

**Inheritance Models in Object-Oriented Hardware  
Using Physical Object Devices**

by  
Vernon Mauery

A thesis proposal 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  
March 2004

# 1 Introduction

## 1.1 *Background*

Historically, the field of computer science has focused mostly on the theory of computing and the application of these theories in the form of software. Because of this emphasis on software in their education, when they enter industry, many computer scientists are really only computer programmers. While this is not necessarily a bad thing, it is limiting in a few ways. When programmers need to interact with any hardware other than through an operating system's many layers of abstraction, many are lost.

For those that don't have a background in circuit design there are starting to be more options available for access to embedded devices. Physical Object Devices (PODs) are one example. They offer abstraction from the hardware and essentially allow the users to treat hardware devices as software objects. PODs currently are not capable of inheritance, which is one of the facets of object-oriented design that offers a great deal of power.

## 1.2 *Research in the Area*

Object-oriented hardware is a relatively new paradigm. Some research has been done in this area to add abstraction and simplify working with hardware. Physical Object Devices (PODs) are probably the most advanced hardware abstraction architecture that we have to date (Sorenson, 2003). PODs give hardware objects a common interface and the ability to interact and connect without a soldering iron. They allow the user to treat the hardware much like a software object in the sense that PODs accept commands and queries and perform tasks on user data or data from their environment.

OOPic (Object-Oriented Programmable Integrated Circuit) is another set of devices that are meant to allow object-oriented programming with hardware (OOPic, 2001). They are able to network together using I2C and access functions that wrap hardware. However, the only facet of object-oriented programming they offer is encapsulation. They encapsulate all the hardware functions into an object and abstract the details away from the user. This is useful, but again, it lacks in some of the more powerful constructs of object-oriented design.

Phidgets (Greenberg, Fitchett, 2001) are another available hardware/software interface. Much like PODs, Phidgets have a software programmable hardware interface. Most of their work has been in applying the ideas of a GUI to hardware. They have met their goal of abstracting the hardware controls into objects a software program can use.

Lego MindStorms offer another simple solution to the hardware problem. All the parts snap together for simple electrical connections. The interface to the sensors and motors is essentially part of the programming language (it is almost entirely visual, using icons and flow charts to create program control.) As evidence of the simplicity of this solution, MindStorms are designed for ages 12 and up.

There have certainly been others that have been working in the area of object-oriented hardware so we will continue our research in the area to keep up to date with current advancements and innovations in the field.

### ***1.3 The Need for Inheritance in Object-Oriented Hardware***

Current object-oriented hardware, advanced as it is, still lacks some basic object-oriented design capabilities. So far, encapsulation, abstraction, and data protection are the only object-oriented principles designers have taken advantage of, yet these can be done in structural programming just as well. The object-oriented paradigm

has much more to offer than just these two concepts. Inheritance is one of the most powerful object-oriented constructs, but although structural programming strives to provide this functionality, it has not yet succeeded. Inheritance in hardware, like software, can be thought of in two ways: extending implementations and implementing interfaces. In Java, we would use the keywords *extends* and *implements* respectively to refer to these types of inheritance.

Computer scientists already know and understand the benefits of object-oriented programming in software. Because we are applying software ideas to hardware, we can still get some of the same benefits – and one of the most important of these is inheritance. Inheritance leads to the rapid development of more complex systems because object extension and implementation offer code reuse and common interfaces. When developers create more complex systems, they can use object-oriented hardware to make larger systems without having to completely redesign when adding or changing a subsystem; if a subsystem breaks, fixing that POD or replacing it with one of the same interface will be trivial.

## **2 Thesis Statement**

The current state of object-oriented hardware lacks some of the features, such as inheritance, that make software objects so useful and powerful. Finding parallels in software and hardware development will enable computer scientists to implement the prevalent software inheritance models in hardware objects, thus allowing for more complex POD systems and cleaner POD designs.

# 3 Methods

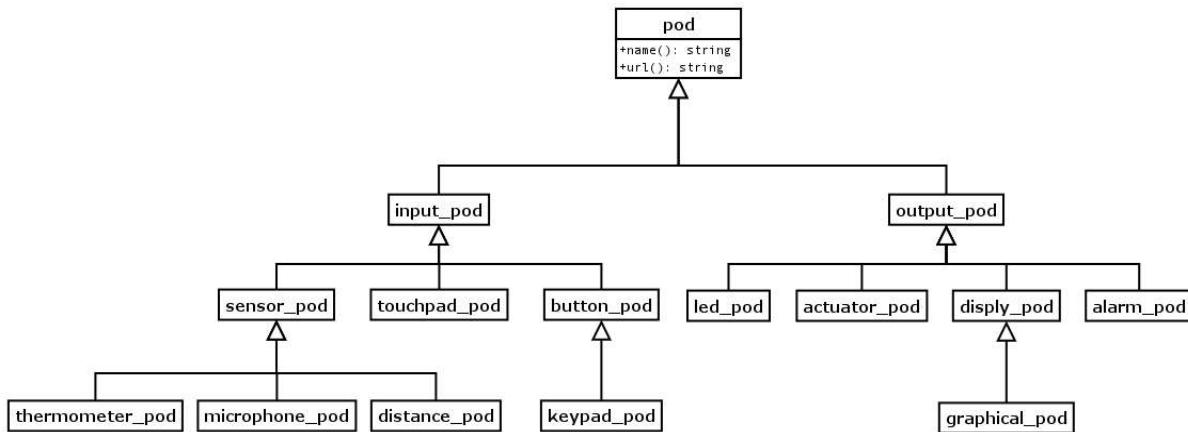
## 3.1 Framework

### POD Interface Hierarchy

In order to deal with the variety of PODs that we will encounter, we must create a versatile hierarchy for interface inheritance. Then, all PODs that implement one of these interfaces will be compatible with other PODs in the same class. Creating a good hierarchy and establishing the functions that will be required at each level of the hierarchy will be essential.

On a basic level all PODs can do the same things. This may be as simple as returning an API or a URL where the user can find more information about that POD. Deeper into the tree, all the PODs in the same class will have at least the same basic interface. This means that a Graphical Display POD can operate at the same level as a text-only POD if necessary.

Preliminary POD Interface Hierarchy



## POD Manager and Device Driver

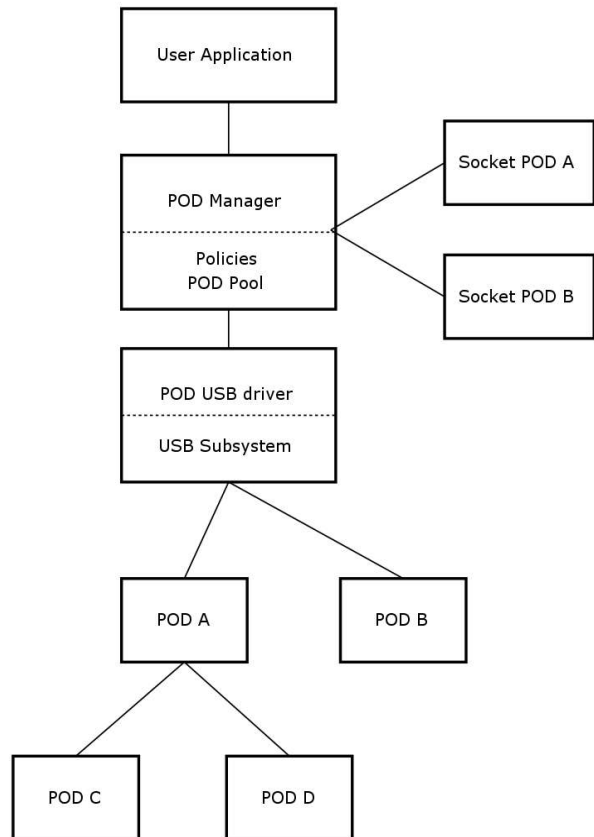
All PODs will be connected by a bus that allows inter-POD communication. The bus we have chosen for our experiments is the Universal Serial Bus or USB. We will write a Linux kernel USB driver to allow direct access to the PODs. The POD Manager will then communicate with the PODs through the USB POD driver. To keep the kernel space code small, all communication policies will be done by the POD Manager. Besides communication control, the POD Manager allows us to view the POD network as a tree of connected PODs rather than as only a bunch of virtual connections to the

host machine. The tree view can help in enforcing POD manager policies concerning PODs that are members of other extended PODs. To allow for network PODs and simulated PODs, the POD manager will also accept socket connections from PODs and allow them to communicate on the POD network. The Application then only needs to connect to the POD manager and to get PODs from the POD Pool for use as objects.

## POD Communication

POD communication on the most basic level is packet based. Each packet is composed of a header and a body of length 0 or more. The header contains information about the packet, such as type, source POD, destination POD, a command, and more information if needed. The body then would contain any additional

POD Manager / Application Framework



information that the command requires such as function parameters or the return value of a function.

## **POD Inheritance**

PODs will offer two basic types of inheritance: extension and interface implementation. As in Java, extending an existing base class gives the new subclass all the same properties plus anything else that is appropriate to add. To extend PODs, we can add new PODs as class members to extend the capabilities of the base class. Java also has interfaces that can be implemented in any way to offer API compatibility between classes. The interfaces inherit methods and members from their base classes.

### *Extension*

Extension is a natural concept for programmers. We take an existing object and add more features to it by adding new methods or even adding new objects as members. We could do this in hardware by extending one POD with another. This would create a new composite object of the base POD with another POD as a class member. The new composite object would have all the members and methods of the base class as well as added functionality from the new members and methods. The advantages of inheritance in hardware are similar to the advantages of inheritance in software. They include:

- 1.** Encapsulation/protection of member PODs
- 2.** Virtual functions to modify subclass POD behavior
- 3.** Better bandwidth usage for inter-member communication

### *Implementation*

We would like to call implementation a form of inheritance because of the inherited members from base classes in the POD hierarchy. This form of inheritance does not deal with creating bigger and better PODs, rather with making sure PODs are

compatible within each class and up the class hierarchy. A POD's interface is specified by the methods that it inherits from all of its base classes. The programmer simply fills in the bodies of the methods to do what he needs and anyone could use that POD because it adheres to the API for its class. For example, say a programmer wants to use a display with his existing POD. The display has basic functionality such as `set_cursor()`, `get_dimensions()`, and `clear()`. Any display that implements these basic functions could be used by the programmer without further knowledge about implementation or other methods this display may have implemented. Therefore, the programmer doesn't need to know the specific interface for any specific display - all the displays implement the same interface and will work fine.

### ***3.2 Experiments***

To show how well inheritance works in hardware objects, we will perform several experiments. These experiments will use PODs that exhibit the ability to extend other PODs and implement some common POD interfaces. Some of these PODs we will create and others we will borrow. The PODs we plan on using for our experiments include a Text Display POD, a Thermometer POD, a Relay POD, an LED POD, and a Button POD.

In order to know the requirements inheritance places on PODs and to better see the benefits of inheritance in object-oriented hardware, we will do some paper designs of various POD systems. We will take ten systems of PODs and show how inheritance models can benefit them and study the interactions of the PODs to refine the communication and structures that POD inheritance requires. These ten systems will come from a variety of applications that range from distributed networks of PODs to rapid prototyping designs. The variety involved will not only show how inheritance is

a useful concept, but also show that it works in more complex applications than we will actually be constructing and testing.

## **Extension Tests**

Extension tests involve testing the attributes that extension offers. The attributes that will be tested are:

1. Extension of the base class (make sure it still complies with the base class interface)
2. Encapsulation (make sure the member POD is protected)
3. Virtual functions (make sure a subclass can override the superclass methods)

### *The Feedback Toggle Button POD – A Textbook Example*

The first test is a textbook example that in its simplicity tests all three attributes yet allows observation of the outcome without the complexity of a truly useful extended POD. The Feedback Toggle Button is composed of a Button POD that is extended by adding an LED POD. The behavior of the new POD is that of a toggle button with the LED lighting up when the button is “closed.” Internally, the `is_pressed()` method of the Button POD is overridden by a new method that queries the LED POD which keeps the state for the Toggle Button. We can test encapsulation by trying to directly access the LED POD. To be sure the subclass can override the superclass's methods, we check to make sure the `is_pressed()` method returns whether or not the LED is on, not whether or not the button is actually pressed. Finally, we will test to make sure the Feedback Toggle Button POD can be accessed as though it were a Button POD to check for extension properties.

### *The Thermostat POD – The Sensor/Actuator Class*

The entire Sensor/Actuator Class of PODs is not only related in function, but in composition as well. It is very common in industry, home, and automotive environments to have a set of electronic components that compose a sensor and an

actuator to monitor and react to external stimuli. Because this class of circuit is so common, we want to show how simple they can be created using inheritance by extension with PODs.

We will create a Thermostat POD that extends a Thermometer POD with a Display POD and a Relay POD. This also shows all three attributes of extension that we are testing, while at the same time makes a useful POD with very little effort. The Thermostat POD will offer the same interface that the Thermometer POD does plus member methods to set the desired temperature, set the temperature thresholds around that temperature, and turn the heater/air conditioner off. We can perform the same tests as with the Feedback Toggle Button to make sure that all three attributes are present and show that the Thermostat POD is in fact a subclass of the Thermometer POD.

### **Interface Tests**

The concept of interface testing is much simpler than extension tests. To test the interfaces, we will create an application that will make sure each POD implemented fulfills the requirements of the interface class it belongs to. This application will use the definition of the class to test the object to be sure it fulfills the required interface. The application will give feedback as to what functions it is testing and whether or not the POD responds in an appropriate manner. This test could be very useful to POD developers to ensure they are producing PODs that comply with the POD hierarchy.

## **4 Contribution to Computer Science**

- 1.** We will show that inheritance can be applied to hardware objects with many of the same benefits that software inheritance offers.
- 2.** This research will allow computer scientists to make greater use of complex embedded hardware systems.

## **5 Delimitations of the Thesis**

We will not pursue the following items as part of our thesis work:

- 1.** A software simulation environment,
- 2.** Abstraction of the bus for the POD Manager, or
- 3.** Quality of service and bandwidth allocations of the bus.

## 6 Thesis Outline

1. Introduction and motivation (2 pages)
2. Overview of related work, referencing similar research (2 pages)
  - 2.1. PODs – Frank Sorenson's research
  - 2.2. Phidgets
  - 2.3. OOPic
  - 2.4. Lego MindStorms
  - 2.5. Other research
3. Foundational material (5 pages)
  - 3.1. Object–Oriented Software Design
    - 3.1.1. Inheritance Models
      - 3.1.1.1. Extensions – (C++ basic inheritance with or without composition, Java “extends”)
      - 3.1.1.2. Implementations – (C++ purely abstract classes, Java “interfaces”)
    - 3.1.2. Composite Models
  - 3.2. PODs Overview
    - 3.2.1. Bus
    - 3.2.2. Connection Manager
    - 3.2.3. Communication
      - 3.2.3.1. Broadcast
      - 3.2.3.2. POD to POD
      - 3.2.3.3. POD networks
    - 3.2.4. Chosen hardware – Atmel USB development boards
4. Innovations (10 pages)
  - 4.1. Extension model in hardware
    - 4.1.1. Taking existing hardware and adding new functionality by extending it with other pieces of hardware.
    - 4.1.2. A very simple hardware extension example – motor/wheel
    - 4.1.3. It is possible to think of many objects in this fashion – a computer extends a screen and adds input and processing features (a keyboard/mouse and a CPU)
  - 4.2. Implementation model in hardware
    - 4.2.1. Simplest inheritance model is interfaces, which is very useful



## 7 Thesis Schedule

<b>Thesis Research Goal</b>	<b>Target Date</b>
Continue literature search and research	Mar 2003
Get development boards and hardware working	Apr 2003
USB PODs driver written / POD hierarchy diagrammed	May 2003
Display POD designed and implemented	Sept–Nov 2003
Keypad POD designed and implemented	Nov–Dec 2003
Study how PODs fit with extension and interface inheritance models	Aug–Sept 2003
Write thesis	Oct 2003 – Jan 2004
Defend thesis	Feb 2004

## 8 Bibliography

Atmel. *Full-speed USB Microcontroller with Embedded Hub, ADC and PWM: AT43USB355*. Rev. 2603B USB. July 2002.

This documentation is the manual for the AT43DK355 development board. It explains the internals to the AT43USB355 microcontroller that our PODs will be using for control and communication. It explains the I/O mapping for pins, the alternate uses for internal registers and pins, and the USB hardware built into it.

Compaq, et al. "Universal Serial Bus Specification." Revision 1.1. 23 September 1998. <<http://www.usb.org/developers/docs/>>.

This specification details the requirements for the USB. It has information about the electrical, mechanical, and protocol specifications for the bus. It explains in detail all the vocabulary used to describe interactions on the bus. Based on this description of the bus, we decided to choose USB as the communication bus for PODs.

Fliegl, Detlef. *Programming Guide for Linux USB Device Drivers*. 18 April 2003. <<http://usb.cs.tum.edu/usbdoc/>>.

Good overview on kernel USB functions and the Linux USB internals. Gives reference to an example kernel USB driver example in the kernel source.

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997.

This book covers many design patterns that fit into three groups: creational, structural, and behavioral. For this topic, structural and creational patterns will be of most interest, particularly the Factory, Composite, State, Creational, and Template patterns. From it, we can learn about interactions between objects, how interfaces can help in object-oriented design, and different views on composite models.

Gowdy, Stephen J. *Linux USB*. 10 April 2003. <<http://www.linux-usb.org>>.

This site is dedicated to helping people get USB devices working in Linux. It has extensive documentation and links to vendors, drivers, and more. It also explains how Linux fits into the USB world. It goes into depth on the Linux USB subsystem.

Greenberg, Saul, and Chester Fitchett. "Phidgets: Easy Development of Physical Interfaces through Physical Widgets." Proceedings of the ACM UIST 2001 Symposium on User Interface Software and Technology, November 11–14, Orlando, Florida. ACM Press.

<<http://www.cpsc.ucalgary.ca/grouplab/papers/>>.

This paper focuses on the need for simple interfaces between hardware and software for programmers. It looks at GUI widgets and tries to make that same connection with hardware devices. While the paper does do well at making this abstraction, this is as far as it goes. It also deals only with user–interface objects – buttons, lights, servos, etc.

"I2C, I2S." *The Educational Encyclopedia*. <<http://users.pandora.be/educyclopedia/>>. 15 March 2003.

This page has several very descriptive explanations of the Philips I2C bus. It details bus arbitration, addressing, and other basics of communication on the I2C bus. It also helped in making the decision on which bus to use for PODs communication.

Kroah–Hartman, Greg. "How to Write a Linux USB Device Driver." *Kernel Korner*. 1 October 2001. <<http://www.linuxjournal.com/article.php?sid=4786>>.

Greg, the Linux USB subsystem maintainer, shares his view on how to customize the skeleton driver that is available in the Linux source. A good second reference to see another point of view on USB programming in Linux. We found the usb–skeleton driver he wrote very useful in writing our own USB driver for PODs.

Lego. *Robotics Inventions System 2.0*. 2000. The Lego Group.

This manual provides instructions on how to program the hardware for the MindStorms and gives some general descriptions of how things work. The Lego MindStorms website has more information on this topic (link from manual is <http://www.legomindstorms.com/>).

Neswold, Richard M., Jr. *A GNU Development Environment for the AVR Microcontroller*. 17 April 2003. <<http://users.rcn.com/rneswold/avr/>>.

This page has a plethora of information on how to get the GNU–AVR tools up and running. This includes AVR–GCC, AVR–Binutils, and AVR–LibC. Together, these are the development environment that we chose to develop PODs in. The software is free, reliable, and open–source. The site also has informative tutorials to help test the tool–chain. This was a very important source of

information at the beginning of the research phase when we were trying to get the hardware and software up and running.

Savage Innovations. "OOPic Home Page." 2001. <<http://www.oopic.com/>>.

This site tells about Object Oriented Programmable Integrated Circuits, an innovative way of looking at hardware. The devices are programmable in Java, C/C++, and Basic. The development libraries associated with the devices export an encapsulation of the hardware as an abstraction layer. In this way, it simplifies the access to the hardware. It offers the ability to network with other OOPic devices using an abstraction of I2C. It pinpoints the need for encapsulation and simplicity in programming but makes the over-generalization of object-oriented design as simply encapsulation.

Sorenson, Frank. "PODS: PEL Object Devices." Diss. Brigham Young University. 2003.

This paper tells of need for abstraction from hardware and object-oriented devices. It tells how PODs could be used to create more complex systems. It explains the basic interface of PODs and some of the possibilities that could be researched in the future.

## 9 Artifacts

1. User space PODs driver using libUSB
2. Alternate USB PODs device driver
3. USB PODs device manager
4. PODs interface hierarchy
5. Example PODs

### 5.1. Implementations

#### 5.1.1. Display POD

**5.1.1.1.** Text Display POD – implements Display POD Interface, text only display

#### 5.1.2. Input POD

**5.1.2.1.** Keypad POD – implements the Keypad POD Interface, 16 key keypad

#### 5.1.3. Output POD

**5.1.3.1.** LED POD – implements Output POD interface, controls lighting an LED

#### 5.1.4. Sensor POD

**5.1.4.1.** Thermometer POD – make sure Frank Sorenson's POD works as a Sensor POD

#### 5.1.5. Actuator POD

**5.1.5.1.** Relay POD – implements Actuator POD interface, controls an electrical relay

### 5.2. Extensions

**5.2.1.** Feedback Button POD (using a Button POD and an LED POD)

**5.2.2.** Thermostat POD (using a Thermometer POD, Display POD, Actuator POD, and Keypad POD)

**Brigham Young University**

**Graduate Committee Approval**

of a thesis proposal submitted by

Vernon Mauery

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

\_\_\_\_\_  
Kelly Flanagan

\_\_\_\_\_  
Date

\_\_\_\_\_  
Dan Olsen

\_\_\_\_\_  
Date

\_\_\_\_\_  
Dan Ventura

\_\_\_\_\_  
Date

\_\_\_\_\_  
David Embley, Graduate Coordinator

\_\_\_\_\_  
Date