PODS: PHYSICAL OBJECT DEVICES

by

Franklin E. Sorenson

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

February 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Franklin E. Sorenson

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	J. Kelly Flanagan, Committee Chairman
Date	Dan R. Olsen, Committee Member
Date	Michael A. Goodrich, Committee Member

ABSTRACT

PODS: PHYSICAL OBJECT DEVICES

Franklin E. Sorenson

Department of Computer Science

Master of Science

Computer control of electronic hardware can be a difficult challenge for programmers. Often, they waste time and misplace effort working through low-level implementation details of electronic devices, rather than using the devices to accomplish higher-level tasks and algorithms.

In this work, we introduce the Physical Object Device (POD), an extensible object-oriented environment that allows simple control of electronic devices through an object-oriented interface. Programming occurs through the use of libPOD, a simple, yet powerful, library designed to simplify the use of PODs by programmers. By building intelligence into electronic devices and by treating these devices as hardware objects, programmers are given a user interface directly targeted for them.

We detail a number of PODs that we have constructed, and demonstrate their use through several case studies. Our work shows that PODs provide a flexible, modular

design, a familiar and uniform programming environment, and rapid design of complex electronic systems.

ACKNOWLEDGMENTS

I'd like to thank my wonderful wife Elizabeth for her support and encouragement. Without her patience and strong faith in me, I could never have finished. I'm grateful to my son Daniel for being a lot of fun, and not breaking any unfixable portions of my thesis.

I thank my advisor, Kelly Flanagan for his guidance, suggestions, and patient encouragement. His moral support helped push me through to the end, and his financial support made possible a Thesis involving numerous electronic parts. Thanks go to Darren Hart and Vernon Mauery for ideas and the encouragement of them nipping at my heels, and to the rest of the Performance Evaluation Laboratory for everything they've done over the years. I'm grateful to the rest of my committee for their reviews and suggestions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Applications for Computer Control of Hardware

Devices which perform actions or measure some element of the environment around them are used heavily in environmental monitoring systems and robotics. Common robotic devices include motorized wheels, gripping hands, and light or distance sensors. When combined with a central controller, these devices allow robots and other systems to interact with the world around them.

Examples of how these devices are combined are the Mars Pathfinder and Sojourner Rover combination. These systems contain several cameras and lasers for distance measurement, several temperature sensors, wind socks to measure wind characteristics, and pressure sensors and accelerometers to gather useful information during their descent [1]. Robots such as Sojourner are specially designed by engineers for the functions they will perform [2].

While some robotic systems have specific constraints which require customization, many others may be purchased pre-built. Additional sensors and actuators may be purchased and attached later. This allows a robot to be used quickly in higher-level applications, including machine learning and control algorithms, or for measuring environmental conditions.

Sensors and actuators are also used in a variety of systems such as security systems, computer room monitoring devices, and as environmental monitors. Common sensors include motion detectors and temperature sensors, and actuators often include alarms, sirens, warning lights, and motors. These devices may be purchased or engineered, but are difficult to combine into a single system.

In situations like Sojourner, NASA had the time, money, and expertise to design and build an entire robot system from the ground up. The constraints of the project also required a custom solution. However, researchers usually do not want to spend a large amount of time on a project to collect and assemble all the pieces they need before they can begin performing their higher-level studies. However, buying pre-built products can be costly, and the product may not have the necessary functionality.

End users often want computer systems with the ability to control these various electronic devices, and they would benefit greatly from a middle-of-the-road solution that allows device control at the right level. They are generally concerned about the time and money involved, and usually have expertise in fields other than electronics. In these cases, a pre-built solution can be very costly, and reverse-engineering current products or designing something from the ground up requires more time, work, and in-depth electronic knowledge.

1.2 Problem to Solve

Common electronic devices are often used to measure some element of their environment, such as temperature, or perform some action, such as sounding an alarm or turning on a motor. Interfacing a computer with electronic devices can be difficult, since the interfaces of many electronic devices are not specifically designed to interface with computers.

Connecting to these devices can require considerable electronics knowledge. Often, the people interested in communicating with electronic devices are artists, industrial designers, or technical researchers. Their main skills usually lie in areas other than electronics. They generally want to concentrate on building and using their entire project, rather than the individual devices.

To illustrate the difficulty in connecting electronic devices, we use an electronic thermometer as an example. Some devices use very high-level interfaces, such as a stand-alone digital wall thermometer or a large weather station. The wall thermometer may be a self-contained device that displays the current temperature on its own screen. Devices like this are generally not intended for direct communication with a computer. The weather station may attach to a computer, however it may require connection through a special computer card, and there may be little information about how it communicates with the software provided by the manufacturer. These interfaces are intended for people who want stand-alone devices, and are not meant to be incorporated into a user's project.

Eletronic devices are also designed with low-level interfaces, with the intention of connecting them to other electronic devices by engineers. This type of device often requires large amounts of additional logic, and assumes that the user has an in-depth knowledge of circuit design and construction. Example thermometers with low-level interfaces include devices with 8 digital outputs, an SPI bus, and a single analog output. Some off-the-shelf devices and construction kits help, though they rarely remove the barriers posed by the electronics. Because these devices aim to serve people familiar with electrical components and microprocessor communications, the electronics experience required remains a challenge to their use.

These are just a few low-level and high-level examples to illustrate the difficulty of finding user interfaces that are appropriate for a variety of people's needs. Even for groups possessing the electronics knowledge to create, attach, and program electronic devices, that may not be where they want to focus their efforts. Marvin Minsky, co-founder of the MIT Artificial Intelligence Laboratory, recently said, "The worst fad has been these stupid little robots. . . Graduate students are wasting 3 years of their lives soldering and repairing robots, instead of making them smart. It's really shocking." [3]. As Minsky states, many interfaces require more low-level work than desired.

These low-level interfaces ignore a large group of people, namely those with the goal of connecting these devices to computers, and using a programming language to add software functionality. Making some of these devices work can mean learning electronic design techniques, soldering, programming, and finding device drivers that provide a programming interface. While some of these challenges can be overcome, as a whole they can become significant barriers to the use of electronic devices.

We believe that by building intelligence into the devices themselves, and by designing the electronics with a common hardware interface and software library, programmers can easily use electronics directly in programs they write. By providing an object-oriented programming framework that encapsulates the electronic devices and abstracts the low-level details, programmers can focus on the tasks that the device will be performing, rather than the electronics. A number of projects have attempted to provide this interface, and we discuss some of these in the next section.

1.3 Previous Work

Some work has been done in an attempt to simplify or standardize communications between a controller and various devices. There are several kits which contain various motors, gears, sensors, and connecting wires. Many of these kits are hobbyist kits, or are designed to connect to small microcontrollers. Some are designed for people specifically researching robotics. While these approaches can be useful in some instances, many people find that they do not meet their needs because they are still difficult to control using their computer.

Examples of hobby kits include Lego Mindstorms. These are Lego kits containing a computer module (the RCX) and a number of Technics parts [4]. The RCX contains a microcontroller which can be programmed from a PC, but can only communicate directly with a limited number of attached devices (3 inputs and 3 outputs). Despite these limitations, Mindstorms have become popular, and many people have designed sensors and other systems for use with them [5].

Additional microcontroller-based solutions include the OOPic (Object-Oriented Programmable Integrated Circuit) [6], a programmable PIC microprocessor that can function as a number of different objects, including hardware circuits (digital I/O, counters, etc), processing, variable storage, and system objects. These objects simplify connection of the OOPic with hardware devices, and allow several OOPics to communicate together. In an NSF Summer Undergraduate Fellowship, students designed and built a small robot from various parts [7]. The pieces contained microcontrollers and were connected with a serial communication bus. One part acts as a master, and gives commands to the other modules.

Several modular systems include Robotix [8] and some work done at Xerox PARC. Robotix is “a motorized modular building system” that includes motors, grippers, jaws, and winches. These parts are controlled directly through a hand-held remote module, so the system is not programmable, and does not apply higher-level ideas. Much of the modular robotic work from Xerox PARC [9] has focused on constructing robots from a number of similar parts, and programming them to work together as a single robot.

Phidgets [10, 11, 12] compare physical interfaces to widgets used in graphical user interfaces. Phidgets are designed to abstract the physical devices, and hide the implementation and construction details. They expose their functionality through an API, and include a simulation mode where the programming interface may be used without an attached device. While Phidgets are an attempt to simplify programming, their interface requires the use of Visual Basic, COM, and ActiveX in Windows, making them complex devices to work with.

While these projects have solved some of the problems associated with controlling and using electronic devices, they fall short of providing an interface that is both easy to program, and requires little electronics knowledge.

Our solution relies on the use of programmable microcontrollers to simplify the interface to complicated electronic devices. These Physical Object Devices, or PODs, have an object-oriented interface that makes them easy to use. We feel that this solution is more complete than any of the previous work, and provides a useful user interface for programmers. The remainder of this work is outlined as follows: Chapter 2 discusses our solution. Chapter 3 describes the general characteristics of PODs, and details the PODs that have been implemented and how they operate. Chapter 4 presents several case studies and demonstrates how PODs can be used. Chapter 5 presents our conclusions, and future work. Some additional details can be found in the Appendices.

Chapter 2

Our Solution: PODs

2.1 PODs are a better approach

We have created a new sort of device, which we call a Physical Object Device (POD), and we have defined a framework under which PODs operate. A POD is an electronic device that is controlled in an object-oriented manner from a computer. When a POD is designed and constructed, many of the technical electronic details are abstracted away, and the POD user is left with a simple and familiar interface. PODs are generally small devices. An example POD is depicted in in Figure 2.1.

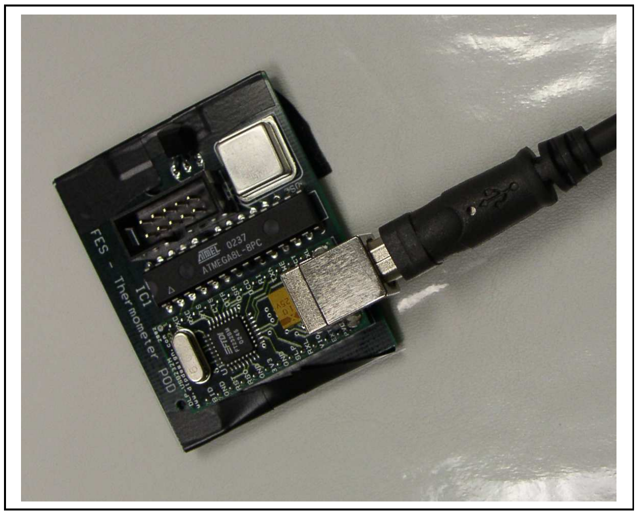


Figure 2.1: Picture of a POD.

Intelligence is built into each POD through the use of small microcontrollers, which allow a POD to be largely self contained. Each POD has a small onboard EEPROM that allows the user to assign the POD a name, and certain other settings may be retained as appropriate to the particular device's need. PODs may be used in programs through an object-oriented C++ library called libPOD. Abstracting the electronic details and utilizing a simple programming library allows users to integrate these smart devices into new or existing programs without requiring an in-depth knowledge of electronics and other POD details.

Each POD is physically connected to the host computer over a USB link, as shown in Figure 2.2. USB is commonly found on computers, can provide power to most PODs, and allows the connection of as many as 127 devices at one time. This, in conjunction with an object-oriented software library, makes PODs easy to attach to nearly any computer, and simplifies the creation of programs to access them.

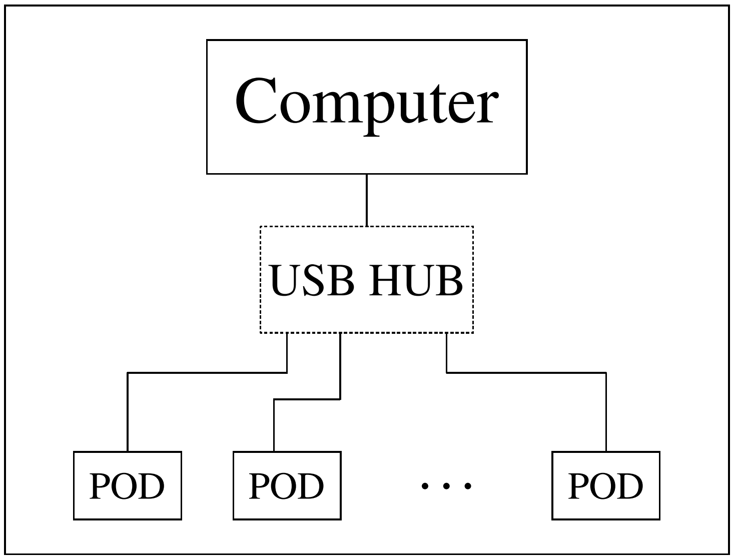


Figure 2.2: Connecting PODs to a computer. If the computer has enough USB ports, the USB Hub may not be necessary.

The POD microcontrollers, when used with the library routines, give PODs a well-defined API that user programs running on the host computer can leverage to easily communicate with a POD. Each POD responds individually, and contains all the methods necessary to function independently. Additionally, PODs are designed to act as objects, so they may take advantage of some of the properties of object oriented design.

We have designed the POD hardware abstraction architecture to be flexible and object-oriented, and have focused this work on demonstrating the usefulness of PODs, as well as how they can interact with each other and the programming environment. Though we designed PODs with object-oriented principles in mind, we have only implemented some of the ideas, and have left the application of some of the more advanced principles for other researchers. We have discussed some of these in more detail in Section 5.2.

2.2 Programming with PODs using libPOD

We have implemented libPOD as a C++ library. This makes programming simpler by abstracting away the low-level details of communicating with the PODs, and helps to facilitate the connection of multiple PODs to a single computer. This also allows user programs to communicate with all connected PODs by invoking simple methods. Currently, libPOD has been compiled under Linux and the Windows Cygwin environment, and PODs have been used in both environments. This gives the POD user great flexibility.

Using libPOD, each POD is represented as an object through the POD base class, and individual POD types inherit from that class. libPOD provides two main objects, POD and PODEnv, as described in Section 2.3.

To program using libPOD, the POD user needs only to connect the POD to the computer, compile their source code, and link in libPOD. Figure 2.3 contains a brief example program that demonstrates how a single Thermometer POD might be used to monitor the temperature in a server room.

```
PODEnv *MyPODs;
ThermometerPOD *Thermometer;
int ReturnValue;
float Temperature;

Thermometer = (ThermometerPOD *)
 (MyPODs->FindPODbyType(POD_TYPE_THERMOMETER));

while (1)
{
   ReturnValue = Thermometer->GetTemperature(&Temperature);
   if (Temperature > HIGH_TEMPERATURE)
   {  Send warning email to the system admin  }
   else {  Everything is okay  }
}
```

Figure 2.3: Pseudo-code to demonstrate a simple POD system.

After the libPOD environment has been initialized, PODs can be queried and controlled through simple method invocations. This speeds the development of user code, and helps PODs be more integrated into user projects. libPOD manages POD connections, and takes care of connecting to and communicating with the PODs, whether it be a single POD, or many PODs.

Since PODs communicate using low-level serial commands with the controlling computer, libPOD implements the lower-level communication with the POD microcontroller. It sends the requests for information, processes the responses, transmits configuration and commands, and listens for asynchronous alerts. To the user, libPOD provides a set of wrapper methods for these functions, and can also respond to some general commands as well. These general methods serve to initialize the POD environment, and allow the user to query and search for connected PODs.

2.3 General Functionality of libPOD

In addition to the actions performed behind the scenes, libPOD also provides the user with a number of more visible methods. These methods function by helping to identify and control the PODs connected. The POD environment is initialized during the constructor for the PODEnv object, a class that contains and manages all connected PODs. This involves opening the various POD devices and determining what is connected. Each POD may then be addressed individually as a POD object, or through methods accessible through the PODEnv object.

PODEnv may be initialized with a number of options, such as POD_ENV_OPTION_TEXT_STATUS, which displays a text-based status readout when detecting attached PODs, POD_ENV_OPTION_GUI_STATUS, which displays a GTK2 progress bar when detecting PODs, or POD_OPTION_DONT_INIT, which tells it not to search for attached PODs. RefreshPODs() may still be called to update the list of PODs, or RefreshPOD() may be used to update each POD individually. The PODEnv object also provides the following helpful methods:

- int PODEnv::GetNumberOfPODs();
  This will return the number of valid PODs currently connected.

- int PODEnv::GetAllPODs(POD *ReturnPODList)
  This returns an array (in ReturnPODList) of all the PODs connected. ReturnPODList should be an array MAXPODS long.

- POD *PODEnv::FindPODbyType(int POD_TYPE, int Number)
  This is used to find a POD of type POD_TYPE. The Number parameter may be ommitted, in which case it defaults to 0 (requesting the first POD of the specified type). If no matching POD can be found, NULL will be returned.

- POD *PODEnv::FindPODbyName(int POD_TYPE, char *PODName)
  This is used to search for a POD by the user-defined name, and may be used with POD::SetName and POD::GetName.

- POD *PODEnv::GetPOD(int PODNumber)
  Returns a POD by number.

- int PODEnv::RefreshPODs()
  This method rescans the system for PODs that have been added, removed, or changed.

- int PODEnv::RefreshPOD(int PODNumber)
  This method rescans the system to see if device #PODNumber has been added, removed, or changed.

Similar to the way that most object-oriented programming environments require software objects to have a default constructor, each POD is required to respond to a few basic commands, as shown here:

- int POD::GetType()
  This method returns the PODType (as an int) of the POD.

- int POD::GetType(char *PODTypeString)
  This method returns a string describing the PODType (PODTypeString should be long enough to hold the description–POD_SIZE_TYPENAME should be long enough).

- int POD::GetVersion(char *VersionString)
  This method returns a string describing the version of the code on the microcontroller.

- int POD::GetURL(char *ReturnURL)
  This returns the URL describing more information about the POD (URL should be long enough to hold the URL–POD_SIZE_URL should be long enough).

- int POD::SetName(char *String)
  This can be used to set a user-defined name for the POD, and can be useful when multiple PODs of the same type are connected (String may be up to POD_SIZE_NAME bytes long).

- int POD::GetName(char *String)
  This method is used to return the user-defined name for the POD (as set with SetName). String should be at least POD_SIZE_NAME bytes long.

- int POD::IsValid()
  This method will respond with a nonzero value if the PORT was successfully opened and the POD responded correctly.

- int POD::TestConnection()
  This method will check the POD to make sure it is still responding.

- int POD::Test()
  This is a virtual method that will test or demonstrate the POD in some way.

Each of these methods requests information from the associated POD, or gathers information from the general POD environment. The return value for each of these methods is an integer result, with positive values indicating success and negative values indicating errors. The details of the methods defined for each specific POD type are given in the next chapter.

Chapter 3

Implementing PODs

3.1 General POD Information

We have focused this research on demonstrating the overall usefulness of PODs, and how the POD environment simplifies the interaction between computers and electronic devices. Section 3.1.1 discusses the hardware common to all of our PODs, while Section 3.1.2 outlines the software associated with each POD. These details help to show that PODs define a useful and convenient programming environment. We have also developed a number of POD Development methods and tools (mentioned briefly in Section 3.3).

3.1.1 Hardware Information

We have created each POD as a 4-layer printed circuit board (PCB), giving the PODs a clean, uniform look. To aid in their development, the PODs all utilize the same microcontroller and USB-Serial device. Since they are intended as prototypes, PODs are not minimized with respect to size or component count, but future work could shrink and minimize PODs and code size.

The PODs have been designed around the Atmel ATMega8L microcontroller. The Atmel AVR microcontrollers are 8-bit processors based on a RISC architecture. The ATMega8L comes in both surface-mount and DIP packages, and has adequate code and memory sizes for many projects. It also has a relatively small package size, and supports a wide variety of on-chip features, making it a practical and convenient chip to work with.

On-chip features include 8 KB of Flash for program storage, 1 KB of SRAM for data storage, 512 bytes of EEPROM for persistant storage, and 23 general purpose I/O ports. It supports external interrupts and Pulse Width Modulation (PWM), and contains a

6 channel Analog to Digital Converter (ADC), as well as an internal RC clock and voltage reference for the ADC.

To allow programming the POD microcontrollers after construction, we have included a programming header. Also, while the ATMega8L contains an internal RC clock, we have built the PODs using an external oscillator running a 3.6864 MHz. We chose this oscillator frequency because it produces accurate UART clock rates up to approximately 230 Kbps, giving us a large degree of freedom when choosing interface speeds. The oscillator is also more stable than the internal RC clock. We have included additional information on this design decision in Section 3.2.1.

To convert between USB and the microcontroller serial port, we utilized the DLP Design DLP-USB232M USB-Serial UART Interface Module [13], which uses the FTDI FT232BM USB-UART chip [14], and comes in a convenient, breadboardable DIP package. It is capable of providing power through the USB bus, and the onboard level converter interfaces with both 3.3V and 5V logic. Several example PODs are shown in Figure 3.1.

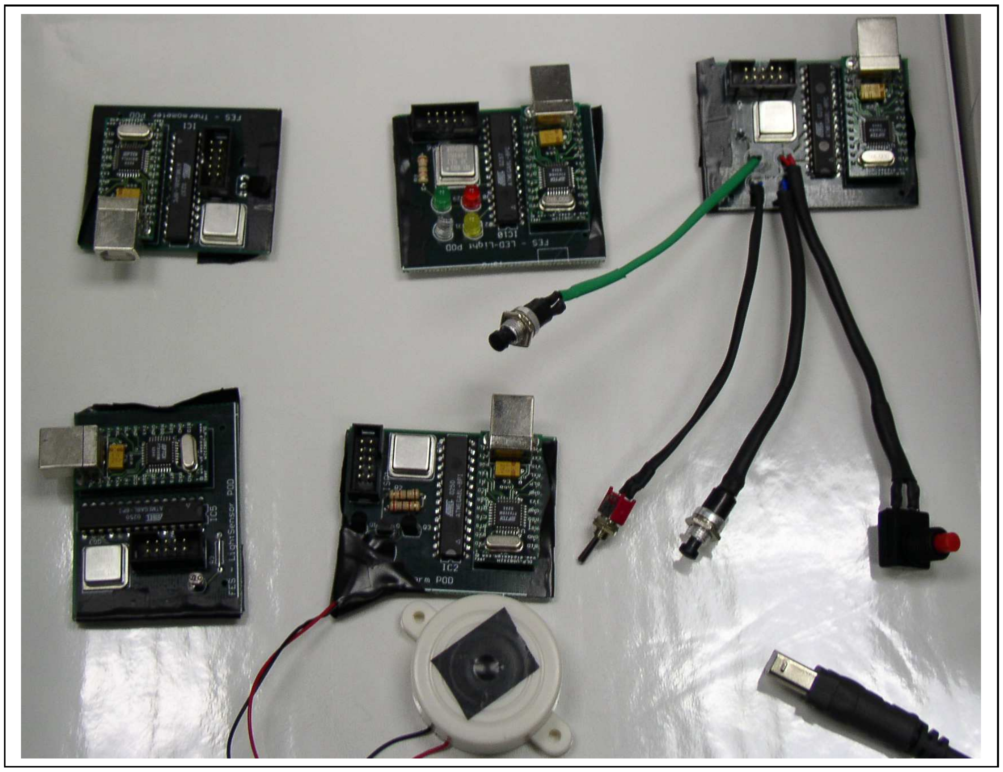


Figure 3.1: Picture of several PODs. Top row, left to right: Thermometer POD, Light Emitting Diode (LED) POD, Buttons POD. Bottom row, left to right: Light Sensor POD, Alarm POD, USB connector.

3.1.2 Software Information

The code on each POD was developed in C, and has been compiled using gcc with the avr target. The microcontroller on each POD communicates by using its built-in UART, and the attached computer receives this communication through a device driver that interprets the connection as a simple serial device. Under Linux, these devices begin with /dev/ttyUSB0, and under Windows, they are COM ports (starting with COM5, for example). libPOD is responsible for opening the appropriate device file and doing the low-level communication.

To assist in the development of the PODs themselves, a number of small header files can be included for additional functionality. This simplifies the microcontroller programming, and makes the POD-specific code simpler and easier to implement and read. Details of these can be found in Appendix B. In the following sections, we discuss some details associated with each POD used in this work.

3.2 POD Implementations

During this project, we constructed a number of PODs, and have defined a number of devices that could potentially be made into PODs. In this section, we discuss details associated with each of the PODs constructed for this work. We briefly discuss some reasons that the particular devices may be desirable and some difficulties an end user might face when designing and building such a device without using PODs. We describe the basic functionality of the POD, give its methods, and discuss implementation details such as electronic parts used during the POD's construction. For the Thermometer POD, we describe additional details related to its construction, including a block diagram and the circuit board layout. For other PODs, some of these details are given in Appendix B.

We have built the following PODs:

- Thermometer POD
  Measures temperature

- Alarm POD
  Sounds an audible alarm

- Motor Control POD
  Provides control of motors

- Light Sensor POD
  Measures light intensity

- Light Emitting Diode (LED) POD
  Turns on various LEDs to provide light or for use as status indicators

- Power Sensor POD
  Detects the power status of 110V AC power

- Sonar Distance Ranger POD
  Measures distance

- Compass POD
  Tells what direction the POD is pointing

- Buttons POD
  Contains various buttons used as input

3.2.1 Thermometer POD

Measuring temperature with and without PODs

There are a number of situations in which people may want to measure temperature from a computer program. This can be very useful when monitoring the temperature in a server room or other temperature-controlled environments, when tracking outside temperature such as in a weather station, or when adding the current temperature to a local web-page. Anyone desiring to measure temperature with a computer has a number of options, few of which are practical. Most require a detailed knowlege of electronic design and construction.

As discussed in Section 1.2, commercially available electronic thermometers are often either stand-alone devices, not intended to be incorporated in user-built projects, or low-level electronic devices that require considerable knowledge regarding circuit construction techniques. When an electronic thermometer is constructed as a Thermometer POD, the technical implementation details become hidden, leaving the user with a clean API that exposes only the necessary functions. This simple software interface represents a significant improvement for the user, as they only require a USB port and some programming knowledge.

Thermometer POD Interface and Methods

The Thermometer POD is a small thermometer capable of reporting temperature in either Fahrenheit or Celsius, as requested by a user program. The POD has a valid temperature range from -40°F to +300°F. General information about the Thermometer POD is shown in Table 3.1.

Temperature Range	-40°F - +300°F
Accuracy	±1°F
Precision	0.25°F
Power	From USB

Table 3.1: Thermometer POD General Data

In addition to the general POD methods defined in Section 2.3, the Thermometer POD adds the following methods:

- int ThermometerPOD::GetTemperature(float *Temperature)
  This method returns the current temperature in Temperature. The return value from GetTemperature indicates whether libPOD had any problems retrieving the temperature from the actual POD. Most of the POD methods return an `int` to communicate error status.

- int ThermometerPOD::SetUnits(char Units)
  This method sets the current temperature units to either F or C.

- int ThermometerPOD::GetUnits()
  This method reports the current temperature units (F or C).

Thermometer POD Implementation

In our project, we built the Thermometer POD using the National Semiconductor LM34DZ [15]. The LM34DZ is a small, precision temperature sensor containing an integrated-circuit with an output voltage linearly proportional to the temperature. It requires very little power, and can provide accurate temperatures over the range -40°F to +300°F.

During implementation of the Thermometer POD, we discovered that the manufacturer's datasheet for the USB-to-Serial device was incorrect in several places. Also, while the microcontroller used contains an on-chip RC (Resistance/Capacitance) clock, its susceptiblity to temperature variations made communication with the POD impossible at anything other than room temperature. Since this was an unreasonable limitation (particularly for a thermometer device), we created all PODs with external oscillators. This represents another detail that is handled by the POD designer, since it allows all PODs to have the same environmental requirements.

A block diagram of the Thermometer POD is illustrated in Figure 3.2. The method invocations are translated in the libPOD code to commands passed across the USB to the POD device. The USB interface on the POD receives these commands and sends them via a serial connection to the microcontroller. The microcontroller-based routines are then executed and the appropriate actions are taken. In the case of the Thermometer POD, this involves reading from the Analog to Digital Converter, changing this reading into the appropriate units, and returning this value. We include an example microcontroller routine to demonstrate the GetTemperature method in Figure 3.3.

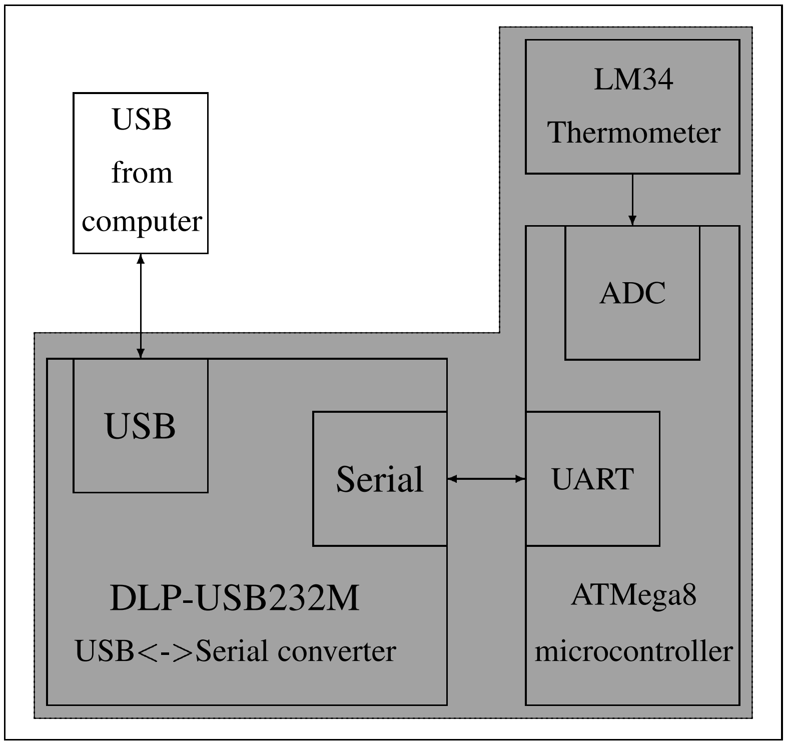


Figure 3.2: Block diagram of the Thermometer POD. The shaded area demonstrates which components make up the POD.

```
void Process_Command_GetTemp()
{
   /* Temperature reading from GetReading (100ths F) */
   unsigned long TemperatureReading;
   /* CurrentTemperature in appropriate units */
   unsigned long CurrentTemperature;

   TemperatureReading = ADC_GetReading(); /* Read the ADC */
   /* convert to the appropriate units */
   CurrentTemperature =
      TemperatureConvert(TemperatureReading, DisplayUnits);
   UART\_TransmitByte(POD\_RESPONSE\_ACK);
   UART\_TransmitWord(CurrentTemperature / 100); /* degrees */
   UART\_TransmitByte(CurrentTemperature \% 100); /* 1/100ths */
}
```

Figure 3.3: Sample code from the Thermometer POD microcontroller. This demonstrates the GetTemperature method.

As previously mentioned, this POD was fabricated using a 4-layer Printed Circuit Board (PCB) process. This PCB is illustrated in Figure 3.4. A production POD could be made much smaller using surface mount technology, removing the programming header, and removing free space from the board. Again, the intent of this work is to demonstrate the concepts and not provide production sized PODs.

The next few sections give the API and overview of the other PODs that have been created. We have included the details of each in Appendix B.

3.2.2 Alarm POD

Sounding an audible alarm

Audible alarms can be a useful method of conveying information. An alarm on a robot could be a warning that the robot is close by or moving, or could be used to signal when the robot has reached its goal. An alarm could also be used as a warning such as when the power goes out, when a system overheats, or when a disk has failed. The Alarm POD is designed to help abstract a number of functions that might be called when sounding an alarm from a computer.

Alarm POD Interface and Methods

The Alarm POD is an audible alarm that can be used to alert people of problems or convey more detailed information. It can sound alarms at 3 different pitches/volumes, allowing the user to choose how intrusive the sound is. The Alarm POD consists of a small board and a

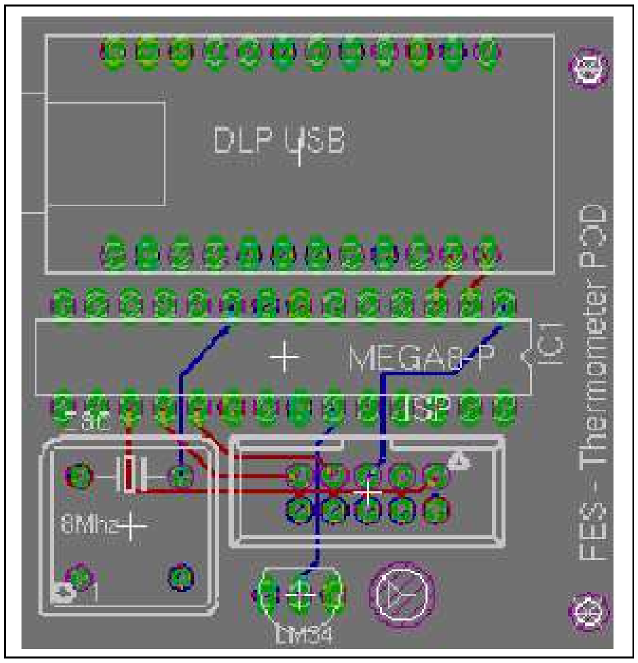


Figure 3.4: PCB artwork for the Thermometer POD.

piezo beeper that sounds at the various pitches. The POD is also capable of playing Morse Code through the beeper in order to convey more detailed information. The Alarm POD extends the base POD class with the following methods:

- int AlarmPOD::SoundAlarm(float Duration)
  This turns on the Alarm POD once for a specified Duration. The Duration parameter represents the number of seconds (or fractions of seconds) that the alarm will sound, with a maximum of 2.5 seconds.

- int AlarmPOD::PlayMorse(char *Data)
  This sends a string of characters to the Alarm POD to be played as Morse Code.

- int AlarmPOD::SetMorseSpeed(unsigned char Speed)

- int AlarmPOD::SetMorseTone(unsigned char Tone)

- int AlarmPOD::Quiet()
  This silences the POD and cancels any currently sounding alarms.

Alarm POD Implementation

We built our Alarm POD using a small piezoelectric buzzer that sounds when power is applied. Since the sound can be very loud and piercing, the power to the buzzer can be switched through several resistors, enabling the POD user to choose the tone that the alarm should sound. Details about the POD can be found in Appendix B.

3.2.3 Motor Control POD

Controlling a Motor

Motors are used in a wide variety of applications, including mobile robots, robot arms, conveyor belts, and fans. They come in both AC and DC varieties, are generally high-power devices, and can be difficult to control by computer. The Motor Control POD helps abstract away many of the detailes required to control a motor.

Motor Control POD Interface and Methods

The Motor Control POD is a DC motor controller that can run the motor either forward or backward, and provides variable speed control. For this work, Motor Control PODs have been attached to two motors from a small remote controlled car, since this provides a convenient base on which to build a robot. The POD can also be used with a number of other motors.

Functions	Forward and backward motor control, with variable speed control.
Power	Logic is powered from USB. The motor requires a separate power supply.

Table 3.2: Motor Control POD General Data

In addition to the base POD class methods, the Motor Control POD adds the following methods:

- int MotorControlPOD::SetSpeed(unsigned char Speed)
  This method sets the relative speed of the motor (from 0-stop to 255-full speed).

- int MotorControlPOD::SetDirection(int Direction)
  This method sets the motor direction. The Direction argument may be MOTOR-_DIRECTION_CLOCKWISE or MOTOR_DIRECTION_CW for clockwise, and MOTOR-_DIRECTION_COUNTERCLOCKWISE or MOTOR_DIRECTION_CCW for counter-clockwise.

- int MotorControlPOD::Clockwise(unsigned char Speed)
  Starts the motor going clockwise at the specified speed (full speed if the Speed parameter is omitted).

- int MotorControlPOD::Clockwise_Slow()
  Starts the motor going clockwise slowly.

- int MotorControlPOD::CounterClockwise(unsigned char Speed)
  Starts the motor going counter-clockwise at the specified speed (full speed if the Speed parameter is omitted).

- int MotorControlPOD::CounterClockwise_Slow()
  Starts the motor going counter-clockwise slowly.

- int MotorControlPOD::Stop()
  Stops the motor completely, cancelling previous commands.

Motor Control POD Implementation

The Motor Control POD uses the STMicroelectronics L298N Dual Full-Bridge Driver [16] as an H-bridge. The L298N can handle motor supply voltages up to 46 V, and can handle up to 4 A of current. It accepts TTL logic levels for the control circuitry, and contains two devices, though both have been tied together in parallel to allow higher power and enhanced heat dissipation.

We designed the Motor Control POD with the flexibility that allows it to be used with a variety of different motors and power supplies. For the case studies, an RC car has been used as a base, and the MotorControl PODs have been attached to its motors. We also used the car's battery as the power source. For other applications, another DC motor and different DC power supply could be used.

Since the POD does not have any speed or position encoding, it is not possible to determine the motor's RPM, though this is clearly desirable. Future work could add these capabilities, and investigate standard motors and power supplies. Details about the POD can be found in Appendix B.

3.2.4 Light Sensor POD

Sensing the Light Level

It can be handy to detect various intensities of light. This can be helpful, for example, when a robot is searching for a bright portion of a room, when charting weather patterns that include the amount of sunlight received, or to detect whether someone has turned on a light in a room. With this light intensity information, the controller can turn the robot to attempt to maximize the light striking the sensor, or the presence of the light can be recorded for future reference.

Light Sensor POD Interface and Methods

The Light Sensor POD is a simple sensor that measures the relative intensity of the light falling on it. It can be queried for the intensity of the light, or a light threshhold can be set, and the POD can be queried as to whether the light is on or off (based on that threshhold).

Functions	Measures the intensity of the light
Range	0 - 1023
Power	From USB

Table 3.3: Light Sensor POD General Data

The Light Sensor POD extends the base POD class with the following methods:

- int LightSensorPOD::GetIntensity()
  This method returns the relative intensity of the light (0-1023).

- int LightSensorPOD::SetThreshhold(int Threshhold)
  This method sets the on/off threshhold of the sensor.

- int LightSensorPOD::GetThreshhold()
  This method requests the current threshhold setting from the POD.

- int LightSensorPOD::IsLightOn()
  This method is used to ask the POD whether the light is on (the intensity is higher than the threshhold).

Light Sensor POD Implementation

The Light Sensor POD contains a Photonic Detectors Inc. PDV-P9007 Photocell [17]. With this type of photocell, the resistance increases as the amount of light striking it decreases. In bright light, the resistance of the photocell drops to approximately 1 KΩ, while in darkness it is approximately 100 KΩ. The Light Sensor POD measures this change in resistance, and produces a value corresponding to the intensity of the light present. This value varies from 0 (darkness) to 1023 (bright light). Details about the POD are shown in Appendix B.

3.2.5 Light Emitting Diode (LED) POD

Light Emitting Diodes

Computer control of lights can sometimes be helpful. The lights could be used as status indicators, or the lights can be used to illuminate an area. On a robot, the lights may be used to aid in navigation, and could be combined with a camera.

LED Light POD Interface and Methods

The LED Light POD consists of four ultrabright white LEDs that may be switched on as desired. The POD was designed to provide light to illuminate a small area, and can be seen easily across a dark room.

The LED Light POD adds the following methods to the POD base class:

- int LightPOD::AllOn()
  This method turns on all the LEDs.

- int LightPOD::AllOff()
  This method turns off all the LEDs.

- int LightPOD::TurnOn(int LED1, int LED2, int LED3, int LED4)
  This can be used to turn on the LEDs individually, with 0 indicating off, and any other value meaning on. If any of the parameters are ommitted, they are assumed to be off.

LED Light POD Implementation

Our initial POD implementation utilized four ultrabright white LEDs. The LEDs were not bright enough to provide as much light as we originally hoped, but did give some illumination to the immediate area, and were bright enough to be seen easily across a dark room. Since it is not as bright as we desired, it is still useful as a beacon light, but does not provide enough light to read by.

The LED Light POD does have promise, though, and we belive that the POD can be useful in a number of different ways. By using different colored lights (red, green, yellow, blue), we have found that the POD becomes a very nice status indicator. To make this effective, we extended the LightPOD as a ColoredLightPOD, allowing us to individually turn on and off the lights by color (ColoredLightPOD.Red() for example).

Future work to increase the amount of light provided by the POD would also increase the PODs usefulness. This could be done through the addition of more or brighter LEDs, focusing the light with a lens and reflector, or by using a technology other than LEDs. When combined with some cameras, infrared LEDs can be used as a light source that cannot be seen by the naked eye. We have included details about the Light POD in Appendix B.

3.2.6 Power Sensor POD

Detecting AC Power Status

In most server rooms, a stable power supply is a mission critical requirement. Though many servers may be connected to an Uninterruptible Power Supply (UPS), if the power outage persists, it may be desirable to shut down the system correctly before the UPS runs out of power. Knowledge of the source power status can be invaluable to the computer or the administrator. Though many UPS systems are capable of reporting the power status, this status may be difficult to access from an administrator's program. In addition, many server rooms have multiple servers, and a box to attach additional servers may be expensive. Working with wall current can also be dangerous for an inexperienced circuit designer.

Power Sensor POD Interface and Methods

The Power Sensor POD is a device that is plugged into both a computer and the AC power line. Power status is detected, is shown on the board with red and green status indicators, and can be communicated to user programs on the host. Assuming that the host computer is connected to a separate power source, such as an Uninterruptible Power Supply (UPS), the Power Sensor POD can be used to detect the power status, giving a computer time to shut down or take the necessary precautions.

Functions	Detects AC power status, and displays status with red and green LEDs.
Connector	Standard PC power connector.
Power	Logic from USB, connection to AC power mains is required.

Table 3.4: Power Sensor POD General Data

The Power Sensor POD extends the POD base object with the following method:

- int PowerSensorPOD::GetPowerStatus()
  Returns 1 if AC power is on, and 0 if power is off.

Power Sensor POD Implementation

The core component of the Power Sensor POD is the Fairchild MID400 AC Line Monitor Logic-Out Device [18]. The MID400 is a small AC line-to-logic interface device. It consists of back-to-back opto-isolators. The POD also uses a Schurter GSP1 Appliance Inlet [19] for the AC power since PC power cables are very common. This also allows the Power Sensor POD to be attached and detached whenever desired, and allows great flexibility of location. The Power Sensor POD also uses standard green and red LEDs for the visual power status indicator. Details about the Power Sensor POD can be found in Appendix B.

3.2.7 Sonar Distance Ranger POD

Distance Measurement

Measuring the distance to objects can give invaluable information about a robot's environment. Distance measurement can also be very helpful in many other applications, such as a security system or when parking a car in a garage. In general, a computer needs to know the distance to objects to make informed decisions.

Sonar Distance Ranger POD Interface and Methods

The Sonar Distance Ranger POD is a self-contained distance sensor, and can provide a high-resolution measure of the range to objects. The distance can be requested in either centimeters or inches, and since the device can detect multiple echos, the POD can return the distance to any of the closest 17 objects, or all 17 can be requested at the same time.

Functions	Measures distance
Range	3cm - 6m
Precision	1cm
Resolution	1cm
Power	From USB

Table 3.5: Sonar Distance Ranger POD General Data

The Sonar Distance Ranger POD extends the POD base class with the following additional methods:

- int SonarPOD::GetDistance_in(int ObjectNumber)
  This method requests the distance in inches to object ObjectNumber. ObjectNumber is an optional argument, and defaults to zero, or the nearest object detected.

- int SonarPOD::GetDistance_cm(int ObjectNumber)
  This method requests the distance in centimeters to object ObjectNumber. ObjectNumber is an optional argument, and defaults to zero, or the nearest object detected.

- int SonarPOD::GetAllDistances_in(int *Ranges)
  This method returns the distance to all objects in inches. Ranges should be an array of 17 integers.

- int SonarPOD::GetAllDistances_cm(int *Ranges)
  This method returns the distance to all objects in centimeters. Ranges should be an array of 17 integers.

Sonar Distance Ranger POD Implementation

The Sonar Distance Ranger POD is built using the Devantech SRF08 Ultrasonic Range Finder [20]. The SRF08 is capable of detecting echos from multiple objects, and can report the distance range in microseconds, centimeters, or inches. It also includes a light sensor that has not been used in this project. Future work could demonstrate how multiple inheritance can work with PODs, and the POD could behave as both a Sonar Distance Ranger POD and a Light Sensor POD. Details about the Sonar POD can be found in Appendix B.

3.2.8 Compass POD

Digital Compass

Electronic compasses are often used to detect the heading of mobile robots, and can be a great aid in navigation. In connection with other sensors and other information about the environment it's operating in, robots may be able to determine their position or where they should go.

Compass POD Interface and Methods

The Compass POD is a digital compass that can be used to detect the direction of the largest magnetic flux. It returns a high-resolution heading, and can also return a text-based description of the heading.

Range	0° - 359.9° [21]
Accuracy	±4°
Resolution	0.1°
Power	From USB

Table 3.6: Compass POD General Data

The Compass POD extends the POD base class with the following methods:

- float CompassPOD::GetHeading()
  This method returns a value corresponding to the compass bearing (negative if an error occurred).

- float CompassPOD::GetHeading(char *HeadingString)
  This method returns a string description of the heading (such as NNW or S). HeadingString should be a string of characters at least 4 characters long. For convenience, this method also returns the value corresponding to the compass bearing.

Compass POD Implementation

The Compass POD is built using the Devantech CMPS03 Robot Compass Module [21], a digital compass that detects the horizontal component of the prevailing magnetic flux. It uses an I^2C bus that the POD microcontroller reads. This heading information is then communicated to libPOD. The CMPS03 uses two Philips KMZ51 magnetic field sensors at right angles to each other to determine the current heading. Details about the Compass POD can be found in Appendix B.

3.2.9 Buttons POD

Input from Pushbuttons

Though many computers have methods to receive input from the user, such as through the use of a keyboard or mouse, some electronic systems do not require the complexity or size of a keyboard. In these cases, a few simple pushbuttons can serve to meet all the input needs of the system. Examples of this include a button to reset the system or a button to silence an alarm. This could also be used to implement touch sensors in various applications. In these cases, small, simple devices may be all that is necessary.

Buttons POD Interface and Methods

The Buttons POD is a simple device designed to accept input from the user. It contains several pushbuttons and an on/off switch. The POD has methods to determine the number of buttons, as well as a method to request the status of each, as shown here:

- int ButtonsPOD::GetNumberOfButtons()
  This method returns the number of buttons on this particular POD.

- int ButtonsPOD::GetStatus(int ButtonNumber)
  This method returns whether the individual button is currently being pressed. A return value of TRUE indicates a button or switch that is currently pressed or On, and FALSE indicates that the button or switch is Off or not pressed.

Buttons POD Implementation

The Buttons POD is a slightly modified LED-Light POD board. It has several small push-buttons, as well as an on/off switch connected to the microcontroller's digital input pins, and the status of each can be individually requested. Additional details about the POD can be found in Appendix B.

We have described a number of PODs that we have built, as well as how they operate. Table 3.7 shows the devices created and their native interfaces. Though the various interfaces may be disparate, by building these devices as PODs, these interfaces have been replaced with a single standard USB interface and a simple API.

POD	**Interfaces used on Microcontroller**
Thermometer POD	One (1) analog input port
Alarm POD	Three (3) digital output ports
Motor Control POD	Three (3) digital output ports
Light Sensor POD	One (1) analog input port
LED Light POD	Four (4) digital output ports
Power Sensor POD	One (1) digital input port
Sonar Distance Ranger POD	One I^2C input/output bus
Compass POD	One I^2C input/output bus
Buttons POD	Four (4) digital input ports

Table 3.7: Thermometer POD General Data

3.3 Interconnecting PODs

Now that we have introduced libPOD and the PODs that we created for this work, we will discuss some of the general ideas related to connecting and using PODs. In this section, we describe hardware and software utilities that are utilized when connecting and using PODs.

While it is often desirable to control a number of PODs from a single computer, many computers do not have enough USB ports. For example, most laptop computers contain only one or two. In these cases, PODs must be connected to an expansion USB hub. One such hub that we found to work quite well is shown in Figure 3.5. This hub is physically small, and expands one USB port to four. It can be either bus-powered (drawing power from the computer it's attached to) or self powered (contains a battery or is attached to another power source). Since most of our PODs require very little power, we found the bus-powered mode to be sufficient.

When PODs are attached to the USB bus, the operating system assigns device or port numbers on the controlling computer. Because they can be connected in any order, or removed and reattached, the port numbers may be reassigned. The user must be able to

Figure 3.5: Small 4-port USB hub.

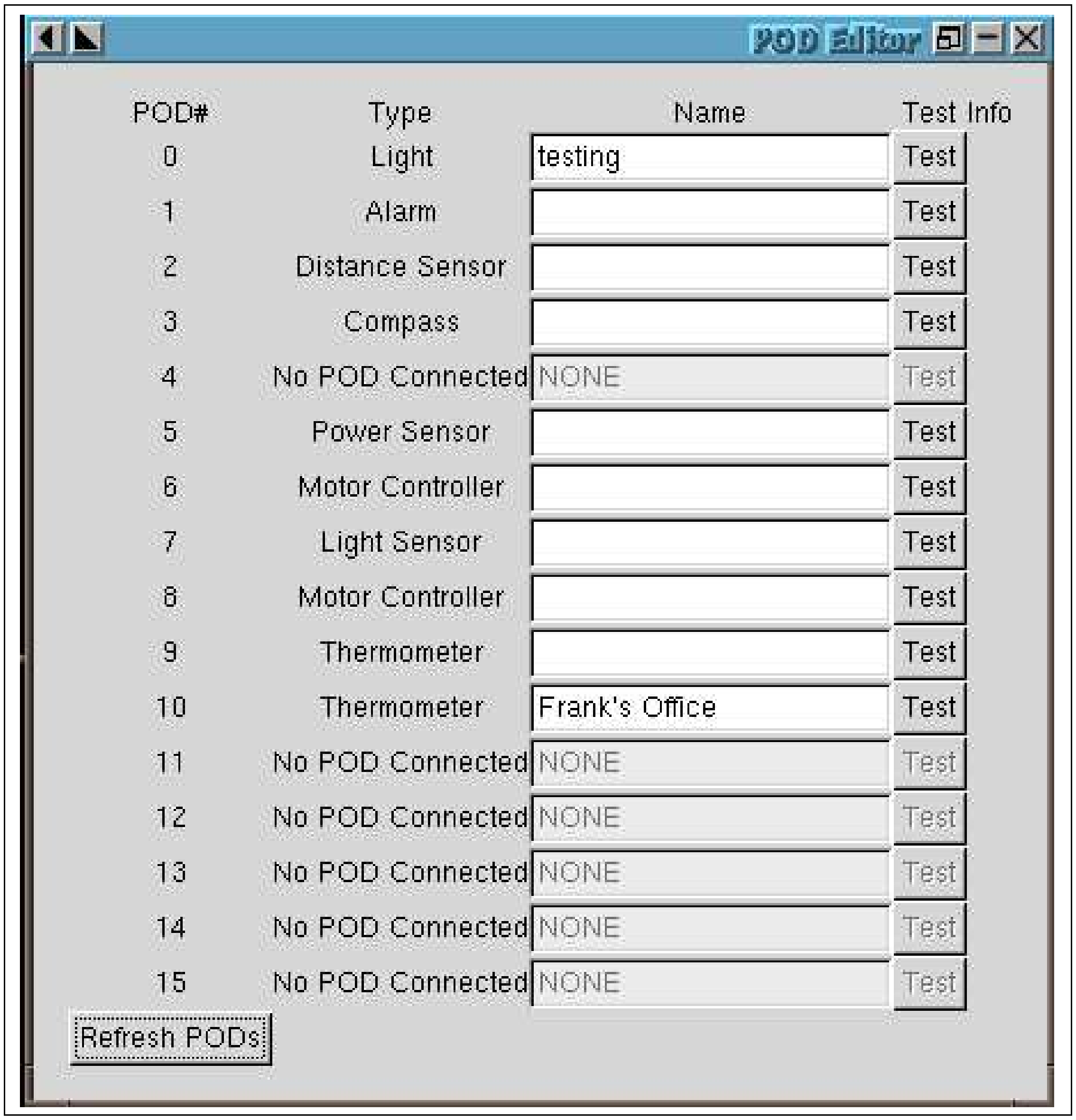
determine what PODs are attached, and to identify which PODs they are communicating with. Overcoming this potential confusion is especially important when multiple PODs of the same type are attached, such as several Thermometer PODs monitoring the temperature in different rooms, or multiple Motor Control PODs on a robot.

As mentioned briefly in Section 2.1, each POD may be assigned a name by the user. This may be done through the `SetName(char *String)` method, and each POD stores this name in its onboard memory. This allows the programmer to determine which device is which (provided the user assigns the names uniquely). Using the FindPODByName() method inside PODEnv, the user can request a particular device by name. Since this name is stored on the POD, it will be retained even if the POD is removed or reattached to a different computer. Each POD also responds to the Test() method, which tests or demonstrates the POD in the manner appropriate for that particular POD type. For example, the Motor Control POD will demonstrate forward and backward motion, and the Compass POD will output the current heading.

To help the POD user detect and identify connected PODs, we have developed a small application called PODEdit that detects all connected PODs and allows the user to individually test and identify each POD. Since each POD can be assigned a name for identification, PODEdit reads the existing name and displays it to the user. The POD can be tested, and the names of each POD may be changed. While the device, port number, and POD number may change as PODs are added, changed, and removed, the name will remain constant, since it is stored on the POD.

PODEdit is written in GTK2. Like libPOD, PODEdit will run under either Linux or Cygwin in Windows. As an example of how PODEdit might be used, a user may have a number of PODs attached, with the resulting screen displayed in Figure 3.6. As shown, Thermometer PODs have been discovered at #9 and #10, and Motor Control PODs were found at #6 and #8. Since one of the Motor Control PODs represents the left wheel, and the other represents the right wheel, it may be difficult to determine which is which. However, the **Test** buttons can be used to test each wheel, assisting the end user in determining which is which, and in labeling each POD appropriately.

The Thermometers may be tested as well, though since they do not have any visible output, they may need to be tested and labeled when they are the lone Thermometer POD attached. In this example, we have labeled POD #10 to be “Frank’s Office”. Again, these names will remain with the particular POD.



POD#	Type	Name	Test Info
0	Light	testing	Test
1	Alarm		Test
2	Distance Sensor		Test
3	Compass		Test
4	No POD Connected	NONE	Test
5	Power Sensor		Test
6	Motor Controller		Test
7	Light Sensor		Test
8	Motor Controller		Test
9	Thermometer		Test
10	Thermometer	Frank's Office	Test
11	No POD Connected	NONE	Test
12	No POD Connected	NONE	Test
13	No POD Connected	NONE	Test
14	No POD Connected	NONE	Test
15	No POD Connected	NONE	Test

Figure 3.6: Screenshot of the PODEdit program.

Chapter 4

Case Studies

In this chapter, we present the results of several case studies involving both simple and complex systems constructed with PODs. The first case study involves a thermometer that can be used to find the temperature at home, in a server room, or even outside. This case study shows how PODs can be used in simple systems. The second case study demonstrates the construction of a Server Room monitoring system. It provides a more complex example of how PODs are used, and extends the use of the thermometer from case study 1 with a number of additional devices. The third case study shows how a number of the same parts can also be used to make a robot. This shows that PODs are extremely modular and flexible, and can be used in a variety of different applications.

For each of the case studies, we describe the system to be constructed, and how it will function. We discuss the benefits that PODs bring to that system's design and construction process, and show that the system is easily constructed or modified in software.

While the case studies will show relatively simple examples, they will demonstrate a number of valuable benefits of using PODs. By first constructing the PODs and libPOD software, we have supplied ourselves with a modular platform that abstracts the low-level electronics, simplifies the construction of these systems, and provides easy reconfiguration. In addition, we show that PODs are flexible and scalable, and that libPOD provides a convenient, yet powerful, user interface to the devices.

4.1 The Process of System Construction

To make for a fair comparison between traditional and POD construction methods, we will first discuss the tasks common to constructing traditional electronic modules and PODs.

For each of the electronic components used in the case studies, a number of tasks must be performed. The user must first investigate various methods of performing a particular task, determine the availability of suitable electronic parts, and determine how best to utilize them to accomplish the target task.

As previously shown, there are often a number of different ways in which various functions may be implemented, and each of these must be evaluated for their feasability and usefulness in the desired systems. This may begin with internet searches and looking through various electronics design books, and may result in several possible approaches. The user then chooses an option, orders electronic parts, and awaits their delivery. When the parts arrive, they must experiment through breadboarding to learn how they might work together, how the electronics are to be powered, and how the device interfaces with the computer.

In designing and building the electronic components, the user must choose the interfaces that will be used. As the electronic parts may communicate with each other, or with the controlling computer, in a number of different ways, the user should design the system with future needs in mind. For power, the device may have a battery attached, a power brick must be plugged into the wall, or it may depend on power from the controlling computer.

The parts are often soldered together, and the resulting component is tested to ensure that it performs its functions correctly. Sometimes, this testing process reveals flaws in the design or construction of the device. The datasheets describing the electronic parts may be incomplete or inaccurate. When these design and construction errors and inconsistancies are found, some or all of the process must be repeated. This design, construction, and testing process is then repeated for each of the other devices to be built.

In the next several sections, we show how choosing to implement a system using PODs reduces the effort to construct devices and systems of devices. This is due to the fact that we pay the implementation costs once instead of once for every system design. Additionally, PODs have a common interface standard and an easy to use API. Choosing PODs gives great flexibility, without significantly limiting a user's future choices.

4.2 Case Study 1 - Thermometer Device

This case study implements a simple system involving a single thermometer. Electronic monitoring of temperature can be useful in a variety of situations. Many web sites contain information about current weather conditions. Historical temperature data is often collected for later reference. Monitoring of environmental conditions can help predict future weather or identify the cause of an environmental problem. Temperature information can be used to warn of potential problems in a climate-controlled area. Noting peak temperatures over time can help determine whether a cooling system is adequate. Rising inside temperatures during hot days, for example, may indicate weaknesses in the cooling, and could indicate the need for a larger system.

The purpose of this case study is to graph the current temperature in our server room, using a tool called MRTG. MRTG is often used for network traffic monitoring and graphing, but its great flexibility allows its use in a number of applications that require data collection and graphing. Under Linux, a cron job starts every 5 minutes, and MRTG calls a program to get the temperature, records the temperature, and updates some graphs.

To implement a simple system, such as monitoring and recording temperature ever 5 minutes, we simply chose an electronic thermometer chip and used it to construct a Thermometer POD. After building and programming the POD, which required approximately the same level of work that a custom implementation would have required, we just plugged in the device and wrote a simple program. To demonstrate just how easy it is to program using PODs, we have included the entire source code for this simple application in Figure 4.1. This code took no more than 5 minutes to write and test.

Our simple thermometer application has been running for over 7 months in the Computer Science Department server room. We have included one of the graphs generated in Figure 4.2, and the web page can be seen at `http://stats.cs.byu.edu/temperature/`.

While this system is extremely simple, it provides a powerful demonstration of just how easy PODs can make a project. After construction of the Thermometer POD, no low-level coding is required, and only a few short lines of C++ are needed. The POD can be easily moved from one computer to another, and the code could quickly be rewritten for use in other applications. This illustrates the benefits of using PODs. While the POD construction effort requires the same level of effort required to build a custom system, the POD design is easily reused and integrated into new systems.

```
#include <stdio.h>
#include <libPOD.h>

int main(int argc, char *argv[])
{
   PODEnv *MyPODs;
   ThermometerPOD *Thermometer;
   float Temperature;

   MyPODs = new PODEnv();

   Thermometer = dynamic_cast<ThermometerPOD *>
      (MyPODs->FindPODbyType(POD_TYPE_THERMOMETER));
   if (Thermometer != NULL)
   {
      Thermometer->GetTemperature(&Temperature);
      printf("\%f", Temperature);
   }
   else printf("0"); /* if no Thermometer POD is connected */
   printf("\n0\n0\n0\n");
}
```

Figure 4.1: Complete source code for the Temperature Sensor system used in Case Study 1 (the additional 0's and newlines printed are to satisfy constraints placed by MRTG, the data collection, storage, and graphing program).

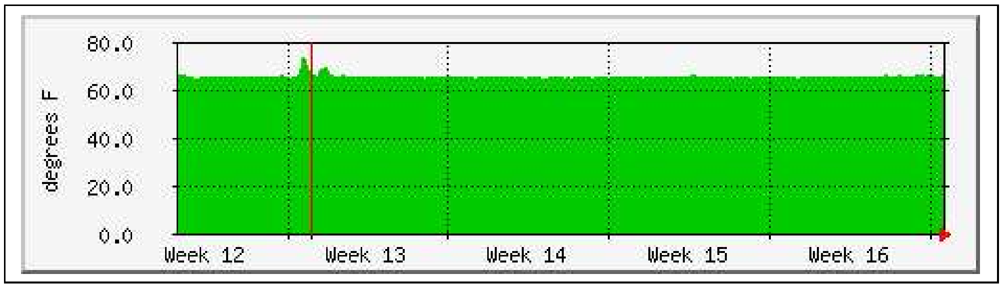


Figure 4.2: Sample monthly graph of the temperature in the Computer Science Department server room. Near the beginning of week 13, the Talmage building experienced some cooling problems, resulting in higher temperatures and the spike in the graph.

4.3 Case Study 2 - Server Room Monitoring System

In this section, we investigate the use of a number of PODs within a single system. This case study will utilize the Thermometer POD used in case study 1, but combines it with a number of other devices to form a more complex server room or datacenter monitoring system.

It is common to carefully secure, monitor, and control various systems in a datacenter or server-room environment. Knowledge of various conditions allows the system to warn of problems, inform people of status changes, or record statistics. A complex system may have various input and output devices, and the results of one element may be used to determine whether to perform some action.

In this case study, we constructed a somewhat complex system that can find the current room temperature, determine whether the lights are on or off, and whether the power has failed. It will warn of problems through an audible alarm, and can indicate status through various-colored LEDs. If the system detects a temperature higher than the configured threshold, email will be sent to a pager, and the alarm will beep. A red light will turn on to indicate a high temperature condition. If the system detects a power outage, similar things will happen (email page, audible alarm, and visual status indicator). Additional pages will be sent periodically, and the alarm and red light will continue, until the system detects that a system administrator has responded to the problem by turning on the lights or clearing the system.

This case study requires a number of different electronic components:

- Lights
- Alarm
- Thermometer
- Buttons
- Light Sensor
- Power Sensor

While USB supports 127 devices, the process of scanning for devices takes time and additional resources. libPOD has been designed to detect and organize up to 16 devices, but support for 127 may easily be accomplished by recompiling and creating additional device nodes (under Linux, this is done with `mknod ttyUSB16 c 188 16`). This practical limitation of 16 is sufficient for all the projects we have done in this work.

Each of the devices used in this case study requires relatively little power, so they can all be powered by the controlling computer. This helps keep the wiring to a minimum, and allows for a clean look to the project. Because each POD contains its own microcontroller, the device communication is simplified. Even though the various electronic components have various interfaces, the controlling computer only needs to communicate with the microcontroller via USB. This is enabled by the microcontrollers we chose. Though small, they contain a large number of features. They can communicate with devices that output analog values, use a digital interface, use the TWI (I^2C) protocol, or communicate using other methods. This enables each device to be a self-contained unit, and allowed us to design and construct each hardware device individually.

After constructing each POD, all that was necessary to build the server room monitor was to create the high level program required to logically interconnect them in the desired fashion. The libPOD library helped abstract away the low-level communication details, which greatly simplified this task. The software required on the controlling computer is illustrated in Figure 4.3.

```
int ShowSystemStatus(SystemPODsStruct &SystemPODs,
   StatusStruct &Status)
{
   int Error = FALSE;

   SystemPODs.Lights->Green(ON); /* system operational */

   Status.CurrentPower =
      SystemPODs.PowerSensor->GetPowerStatus();
   if (! Status.CurrentPower)
   { /* power has failed */
      printf("Power OFF ");
      SystemPODs.Lights->Red(ON);
      Error = TRUE;
      if (Status.OldPower != Status.CurrentPower)
      { /* failed now */
         SendEmail("Power Failed");
         Status.AlertAcknowledged = FALSE;
      }
   }
   else SystemPODs.Lights->Red(OFF);
   Status.OldPower = Status.CurrentPower;

   SystemPODs.Thermometer->GetTemperature(Status.
      CurrentTemperature);
   if (Status.CurrentTemperature >
      (MAX_TEMPERATURE + HYSTERESIS))
   { /* high */
      printf("Temperature HIGH ");
      SystemPODs.Lights->Yellow(ON); /* indicate high temp. */
      Error = TRUE;
      if (Status.OldTemperature <=
         (MAX_TEMPERATURE - HYSTERESIS))
      { /* new problem */
         SendEmail("High Temperature");
         Status.AlertAcknowledged = FALSE;
      }
      Status.OldTemperature = Status.CurrentTemperature;
   }
   else /* temperature normal */
   {
      SystemPODs.Lights->Yellow(OFF);
      if (Status.CurrentTemperature <=
         (MAX_TEMPERATURE - HYSTERESIS))
         Status.OldTemperature = Status.CurrentTemperature;
   }

   Status.CurrentLightLevel =
      SystemPODs.LightSensor->IsLightOn();

   if (Error == TRUE) /* problem...sound the alarm */
   {         /* unless alert has been acknowledged */
      if ((Status.OldLightLevel == FALSE)
         && (Status.CurrentLightLevel == TRUE))
      {
         Status.AlertAcknowledged = TRUE;
         printf("Acknowledged by lights being turned on\n");
      }

      if (SystemPODs.Buttons->GetStatus(BUTTON_PUSHBUTTON))
      {
         Status.AlertAcknowledged = TRUE;
         printf("Acknowledged by button\n");
      }

      if ((! Status.AlertAcknowledged)
         && SystemPODs.Buttons->GetStatus(BUTTON_ONOFF))
      {
         SystemPODs.Alarm->Sound(SOUND_LENGTH);
         printf("Alarm ON                           \r");
      }
      else printf("Alarm OFF                           \r");
   }
   else
   {
      printf("Status NORMAL                            \r");
      Status.AlertAcknowledged = TRUE;
   }
   Status.OldLightLevel = Status.CurrentLightLevel;

   fflush(stdout);
   return(OK);
}
```

Figure 4.3: Sample code from the Server Room monitoring system.

4.4 Case Study 3 - Mobile Robot System

As shown in the previous case studies, PODs simplify the construction of systems with their modular design, and by providing a method to bridge the gap between high-level programming and the low-level electronics. Because PODs can easily be added or removed, the same components can be used in a variety of different systems. This makes them ideal for many applications, because once an electronic component has been designed as a POD, that device can be used interchangably in completely different systems. After the desired PODs are attached, and a new controlling program is written, the system has taken a completely new form.

While the previous case studies focused on demonstrating how PODs allow for a consistant user interface, and simplify the construction of a system, this case study demonstrates how components built for one system can be used interchangably to build an alternate system. Designing a separate system using many of the same parts shows the flexibility and power of both the electronics and libPOD.

Here, we present the construction of a simple mobile robot. We use some of the PODs built for the other case studies, as well as constructing several more. Our goal in programming the robot was to have the robot seek out light. While this is a basic task, it will show how quickly a programmer can move from the robot building stage to implementation of interesting algorithms. Again, we will not be designing a complex robot, but rather demonstrating the flexibility and power of our modular POD system by building a simple robot in just a few steps, and using PODs previously built for other purposes.

Our mobile robot moves around the halls of the department, and is designed with a sonar distance ranger to help it avoid running into walls or other objects. The robot has several light sensors so that it can detect areas of bright light, and it has a compass to track its heading. An alarm has also been attached, and sounds when the robot is backing and to indicate its status.

To provide a chassis for our robot, we disassembled a remote control vehicle, and used its base as a platform to build on and attach the parts to. We removed the car's superstructure, and replaced it with a platform on which the battery, sensors, other devices, and computer controller have been mounted. For mobility, we used the car's motors, since they are powerful enough to move the robot around, and are securely attached to the base. We also used the RC car's internal battery to power the motors.

For control, we chose to use a single board computer (SBC), the Advantech PCM-5824 [22]. This compact computer runs a basic Linux operating system off of a laptop hard drive. It has a network port and can also communicate over a wireless card. The SBC has serial, parallel, and USB ports, and is capable of running on +5 volts from a battery or other source. We mounted this computer inside a custom-designed enclosure. This helps insulate and contain the computer and components and allows for easier mounting on top of our robot. It receives its power from a 6 Volt gell-cell battery mounted on the robot. We tested several batteries before choosing one that provides a little over two hours of runtime on our computer.

The sensors and other electronics have also been insulated and mounted to the robot. Several small USB hubs help to connect all the devices with the SBC. Except for the robot's motors, all the devices can draw power directly from the USB bus. Because the computer runs a full Linux system, we were able to program it using the same tools, libraries, and methods that we used in the previous case studies. After we had everything in place, we wrote our program to test out our algorithm for finding light.

Because the PODs may be attached in any order, and to differentiate the various motors and sensors, we used PODEdit to test and label the devices appropriately. For example, in our robot, the motor control PODs have been labeled **Left** and **Right** to simplify their identification in the programs the robot runs.

Our robot follows a fairly simple light-seeking algorithm, and performs relatively well. Our program took about an hour to write, then another hour or two to debug and improve. While the entire code for our robot is lengthy, we wish to highlight several sections of code that illustrate interesting points. We implemented our robot as an object that extends the PODEnv class, as depicted in Figure 4.4.

We have included a picture of the completed robot in Figure 4.5, and the source code for the light-seeking algorithm our robot executes is displayed in Figure 4.6. Our robot currently checks the light levels forward and to each side. The robot then turns in the direction of the greatest light, moving forward if that sensor measures the highest light intensity. It will stop if the robot detects that the light is decreasing in intensity, or if it is traveling forward, and determines it cannot safely go further.

```
class Robot : public PODEnv
{
   public:
      Robot();

      int Forward(unsigned char Speed);
      int Backward(unsigned char Speed);
      int Left();
      int Right();
      int Stop();

      float Heading();
      int Distance();
      int DisplayLightReadings();

      int ChooseBrightest(int &Brightness);
      int SeekBrightest();

   protected:
      MotorControlPOD *LeftMotor;
      MotorControlPOD *RightMotor;

      LightSensorPOD *LeftSensor;
      LightSensorPOD *CenterSensor;
      LightSensorPOD *RightSensor;

      SonarPOD *Sonar;
      CompassPOD *Compass;

      AlarmPOD *Alarm;
   private:
      POD *FindPOD(int POD_Type, char *POD_Name);
};
```

Figure 4.4: Object-oriented Robot.

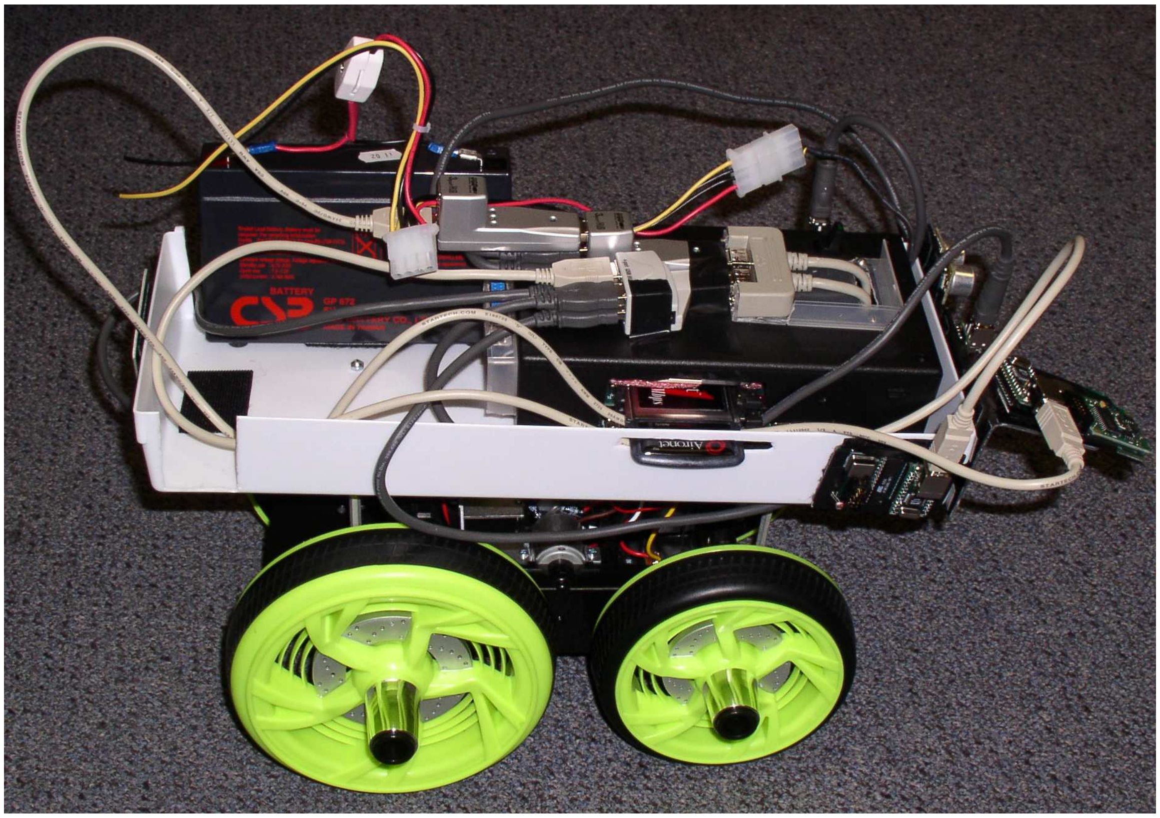


Figure 4.5: Picture of our working robot.

```
int Robot::SeekBrightest()
{
   int LastBrightest;
   int Brightest;
   int LastBrightness;
   int Brightness;
   int Distance;
   int Done = FALSE;

   while (! Done)
   {
      Brightest = ChooseBrightest(Brightness);

      if (Brightest == LEFT)
      {
         if (LastBrightest != LEFT)
         {
            Stop();
            Left();
         }
         LastBrightest = Brightest;
         LastBrightness = Brightness;
         printf("Turning Left\r");
         fflush(stdout);
      }

      else if (Brightest == RIGHT)
      {
         if (LastBrightest != RIGHT)
         {
            Stop();
            Right();
         }
         LastBrightest = Brightest;
         LastBrightness = Brightness;
         printf("Turning Right\r");
         fflush(stdout);
      }

      else /* forward */
      {
         if (LastBrightest != FORWARD)
         {
            Stop();
            Forward(SPEED);
         }
         else
         {
            Distance = Sonar->GetDistanceInches();
            if (Distance <= DISTANCE_THRESHOLD)
            {
               Stop();
               printf("Forward is brightest, but it's close-range.
                    Stopping\n");
               Done = TRUE;
            }
            if ((Brightness + BRIGHTNESS_THRESHOLD) < LastBrightness)
                 /* getting darker again */
            {
               Stop();
               printf("Getting darker again.  Stopping\n");
               Done = TRUE;
            }
         }
         LastBrightest = Brightest;
         LastBrightness = Brightness;
         if (! Done) printf("Forward\r");
         fflush(stdout);
      }
   }

   Stop();
   printf("Forward should now be brightest\n");
   return(0);
}
```

Figure 4.6: Sample code demonstrating the algorithm the robot follows.

4.5 Analysis and Comparison of POD Benefits

When comparing the tasks, decisions, flexibility, and time requirements when building these systems, it is obvious that PODs make things simpler. Many of the key implementation details have been decided. The system scales easily, and each POD becomes easier to build than its predecessor. After the PODs have been built, the user merely connects the devices, and can spend their time working to perfect their software algorithm.

Our case studies have demonstrated that PODs provide a great many advantages over traditional design methods. PODs allow for quick development through the reuse of modular hardware. The choice to use PODs removes many detailed decisions, but does not cause significant sacrifices. Once built, PODs can be treated as just another software object. PODs provide a simple and uniform user interface, as well as a flexible, modular design.

As we have shown, each POD can be used in a variety of different ways. During our work, we found a plethora of additional projects that could be implemented using just the PODs that we had created. The following list of such projects is obviously not complete, but illustrates the usefulness and versatility of PODs.

- The Power Sensor POD could be attached to a switch and used as a timeclock. An employee could flip the switch on when they arrive at work, then turn it off again when they clock out for lunch or at the end of the day. The system could tally up the time spent working each day, and report this total for billing purposes.

- The Compass POD could be attached to a door and calibrated to detect when the door is open and closed. This could be used to tell when someone enters the room, and could easily serve as a burgler alarm in a variety of situations.

- The Sonar Distance Ranger POD could be mounted in such a way as to work like a motion detector, or to indicate that someone is present just outside a door. This might be useful for applications such as airline security, where a pilot must know when a person might be directly outside the airplane cabin door.

- A Sonar Distance Ranger POD and an Alarm POD could be used together to help someone parking an unwieldy vehicle determine how much farther to pull into the garage (this could be implemented in either the garage or the vehicle).

- A Light Sensor POD placed under or behind an object could be used to warn someone that the object has moved or been taken.

- An Alarm POD could be used to read email or give information through Morse code to someone who was visually impaired or enjoys Ham Radio.
- A small pinwheel could be attached to the Motor Control POD, allowing it to function as a small fan, or as a visual indicator or alert.
- A Light Sensor POD placed alongside a plant could help the gardner tell whether the plants are getting the appropriate amount of sunlight.

As stated earlier, this list is not exhaustive by any means, but simply illustrates the usefulness of PODs in creating otherwise complex systems.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In our work, we concerned ourselves mainly with providing a user interface to hardware devices for programmers. These people often have higher-level ideas and algorithms that they would like to implement, and would rather not work through low-level implementation details. As the study of algorithms is the goal, time spent designing and building custom electronics may be time wasted or misplaced.

We have created object-oriented hardware devices that we call PODs (Physical Object Devices). We designed and constructed a number of PODs, provided a C++ programming library that makes their use simple, and demonstrated the usefulness of PODs in several case studies. Our work shows that object-oriented hardware is not only possible, but convenient and useful. Our approach simplifies and standardizes both the hardware and software interfaces, and allows the user to focus on writing their programs to control devices. For the POD designer, each new POD becomes easier to create, and even after just a few PODs have been built, a large variety of systems can be constructed.

There are a number of easily identifiable benefits that PODs provide:

- **Fewer decisions** - Most people don't really care how their system is powered or whether it is connected over USB, serial, or another interface. They just want the device to work. While there certainly exist cases in which the builder may want different choices than those picked by the POD designer, PODs remove many of these detailed decisions, and allow the majority to build without having to deal with them at all.

- **Modular design** - The modular design of PODs makes both individual devices and entire systems easier to test, identify problems with, and modify. For example, if a particular thermometer sensor becomes damaged in some way (water sometimes gets into devices placed outside), the user may need to desolder the temperature sensor or some part of it, find a new component, and resolder it. The new sensor may not have the same power, pinout, and space requirements, or may require differently sized resistors. Using PODs, the worst case is that a new POD is ordered, and the existing POD is unplugged from the USB bus, and the new one connected in its place. Since USB supports up to 127 devices, this modular design also makes it simple to add new devices later, such as when adding the LED lights to an existing monitoring system. It is also possible to swap out one device for another with slightly different capabilities. This may come in handy if the initial system included a power sensor that could tell whether the power was on or off, and an updated version of the power sensor POD could monitor the current line voltage, and warn of spikes and brownouts.

- **Simple to program** - PODs exhibit a number of object-oriented software design principles. Encapsulation is demonstrated through abstracting away the low-level details. Each new object inherits from the base POD object, and the ColoredLightPOD shows how a POD can be extended. The Test method demonstrates polymorphism by demonstrating the POD in the appropriate ways. This object-oriented interface is familiar to programmers, and allows for very simple programs to control the devices in complex ways. Once a POD has been constructed, the user does not need to know anything about how the electronics work. The PODEnv access methods provide a simple user interface that requires only a basic understanding of object-oriented programming.

- **Hardware Reuse** - Like software objects, electronic hardware can achieve great benefits by reusing previous work. We have shown that the object-oriented devices used in each case study can be used in other applications with little or no modification necessary. In addition, construction of each new POD becomes easier, since a number of interfaces have been completed. For example, the Buttons POD was constructed out of a Light POD as an afterthought, and the Sonar POD and Compass POD are essentially physically identical (other than board shape). Because the interfaces for some low-level devices are the same, the circuit board can be reused for a slightly different application.

- **Uniform Interface** - Since all PODs share a common library, each device becomes a software object with a similar user interface. This user interface makes a seamless interaction between the hardware and software, and ensures that a user can replace one POD with a similar POD without much work. This uniform interface becomes especially important when replacing devices or using a new one for the first time. For example, our current distance sensor has a precision of 1 centimeter, and a range between 3 centimeters and 6 meters. Because of its accuracy and range, it can be an expensive device to purchase. Because PODs call for the same interface, an inexpensive distance sensor with different characteristics could easily replace our more expensive one.

- **Rapid Development** - Using PODs significantly cuts down on the number of steps required for the user to move from the conceptual stage to the usage stage. This rapid development allows them to concentrate on applying algorithms and doing research, rather than investigating potential options and soldering. Since the user does not need to spend time working on these low-level details, they are free to work on those aspects that they find more interesting, and where their skills lie.

- **Flexibility** - Using PODs in a system does not stop the user from using non-POD electronics in their projects. While the user will be at a disadvantage by not having the great benefits that PODs bring, they are not limited in their options, and are free to do any extra work they desire. This could involve using both PODs an non-POD systems side-by-side.

5.2 Future Work

5.2.1 Minimization and Miniaturization

We focused on demonstrating the concepts behind PODs, and have not made much effort to minimize the devices in size or complexity. The PODs we built were constructed using prototype 4-layer PC Boards and electronic parts that we could solder. While we did attempt to find and use smaller components, and to position them close to each other on the device, none of the PODs we built could be considered minimal. Here, we discuss a few of the considerations that could be used to minimize and miniaturize PODs in terms of size, cost, and complexity.

The microcontrollers we used have 23 programmable IO ports, 6 of which can also serve as Analog to Digital Convertor channels. None of our PODs required more than 4 digital IO ports, or 1 Analog to Digital Convertor channel. In our prototype work, it was clearly to our advantage to have a lot of extra ports that we might not use, but a smaller and simpler microcontroller would most likely suffice. In most cases, this could save quite a bit of money and space on the circuit board.

The ATMega8 can hold up to 8 KB of program code in its flash memory, and also has 1 KB of main memory and 512 bytes of EEPROM. Since we did all of our microcontroller programming using compiled C, and everything fit into the memory constraints, we were glad to have the large capabilities. It is quite likely that using raw assembler, or utilizing different programming practices would allow the POD designer to get away with using a microcontroller with much less memory. This would conserve space and save money.

Our microcontrollers contain a built-in clock, but because of stability issues, we did not use it, but instead used an external oscillator. The oscillator was much larger than if we had used a crystal and capacitors, however it reduced chip count, thereby making the design simpler. We feel that the temperature stability is important, and believe that an external clock continues to be the best solution. A surface-mount crystal and capacitors should be used to help reduce the space requirements.

The ATMega8 has an onboard voltage reference circuit for use with the Analog to Digital convertors. We did not realize this initally, and we sent out our first POD for construction with a design that had three additional components to perform this function. After we discovered the built-in voltage reference circuit, we utilized it in all future designs, rather than using the circuit we had originally designed. We found the voltage reference to be suitably stable and accurate, and removing the extra circuit saved significant space and expense.

When designing our PODs, we made every effort to find through-hole components. Though they are usually larger than their surface-mount counterparts, and require both sides of the board, they are much easier to solder. This alone saved us many hours when constructing our PODs. For prototype work, we believe these are reasonable trade-offs, but for any end-user devices, it is more practical to use the smaller components, and to have the parts on both sides of the board.

We used a pre-built USB-Serial converter device that contained a number of components. Placing these components directly onto our board would help conserve space.

The DLP board also contained a full-size USB B-connector. There are smaller connectors that could be used, even for USB, and this could cut down on the space required. Alternate interfaces exist, and some of these are mentioned in Section 5.2.3.

5.2.2 Interesting PODs to Build in the Future

The PODs we built for this work merely scratch the surface of what could be done. In order for PODs to be a truly viable option for many people, a POD must exist for each of the devices desired. Ideally, the user would be able to do what they want by finding a POD that provides the perfect solution. During our work, we thought of a number of devices that we think would be useful and interesting to have. Some of these are listed here:

- LCD Display
- Keyboard POD (plug in regular keyboard)
- Battery with battery monitor
- Water sensor
- Air flow sensor
- Smoke alarm
- Motion detector
- Door or window entry detector
- Voice recognition device
- Motor with wheel encoder (or other position detecting device)
- GPS receiver
- Tachometer
- Scale (weight sensor)
- Accelerometer
- Pressure sensor
- Strain gauge
- Barcode Reader

5.2.3 Interesting Directions to go with PODs

We enjoyed working with USB for a number of reasons. USB is a common interface, and the hardware (FTDI chip and DLP-Design board) we used greatly simplified its use. USB provided power to most of our PODs, and we generally found it to be very handy to use. Devices are easy to add and remove at will, and some of the low-level details were already implemented for us.

However, USB does have some limitations, such as each device requiring a USB cable. When working with a number of PODs, there can be quite a few USB hubs and cables connecting everything together. This often creates a large rat's nest of cables that sometimes must be untangled or traced. Also, USB can only go a limited distance from the controlling computer.

During our implementation, we found that the concepts behind PODs are simple, and are not tied to USB. The ideas upon which PODs are built would work well when applied to alternate interfaces. A few of these options could include connecting to a POD across a network, with the POD being it's own networked device, or the POD connected to some server application. This would allow some separation between the controlling computer and the device.

Another interface that would be useful to pursue is wireless radio systems. There are a number of possible systems currently existing, such as Bluetooth [23], Zigbee [24], or small, individual transcievers, such as RF Monolithics Virtual Wire® [25].

Because we designed our PODs as prototypes, we simply used the serial ports created through the use of the provided device driver (the `ftdi_sio` module under linux). Since the USB devices could be reprogrammed with alternate USB device numbers, assigning our own unique device numbers just for PODs would allow other (non-POD) devices to use the original driver. This could also provide a more uniform interface between Windows and Linux environments.

Some of the PODs lend themselves quite well to graphical status display and control interfaces. Since libPOD has some GUI capability already (through GTK2), it would not be difficult to add functionality to allow a graphical user interface to be built on top of the individual PODs. A Compass POD could take on a graphical interface that pointed north, or the temperature from a Thermometer POD could be displayed as a graphical analog thermometer.

Currently all PODs are synchronous, and require polling. We found this to be acceptable in our work, but a more elegant solution would allow an asynchronous system as well. This could be accomplished through callbacks, through signals, or by multithreading libPOD.

5.2.4 Object-Oriented Principles

While PODs are physical devices, they implement and exhibit a number of Object-Oriented software design principles. As a result, they are very flexible. They behave like software objects, and can be treated as such. Each POD applies the principle of encapsulation to abstract its internal workings and expose only the necessary methods. Each particular POD type inherits a number of properties from the POD base class, and adds their own features and capabilities. This allows each POD type to leverage the methods and properties of a POD, and simplifies the creation of future PODs. Polymorphism is shown through several methods that test and demonstrate each POD type. In addition, PODEnv is a composite class, containing potentially any number of base PODs and derived POD classes.

We have included a diagram of an example object hierarchy in Figure 5.1. All PODs used in this work are shown. The dashed objects represent potential PODs that could be built. The dashed lines indicate possible relationships between the objects. Most PODs are relatively simple extentions of the base POD class. All PODs are derived from the POD base class, either directly or indirectly. Though many of the methods implement relatively simple functions, our example demonstrates a few ways in which PODs may become more complex and powerful.

In this example, a Backup Sensor POD (used to aid a driver in parking) could be constructed using a Sonar POD and an Alarm POD. The POD could be implemented as either a composite class or multiply-inherited class. A Thermostat POD (or class) extends a Thermometer POD by adding the ability to set alarms and warn when the temperature leaves a specified comfort zone. A Motor Control POD may provide a rudimentary control for a motor, understanding only direction of spin, and its own measure of how fast it is attempting to rotate, but after attaching a wheel to the motor, a Powered Wheel POD could contribute information about the wheel radius and control over the actual linear speed.

By creating a composite class containing a number of motorized wheels, a movement platform could be constructed as an individual class in the hierarchy. A control device of this type could act as an intermediary between several PODs. It could manage the component wheels, abstracting away lower-level control details, and providing higher-level functions to users. These higher-level functions could involve commands to move to the

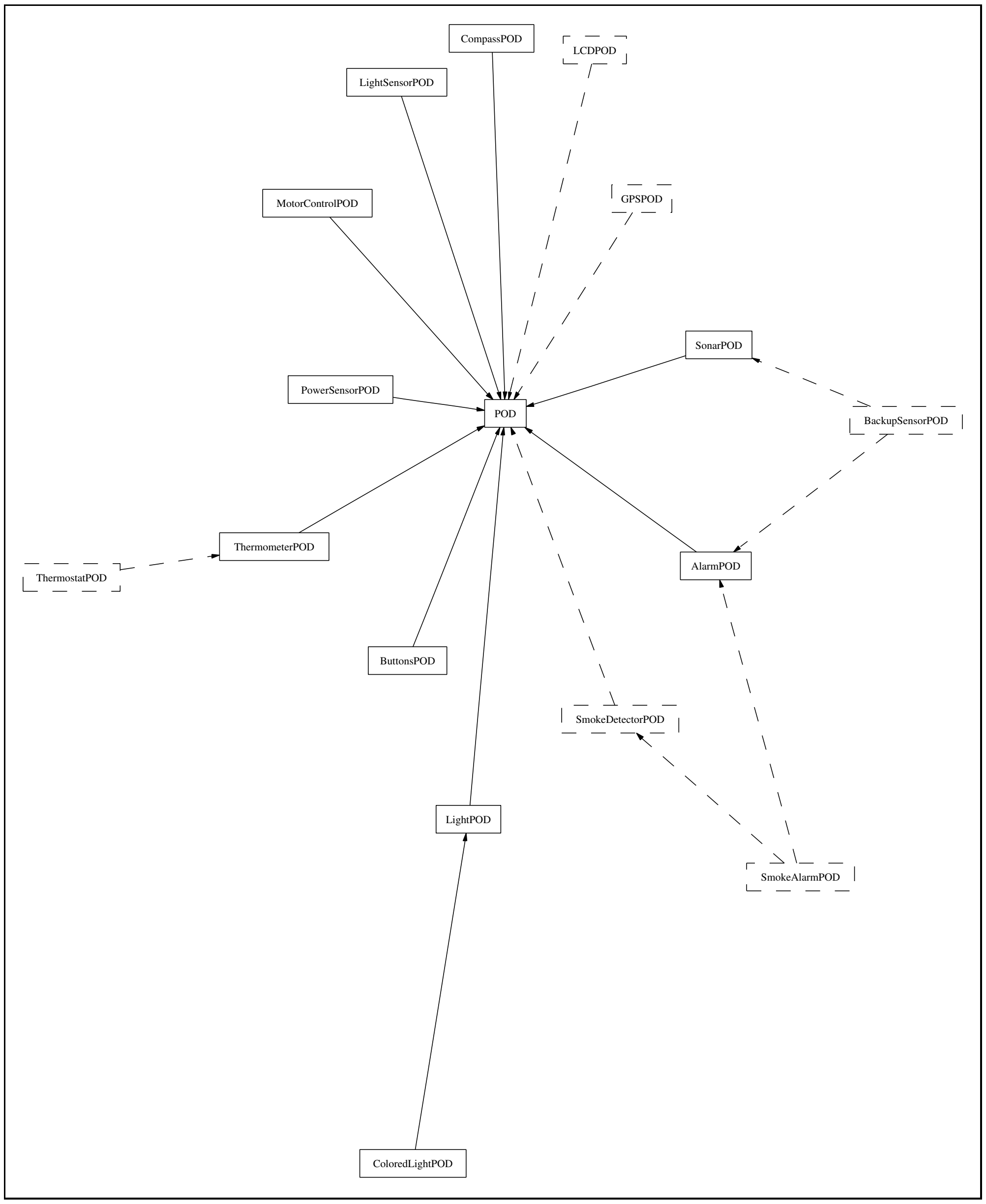


Figure 5.1: Object Hierarchy in libPOD. All PODs are derived from the POD base class, either directly or indirectly. The dashed objects and lines indicate potential PODs and their relationships.

left, or rotate. Implementing such a system in hardware results in a physical hierarchy of devices similar to a logical hierarchy of software objects.

5.3 Final Remarks

In this work, we have introduced the Physical Object Device (POD), an extensible object-oriented environment that allows simple control and communication with electronic devices. We have also described libPOD, a library designed to facilitate the use of the PODs by programmers. We believe that this simple, yet powerful, environment solves many of the challenges faced when working with electronic devices. PODs provide programmers with a user interface directly targeted for them.

Through several case studies, we have demonstrated the use of PODs and libPOD in building both simple and complex electronic systems. These case studies have also shown a number of great benefits of using PODs such as a flexible, modular design, a familiar and uniform object-oriented programming environment, the easy reuse of hardware objects, and the ability to rapidly develop complex electronic designs.

Our work has shown that by building intelligence into electronic devices and by treating these hardware objects similar to the way that software objects are used, great benefits can be realized. These benefits apply both in the creation of the PODs themselves, as well as the software that uses them. PODs are a useful method for computer control of hardware.

Appendix A

Tools and Software Libraries

As with many projects such as this, finding and using the correct tools can pose a significant challenge. In this Appendix, we briefly describe a number of tools that we found to be invaluable. Additional details can be found at our web site at `http://pel.cs.byu.edu/~sorenson/Thesis/`.

The code on the microcontrollers was written on a Linux machine, and compiled using the AVR port of the GCC tools. These tools can be found at `http://savannah.nongnu.org/download/avr-libc/released/`, and include:

- avr-binutils-2.14-1.i386.rpm
- avr-gcc-3.3.1-1.i386.rpm
- avr-gcc-c++-3.3.1-1.i386.rpm
- avr-gdb-6.0-1.i386.rpm
- avr-libc-1.0-1.i386.rpm
- avr-libc-docs-1.0-1.i386.rpm

Once these tools have been installed, the microcontroller binaries can be compiled. An example Makefile is shown in Figure A.1.

Each POD has been built with a programming header, and can be programmed using UISP (Universal In-System Programmer). This tool can be found at `http://savannah.nongnu.org/download/uisp/`, and utilizes a parallel port programming cable.

```
CC=avr-gcc
STRIP=avr-strip
OBJCOPY=avr-objcopy
LIBS=-L/usr/avr/lib/avr4
DEVICE="atmega8"
INCLUDES=-I/usr/avr/include -I/root/programs/pods/Includes
CFLAGS=-Wall -Wl,$(LIBS)

# speed at which to write to flash (3125 is probably the minimum value)
#SPEED="-dt_wd_flash=3125"

TARGET=Temperature.hex

.SUFFIXES: .c .out .hex

#.SECONDARY: Temperature.out
.PRECIOUS: %.out

all: $(TARGET)

.out.hex:
        $(OBJCOPY) -O ihex $< $@

.c.out:
        $(CC) $(CFLAGS) -mmcu=$(DEVICE) $(INCLUDES) $< -o $@
        $(STRIP) $@

erase:
        uisp -v=3 -dprog=stk200 --erase

program:
        make erase
        uisp -dprog=stk200 -v=2 --upload if=$(TARGET) --verify $(SPEED)

set_fuses:
#       Internal 8Mhz
#        uisp -v=3 -dprog=stk200 --wr_fuse_l=0xe4
#       External Oscillator
        uisp -v=3 -dprog=stk200 --wr_fuse_l=0xe0

reset:
        uisp -dprog=stk200

clean:
        @rm -f $(TARGET) *.out
```

Figure A.1: Sample Makefile to compile code for the microcontrollers.

Because our single board computer did not have a monitor attached, we found it helpful to use a program to generate audible beeps to indicate its status. We used the beep package located at `http://www.johnath.com/beep/`.

We used the CAD software Cadsoft Eagle [26] to design the schematics and boards for our PODs.

To visualize how our objects interrelated, we used the tool GraphViz, found at `http://www.graphviz.org/`.

Appendix B

POD Details

At the very lowest level, PODs communicate using a simple binary language over the bus. Since most POD users will never need to know the intricate details in order to implement or use PODs, those wishing to see these details are referred to our web site at `http://pel.cs.byu.edu/~sorenson/Thesis/`.

Each POD knows and responds to a few general commands, and POD-specific commands are added onto those. These general commands are shown in Table B.1.

Each POD type (Thermometer, Motor Control, etc.) is assigned a unique identifier to distinguish it from the other types of PODs. The POD types currently defined are shown in Table B.2. This type ID may be found using the Get POD Type command.

We created a C library to simplify the programming of the microcontrollers on the PODs. This library is accessed through small header files. These header files are depicted in Table B.3. Full source code can be found at our web site at `http://pel.cs.byu.edu/~sorenson/Thesis/`.

Command Code	Command	Details
0x0000	PING	This tests to make sure the POD is still responsive.
0x0001	Get POD Type	This command queries the POD for its type (Types are shown in Table B.2).
0x0002	Get POD Info	This command requests additional information from the POD, such as version information.
0x0003	Reset POD	This resets the POD to the powerup state.
0x0004	Get Name	This method requests the user-assigned name from the POD.
0x0005	Set Name	This method allows the user to assign a name to the POD.

Table B.1: General POD commands

POD ID	Description
0x0001	Thermometer POD (Section 3.2.1)
0x0002	Alarm (audible) (Section 3.2.2)
0x0003	Motor Control (Section 3.2.3)
0x0004	Light Sensor (Section 3.2.4)
0x0005	LED-Light (Section 3.2.5)
0x0006	Power Sensor (Section 3.2.6)
0x0007	Compass (Section 3.2.8)
0x0008	Sonar Distance Ranger (Section 3.2.7)
0x0009	Buttons (Section 3.2.9)

Table B.2: POD Type IDs

File Name	Description
Serial.h	Routines to implement higher-level communication using the USART.
ADC.h	Routines to simplify the use of the Analog to Digital Converter (ADC).
Sleep.h	Routines to place the microcontroller into the various sleep modes.
Delay.h	Implements *nop* delay loops.
TWI.h	Routines to assist in communication over the Two Wire Interface (also known as I^2C).

Table B.3: Header files used when developing POD microcontroller code.

Bibliography

[1] NASA JPL. *Mars Pathfinder Instrument Descriptions*. `http://mars.jpl.nasa.gov/MPF/mpf/sci_desc.html`.

[2] Foster-Miller. *Foster-Miller Robotics Homepage*. `http://www.foster-miller.com/robotfr.htm`.

[3] Mark Baard. AI Founder Blasts Modern Research. *Wired News*, 2003.

[4] *Lego Mindstorms Internals*. `http://www.crynwr.com/lego-robotics/`.

[5] *lego mindstorms sensor input page*. `http://www.plazaearth.com/usr/gasperi/lego.htm`.

[6] *OOPic Home Page*. `http://www.oopic.com/`.

[7] Kapil Kedia and Vincent Marshall. An architecture for a modular robot. NSF Summer Undergraduate Fellowship, 1999.

[8] Robotix. *Robotix Homepage*. `http://www.roboticsandthings.com/robotix/robotix.html`.

[9] Xerox. *Modular Robotics at PARC*. `http://www2.parc.com/spl/projects/modrobots/index.html`.

[10] Chester Fitchett and Saul Greenberg. The Phidget Architecture: Rapid Development of Physical User Interfaces. In *UbiTools '01 Workshop on Application Models and Programming Tools for Ubiquitous Computing*, 2001. Held as part of UBICOMP 2001.

[11] Saul Greenberg and Chester Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM Press, 2001.

[12] Saul Greenberg and Chester Fitchett. Phidgets: Incorporating physical devices into the interface. In M. Newman, K. Edwards, and J. Sedivy, editors, *Proceedings of the Workshop on Building the Ubiquitous Computing User Experience*, 2001. Held at ACM CHI '01.

[13] DLP Design. *DLP-USB232M USB-Serial UART Interface Module*. `http://www.dlpdesign.com/usb/usb232.html`, 2002.

[14] FTDI. *FT232BM Data Sheet Version 1.1*. `http://www.ftdichip.com/Documents/ds232b11.pdf`, 2002.

[15] National Semiconductor. *Precision Fahrenheit Temperature Sensor*. `http://www.national.com/ds.cgi/LM/LM34.pdf`, 2000.

[16] STMicroelectronics. *Dual Full-Bridge Driver*. `http://eu.st.com/stonline/books/pdf/docs/1773.pdf`, 2000.

[17] Photonic Detectors Inc. *Cadmium Sulfoselenide (CdS) Photoconductive Photocells*. `http://photonicdetectors.com/pdf/pdvp9001.pdf`, 2003.

[18] Fairchild Semiconductor. *AC Line Monitor Logic-Out Device*. `http://www.fairchildsemi.com/ds/MI/MID400.pdf`, 2001.

[19] Schurter. *Appliance Inlet GSP1*. `http://www.schurter.ch/pdf/e_d/c_gsp1.pdf`, 2002.

[20] Devantech Ltd. *Devantech SRF08 Ultrasonic Range Finder*. `http://www.robot-electronics.co.uk/htm/srf08tech.shtml`, 2002.

[21] Devantech Ltd. *Devantech CMPS-03 Digital Compass*. `http://www.robot-electronics.co.uk/htm/cmps3doc.shtml`, 2003.

[22] Ltd. Advantech Co. *PCM-5824*. `http://www.advantech.com/products/Model_Detail.asp?model_id=1-D6M0I`, 2003.

[23] Bluetooth. *The Official Bluetooth® Wireless Info Site*. `http://www.bluetooth.com/`.

[24] Zigbee. *The Official Website of the ZigBee Alliance*. `http://www.zigbee.org`.

[25] Inc. RF Monolithics. *RF Monolithics, Inc.* `http://www.rfm.com/products/vwire.htm`.

[26] Cadsoft. *Cadsoft Online: Home of the Eagle Layout Editor*. `http://www.cadsoftusa.com/`.