Carleton
UNIVERSITY

Automotive Computing with Game Console Hardware

Pat Suwalski (294246)
2005

Supervisor: Professor G. Wainer

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

April 2005

Abstract

This project describes a general-purpose automotive computer based on the Microsoft Xbox game console. Several components that can be used to construct a general-purpose automotive computer are designed and implemented. While current solutions are likely to use a highly-integrated and simple client, this project proposes the concept of a fully-functional mobile server. Consideration of safety and usability research throughout the design results in a unique product.

Contents

1 Introduction **1**
- 1.1 DashBox . . . 1
- 1.2 Microsoft Xbox . . . 3
- 1.3 System Overview . . . 4
- 1.4 Report Organization . . . 6

2 Software Design **7**
- 2.1 DashUI . . . 7
  - 2.1.1 Secondary Window Manager . . . 8
  - 2.1.2 Simple Configuration . . . 9
  - 2.1.3 Usability . . . 10
  - 2.1.4 Program Design . . . 10
- 2.2 Vehicle State Daemon . . . 11

3 Hardware Design **13**
- 3.1 Power Control Unit . . . 13
- 3.2 Binary State Sensor . . . 18
- 3.3 Miscellaneous Hardware . . . 19
  - 3.3.1 USB Dongle . . . 19
  - 3.3.2 VGA Adapter . . . 21

4 Human Factors & Safety **22**
- 4.1 Hardware Interaction Concerns . . . 23
- 4.2 Software Interaction Concerns . . . 25

5 Results **27**
- 5.1 Touch-screen Interface . . . 27
- 5.2 Usable Software . . . 29
- 5.3 Other Peripherals . . . 30
- 5.4 Web Page . . . 31

6 Conclusions **32**

A ATTiny2313 Introductory Material **35**

B Weblog **40**

List of Figures

1.1 DashBox Logo 2
1.2 Microsoft Xbox 3
1.3 Top-level system design 5

2.1 DashPC interface 8
2.2 DashUI interface 9

3.1 Power Control waveforms 14
3.2 Power Control circuit schematic 15
3.3 Power Control execution flowchart 16
3.4 Power Control test implementation 17
3.5 Binary State Sensor schematic 18
3.6 State Sensor test implementation 19
3.7 USB Dongle 20
3.8 VGA Adaptor 20

4.1 Driver control reach envelope 24

5.1 Center console touch-screen design 28
5.2 Comparison of GTK Themes 30

Chapter 1

Introduction

In recent years, computing devices designed to be embedded in vehicles have become fairly commonplace. Their general purpose ranges from Global Positioning System navigation to passenger entertainment. However, there is a growing movement of hobbyists who are interested in installing standard computing devices in their vehicles because they:

- cannot afford a vehicle with a built-in computer;
- stand to gain from having a more sophisticated device.

The goal of this project is to investigate and build upon the latter point. Specifically, using standard personal computer parts is a low-cost and highly-flexible method of testing new ideas and expanding upon existing ways.

Current car-computing devices are generally not very powerful or expandable: they are very integrated and have a very closed architecture. In many ways they are little more than a PDA (personal digital assistant) built into the dashboard. From a systems point-of-view, they are simple devices fulfilling the role of clients for simple tasks.

1.1 DashBox

This project, codenamed DashBox, aims to take the opposite viewpoint. Whereas car computing traditionally uses embedded hardware, DashBox uses off-the-

Figure 1.1: DashBox Logo

shelf computer components. A closed architecture is replaced with the most widely deployed open architecture, Linux. Whereas traditional solutions are not expandable, DashBox can accommodate any number of software and hardware extensions.

The sum of these differences is that contrary to the embedded client model, DashBox is a server that serves any number of applications to any number of clients, be they PDAs, laptops, or a client to itself via a direct touch-screen interface. Vehicles are large enough to support more than highly-embedded devices. DashBox proposes the concept of a *mobile server*.

The problem underlying this project is that "serious" car-computing appears to currently be limited to hobbyists who have little regard for standardization or configurability. A prime example of this philosophy is DashPC[1], with a hard-coded interface and little thought to usability.

From the hardware standpoint, there is little freely-available information for constructing hardware to support the automotive computer.

DashBox goal is to provide a solution to the aforementioned problems by being completely open-source. The philosophy is to provide the simplest tools to accomplish the task. All hardware designed is composed of readily available parts and relatively cheap to construct. To further lower the cost, amongst other things, the feasibility of using readily-available gaming consoles as the

[1] http://www.dashpc.com

Figure 1.2: Microsoft Xbox

the central computer is considered throughout.

A significant portion of this project is to accomplish the goals using game console hardware. At first, this may seem unusual, but game consoles have several benefits over regular personal computers:

- Durability: they are designed to withstand the rigor of children;
- Low power consumption: not designed to carry add-on hardware;
- Relatively powerful for price.

The first two points are clearly beneficial to a vehicle environment, and the third is a bonus which comes from the processing-intensive nature of current video games.

1.2 Microsoft Xbox

The Microsoft Xbox [Figure 1.2] is a modern gaming console built from partially proprietary, but mostly standard x86 hardware. Because of this, its

main strength is its price and compatibility with common third-party hardware. While its BIOS is designed to boot into a gaming console that understands the Xbox game disc format, it can be re-flashed to accommodate a Linux bootloader, which allows the unit to function as a standard computer.

With its relatively cheap hardware and ability to do more than it was designed to do, the Xbox is a good candidate for automotive computing. It has four USB ports, a network interface, an optical disc reader, a hard drive, as well as video output. From a hardware perspective, the Xbox has all of the components needed to allow for user interaction, as well as data collection from the vehicle.

At this point in time, Xbox hardware has been well documented. A stock Xbox requires only slight hardware modification to allow for alternate software. The process, including the hardware encryption present on an Xbox, is explained in detail in Huang's *Hacking the Xbox*[1]. A working Debian-based Linux distribution, Xebian[2], specifically for the Xbox is available.

1.3 System Overview

The result of research into the problem domain has led to the top-level system pictured in Figure 1.3. In this design, the Microsoft Xbox is at the center of the system. Any number of peripherals are attached to it through its Audio/Video, USB, and network jacks. The entire system is integrated so that when the vehicle is turned on, the system boots and no additional tasks are required to make it work.

To achieve the desired design, the project was divided into small components, both hardware and software. This approach is beneficial for unit testing the individual components prior to system integration.

Specifically:

- a software user interface was designed and implemented;
- a hardware power control unit was designed and implemented;

[2] http://www.xbox-linux.org

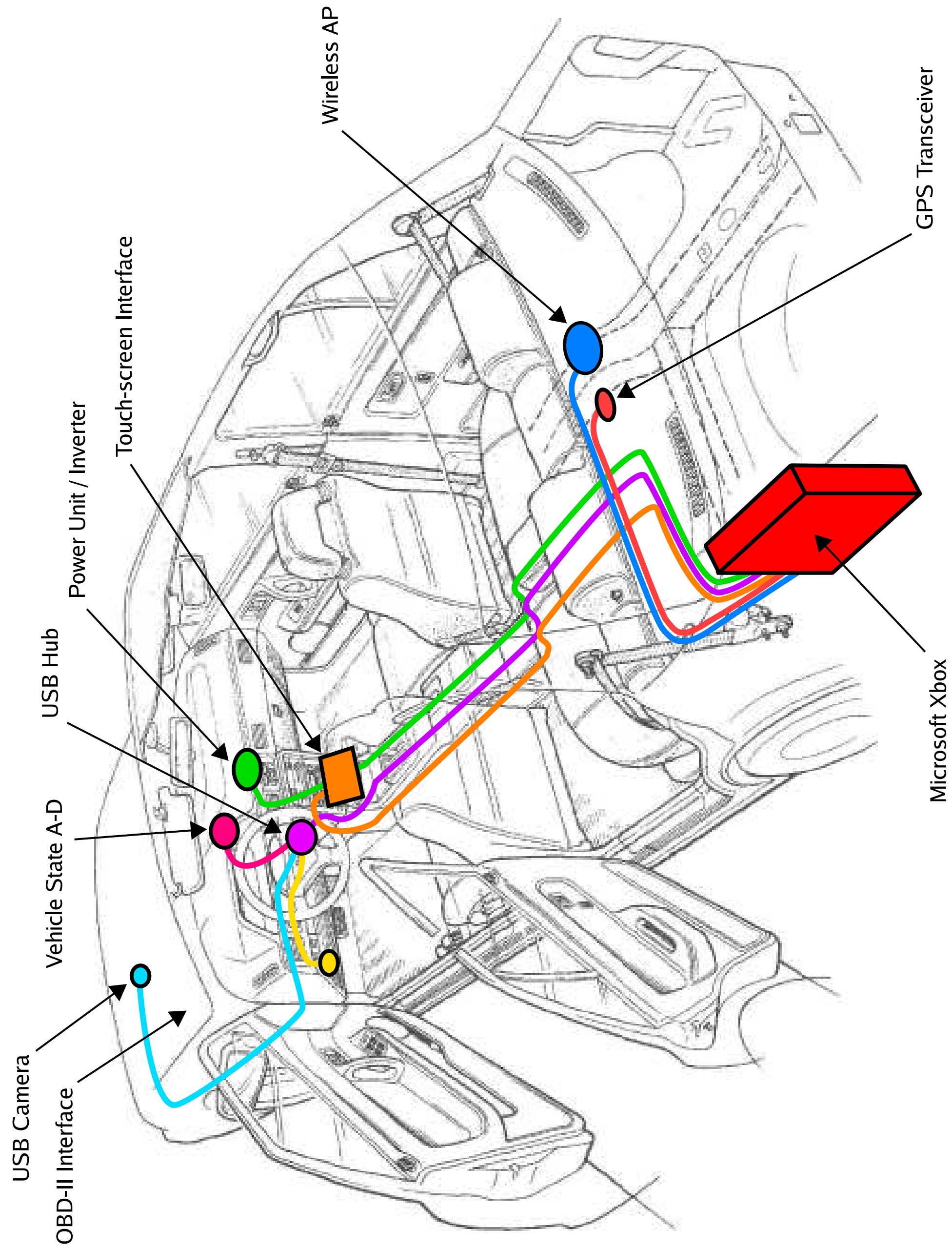


Figure 1.3: Top-level system design

- hardware and software to read the state of equipment within a car was designed and implemented;
- the topic of human factors, including safety, was researched;
- several peripherals beyond those designed were integrated and tested.

1.4 Report Organization

This report is organized into chapters and sections. The chapters immediately following this introduction explain the technical aspects of the system design. These are followed by a research chapter about designing with a focus on human-to-system interaction, especially with regard to safety. The culmination of the technical and research chapters is explained via the Results chapter. Finally, ther are concluding statements and various appendices.

Chapter 2

Software Design

The initial intent of this project was to use existing software with new hardware combinations. The initially-proposed software was DashPC[1], a simple graphical program launcher with large buttons, designed to be used via touchscreen. However, the latest version of DashPC (0.45) quickly proved inadequate in terms of configurability, user interface design, as well as expandability. Specifically, all of these criteria are problematic because DashPC hardcodes everything in its binary. The interface is based on fixed-image panes, as can be seen in Figure 2.1.

2.1 DashUI

Based on the analysis above, a new design was sought, with emphasis on ideals of usability and configurability. The software project is named DashUI. Specifically, goals of the design are that it takes the role of a secondary window manager, it has a simple configuration mechanism, and its interface places a high priority on usability.

[1] http://www.dashpc.com

Figure 2.1: DashPC 0.45 interface when first started

2.1.1 Secondary Window Manager

The unique feature that makes DashUI different from other interfaces is the way it handles launched programs. The interface consists of large tabs, each containing an instance of a program. While other interfaces are simple launchers, DashUI actually *swallows* the programs it launches into the tab sheets. This interface can be seen in Figure 2.2.

Effectively, DashUI becomes a secondary window manager. It occupies the entire screen and allows easy switching between programs by selecting the appropriate tab on the driver's side. The advantage of DashUI not playing the role of window manager is that should programs actually need to show transient ("pop-up") windows, they still have the familiar titlebar and controls. However, with usability and safety in mind, transients should be exception rather than the rule.

The majority of the swallowing code is based on a small Gnome applet called gnome-swallow[2], with extra wrapper code to allow for multiple tabs

[2]http://interreality.org/ tetron/technology/swallow

Figure 2.2: DashUI interface with swallowed movie player

and configurability. The swallowing code was also optimized and updated for recent compilers, including a bug fix submitted up-stream. These changes make the swallowing code suitable for DashUI.

2.1.2 Simple Configuration

With its unique tabbed interface, DashUI requires a method for the user to specify the contents of each page of the tabs. A simple configuration file, which conforms to the IniFile convention, can be used for this. A flexible parser was written for this purpose. The configuration file is made up of any number of blocks. A tab is created for each new program block. An example block, containing instructions on how to swallow the X-Eyes demo application follows:

```
[xeyes]
program = "/usr/bin/xeyes"
window = "xeyes"
icon = "applet3-48.png"
resize = 1
```

In this example, the swallowed window has the *window* title `xeyes` belonging to the *program* binary `/usr/bin/xeyes`, has an icon specified by the PNG file, and is resized to fill the entire tab sheet (much like the Totem media player in Figure 2.2). Based on these settings, the program flow is as follows:

1. create a new tab with icon `applet3-48.png`
2. execute the binary `/usr/bin/xeyes`
3. search for a window that is labelled `xeyes`
4. obtain window control from the X server
5. reparent the window to the area in the new tab
6. resize the swallowed window to match the size of the tab sheet

2.1.3 Usability

DashUI's design is specifically such that once an individual has set their configuration file, the swallowed applications can be accessed without much thought. Because of the even spacing of tabs along the edge of the touch-screen closest to the driver, knowing where to push becomes second nature.

Lighting considerations are important for vehicle computing, as light levels change dramatically. As such, it is important that all user-interface elements stand out as clearly as possible. A nearly monochrome, high-accessibility theme like the GTK theme *HighContrast* works well. This theme would benefit from having thicker lines around controls. However, it is generally preferable as-is over other themes.

2.1.4 Program Design

DashUI is written in C and uses GTK+[3] for the user interface. It operates using POSIX threads. The main thread is the interface as seen by the user.

[3]Gimp Tool Kit, http://www.gtk.org

When new programs are launched into tabs simultaneously at start-up, each program is launched and swallowed from a child thread. There are two benefits to this approach:

1. the swallowed programs are terminated when the main program closes;
2. all tabs are loaded in parallel, providing better start-up time.

A third thread is maintained to stay in constant communication with the Vehicle State Daemon, described below.

2.2 Vehicle State Daemon

It is useful to know the state of various devices within a vehicle. To that end, a simple piece of hardware based on a USB mouse was constructed to read the 12V states native to the vehicle. Specific information can be found in Section 3.2. The goal of the daemon, which is the term for a program that executes in the background to perform any number of utility functions, is to read changes in the state of the mouse, which is equivalent to reading state changes in the vehicle.

The daemon is a short program written in C. Its output is a standard UNIX IPC pipe, with DashUI occupying the other end. Changes in mouse button states are passed through this pipe, and it is up to DashUI to react accordingly. The program efficiently avoids polling the mouse by simply opening the mouse device node with the blocking flag. Therefore, only as new data packets are sent by the mouse does the daemon process and pass the information through the pipe. This program was designed as an external daemon so that is it an optional element of the DashUI interface; DashUI will run without it.

Currently, DashUI reacts to mouse button 1 by tinting the screen. The intended goal for this behaviour is to minimize glare from the computer screen when driving in the dark. To accomplish this, the 12V signal corresponding to headlight state is read. This is the same signal that is often passed to clocks and radios to dim their display. The screen is tinted red using existing

X-Windows extension *XFree86-VidModeExtension*, designed, amongst other things, for colour gamma-level manipulation. When the signal drops, the screen brightness and colour are returned to normal.

Chapter 3

Hardware Design

Standard computing devices are designed to be stationary units, powered by 120 or 240 volts, operated directly by a human. When the conventional computer is to be used within a vehicle where there is a 12V supply and limited direct interaction with the hardware, there are several extensions to its hardware that must be considered, and others that provide for better integration. This chapter discusses several simple designs that make it viable to use a standard computer or gaming console alike.

3.1 Power Control Unit

When using conventional computing devices, it is important to have definitive control over their power consumption. Unlike embedded devices, they can drain a car battery if left on overnight.

The first step to enforcing good behaviour is to notify the computer of when it is safe to turn on and turn off. With ACPI (Advanced Configuration and Power Interface)[1], it is possible to allow for clean software shutdown. For example, when the power button is pushed, the computer can be informed that it must begin its shutdown sequence. However, this is not sufficient as there is always the possibility of an unexpected malfunction in the shutdown

[1] ACPI4Linux, http://acpi.sourceforge.net

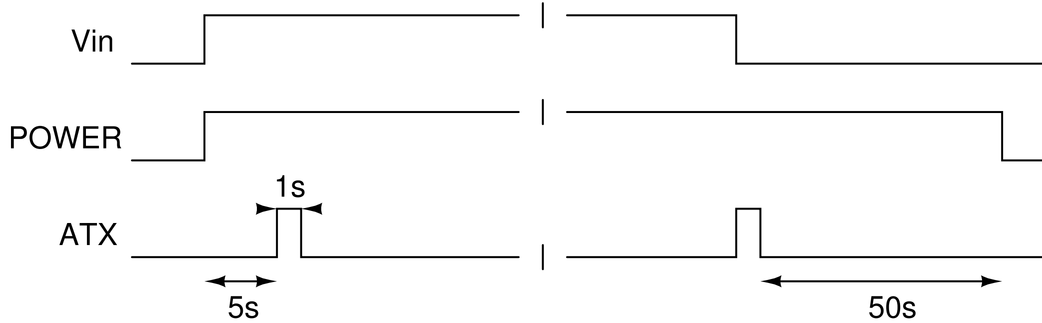


Figure 3.1: Power Control input signal Vin controlling outputs

sequence that would prevent the computer from switching itself to a low-power (off) state.

The solution to this potential problem is to externally control power to the unit. A high-reliability system that physically enables power, turns on the computer, soft-shuts down the computer, and then eventually removes its current supply altogether is the answer.

The designed Power Control module uses an input signal Vin to control two output signals, called POWER and ATX [Figure 3.1]. The signal represented by Vin is the same 12V signal that informs vehicles appliances such as radios to turn on when the ignition switch is turned. Therefore, once the ignition is on, there is a 5 second countdown until the ATX power switch is shorted, causing the computer to turn on. This time should be enough for power usage throughout the vehicle to steady so as not to strain the alternator. The computer is provided power for an indefinite period of time. Once the ignition signal pulls low, the Power Control unit once again shorts the ATX signal, which begins a soft-shutdown of the computer. After 50 seconds, by which point the computer should be safely powered off, the POWER signal is dropped, removing the computer's supply current.

The implementation of the Power Control unit consists of an Atmel ATTiny2313 microcontroller driving two relays, one for the output signal POWER, and the other for ATX [Figure 3.2]. The ATTiny chip contains two 7-bit I/O ports. See Appendix A for an overview of the pin layout and I/O port infor-

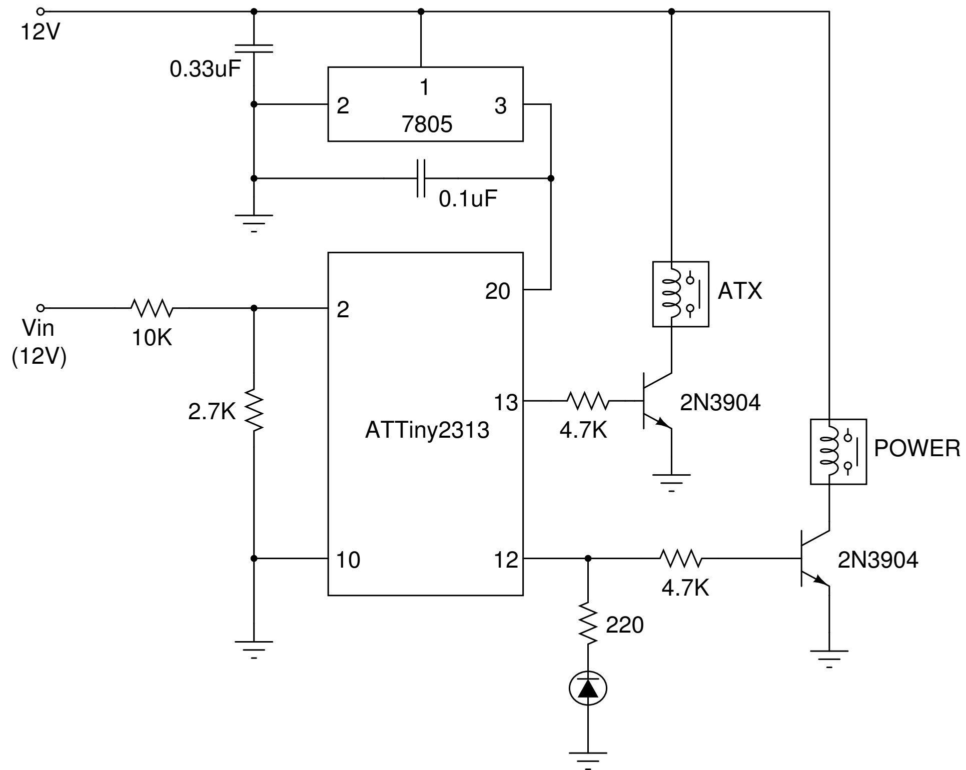


Figure 3.2: Power Control circuit schematic

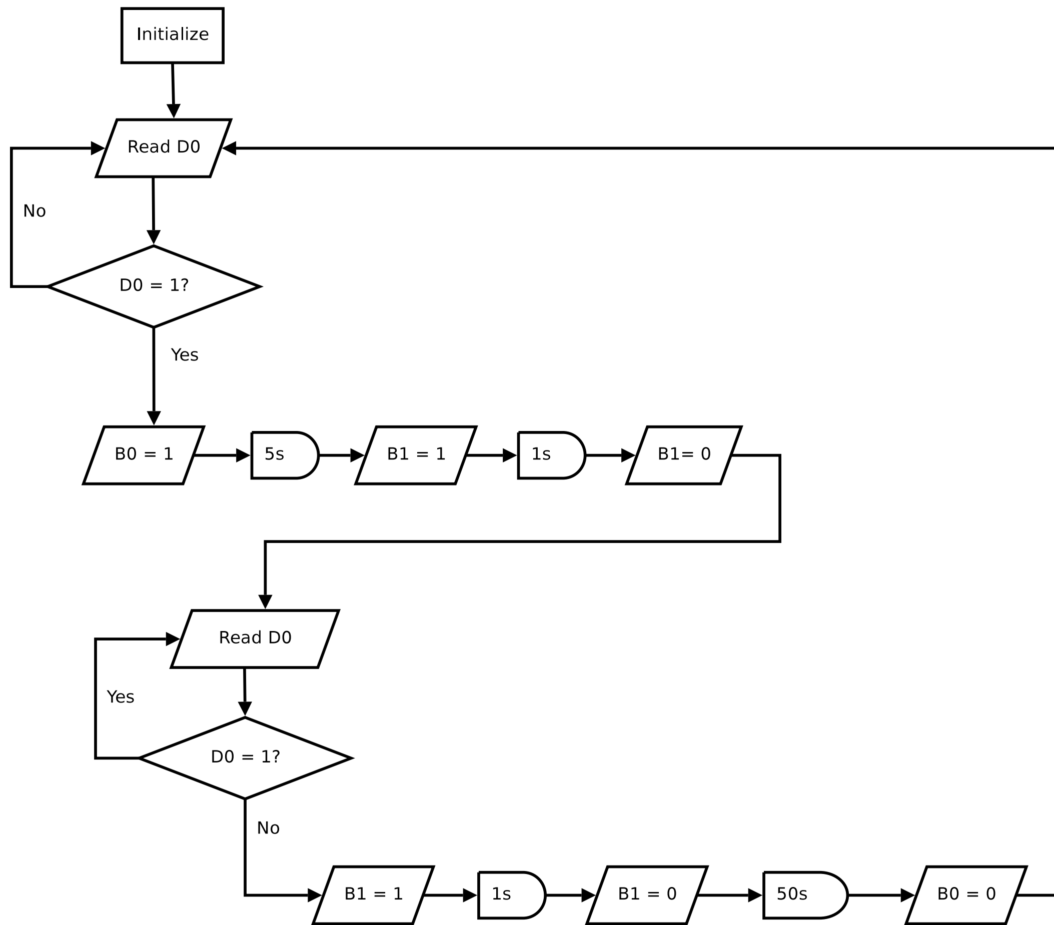


Figure 3.3: Power Control execution flowchart

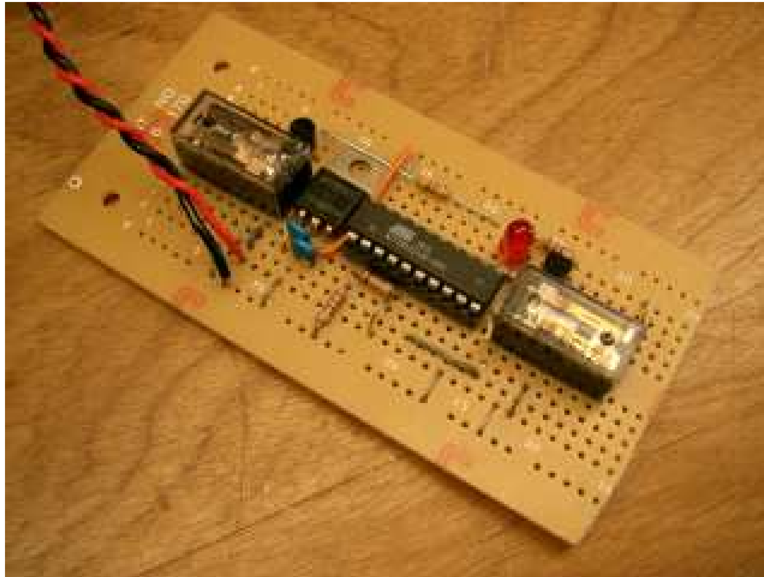


Figure 3.4: Power Control test implementation

mation for the ATTiny2313. One port is used to read the state of ignition (pin 2), using a voltage divider to achieve the correct voltage. Both outputs (pins 12 and 13) control 2N3904 NPN transistors, which in turn drive 12V relays. In the case of the POWER relay, it controls one of the power wires to the computer power supply, or 120V inverter as the case may be. For the ATX signal, the relay simply shorts the power button header pins provided on the ATX-compliant motherboard. The entire ATTiny is fed by a MC7805 5V voltage regulator. A LED is provided to display the state of the POWER signal, though it is not required.

The program execution can be modeled as a flowchart depicted in Figure 3.3. Input and output operations are specified by port letter and bit number. Therefore D1 is Port D, bit 1, or pin #13 on the microcontroller.

The Power Control Unit design was proven by a hardware test implementation [Figure 3.4]. Although the relay designated for reacting to the POWER signal is potentially too small to carry the current requirements of a computer, it can be replaced with any larger 12V relay that can be driven by the transistor. The board demonstrates that the design works.

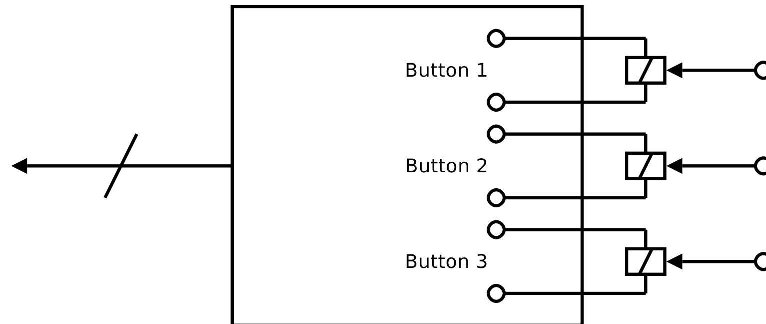


Figure 3.5: Binary State Sensor schematic

3.2 Binary State Sensor

No car-computing device is complete without some level of integration with its vehicle's systems. Much of what such a machine does is in response to the state of the vehicle. Automobiles generally maintain states of various components as logic levels 0V and 12V. For safety, these signals need to be electrically isolated from the computer. In most implementations, the states are read through a relay system over a parallel or serial port on the computer. In the case of the Xbox and many new small desktop computer motherboards, this cannot work, as they do not support either of these legacy ports.

A simple solution to the problem is to build upon previous solutions involving relays and serial ports. Simplifying further, a USB computer mouse provides all of the required functionality with an easy to decode interface. The initial design is pictured in Figure 3.5, with relays on the right shorting the contact pads where the mouse buttons normally are. Using the signals from the mouse as a sensing interface is described in more detail in Section 2.2.

The design specified in Figure 3.5 was constructed from a Cicero USB mouse to produce what is seen in Figure 3.6. The mouse, connected to the Microsoft Xbox via a USB-to-Xbox converter cable allows the state of the relays to be read. Using 12V relays, any signal in the car can be read relative to ground.

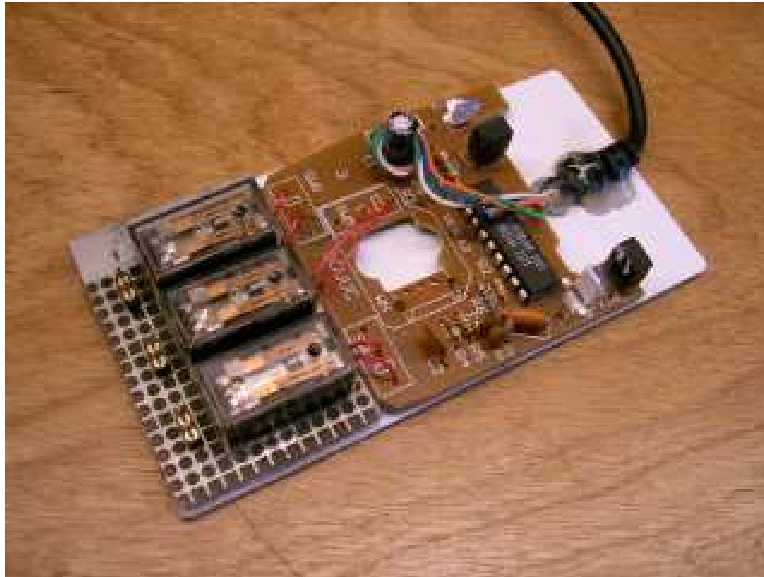


Figure 3.6: State Sensor test implementation

3.3 Miscellaneous Hardware

Working with the Xbox and standard peripherals necessitates the construction of several miscellaneous devices. These are for the most part simple pass-through adaptors that are placed between the Xbox and a peripheral to allow the two to communicate.

3.3.1 USB Dongle

The ports on the front side of Microsoft Xbox implement a standard USB interface despite their proprietary connector. To be able to interface with the unit, a USB converter cable was designed, as pictured in Figure 3.7 with USB flash memory.

The dongle is a passive device. In fact, the pins are connected straight-through, pin-for-pin.

Figure 3.7: USB Dongle

Figure 3.8: VGA Adaptor

3.3.2 VGA Adapter

For the Xbox to be able to output to a high-resoution video device, such as a touch-screen, a standard set of VGA signals is required. While the Xbox cannot produce VESA-compliant VGA output, because the DAC (digital-analog-converter) is tuned for television output, enough signals are present so that with the help of a sync-converter microchip, most VGA screens are able to display high-resolution (640x480 and up) output.

The converter [Figure 3.8] is based on a design provided on the Xbox-Linux site[2]. While the Xbox only outputs red, green, and blue signals. The green signal also carries a composite sync signal. The adaptor contains a LM1881N chip, which generates proper horizontal- and vertical-sync signals. The output of this adaptor is nearly VESA-compliant, differing only in voltage levels. However, most modern display devices can cope with this.

Added value from this dongle comes from the fact that the Xbox output connector (AVIP) contains both video and audio lines. For the purposes of car computing, this is beneficial because both can be carried to the front of the cabin through one cable, where they are both likely to be used. The adaptor in Figure 3.8 contains an audio connector.

[2] http://www.xbox-linux.org/Xbox_VGA_HOWTO

Chapter 4

Human Factors & Safety

Human factors are a very important concern when considering automotive technology, especially an interactive process such as computing.

The concern over human usability factors as they apply to ergonomics and safety has been heavily studied over the past decade or so, as a multitude of new interfaces for new technologies need to be considered. This chapter makes references to several of these studies.

Whenever there is discussion of safety regarding using technology within the car, there are often strong opponents whose position is that the driver must be completely concerned with the task of driving at all times. There are also those who believe that good judgment permits for a certain level of distraction, for example, the use of a mobile telephone. In a study conducted in the mid-1980's, researcher Hughes Cole concluded that between 30 and 50 percent of visual attention of a driver can be allocated for tasks not driving-related. Their basis for this conclusion was that most drivers do not utilize their full mental capacity when driving, and that only 50 to 70 percent concentration is required to be safe[2]. It must be noted that Hughes and Cole refer only to visual attention and their study predates the wide deployment of mobile telephones, which are currently being scrutinized for causing many distraction-related accidents. However, the system being considered here concerns visual multitasking above all else. The system is far less personal than a telephone conversation, its likelihood to produce emotional response and distraction far

lower that of a personal communication.

4.1 Hardware Interaction Concerns

Vehicle-driver ergonomic concerns are a broadly researched area in recent years. Researchers tend to focus on aspects of visibility and reach as it relates to drivers of all sizes and impairments. Recent advances in computer technology have researchers considering trade-offs of various possible computer display technologies and their physical situation within the vehicle. As early as 1993, in-dash computer navigation systems were actively tested (see *Antin, 1993* [3]).

For the purpose of this project, which aims to be useful as an add-on technology to a pre-fabricated vehicle, overall dash panel layout must be considered fixed. The best solution is one that can be integrated such that it follows as many safety and usability guidelines as possible. For the purpose of this project, the 1994 Mazda Protegé is considered.

The most suitable location for a small liquid-crystal display is the center console, as it could benefit both the driver and a passenger. Considering the use-case where a driver is accompanied a navigator on a road trip, safety could be higher than with the navigator using a paper map which can occlude the driver's vision. However, in considering that the system would often be used by a single occupant, the driver, there are several concerns that must be addressed. In researching placement of mirrors and electronic screens, Flannagan and Sivak concluded that:

> For horizontal extent of eye movements, an 'optimal' range is given as 15 degrees left or right of the principal line of sight, and an 'acceptable' range is given as 30 degrees left or right. For vertical extent of eye movements, the 'optimal' range is from 15 degrees up or down, and the 'maximum' range is 45 degrees up to 65 degrees down. [4]

The large difference between "optimal" and "maximum" ranges can be at-

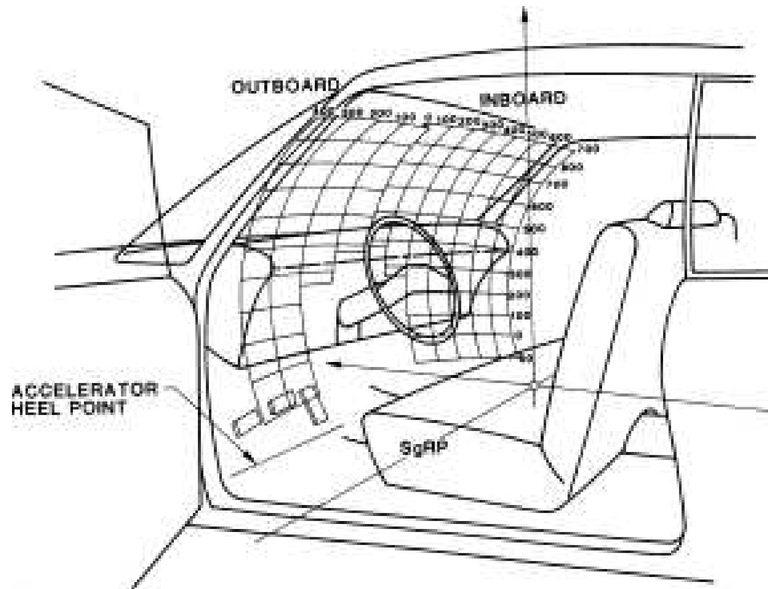


Figure 4.1: Driver control reach envelope

tributed to their observation that "People are good at monitoring the periphery of their visual field for potentially important events" [4]. The horizontal angle from the driver's head looking forward and toward the center console is largely proportional to the width of the vehicle. Therefore, large vehicles could be problematic with this approach. For smaller vehicles, the 30-degree range can be satisfied. Vertical placement is completely dependent on the vehicle design and optimal placement of a screen. However, Figure 4.1 is the outcome of a study by Roe in 1972, showing that the center console is properly within reach of the driver at most vertical levels in a typical car.

In considering the concern about video screen placement, Wierwille suggests:

> It is generally recognized that as the angle between the forward view and the in-car task increases, transition time for the eyes also increases. Furthermore, accommodation time (time to refocus from the forward scene to the in-car task) increases with change of distance from the viewer and also with age. Finally, the further down into the car the task is, the less likely it is that peripheral

> vision could be used to detect a hazard in the forward scene. These well-known results suggest that it is best to locate the in-car task display as high on the IP [(instrument panel)] as possible (or even above the IP) and the focus distance to the in-car display should be further away rather than nearer. [6]

The significance of this statement regards human eye response in addition to the findings of Flannagan and Sivak. The task of focusing ones eyes between the road and a screen is tedious, and increasingly so as eye motion increases. At the same time, Wierwille's statement is applied to the panel design of small cars such as the Toyota Echo, whose instrument cluster is at the center of the panel. Toyota's ergonomics designers may have considered the trade-off of eye motion versus forward-visibility when choosing this design. Whatever their intentions, it is clear that the center console design is viable.

4.2 Software Interaction Concerns

Software usability is currently a prime concern for many software designers. Recent trends have proceeded beyond simply creating an attractive interface to actual studies of highly accessible user interfaces. These studies often result in standard guidelines to be used throughout a project, as is the case with the Gnome Human Interface Guidelines[1]. For automotive purposes, clear design is even more crucial and should satisfy simplicity above all else. In addition to purely software concerns, the effects of light and glare can affect how the software is perceived by an individual. In considering controls drivers operate, Graesser and Marks find that:

> The time to find a control should be faster when there is a word, letter string, or graphic symbol that distinctively conveys the function of the control. The time should be faster when there is a contrast in color, texture and brightness between the controls and the background. [7]

[1] http://developer.gnome.org/projects/gup/hig

Since texture on a video screen is not a variable, efforts must be focused on using high contrast with brightness appropriate for driving conditions, and using symbols effectively. A quicker response time is desirable because it accounts for less concentration expended to perform the overall task.

Tight coupling between hardware and software usability is required to produce a powerful, but safe system.

Chapter 5

Results

This section of the project culminates the research and design of software, hardware, and human factors, and combines them into a possible solution. The overall model used is the one in Figure 1.3. At the center of the system is the Xbox loaded with Xebian Linux and attached to other hardware via its custom peripherals.

5.1 Touch-screen Interface

In considering all possibilities for direct interaction with the system, the most feasible is via a touch-screen. Other options, including a wireless keyboard or voice recognition are less suitable.

Placement of the touchscreen is a serious concern due to safety issues. As stated in previous chapters, it is important that the screen be as high up as possible and as far away from the user so that the eye is not forced to look away and refocus. The other criteria for a touch-screen is that it be easy to reach.

A rendering of the chosen design is depicted in Figure 5.1, based on the interior of a 1994 Mazda Protegé. The idea is to use the space left in most consoles for compact disc changer. This area is often closed off or made into a small storage compartment. However, it is the ideal height for a touch-screen drawer. As can be seen from the rendering, the screen would slide out and

Figure 5.1: Center console touch-screen design

hinge down to an ergonomic angle.

Incorporating a design within close proximity to an existing user interface has a distinct benefit over choosing other locations: it is very likely that significant resources went into research & development to ensure that the radio is properly positioned for easy reach and minimal distraction. A touch-screen interface benefits from the same research.

Alternative options considered for the interface are:

- heads-up-display: this option is expensive, difficult to implement with full colour, would only benefit one person in the vehicle, and would not work well during daylight hours.
- sun-shade touch-screen: an alternative position for the touch screen, swinging down on the sun shade has potential, but it also places the screen much closer to the user's eyes, forcing refocusing. It, too, would only benefit one person.
- top-mounted screen: mounting the display on top of the dash panel would incur more obstruction. This could be unsafe.

A touch-screen sliding out from under the car radio is within reach, sufficiently far from the driver's head, and at an acceptable angle to the driver.

5.2 Usable Software

Using DashUI has the potential to decrease the concentration required to operate the car computer. However, it is still very important that the contents of the screen are visible under all lighting conditions. As suggested in a previous chapters, high-contrast user interface elements for the programs to be displayed on the touch-screen are a necessity.

GTK (Gimp Tool Kit), due to a high concern for usability, provides a theme called *HighContrast*. This theme is mostly sufficient for automotive uses, with controls clearly separated. However, because touch-screens tend to be smaller than average computer monitors, the theme would benefit from

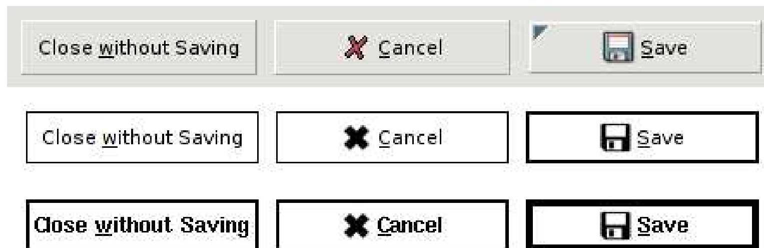


Figure 5.2: Default, HighContrast, and Modified HighContrast Themes

using thicker lines to throughout. Figure 5.2 shows the default GTK theme, HighContrast, and HighContrast modified to have double the line thickness. Also, it is suggested that a high-visibility interface would benefit from screen fonts that are bold and not anti-aliased, as in the figure. With the variability in TFT-LCD display clarity under certain angles, non anti-aliased text is easier to read.

5.3 Other Peripherals

It was suggested in the Introduction that an important aspect of the philosophy of this project involves the concept of a "mobile server." This works very well when a wireless access point is cross-over wired into the network port of the Xbox. The Xbox runs a DHCP server, enabling laptops and other wireless devices to connect to it very easily. These devices can control the computing system or exchange data with it.

With computer aided navigation a primary idea, a Garmin eTrex global positioning system transceiver was tested with the Xbox. This worked quite well, though there were some issues with swallowing the correct window of GpsDrive[1], a popular open-source GPS mapping tool, into DashUI. However, GpsDrive otherwise worked properly through the Xbox hardware.

OBD-II (On-Board Diagnostics II) would have provided for interesting possibilities for tracking a multitude of information about the vehicle. Unfortunately, the testbed 1994 Mazda Protegé only supports OBD-I, making it

[1] http://www.gpsdrive.cc

impossible to extract any reasonably interesting data.

Overall, the tested components worked and did not interfere with each other.

5.4 Web Page

In the interest of continuing this project, hosting was sought at SourceForge.net. SourceForge is the world's largest host of open-source projects. DashBox succeeded in the approval process. At present, a preliminary web site has been set up for the project, with some introductory information and an archive of DashUI and the DashUI Daemon.

The web site is located at **http://dashbox.sourceforge.net**.

Chapter 6

Conclusions

The goal of this project was to implement a system based on gaming console hardware that makes it suitable for use in automotive environments. Existing software and hardware solutions were problematic due to proprietary components, hard-coded software, and little regard for uniformity. The solution required the design and implementation of software such as DashUI and hardware such as the Power Control Unit and Vehicle State Sensor. The solution was termed *mobile server* in contrast with existing, tightly-integrated solutions.

DashUI uses a unique tabbed interface to improve driver concentration by presenting a uniform interface, allows for full user control over configuration, and allows for reacting to state changes in the vehicle itself. The Power Control Unit enables game consoles and desktop computers alike to be safely powered by the vehicle. The State Sensor allows for reading changes in the state of the vehicle's systems in a simple and efficient manner. Consideration of safety and usability throughout the design resulted in a unique product. The DashBox project will benefit others, as it is an open-source project easily obtainable from SourceForge.net.

Throughout the course of the project, several weaknesses in the original goal became apparent. While a game console like the Microsoft Xbox is sufficient for the task of in-vehicle computing, it cannot be recommended for this purpose. The Xbox's ruggedness, low power consumption, and low price are

offset by its low (64MB) memory capacity, non-compliant video output, and the need to void the warranty by opening it to make it useful for this purpose. The latter point alone limits the number of people who would consider using the designs suggested in this project. Using a standard micro-ATX computer motherboard would overcome this issue, possibly for a negligible difference in price once all of the conversion dongles are accounted for. Because the operating system used is Linux, using the software created for this project on standard x86 computer components involves no extra work.

In conclusion, as a concept, game consoles can be used for in-dash automotive computing, though the shortcomings outweigh the benefits of this approach.

Bibliography

[1] Andrew “Bunnie” Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. San Francisco: No Starch Press, 2003.

[2] Hughes, P. K. and Cole, B. L. “What attracts attention when driving?” *Ergonomics* 29 (1986): 377-391.

[3] Jonathan F. Antin. “Informational aspects of car design: Navigation.” *Automotive Ergonomics*. Ed. Brian Peacock and Waldemar Karkowski. London; Washington, DC: Taylor & Francis, 1993. 321-337.

[4] Michael Flannagan and Michael Sivak. “Indirect vision systems.” *Automotive Ergonomics*. Ed. Brian Peacock and Waldemar Karkowski. London; Washington, DC: Taylor & Francis, 1993. 205-217.

[5] R. W. Roe. “Occupant Packaging.” *Automotive Ergonomics*. Ed. Brian Peacock and Waldemar Karkowski. London; Washington, DC: Taylor & Francis, 1993. 11-42.

[6] Walter W. Wierwille. “Visual and manual demands of in-car controls and displays.” *Automotive Ergonomics*. Ed. Brian Peacock and Waldemar Karkowski. London; Washington, DC: Taylor & Francis, 1993. 299-320.

[7] Arthur C. Graesser and William Marks. “Models that simulate driver performance with hand controls.” *Automotive Ergonomics*. Ed. Brian Peacock and Waldemar Karkowski. London; Washington, DC: Taylor & Francis, 1993. 383-399.

Appendix A

ATTiny2313 Introductory Material

Features

- Utilizes the AVR® RISC Architecture
- AVR – High-performance and Low-power RISC Architecture
  - 120 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20 MHz
- Data and Non-volatile Program and Data Memories
  - 2K Bytes of In-System Self Programmable Flash
    Endurance 10,000 Write/Erase Cycles
  - 128 Bytes In-System Programmable EEPROM
    Endurance: 100,000 Write/Erase Cycles
  - 128 Bytes Internal SRAM
  - Programming Lock for Flash Program and EEPROM Data Security
- Peripheral Features
  - One 8-bit Timer/Counter with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare and Capture Modes
  - Four PWM Channels
  - On-chip Analog Comparator
  - Programmable Watchdog Timer with On-chip Oscillator
  - USI – Universal Serial Interface
  - Full Duplex USART
- Special Microcontroller Features
  - debugWIRE On-chip Debugging
  - In-System Programmable via SPI Port
  - External and Internal Interrupt Sources
  - Low-power Idle, Power-down, and Standby Modes
  - Enhanced Power-on Reset Circuit
  - Programmable Brown-out Detection Circuit
  - Internal Calibrated Oscillator
- I/O and Packages
  - 18 Programmable I/O Lines
  - 20-pin PDIP, 20-pin SOIC, and 32-pin MLF
- Operating Voltages
  - 1.8 - 5.5V (ATtiny2313V)
  - 2.7 - 5.5V (ATtiny2313)
- Speed Grades
  - ATtiny2313V: 0 - 4 MHz @ 1.8 - 5.5V, 0 - 10 MHz @ 2.7 - 5.5V
  - ATtiny2313: 0 - 10 MHz @ 2.7 - 5.5V, 0 - 20 MHz @ 4.5 - 5.5V
- Typical Power Consumption
  - Active Mode
    1 MHz, 1.8V: 230 µA
    32 kHz, 1.8V: 20 µA (including oscillator)
  - Power-down Mode
    < 0.1 µA at 1.8V

8-bit AVR® Microcontroller with 2K Bytes In-System Programmable Flash

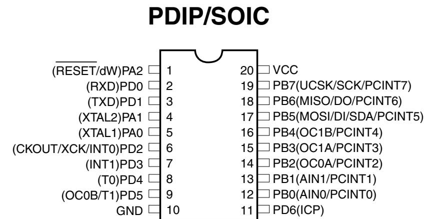
ATtiny2313/V

Preliminary

Rev. 2543F–AVR–08/04

Pin Configurations

Figure 1. Pinout ATtiny2313

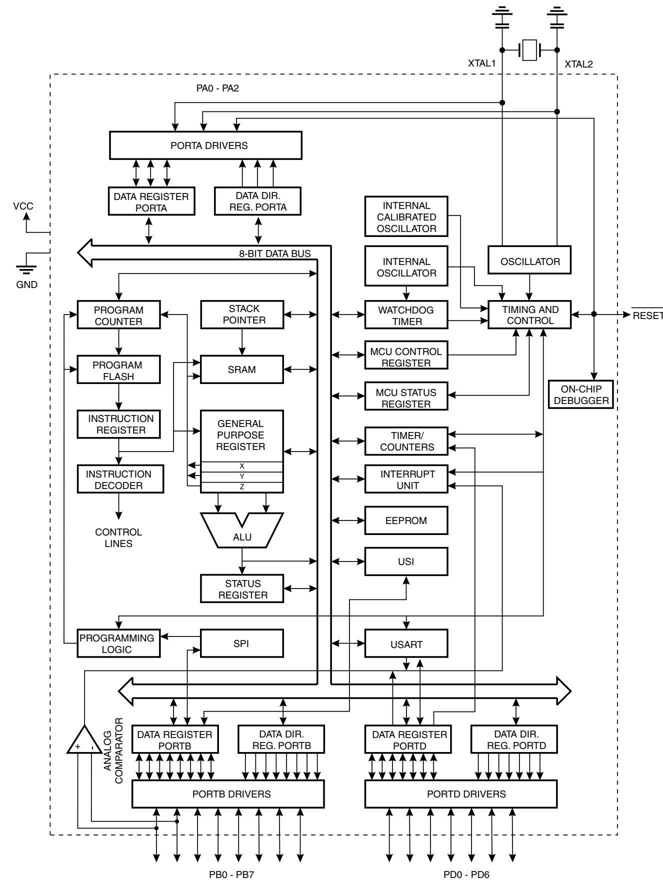


Overview

The ATtiny2313 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATtiny2313 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Block Diagram

Figure 2. Block Diagram



2543F–AVR–08/04

The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATtiny2313 provides the following features: 2K bytes of In-System Programmable Flash, 128 bytes EEPROM, 128 bytes SRAM, 18 general purpose I/O lines, 32 general purpose working registers, a single-wire Interface for On-chip Debugging, two flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, Universal Serial Interface with Start Condition Detector, a programmable Watchdog Timer with internal Oscillator, and three software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next interrupt or hardware reset. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption.

The device is manufactured using Atmel's high density non-volatile memory technology. The On-chip ISP Flash allows the program memory to be reprogrammed In-System through an SPI serial interface, or by a conventional non-volatile memory programmer. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATtiny2313 is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

The ATtiny2313 AVR is supported with a full suite of program and system development tools including: C Compilers, Macro Assemblers, Program Debugger/Simulators, In-Circuit Emulators, and Evaluation kits.

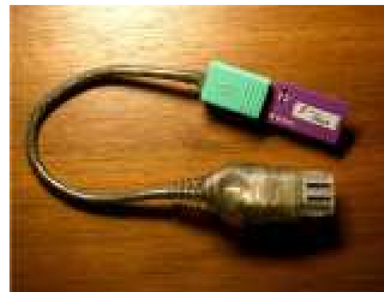
Appendix B

Weblog

USB Cable

September 26, 2004

Today's accomplishment was building a USB⟷Xbox cable. The Xbox end was purchased a part of a controller extension cord. The USB end used to be a Logitech USB⟷PS/2 adaptor. It works really well with a 64M IntelligentStick. The Xbox thinks it is its proprietary storage unit. Copying the MechAssault Linux Installer to it actually shows nice Tux icons in the built-in file browser. Now I need to go out and get MechAssault to really get going in the 4th-year project by installing Linux.



Xbox and USB, Continued

October 2, 2004

With the remaining female end of an Xbox USB cable, as well as the male end of a broken cable, I decided it would be interesting to try doing the opposite of what I did last weekend. Upon soldering together the cable ends, I plugged a controller unit into my new PC.

The controller is a standard USB hub (since it has room to plug in other accessories), as well as a somewhat less standard joystick. Searching the internet, I quickly came across drivers and it worked fabulously. A quick game confirmed that everything works as it should.

This cable may prove helpful in debugging Xbox accessories in case they do not seem to work, when Linux is on the Xbox. The accessory with the standard USB end could plug into the Xbox adaptor, which could then plug into this new adaptop, bringing it back to standard USB. Though seemingly pointless, it might help find an error somewhere along the way. For now, it is fun to have a game pad connected to my PC.

My supervisor has asked me to keep a web log of progress on this project, so this entry is part of the new Xbox section of my log.

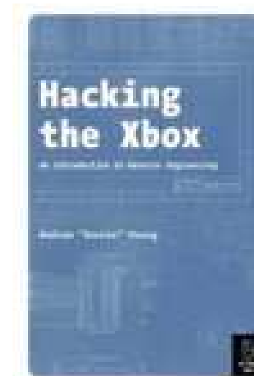
I am currently in the search for the MechAssault game needed to gain entry into the unit. Looking at the stores, the game sells for about 25 dollars, but is the fixed “Platinum Hits” edition. This one is patched to prevent the installer from working. There is also a strong possibility that my Xbox has firmware that makes this process more difficult. If that is the case, I will either have to find an older used unit, or go directly to the hardware modification. Either way, I will have to rent the original version of MechAssault from the rental store to find out.

I am over half way through reading Hacking the Xbox. The chapters get more and more interesting. I am currently reading about the internal hardware encryption methods the units use. I am glad that this part has alrady been figured out by others.

Hacking the Xbox

October 17, 2004

I finally finished reading Hacking the Xbox by Andrew “Bunnie” Huang.



I consider this book to be the de facto textbook for getting anywhere with the project. Though all of the chapters were very interesting, only several will actually be used to develop the “CarBox,” seeing as there is no reason or desire to further knowledge of the Xbox’s security system. Regardless, the book covered many aspects of what will be required to have the unit operate with a touchscreen device, network device, and other peripherals.

Last weekend I ordered the original version of

MechAssault, which is needed to install Linux on the Xbox. I was hoping to have it by this weekend, but it did not come in time. Hzopefully, it will be here soon.

The DashPC Project

November 4, 2004

While I'm still waiting to get the Xbox hardware to run Linux, I decided it would be a good idea to look ahead and see what front-end software I should be running.

DashPC has always seemed the logical choice. It has an intuitive interface, what appears to be decent integration, and of course, it runs on Linux using the GTK toolkit. However, building and running it today, I was somewhat disappointed:

- The tarball is not conventionally organized, and all of the files uncompressed into the current directory.
- The UI is incomplete and buggy.
- The integration that was implied is non-existent. It's simply a shell with pretty graphics that does callouts to other programs.
- It's not configurable, everything appears to be hardcoded.
- It seems disorganized: I downloaded a tarball versioned 0.45, and I got version 0.5.9 of the program.

With these shortcomings, it would probably be easier to start a new interface from scratch. There is really very little that could make this project useful. However, that is not a bad thing, as the project's major goal is to consider user interfaces, both hardware and software.

Here are some ideas for a new project that would differentiate it from DashPC:

- Interface elements should not be graphics. Graphics do not allow for scalability or contrast modification.

- Interface "screens" should be user-configurable, so that a new section is easy to add.
- The UI could be written in Glade, to allow for easier design of various panels/screens.

This software would be straightforward, with focus on two areas: user interface design and plugins/extensibility. Seeing as the back-end work is simple, the first step of the project should be to do some Glade mockups of a new user interface.

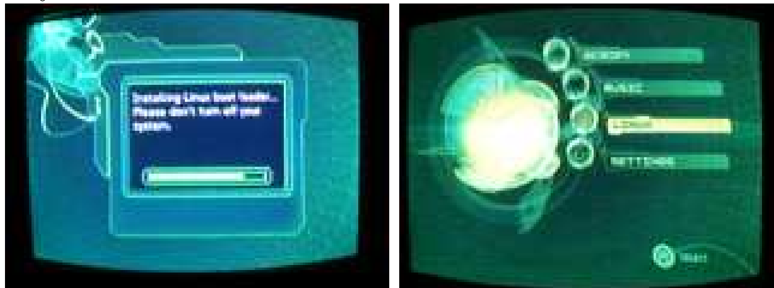
Full Steam Ahead

November 13, 2004

At this point, it seems it is high time to get the project really going. The plan is to do the bulk of the work during the Christmas holidays, so it is imperative to have all of the ground work done by then. By December, all work must be R&D.

To that effect, this evening I finally got around to trading my brand new (version 1.6) Xbox with a friend who had the original (version 1.0/1.1) equipment. This is a win-win situation, because he gets a new unit, with a new warranty, and I get a unit that is more suitable for this project. Knowing the difficulties with the 1.6 hardware also allows for another section of the project report, where I can write about Microsoft's continued attempts to prevent people from running Linux on their gaming consoles, as they get more experience with their ideas about *trusted computing*, of which the Xbox is a prototype platform.

I was able to get Linux loaded into the two megabyte base image fairly quickly:



Next, I will attempt to strip the unit of its ability to play games, in favour of behaving as a small, cheap computer.

Xebian Installed

November 14, 2004

Today was devoted to installing Xebian, a flavour of Debian tailored specifically to Xbox. After installation, it is 100% compatible with Debian's pool of software, but has a slightly different bootup process, as well as a kernel to be used specifically with this hardware. It is now running fairly well, with video output to a video camera.

The first task to install a full Linux system was to overwrite the Flash with one that can boot Linux. This ROM, called Cromwell, was put together to be able to boot Linux. The author did not release the source code publically due to concerns that it could be used to do things considered illegal. To overwrite the normal Flash, two tiny connections that were deliberately left out by Microsoft, namely the traces the write pins, had to be filled in on the motherboard. Here is one of them next to a pin head:

That was a fairly easy task, and the chip flashed flawlessly using the tiny Linux install from yesterday. Next, the hard drive was replaced with one that is expendable. Finally, a CD with Xebian was burned, and a boot attempted. This is where the troubles started. The CD drive was having troubles reading writable media, since writable optical discs tend to be of lesser quality than commercially pressed discs. After four hours of trying various discs burned at various speeds, a Google search revealed that many Xbox optical drive lasers are not set powerfully enough to read writable discs. It was suggested that the laser should be "tuned" to be more powerful. As hinted, right next to the laser diode there was a potentiometer that could be set to lower resistance. It was rotated from approximately 1250Ω to approximately 1050Ω. This setting worked, Xebian installed:

There are still issues to work out. The biggest one is that the system does not want to boot directly from the hard drive. Instead, the Xebian developers provide a CD that is simply a bootloader to the installed system. There is confusion because documentation indicates the Cromwell Flash should be able to boot directly from the hard drive. More research is required here.

While the system was installing, I looked into the complexity of creating a VGA converter to output better video. Apparently, with the signals coming out of an Xbox designed to work with Composite, S-Video, and HDTV, there are enough signals to drive most VGA monitors with a straight-through connection. If the touch-screen LCD I purchase for this unit does not work in this mode, there is a very simple circuit that can be built between the Xbox and the monitor that transforms the synchronization signals into pure VGA.

All in all, this weekend made for good progress.

Some Progress

November 18, 2004

After some eMailing to the xbox-linux mailing list with no success in finding a cure for the Xbox not booting automatically from the hard drive, I discovered that the Cromwell BIOS I was running was somewhat out of date. Flashing the latest (2.32) fixed the issue, and the unit now powers up and boots itself properly. The xbox-linux guys are a friendly and supportive group, I look forward to contributing to the project.

Another good discovery was that with Cromwell, as opposed to the Microsoft firmware, any arbitrary ATAPI CD-ROM drive can be used. This is good, because the built-in optical drive was causing no end of problems reading burned media. The flipside to this is that now that the unit boots properly, the usefulness of the optical drive is much lower. But it is still good to know.

The progress report for this project is due fairly soon. Now that the system is fully running, it would be good to start transcribing ideas about how the user interface should work in time to write about it. Then, the Christmas

Break can be used to start implementing these things. Also, over the exam period, it would be good to start looking for important components like the touchscreen and the GPS unit.

VGA Output

December 7, 2004

The majority of this evening was spent constructing a simple converter that takes output from the Xbox and converts it into mostly spec VGA.

The Xbox has a video encoder chip that takes the raw digital input and converts it to (usually) composite video, although it also outputs luminance and chrominance for S-Video, as well as both current HDTV standards. From a combination of signals from all of these formats, a VGA signal can be formed. The problem is that the sync signal for the whole picture is encoded within the Green channel. This is not VESA-compliant.

The solution to creating a more-or-less VESA compliant signal is to use a video sync splitter chip. In this case, the National LM1881N was used. The input to the chip is the Green channel and the outputs are the horizontal and vertical sync signals, which go directly to pins 13 and 14 on a VGA connector.

While the rest of the PCB is mostly pass-through for the video signals and their ground lines, there Xbox also outputs audio through the same connector as video. Therefore, there is a stereo headphone jack, and it works surprisingly cleanly.

Unfortunately, the video portion of the converter is not ideal. It was expected from the start that there would be a slight greenish tint, since the colours are not weighted for VGA display. However, the green tint is quite heavy wherever other colours are not displayed. Also, there are occasionally some sync issues. Upon investigating the completed circuit, it appears that the resistor is approximately half as resistive as it is supposed to be. Perhaps the resistor is faulty. This may explain some of the issues, so it will be replaced at some point in the near future.

User Interface Idea

December 25, 2004

Now that exams are over, as are Christmas festivities, I am proceeding full-steam with producing software I envisioned while studying for exams.

It occurred to me that the flaw in DashPC's design philosophy is that it's just a launcher. It sits on top of a window manager and launches applications (and a few other things). What is needed in an automobile is an interface, something that does not hide behind windows, but rather allows control over them, and is always available to switch to other programs. However, it should not be a window manager, as that would have to take control of any child windows that a program starts, and that is not the goal. The idea is that child windows are the exception rather than the rule, and those should be handled by a window manager when they do occur.

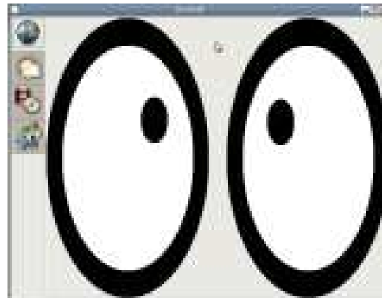
The result of my ideas is a tab interface with large, simple tabs on the driver side of the screen. Tabs can be added or removed from a configuration file, and each tab corresponds to a different program, always running. The program corresponding to each tab is held within the tab's content area, as pictured for gpsdrive (this is only a concept graphic).

The difficulty lies in embedding the program. Several protocols were considered, including xembed. However, it needs to set up a socket and communicate with the client program, which would require modification for each program to be used. Finally, today, I came across gnome-swallow, which literally swallows any window into its panel window. The whole program is approximately 500 lines, with 75% of that being the panel code. The remaining swallow code should be very easily portable to any other GTK application. If all goes as planned, this means the program will be fairly trivial to write.

To make things even easier on the user, a high-contrast GTK theme could be easily designed to make it very clear which tab is selected. This theme could also enforce larger text, as needed for readability on smaller screens. I am also thinking of designing a hardware switch that would make the software automatically tint to a dark green or red when the vehicle's lights are on. This could be executed right at X11's level via colour-correction functions.

Beyond a Mock-up

December 27, 2004



The previously-designed mock-up is now a reality. Written in GTK, with code borrowed from gnome-swallow, the xeyes program was swallowed into this preliminary design. The program is actually attached to the tab pages, so when a tab is switched, the eyes are hidden. They return when the first tab is once again focused.

There are still many things to work out. The immediate concern is whether it is possible to have multiple applications swallowed. There is no reason why this should not be possible, but the current code does interesting synchronization magic by forking the swallowed program off and then attempting to control the main program loop. This, obviously, cannot work with more than one program at a time. At the same time, with a small number of programs swallowed, perhaps it is not a significant performanceissue to overcome.

The step after that is to be able to dynamically specify which programs belong to which tabs, rather than hard-coding. Related to this is the ability to set the swallowed applet's size. Currently, the eyes are scaled to fit the tab content area via command-line argument to xeyes. This solution is not practical when there is a configuration file, as the size of the tabs is not known in advance. Since the window identifier of the swallowed program is known, it would be cleaner to suggest to the program to resize to given dimensions via an Xlib call after the program is started, much like a window manager would.

Overall, this design is working out fairly well.

Success

December 30, 2004

The DashUI program is now moderately useful. It runs cleanly, has a low overhead, and looks pretty good too.

At this point, it can swallow just about any window, even complex programs like media players such as Totem. However, one problem that needs solving is selection of which window to swallow. In programs that have unpredictable titles, the window cannot be specified in advance. In others, like gpsdrive, the splash

screen gets in the way. Have not decided how to get around this yet.

From using DashUI with a mouse, it is fairly clear that the solution would work very well on a touch screen. Large tabs that are always where they should be, quick switching between applications, and the ability to swallow anything really work. From here on, it should be mostly cleaning up the code, as well as adding configuration file parsing.

DashUI Progress

January 11, 2005

Tonight was dedicated to adding configuration file parsing to DashUI. I decided to use the standard IniFile format, as it's easy to understand and easy to parse.

The C parser is about one hunded lines in length, and handles comments, whitespace, any field order, and the various data types that would be found in a configuration file.

The format is extremely simple, For example, to embed the xlogo program:

```
# This example swallows the xlogo program
[X Logo]
program = "xlogo"
window = "xlogo"
icon = "applet3-48.png"
resize = 1
```

However, the main problem with the approach to DashUI is that it still requires a window title to swallow any application. This is problematic for some programs because of either synchronization, or unpredictable window titles. That should be the next problem to solve.

A Possible Interface

January 23, 2005

When this project began, there was a fairly clear vision of what the interface to the car computer could look like. Seeing as the presentation is coming up this week, I felt it was high time to put the idea into a format where it can be shared with others. This touch-screen would stow-away in the slot designed for optional cd-changer units, and would slide out like a drawer, then hinge down so that it could be easily usable from the front seats.

While I had initially intended to build this unit into a car, I am having difficulty locating a reasonably-priced touch-screen that has a diagonal of 6 inches or less. The units I found generally tend to be for medical equipment or factory equipment purposes, and as such, are far more expensive than common sense would indicate. Understandable, seeing as the general consumer would not find such equipment very useful.

The difficulty in obtaining a screen coupled with the Xbox's green-tint VGA output are making the unit look less and less appealing for car-pc purposes. The initial benefits of low cost (~$200), good durability (designed to be handled by children), and fairly low power requirements (~100W) seem to be offset by minor annoyances that make it impractical to build a fully-functional in-dash unit. The very cold weather of this January has not been helpful either. Would it even be safe to spin up a hard drive at -30°C temperatures?Would the electronics be able to handle the condensation that would follow shortly thereafter?

For the project, it should be possible to work without actually building the computer in. For the poster-fair/demo I have come up with a simple method of exporting the display to a touch-screen laptop that should show how things are to work. This model may even be considered as an alternative to VGA output, using a PDA mounted on the dash to interact with the car computer. Still, I was hoping to have a fully-functional car computer in the end.

Project Progress and Setbacks

February 25, 2005

This reading week was to be used for completing several key phases in the Dashbox project.

An initial goal of the project was to produce a plugin or some sort of application that could communicate over OBD-II to the vehicle's CPU and extract meaningful statistics about its usage. While this is an attainable goal, my vehicle is a year too old and only contains a non-standardized OBD-I interface. This means that custom hardware would have to be built to communicate over the interface, and it would be useless to others. Therefore, this component of

the project report will be more of a research area. In searching for relevant information, I came across a professor at NMSU.edu who is currently doing a project with his students whereby a USB OBD-II interface communicated via a kernel module. Also, freediag.sf.net has OBD-II tools, without a GUI. These tools would be a good candidate for existing serial port hardware.

Progress has been made on another component of the project, the relay-mouse status sensor. This plan has been modified to use a USB mouse, as it makes more sense then using a serial-USB dongle. The daemon is nearing completion, with the ability to execute and terminate programs based on button status. There is no fancy IPC, though such a facility could be implemented to communicate with the DashUI element. I cannot see any use for this, however. The software and hardware should be completed shortly.

Two software elements that should be examined over the next week or so are the high-contrast/visibility GTK theme, and the project web page. As more and more content is added, it would be good to house it somewhere. A web page could also focus the work in these final stages. There is little over a month left to complete the whole project!

Critical Mass

March 5, 2005

As the project deadline quickly approaches, I have been spending considerable time putting my ideas into motion.

Last week I decided that the power control unit, while simple, would benefit from being microcontrolled. I have spent a good portion of this week researching the options, and concluded that the Atmel Tiny series is perfect for this and other applications. It is amazingly versatile, simple to program, and conveniently has an on-chip clock (other chips need an external crystal). After programming, literally all that is needed is Vcc, Vss, and any inputs/outputs. Having spent over 20 hours so far practicing writing software for it, I am completely inpressed with this chip. The software will be simple to write, and the hardware

will hopefully be easy to design.

Additionally, some time this week was spent creating the GTK theme for DashUI. A reference engine, *HighContrast*, is being used. The goal will be to thicken up borders so that they are several pixels thick, and possibly modifying the colour scheme so it is not as high contrast as it was originally designed to be. This still needs work.

Lastly, library research was performed to obtain research about ergonomics of vehicle multimedia hardware and its interaction with humans. Two excellent books were found: *Automotive Ergonomics* and *Human Factors in Driving, Seating, & Vision*. These books contain invaluable information and considerations regarding revolutionary human interaction mechanisms, as well as numerous related safety studies.

Microcontroller Success

March 6, 2005

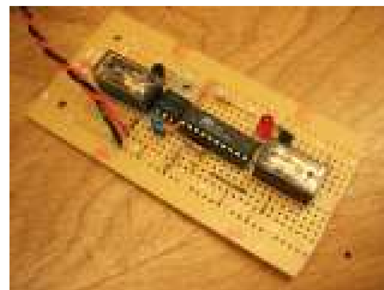
After staying up half of the night, the program for the Atmel microcontroller power control unit was deemed satisfactory. Then I spent half of today ironing out glitches inherent to moving from the test chip to the real one (they are different models). After all of that, it only took 15 minutes to build a breadboard prototype of the final circuit. Though the prototype is designed to work on only 5V, it should not take long to modify it to handle 12V and build the final unit.

These past three days have been extremely educational.

Poster Fair

March 17, 2005

The project poster fair was on Monday. I decided that it would represent the deadline for all of the aspects that were on the to-do list in the presentation.



As such, over the two weeks preceding the poster fair, I invested a total of about 60 hours of time. While not by any means technically advanced, the power control unit represented a lot of that time, and it certainly represented many hours of learning Atmel microcontroller technology and how to actually design simple circuits with transistors. Looking back,

it seems completely straightforward, but there was some resistance in moving from lecture-hall theory to actual design.

The poster show went well, the demo mostly worked. Of course, not when the professor came to see it. Report-writing is slated to start any day now. I seem to be well-off in terms of quantity of content. The quality is also present, but it could still use some polish. An end is in sight.