software bugs may cause incorrect system operation. By performing at least some level of system testing, issues can be found and corrected as early as possible within the project's development process.

6.1.1 Board Level Testing

During and after the construction phase of the robot's hardware, a number of verification tests were made to ensure that the various hardware modules were operating as expected. This step was critical in ensuring the correctness of the hardware before the main firmware was written, to rule out hardware errors in the software verification phase.

6.1.1.1 Main Power Supply

The main power supply was tested with an oscilloscope, after the module components were installed onto the PCB. A static load of 300mA was placed on the switchmode supply output, and the voltage ripple on the supply measured with an oscilloscope (see Figure 6.1). This yielded an output ripple of approximately 80mV, well within the tolerances of the system. At the given static load, the regulated output voltage was measured at 4.98V, within the specifications of $4.8V \leq \mathrm{V}_{\mathrm{out}} \leq 5.2V$ listed in the LM2595-5 regulator datasheet.

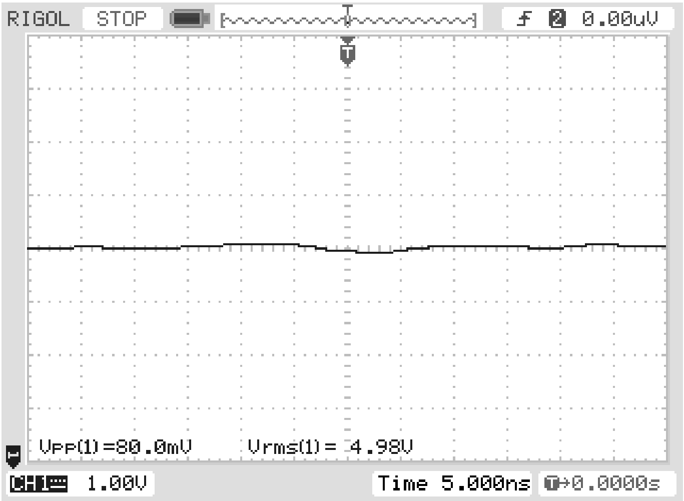


FIGURE 6.1: Oscilloscope trace of the main 5V switchmode power supply showing ripple under static load.

6.1.1.2 Sensor Power Supply

For the secondary 3.3V LDO power supply used by the board's sensor modules, an identical set of tests to those used in the verification of the main power supply were performed, using the sensor boards as the regulator output load. This yielded a ripple of 120mV, and an output voltage of 3.27V. When compared to the ADP3308 datasheet output's electrical characteristics, this was within the stated ±1.2% accuracy ($3.26V \leq \mathrm{V}_{\mathrm{out}} \leq 3.34V$).

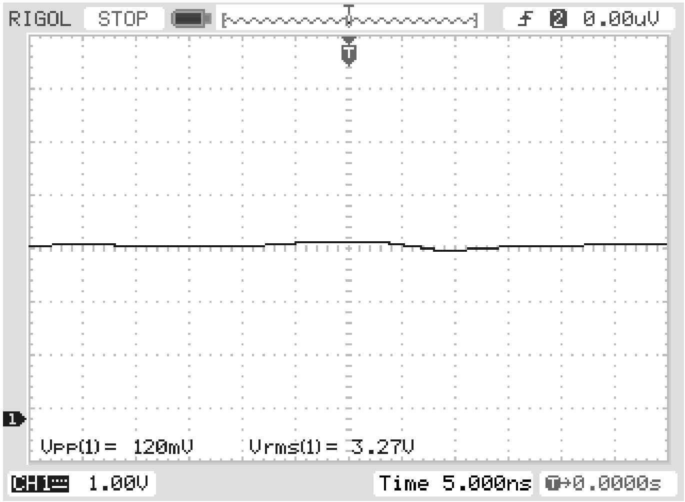


Figure 6.2: Oscilloscope trace of the sensor 3.3V LDO power supply showing ripple under static load.

6.1.1.3 Motor PWM Outputs

To test the motor outputs (and, in turn, the entire motor controller circuit) the two motor output channels were connected to an oscilloscope in sequence, and a small test wrapper written over the motor driver firmware module. Each output was switched backwards and forwards at the maximum (safe) PWM rate, and the resulting waveforms verified against the expected waveforms. Figure 6.3 shows one such waveform, showing the left motor output at full speed.

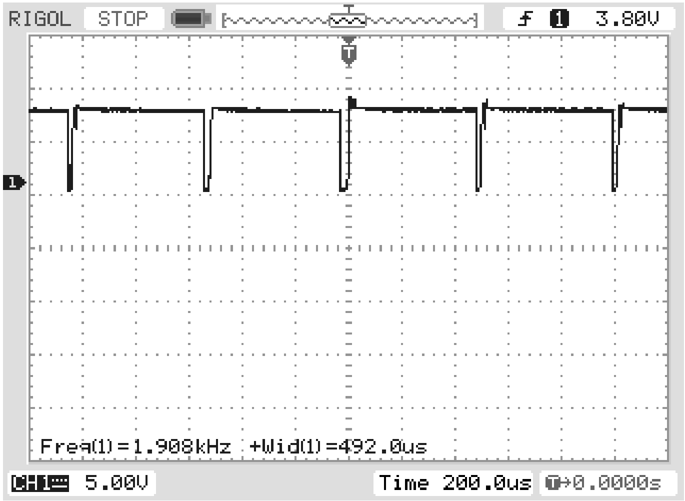


Figure 6.3: Oscilloscope trace of one motor channel PWM output while active, showing the frequency and duty cycle at full speed.

6.1.1.4 RGB Status LED

A simple test routine was written to cycle through all possible colours on the mounted RGB status LED; this demonstrated that the three GPIO channels were configured and connected correctly, and that the appropriate brightness balancing for each subcomponent in the LED package was set correctly by the chosen current limiting resistors on the board. As this is a simple but visually attractive piece of self-diagnostics, the test routine code was eventually retained in the final robot version during the robot's start-up sequence.

6.1.1.5 LCD and Backlight

To test the board's LCD functions, a routine calling various formatting commands was written and wrapped around the completed LCD driver software module. These commands—containing various cursor placements, custom character definitions and formatted strings—indicated that the display was indeed working as expected. For the LCD backlight, code was written to slowly fade in the backlight brightness from the minimum value, up to the maximum in 10ms increments. Like the RGB LED test routine, this functionality was eventually incorporated into the start-up routine of the final robot firmware.

6.1.2 Software Testing

At the conclusion of the hardware testing, a new set of software tests were performed on the robot firmware to ensure correct operation and compatibility with the appropriate specifications. This presented its own unique set of challenges, as suitable testing environments and testing procedures had to be developed.

6.1.2.1 USB Integration

Small software tests were performed for the three supported USB classes; HID, Mass Storage and Bluetooth Adapters. Doubling as a test of the hardware drivers, the HID management driver was tested against a PS3 Controller and two generic HID class USB gaming controllers, to ensure that the appropriate buttons were mapped to the robot's functions. During these tests, a brown-out failure condition was observed on the USB bus when the robot's motors were switched on and off (or changed in direction) rapidly. This issue was traced back to the original "AA" cell bank battery's insufficient instantaneous current sourcing capabilities, and was temporarily corrected in the robot prototype with a large value capacitor on the battery input terminals (see Figure 6.4) until the more powerful Lithium-Ion battery pack was substituted.

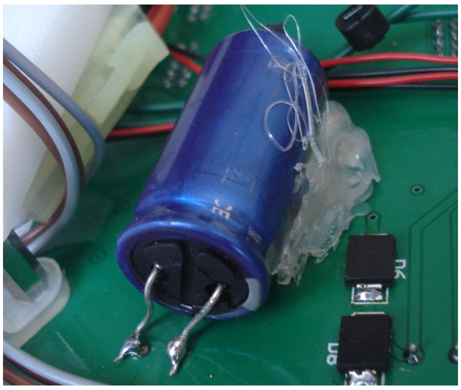


Figure 6.4: Photo of the temporary main input power capacitor, to prevent brown-out conditions from motor-induced current spikes.

The Mass Storage management module (and, by extension, the FATFs library it depends upon) was tested through a procedure of disk insertion and removal, with files written to and read from the disk filesystem using dummy data. Thus verified, the complete Mass Storage management file parsing routines were checked by printing the parsed file output to the robot's LCD display.

As the Bluetooth Adapter management code relied upon the upper Bluetooth Stack logical layers for correct operation, the functionality of the USB transceiver device management could not be examined and verified until the base layers on the Bluetooth Stack were completed. However, in this phase of the software testing, the module's ability to correctly bind to compatible Bluetooth devices in a generic manner (using several adapters from different manufacturers) was tested and found to function as expected.

6.1.2.2 Sensor Platform

The sensor platform required a significant amount of time to complete and verify; the final development time was close to one and a half weeks, much more than the expected two or three days. A number of flaws were found in the written code when it was tested on the physical robot hardware, including a bug in the LUFA USB stack's I^{2}C driver for packet read and writes. Timing and other issues in the written driver for the Bosch BMA150 accelerometer resulted first in no sensor output, followed by output on only one channel. The root cause of this problem was eventually tracked down to misleading datasheets, where the sensor would react correctly only if certain configuration registers were altered in specific manners to what should have been—according to the values given in the datasheet—the sensor's initial power-on defaults.

During the sensor platform testing phase additional small errors relating to the formatting and endianness of the retrieved values were also identified and corrected.

6.1.2.3 Bluetooth Stack

The completed Bluetooth stack was tested in an iterative manner; as additional features and layers were completed, they were verified against existing third-party Bluetooth devices. An invaluable tool used in the verification and debugging of each layer and protocol was the Linux "Bluez" Bluetooth stack's `hcidump` tool, which allowed the robot's Bluetooth stack packets to be captured in real time and decoded, indicating any issues in the implementation of one or more protocols used in the device.

Listing 6.1 shows a fragment of a HCI layer packet capture between a virtualized Linux environment and the robot, while successfully establishing a new L2CAP channel between the two devices for SDP discovery.

```
dean@dean-VirtualBox:~/Robot$ sudo hcidump l2cap

HCI sniffer - Bluetooth packet analyzer ver 2.1
device: hci0 snap_len: 1028 filter: 0x8
< L2CAP(s): Info req: type 2
> L2CAP(s): Info rsp: type 2 result 0
    Extended feature mask 0x0000
< L2CAP(s): Connect req: psm 3 scid 0x0040
> L2CAP(s): Connect rsp: dcid 0x0040 scid 0x0040 result 0 status 0
    Connection successful
< L2CAP(s): Config req: dcid 0x0040 flags 0x00 clen 4
    MTU 1013
> L2CAP(s): Config req: dcid 0x0040 flags 0x00 clen 4
    MTU 1024
< L2CAP(s): Config rsp: scid 0x0040 flags 0x00 result 0 clen 4
    MTU 1024
> L2CAP(s): Config rsp: scid 0x0040 flags 0x00 result 0 clen 0
    Success
< L2CAP(d): cid 0x0040 len 4 [psm 3]
> L2CAP(d): cid 0x0040 len 4 [psm 3]
```

Listing 6.1: HCI packet capture during a L2CAP channel configuration.

The RFCOMM layer was examined and verified at both the L2CAP layer and the RFCOMM service layer, using the Linux `rfcomm` utility to bind the robot's virtual serial port to a virtual device node named `/dev/rfcomm0`. Connecting to this device through a standard serial terminal emulator yielded the RFCOMM layer packet dump as shown in Listing 6.2, showing the successful negotiation and configuration of a new RFCOMM layer multiplexer channel.

```
dean@dean-VirtualBox:~/Robot$ sudo hcidump rfcomm

HCI sniffer - Bluetooth packet analyzer ver 2.1
device: hci0 snap_len: 1028 filter: 0x10
< RFCOMM(s): SABM: cr 1 dlci 0 pf 1 ilen 0 fcs 0x1c
> RFCOMM(s): UA: cr 1 dlci 0 pf 1 ilen 0 fcs 0xd7
< RFCOMM(s): PN CMD: cr 1 dlci 0 pf 0 ilen 10 fcs 0x70 mcc_len 8
  dlci 2 frame_type 0 credit_flow 15 pri 7 ack_timer 0
  frame_size 1008 max_retrans 0 credits 7
> RFCOMM(s): PN RSP: cr 0 dlci 0 pf 0 ilen 10 fcs 0xaa mcc_len 8
  dlci 2 frame_type 0 credit_flow 0 pri 7 ack_timer 0
  frame_size 1008 max_retrans 0 credits 7
< RFCOMM(s): SABM: cr 1 dlci 2 pf 1 ilen 0 fcs 0x59
> RFCOMM(s): UA: cr 1 dlci 2 pf 1 ilen 0 fcs 0x92
< RFCOMM(s): MSC CMD: cr 1 dlci 0 pf 0 ilen 4 fcs 0x70 mcc_len 2
  dlci 2 fc 0 rtc 1 rtr 1 ic 0 dv 1 b1 1 b2 1 b3 0 len 0
> RFCOMM(s): MSC CMD: cr 0 dlci 0 pf 0 ilen 4 fcs 0xaa mcc_len 2
  dlci 2 fc 0 rtc 1 rtr 1 ic 0 dv 1 b1 1 b2 1 b3 0 len 0
< RFCOMM(s): MSC RSP: cr 1 dlci 0 pf 0 ilen 4 fcs 0x70 mcc_len 2
  dlci 2 fc 0 rtc 1 rtr 1 ic 0 dv 1 b1 1 b2 1 b3 0 len 0
> RFCOMM(s): MSC RSP: cr 0 dlci 0 pf 0 ilen 4 fcs 0xaa mcc_len 2
  dlci 2 fc 0 rtc 1 rtr 1 ic 0 dv 1 b1 1 b2 1 b3 0 len 0
> RFCOMM(d): UIH: cr 0 dlci 2 pf 0 ilen 47 fcs 0x40
```

Listing 6.2: RFCOMM service packet capture during a multiplexer channel establishment.

Finally, the SDP layer was verified both at the L2CAP layer packet level, and at the SDP layer using the Linux tool `sdptool`. This utility was used to browse the services offered from the robot's SDP server service, to ensure that the correct services and service attributes were being sent to the requesting device correctly. A sample of this SDP output is shown in Listing 6.3.

```
dean@dean-VirtualBox:~/Robot$ sudo sdptool browse --tree

Browsing 00:11:67:BE:07:26 ...
Attribute Identifier : 0x0 - ServiceRecordHandle
  Integer : 0x10000
Attribute Identifier : 0x1 - ServiceClassIDList
  Data Sequence
    UUID128 : 0x00001101-0000-1000-8000-00805f9b-34fb
Attribute Identifier : 0x4 - ProtocolDescriptorList
  Data Sequence
    Data Sequence
      UUID128 : 0x00000100-0000-1000-8000-00805f9b-34fb
    Data Sequence
      UUID128 : 0x00000003-0000-1000-8000-00805f9b-34fb
      Channel/Port (Integer) : 0x1
Attribute Identifier : 0x5 - BrowseGroupList
  Data Sequence
    UUID128 : 0x00001002-0000-1000-8000-00805f9b-34fb
Attribute Identifier : 0x6 - LanguageBaseAttributeIDList
  Data Sequence
    Code ISO639 (Integer) : 0x454e
    Encoding (Integer) : 0x6a
    Base Offset (Integer) : 0x100
Attribute Identifier : 0x100
  Data : 57 69 72 65 6c 65 73 73 20 53 65 72 69 61 6c 20 50 6f 72 74 00
Attribute Identifier : 0x101
  Data : 57 69 72 65 6c 65 73 73 20 53 65 72 69 61 6c 20 50 6f 72 74 20 53 65 72
    76 69 63 65 00
```

Listing 6.3: Capture of the services and attributes exposed by the robot's SDP server.

6.2 Achieved Results

At the conclusion of the project time frame, much of the project goals had been achieved. The completed robot hardware was demonstrated as working with a variety of off-the-shelf consumer grade Bluetooth products, including the Playstation 3 controller, Nintendo Wii controller and a Sony Ericsson z550i mobile phone running its included *Bluetooth Remote Control* application (see Figure 6.5). The Bluetooth stack developed for the project was found to be functional enough to operate with these devices without special modifications being required to either the robot or the Bluetooth controllers.

All the robot's sensors and other hardware drivers worked correctly, with the user able to control the robot's movement, headlights and horn over both wired USB and wireless Bluetooth HID connections. Sensor data was able to be streamed remotely to a PC or other Bluetooth enabled device supporting the RFCOMM wireless serial port profile. Listing 6.4 shows a sample of the streaming sensor data captured from the robot.

A simple C# application was written to display the streaming sensor data graphically to the user in real time (see Figure 6.6). The robot firmware was capable of maintaining both the streaming sensor data connection as well as a secondary connection to a Bluetooth controller simultaneously, allowing for user control of the robot while a PC graphs the retrieved sensor values.

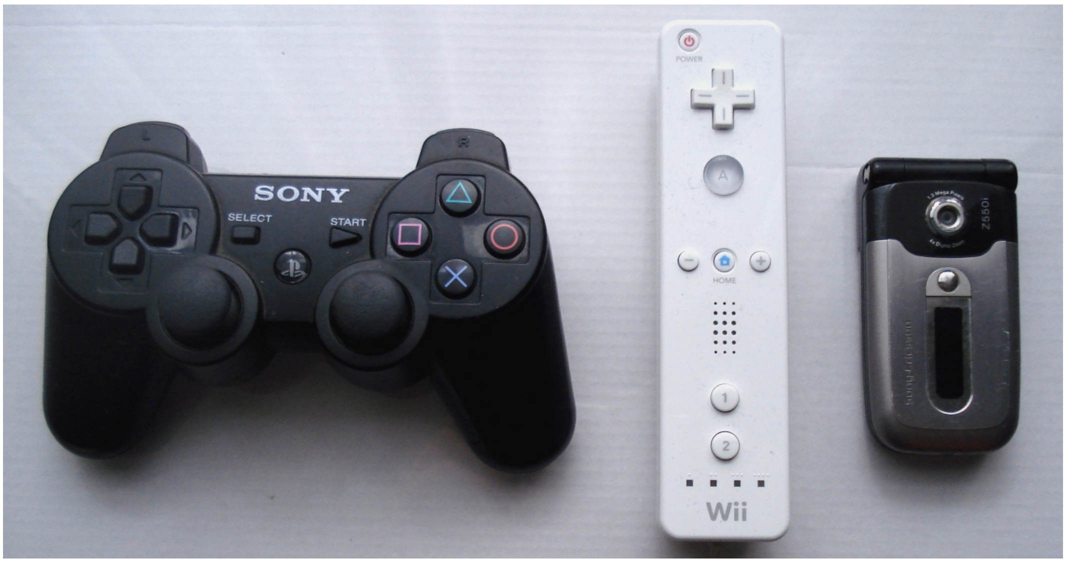
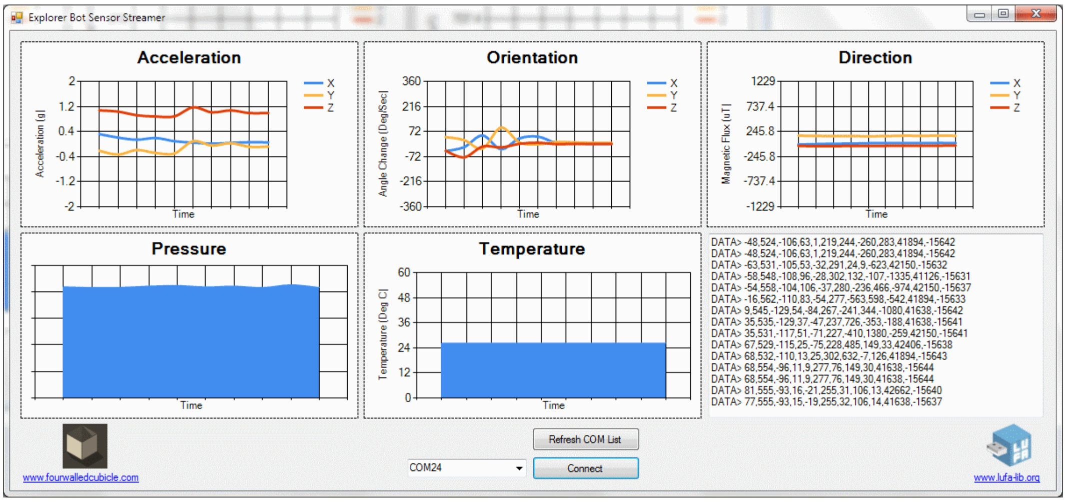


FIGURE 6.5: Tested consumer grade Bluetooth enabled devices: Playstation 3 controller *(left)*, Nintendo Wii controller *(middle)*, Sony-Ericsson z550i mobile phone *(right)*.

```
dean@dean-VirtualBox:~/Robot$ cat /dev/rfcomm0

204,516,-54,-51,19,243,40,120,39,42856,-16668
199,513,-55,-52,21,244,45,118,33,42858,-16668
206,512,-56,-51,21,243,45,117,31,42854,-16668
201,521,-57,-51,21,241,42,127,33,42856,-16672
205,511,-59,-52,22,243,34,128,37,42858,-16667
200,514,-58,-52,20,242,36,121,36,42865,-16666
204,516,-56,-54,22,242,39,121,34,42856,-16666
204,510,-58,-53,22,244,40,124,38,42865,-16673
205,517,-53,-54,21,243,41,128,40,42859,-16662
```

LISTING 6.4: Captured RFCOMM streaming sensor data.



```
DATA> -48,524,-106,63,1,219,244,-260,283,41894,-15642
DATA> -48,524,-106,63,1,219,244,-260,283,41894,-15642
DATA> -63,531,-105,53,-32,291,24,9,-623,42150,-15632
DATA> -58,548,-108,96,-28,302,132,-107,-1335,41126,-15631
DATA> -54,558,-104,106,-37,280,-236,466,-974,42150,-15637
DATA> -16,562,-110,83,-54,277,-563,598,-542,41894,-15633
DATA> 9,545,-129,54,-84,267,-241,344,-1080,41638,-15642
DATA> 35,535,-129,37,-47,237,726,-353,-188,41638,-15641
DATA> 35,531,-117,51,-71,227,-410,1380,-259,42150,-15641
DATA> 67,529,-115,25,-75,228,485,149,33,42406,-15638
DATA> 68,532,-110,13,25,302,632,-7,126,41894,-15643
DATA> 68,554,-96,11,9,277,76,149,30,41638,-15644
DATA> 68,554,-96,11,9,277,76,149,30,41638,-15644
DATA> 81,555,-93,16,-21,255,31,106,13,42662,-15640
DATA> 77,555,-93,15,-19,255,32,106,14,41638,-15637
```

FIGURE 6.6: Sensor logging application, showing data streaming wirelessly from the connected robot via a Bluetooth virtual serial port.

The completed robot PCB hardware was integrated into a pre-fabricated off-the-shelf robot platform base, and linked to the two wheel mounted DC motors. Additional parts for the rear PCB retention clip and frontal headlight and horn mounting were designed by the project's Co-Supervisor Robert Ross, and printed out on the University's 3D printer. This resulted in the compact and functional robot prototype shown in Figure 6.7.

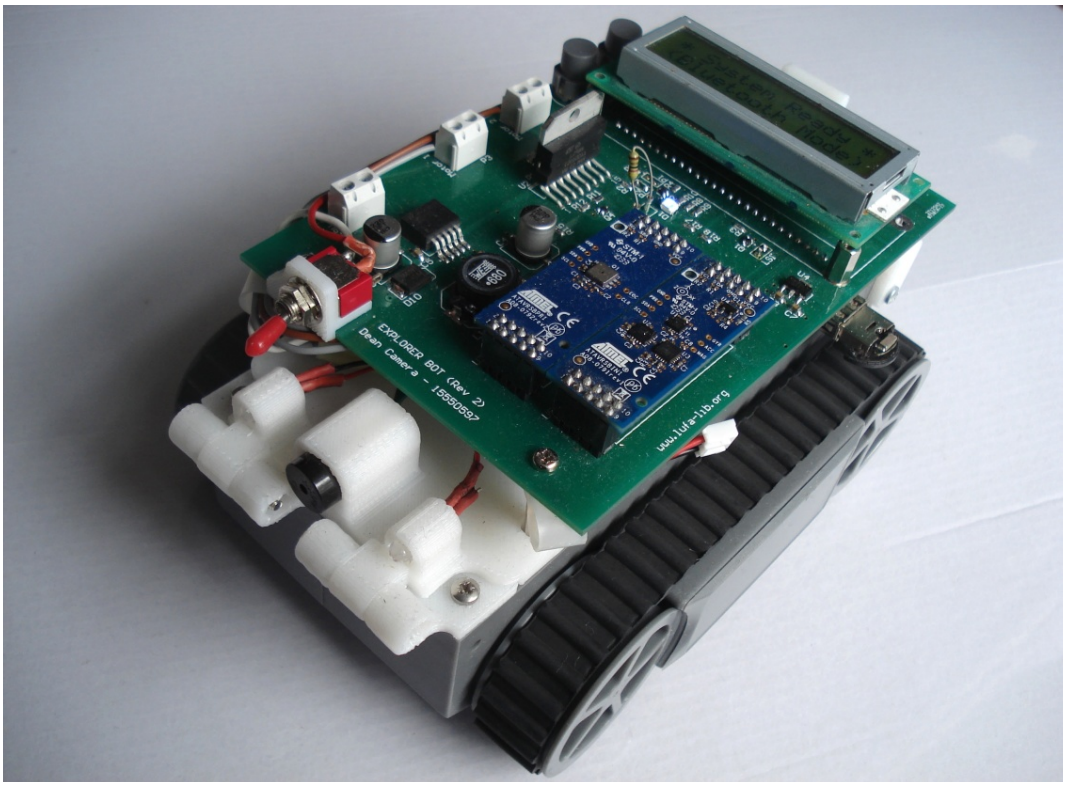


Figure 6.7: Photo of the completed *ExplorerBot* prototype robot.

Chapter 7

Project Discussion

In this chapter, the project's issues (and their solutions) are discussed in detail, as is the implications and impact of the project as a whole. The various phases of the project provided ample opportunities for problem solving, both in hardware and software as the project's development progressed.

7.1 Issues Faced

During the development of the project, as expected a number of issues were faced and overcome. The probability of unexpected issues arising during the development of any non-trivial project rises exponentially with its complexity, and this project was no exception. The main issues are outlined below, along with the solution chosen.

7.1.1 Micropendous Pinout PCB Error

After the manufacture of the first revision PCB was received some weeks after the order was placed, a problem was discovered in the board layout of the main *Micropendous* microcontroller module: the pinouts appeared to have been shifted by one around the entire module perimeter. The root issue cause was traced back to the custom Altium component created for the module; for an inexplicable reason, Altium begins part component schematic pin numbering from pin 0, instead of the conventional pin 1. This resulted in the pin numbering around the module to be rotated by one place during board layout.

Coincidentally, as the unroutable pins were marked as *No-Connect* in the robot schematics (due to their unused functions) the Altium DRC module reported no schematic or routing errors. This resulted in the error not being caught until the board manufacturing had been completed, as a cursory visual inspection of the board layout before production did not catch the subtle pin numbering problem.

While it is possible that the board pinouts could have been corrected with manual trace cutting and re-wiring, the scale of the error—some fifty two pins—and the need for other slight board adjustments forced the decision to produce a second, corrected, version of the PCB.

7.1.2 Incorrect Transistor Pinout

During the component ordering phase of the project, an issue was discovered; no NPN transistors could be found in the appropriate SOT-323 footprint that matched the pinouts given by the Altium library component used. To correct this error, it was necessary to mount the transistors flipped upside-down. This inverted orientation corrected the transistor pinouts (see Figure 7.1) so that they could be mounted to the board with the application of a small amount of extra solder to bridge the inverted pins to the PCB.

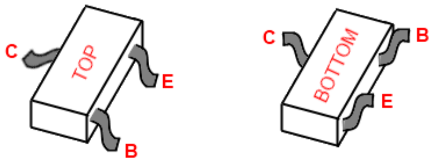


Figure 7.1: Diagram showing the pin-outs of the SOT-323 transistor when mounted normally (*left*) and flipped (*right*).

7.1.3 Motor Induced Current Spikes

During the testing phase of the robot, a failure of the USB AVR microcontroller's USB interface was observed when the motor direction was altered rapidly. Further observation narrowed this down to the switching latency of the chosen inverter transistor (30nS) and H-Bridge (approx. 2μS). During this time both halves of the H-Bridge are enabled, resulting in the possibility of a momentary short in the system battery, if the motor PWM output signal is enabled during the switching period.

To counteract this, a delay was introduced into the robot's motor driver firmware, to disable the motor output PWM during the switching time. This delay eliminated the high switching current due to the momentary short of the battery, thus preventing the microcontroller USB interface from failing.

Related to the above issue was the motor inrush current, present any time the motor was started from a stationary state. This current, present in all motors under load while the static friction in the motor is overcome, caused large current draw spikes in the main power supply, triggering a destabilization of the main switchmode regulator and thus a brownout of the main microcontroller. This issue was corrected once the main battery was switched from a series of six “AA” cell batteries to a much higher rated Lithium-Ion based battery pack.

7.1.4 PCA9306 Physical Package

Despite having some familiarity with hand-soldering somewhat-small multi-pin surface mount IC components, the package used by the PCA9306 I^{2}C level converter presented a serious challenge. This was due to its US-8 package; measuring just 2.7mm x 4.5mm, the component contains eight pins, each just .3mm wide.

Rather than risk destroying the component and waiting additional weeks for a replacement, the LaTrobe University Workshop was requested to use a hot-air gun to solder the device to the robot PCB. This ensured that the component would be correctly placed and soldered without a high risk of component damage. In future projects, relatively cheap components such as this should be ordered with one or more extras added to the quantity, so that damaged components (if any) can be replaced immediately without incurring additional delays while new components are ordered.

7.1.5 L298D Unavailability

In the first PCB revision, the motor controller design centered around the L298D H-Bridge component, which offered logic level inputs, high current/voltage driver outputs, and convenient internal flyback diodes. These flyback diodes—used to dissipate reverse EMF generated by the motor coils’ inductance as they are turned on and off—are a critical part of any such circuit involving inductive loads to prevent damage to the driver.

At the time of ordering however, it was discovered that the L298D variant was not readily available from the University’s approved component suppliers. As a second revision of the PCB was already required due to the pinout error of the Micropendous module discussed earlier, a set of external flyback diodes were added on the underside of the board, and the L298N H-Bridge IC used as a substitute. Identical in every other way to the L298D, the L298N variant does not contain internal flyback diodes.

7.1.6 Unreliable Bluetooth Packet Buffering

During development, it was found that occasionally packets sent via the Bluetooth interface were (apparently) being ignored by the receiving device — this was especially apparent during the L2CAP layer channel initialization and configuration phases. In many instances, a channel would fail to configure at all, causing the remote device connection to time out.

After further investigation, it was determined that the problem source was the lack of a reliable packet buffer within the device; if a packet was delayed within the external Bluetooth HCI controller silicon, the controller would reject new packets for transmission. Without a buffer, these rejected packets for transmission would be lost. Ideally, this could be solved by indicating to the controller that only buffer-related event packets should be returned to the host while the controller is busy, to allow for a polled busy-wait scheme to be used to determine the controller's readiness. However, there is no mechanism to delay incoming L2CAP data packets from the controller during this stage in the HCI specifications, which could potentially result in dropped received packets.

To solve this, the stack's L2CAP layer was extended to include its own internal reliable packet transmission scheme, in the form of a small event queue. As channel events were received or generated internally, the pending event would be added to a reliable queue. When a transmission is attempted, these events are only removed from the queue if the transmission was successful (see Chapter 3).

This scheme results in a reliable L2CAP layer, without the use of a large internal packet buffer within the device. Additional similar schemes could potentially be added to the higher level service layers, however this was not found to be necessary for the project's firmware.

7.1.7 PS3 Controller Compatibility

Sony's *Playstation 3* controller presented several small difficulties in making it compatible with the rest of the system. While the controller itself implements the standard HID profiles in both wired USB and wireless Bluetooth modes, additional device-specific software tweaks are necessary for compatibility.

When plugged in as a wired controller via the USB interface, a special HID Feature packet must be sent to the controller over the logical HID interface, via the USB Control endpoint. This packet, directed to the HID report ID `0xF4` and containing the special "magic" bytes `0x42 0x0C 0x00 0x00`, it is used to enable general controller reports through the regular HID data endpoints. Without this packet, the controller will not send any state change information to the host. In Bluetooth mode, the magic packet required for Bluetooth HID reporting is `0x42 0x03 0x00 0x00`.

A second packet during initialization is also required, another HID Feature packet directed at the HID Report ID `0xF5`. Prefixed with the bytes `0x01 0x00`, this request packet data is followed by the six octets of the host's Bluetooth device address, to pair the controller to the specified address. This is in contrast with the general Bluetooth pairing mechanism, which usually involves a device discovery and authentication phase over the Bluetooth link.

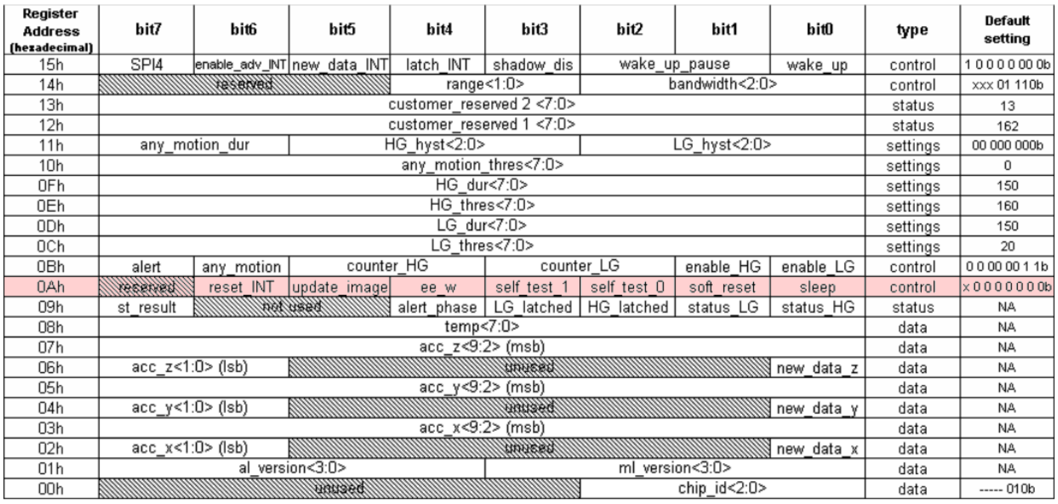
The two initialization packets for USB HID reporting and Bluetooth address pairing are shown in Listing 7.1.

```
if (Joystick_IsPS3Controller)
{
   /* Read out the latest inserted bluetooth adapter address stored in EEPROM */
   BDADDR_t TempAddress;
   eeprom_read_block(TempAddress, BluetoothAdapter_LastLocalBDADDR,
                     sizeof(BDADDR_t));

   /* Send PS3 bluetooth host pair request report to the adapter */
   uint8_t AdapterPairRequest[] = {0x01, 0x00, TempAddress[5], TempAddress[4],
                                               TempAddress[3], TempAddress[2],
                                               TempAddress[1], TempAddress[0]};
   HID_Host_SendReportByID(&Joystick_HID_Interface,
                           0xF5, HID_REPORT_ITEM_Feature,
                           AdapterPairRequest, sizeof(AdapterPairRequest));

   /* Instruct the PS3 controller to send reports via the HID data IN endpoint */
   uint8_t StartReportingRequest[] = {0x42, 0x0C, 0x00, 0x00};
   HID_Host_SendReportByID(&Joystick_HID_Interface,
                           0xF4, HID_REPORT_ITEM_Feature,
                           StartReportingRequest, sizeof(StartReportingRequest));
}
```

Listing 7.1: Playstation 3 specific initialization code.

7.1.8 Misleading Sensor Datasheets

During the development of the sensor hardware drivers, inconsistencies and misleading portions of the datasheets caused significant confusion and difficulties in attempting to establish a functional data stream from the attached sensor devices. In the case of the Bosch BMA150 accelerometer, this issue was traced back to an apparent error in the datasheet; despite indicating an initial value of `0x00` for the control register at address `0x0A` (see 7.2), the *sleep* control bit had to be manually cleared on start-up for the sensor to function correctly.



Register Address (hexadecimal)	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	type	Default setting
15h	SPI4	enable_adv_INT	new_data_INT	latch_INT	shadow_dis	wake_up_pause		wake_up	control	1 0 0 0 0 0 00 0b
14h	reserved			range<1:0>		bandwidth<2:0>			control	xxx 01 110b
13h	customer_reserved 2 <7:0>								status	13
12h	customer_reserved 1 <7:0>								status	162
11h	any_motion_dur		HG_hyst<2:0>			LG_hyst<2:0>			settings	00 000 000b
10h	any_motion_thres<7:0>								settings	0
0Fh	HG_dur<7:0>								settings	150
0Eh	HG_thres<7:0>								settings	160
0Dh	LG_dur<7:0>								settings	150
0Ch	LG_thres<7:0>								settings	20
0Bh	alert	any_motion	counter_HG		counter_LG		enable_HG	enable_LG	control	0 0 00 00 1 1b
0Ah	reserved	reset_INT	update_image	ee_w	self_test_1	self_test_0	soft_reset	sleep	control	x 0 0 0 0 0 0 0b
09h	st_result	not used		alert_phase	LG_latched	HG_latched	status_LG	status_HG	status	NA
08h	temp<7:0>								data	NA
07h	acc_z<9:2> (msb)								data	NA
06h	acc_z<1:0> (lsb)		unused					new_data_z	data	NA
05h	acc_y<9:2> (msb)								data	NA
04h	acc_y<1:0> (lsb)		unused					new_data_y	data	NA
03h	acc_x<9:2> (msb)								data	NA
02h	acc_x<1:0> (lsb)		unused					new_data_x	data	NA
01h	al_version<3:0>				ml_version<3:0>				data	NA
00h	unused					chip_id<2:0>			data	----- 010b

Figure 7.2: Table from the BMA150 sensor datasheet, showing the initial start-up values of the sensor's registers. *Note the highlighted control register's stated default value.*

7.1.9 Cheap Bluetooth Dongles

A collection of cheap USB Bluetooth dongles were purchased for the development of the robot; it was intended that these be used for testing, as well as for the final robot communications to a laptop computer running the developed sensor graphing application. However, after numerous connection failures between to similar dongles, it was discovered that several of the purchased transceiver modules contain invalid Bluetooth Device Address values.

The BDADDR values present in each Bluetooth device is intended to be used as a unique MAC identifier to distinguish between devices at the baseband level, however a number of the dongles—despite different physical appearances in some cases—contained the same "unique" address value (see Figure 7.3). When presented with the task of connecting to the same BDADDR as the local adapter, modern PC operating systems would reject the otherwise valid request.

Other oddities in the Bluetooth HCI implementation within the several of the purchased devices was observed; in one instance, the Bluetooth adapter would accept a Bluetooth HCI layer reset command, but would fail to reset the HCI layer state correctly.

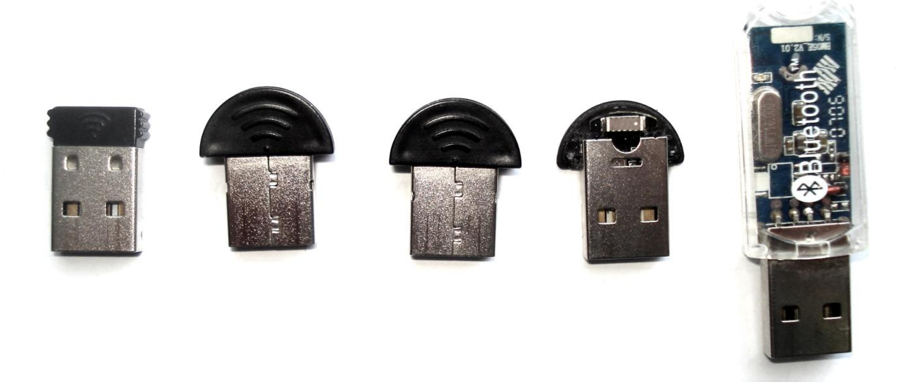


Figure 7.3: Photo of purchased Bluetooth USB adapters with incorrect Bluetooth Device Addresses.

7.2 Project Significance

This project represents a significant impact to the open source and embedded systems communities. It gives a free, open source Bluetooth stack in a new form, paving the way for a new generation of low cost Bluetooth devices. These new devices, using low powered processors previously deemed unsuitable for the task of complex Bluetooth interactions, will give users a new rich landscape of system integration and communication.

While other wireless technologies already in use will remain just as important as they are today, this project gives a new avenue of development for products that can interact directly with off-the-shelf consumer devices. Modern smart-phones, tablet computers and laptops now contain Bluetooth radios as standard features; the stack presented in this project will open up these devices to direct communication with embedded systems without the need for obscure wireless transceiver dongles. In the future, this type of technology will allow for system monitoring and configuration over a wireless link with compatible consumer devices using a rich UI, instead of wired solutions or expensive local displays and buttons.

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 practicality 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 time frames 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 time frames allotted 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 allotted project time frame.

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 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 specify 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 transport—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 environment 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 management 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.
BDADDR	**B**luetooth **D**evice **Addr**ess
CPU	**C**entral **P**rocessing **U**nit
DC	**D**irect **C**urrent
DRC	**D**esign **R**ule **C**heck
EEPROM	**E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory
EMF	**E**lectro**m**otive **F**orce
FAT	**F**ile **A**llocation **T**able
GPIO	**G**eneral **P**urpose **I**nput/**O**utput
HCI	**H**ost **C**ontroller **I**nterface
HID	**H**uman **I**nterface **D**evice
IC	**I**ntergrated **C**ircuit
$\mathbf{I^2C}$	**I**nter **I**ntegrated **C**ircuit Bus
IP	**I**ntellectual **P**roperty
JTAG	**J**oint **T**est **A**ction **G**roup
L2CAP	**L**ogical **L**ink **C**ontrol and **A**daption **P**rotocol
LCD	**L**iquid **C**rystal **D**isplay
LDO	**L**ow-**D**r**o**pout (Regulator)
LED	**L**ight **E**mitting **D**iode
LUFA	**L**ightweight **U**SB **F**ramework for **A**VRs
MAC	**M**edia **A**ccess **C**ontrol
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	**S**ervice **D**iscovery **P**rotocol
SRAM	**S**tatic **R**andom **A**ccess **M**emory
TCP/IP	**T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocol
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
UI	**U**ser **I**nterface
USB	**U**niversal **S**erial **B**us
UUID	**U**niversally **U**nique **Id**entifier

Appendix A

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 A.1 below.

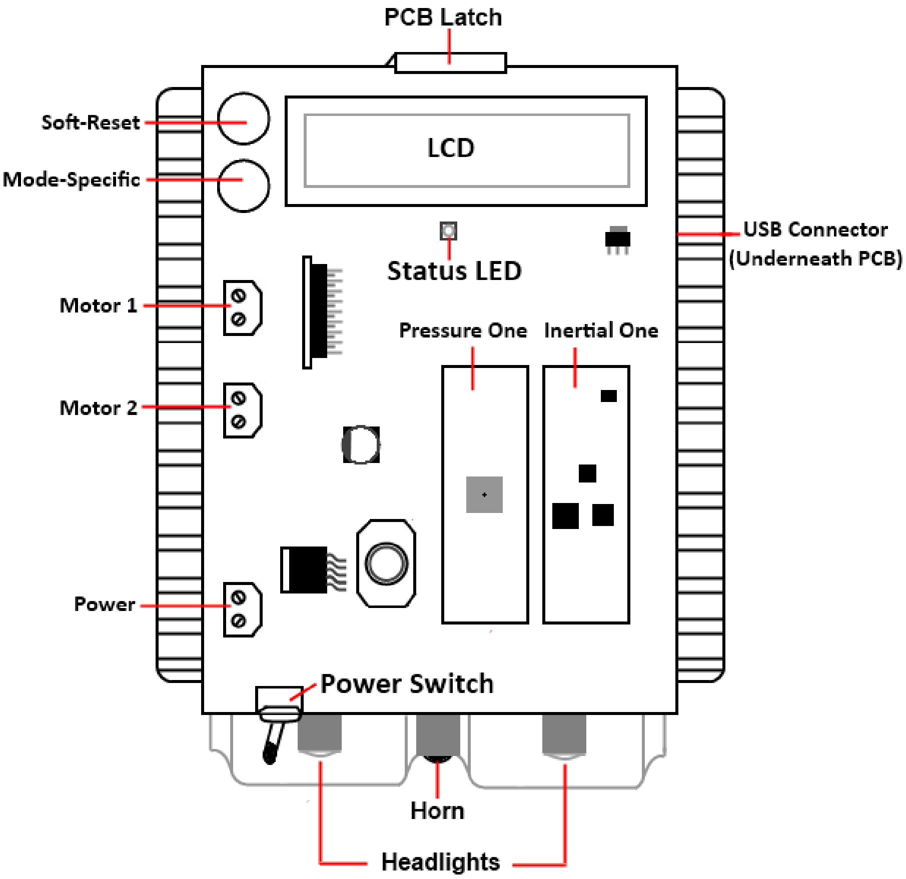


Figure A.1: Annotated diagram of the robot.

A.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 ability to supply large amount of instantaneous current, used by the motors when switched on).

A.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.

A.2.1 HID Class Devices

For robot control over a wired interface, USB HID class devices may be inserted into the robot's USB receptacle. 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 A.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 8
Horn	L2	Button 6
Headlights (Toggle)	R1	Button 7
Novelty Horn	L1	Button 5

Table A.1: Button mappings between various supported USB HID devices and the robot's 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.

A.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.

Upon insertion, the robot will attempt to read a file named `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. The format of the stored address is `XX:XX:XX:XX:XX:XX`, where each `XX` represents one of the six consecutive octets of the remote device's address in hexadecimal format. If such a file does not exist on the attached disk, a new file is created and populated with the currently stored remote device address.

Also during insertion, a second file called `SENSLOG.CSV` will be opened. If this file does not exist, a new file is created and populated with the names of the robot's sensors in Comma Separated Values (CSV) format. To begin logging of the robot's sensor data to the attached disk, press the Mode-Specific button next to the LCD.

A.2.3 Bluetooth Adapter Devices

All USB Bluetooth adapters conforming to the Bluetooth 2.1 specification for USB Bluetooth HCI data transport are supported by the robot for Bluetooth mode. When a new adapter device is inserted, the fixed local Bluetooth Address of the adapter is stored into the robot's internal non-volatile EEPROM, for later pairing with any attached Playstation 3 controllers.

To initiate a connection to a remote device, using the address previously stored in via a configuration file read from an inserted Mass Storage Device, the Mode-Specific button next to the LCD should be pressed. This will attempt a connection to the stored remote device address, with the connection result being displayed onto the LCD.

All incoming connection and channel requests to the robot will be automatically accepted, however only data carried over the robot's supported services will be processed. Connection and channel information is displayed onto the robot's LCD.

A.3 Supported Bluetooth Services

The robot currently supports three externally exposed services; Service Discovery Protocol, Human Interface Devices, and RFCOMM serial. Other services are not supported and data sent via non-supported services will be ignored by the robot firmware.

A.3.1 SDP Service

The Service Discovery Protocol service is implemented in a Server role only, and is used to expose the remaining services within the robot to external Bluetooth devices during the service discovery phase. All types of server-role requests are supported by the robot.

When requested, the robot will indicate that the device supports the RFCOMM as a server role. This information is used by PCs to automatically allocate and configure a Virtual Serial Port for the robot, used to carry streaming sensor information.

A.3.2 HID Service

The Human Interface Device service is implemented as a client only; the robot accepts reports from external devices, and processes them to determine what actions (if any) must be taken. At present, only three Bluetooth HID devices are supported:

1. Sony Playstation 3 controllers,
2. The Sony-Ericsson z550i mobile phone, and
3. Nintendo Wii controllers.

With a client implementation of the SDP service, this service could be extended to support Bluetooth HID devices in a generic manner. In the case of the first two supported devices, connections may be established to the robot from the HID device itself. For the Nintendo Wii controller, the connection must be initiated locally from the robot. The robot button mappings for these devices are shown in Table A.2 below.

Function	**PS3 Controller**	**z550i Phone**	**Wii Controller**
Left	D-Pad Left	Navigation Left	D-Pad Left
Forward	D-Pad Up	Navigation Up	D-Pad Up
Backward	D-Pad Down	Navigation Down	D-Pad Down
Right	D-Pad Right	Navigation Right	D-Pad Right
Headlights (Momentary)	R2	Right Button	Button B
Horn	L2	Left Button	Button A
Headlights (Toggle)	R1	*N/A*	Button +
Novelty Horn	L1	*N/A*	Button -

Table A.2: Button mappings between the various supported Bluetooth HID devices and the robot's functions.

A.3.3 RFCOMM Service

For remote wireless streaming of the robot's sensor data, the RFCOMM wireless Virtual Serial Port service is implemented as a server role. Connecting to the robot via this service using a PC will cause the robot to stream out the current values of the sensors in CSV format. When the virtual serial port connection is first opened, the robot will output the human readable names of each sensor before the sensor values. Only one RFCOMM session may be active at one time.

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*.

B.2 Build Dependencies

The Bluetooth stack and robot firmware 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 download 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**

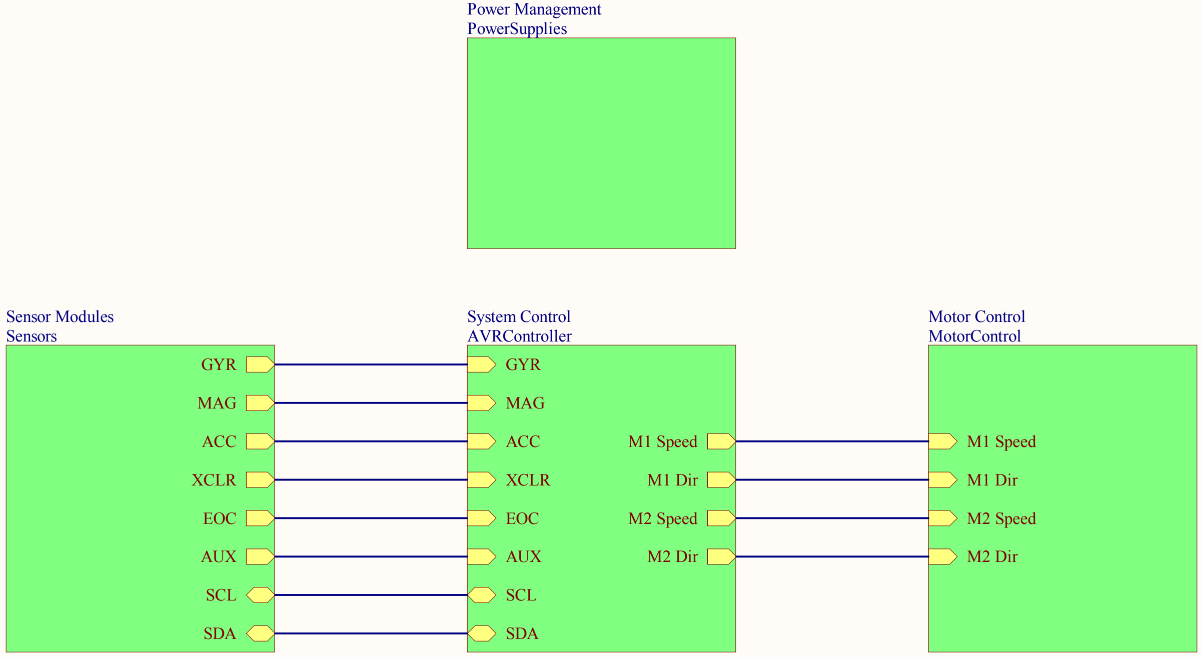
Which can be installed via the command prompt using the command `sudo apt-get install gcc-avr binutils-avr avr-libc avrdude`.

Appendix C

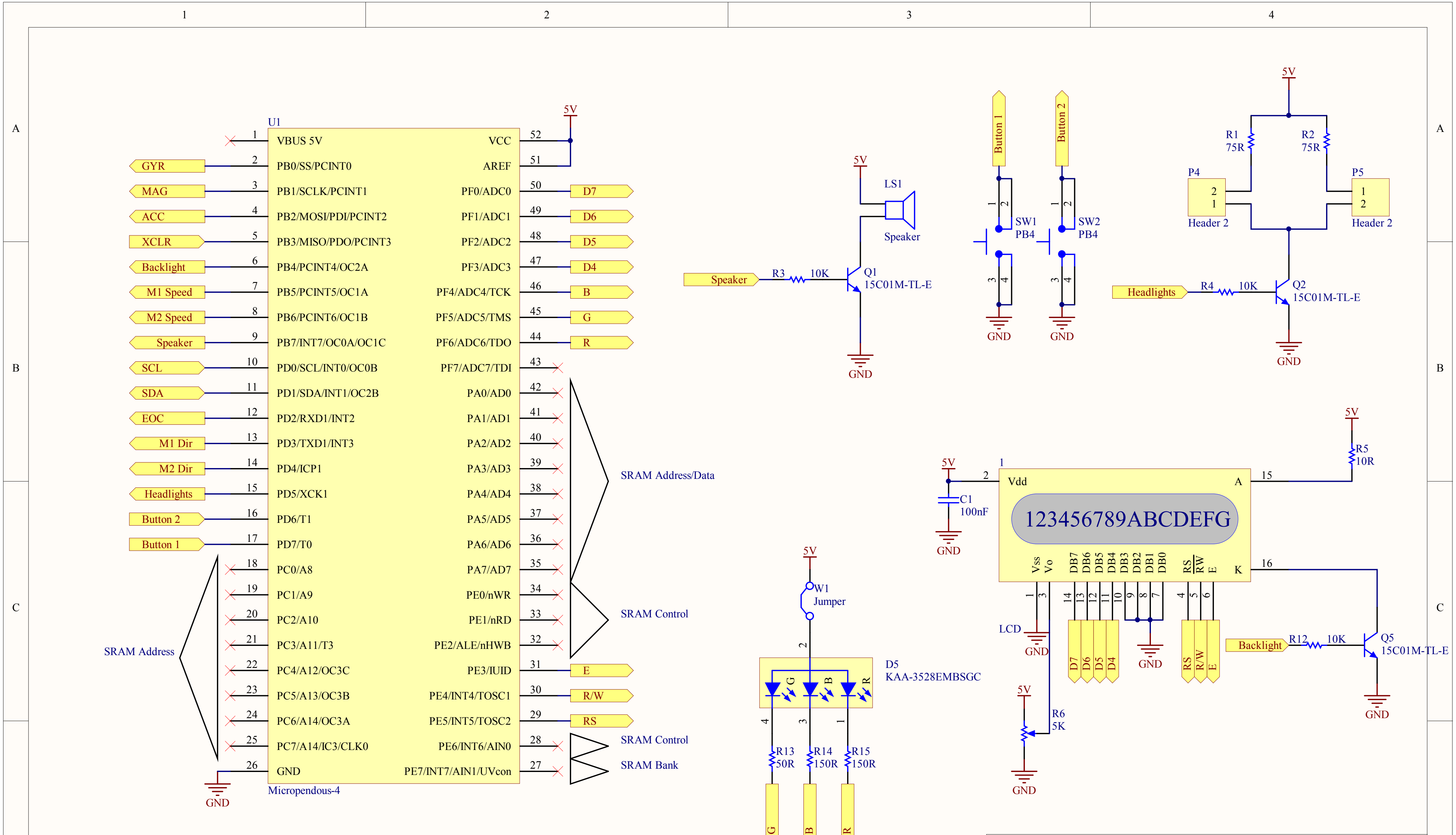
Schematics

C.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.



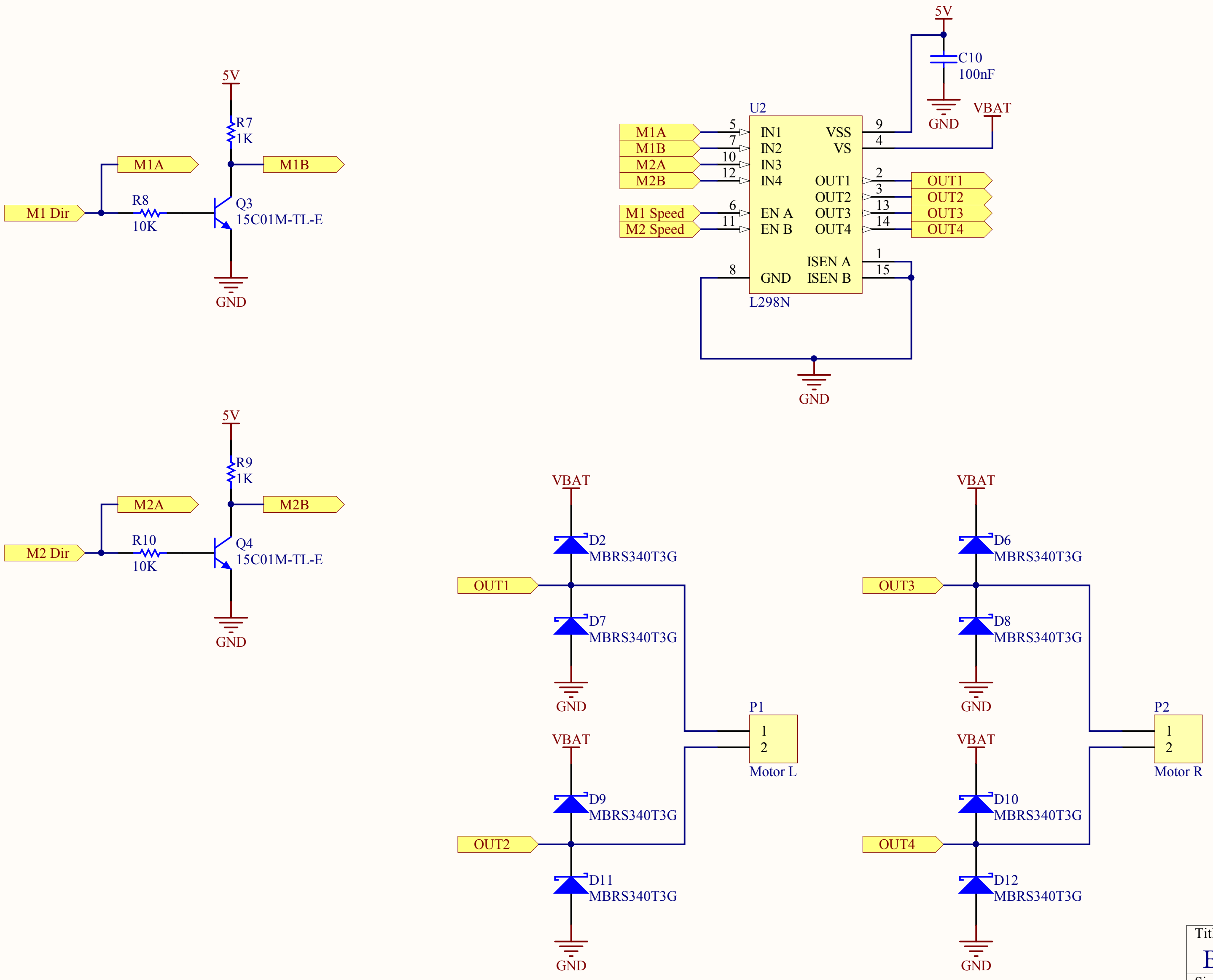
Title		
Size A4	Number	Revision
Date:	24/06/2011	Sheet of
File:	C:\Users\..\Project.SchDoc	Drawn By:



Title
Bluetooth Explorer Robot - Controller Schematic

Size	Number	Revision
A4		

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

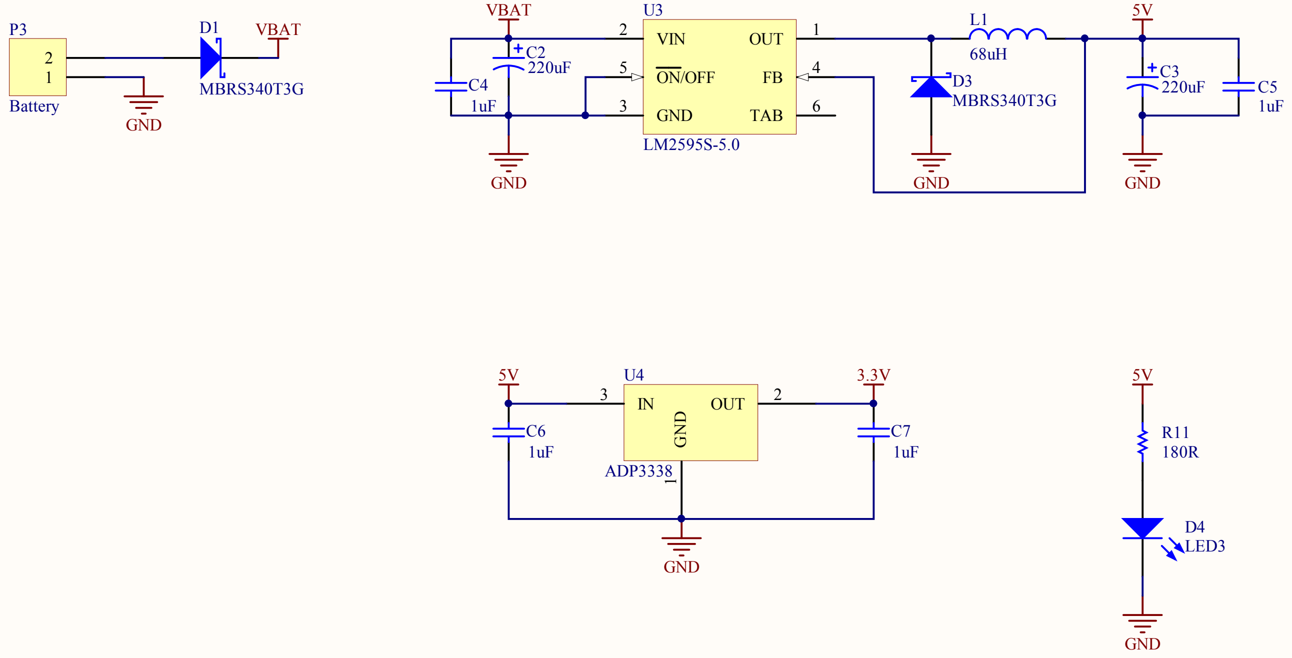


Title

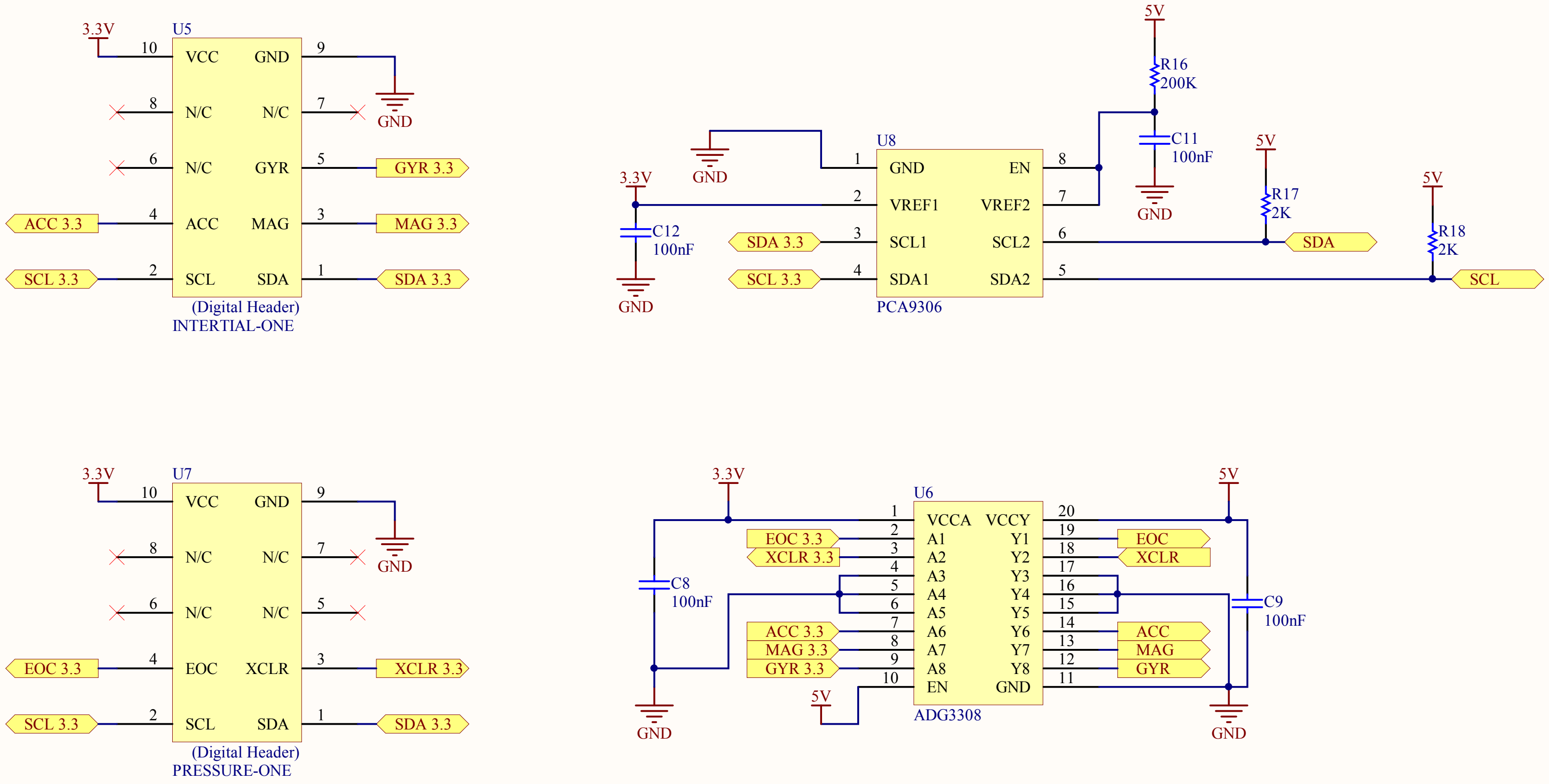
Bluetooth Explorer Robot - Motor Control Schematic

Size	Number	Revision
A4		

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



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



Title

Bluetooth Explorer Robot - Sensors Schematic

Size A4

Number

Revision

Date: 24/06/2011

Sheet of

File: C:\Users\..\Sensors.SchDoc

Drawn By:

Bibliography