Design of a Secure RTU for Use in SCADA Systems
M. Eng. Progress Report
Summer 2010

Brad Luyster

Table of Contents

Section 1 – Introduction

This document is a summary of the work completed through the Summer of 2010 in the construction of a prototype for a Secure Remote Terminal Unit (RTU) utilizing the architecture developed by Graham and Hieb. [1] This architecture separates components into three distinct blocks, with some degree of separation existing between each block.

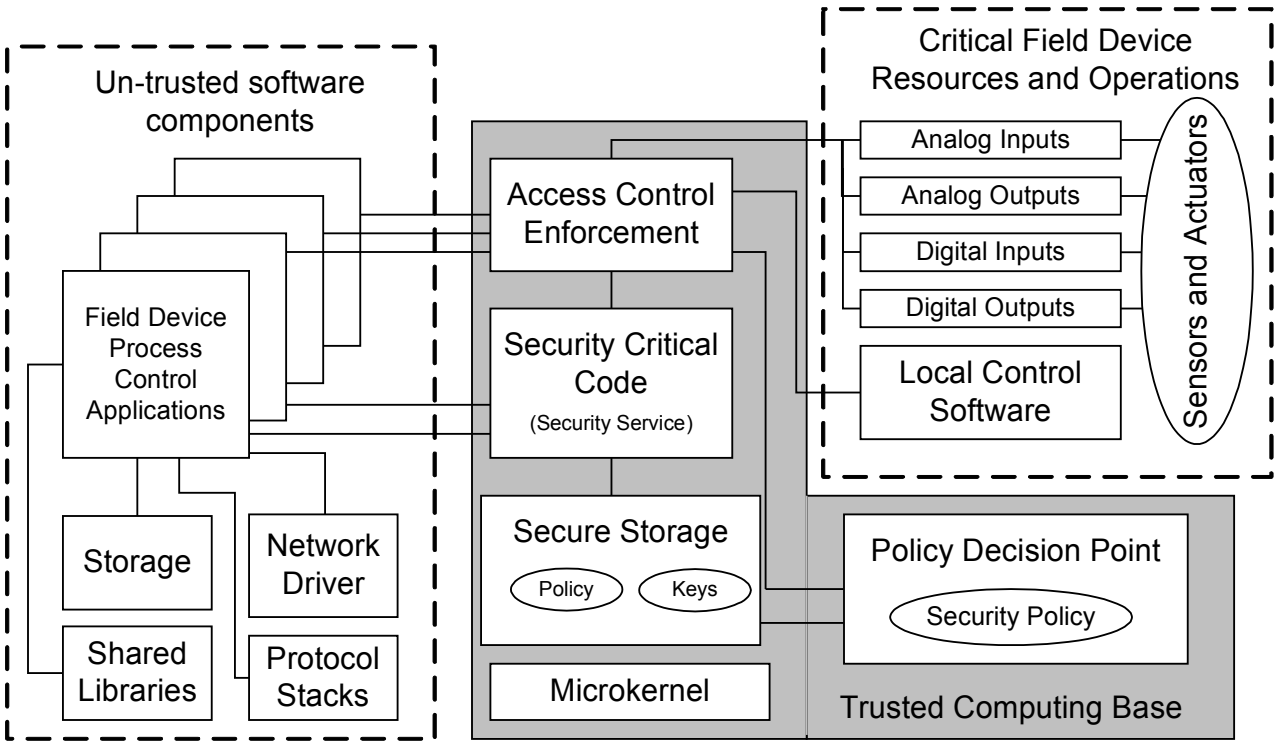


Figure 1: Target Architecture from [1].

An RTU is a device used in Supervisory Control and Data Acquisition (SCADA) systems to control remote devices and sensors. Theses SCADA systems are control devices which operate systems such as the power grid, the water distribution system, and many industrial plant processes. The pumps, switches, sensors and other control devices which interact directly with the process in question are unintelligent devices with no computing or decision making power of their own. They simply accept an input, and provide some output. The RTU device provides the input required to actuate these devices. The RTU communicates to a central control room using any number of protocols and methods. Today, Ethernet is an increasingly prevalent industrial standard, and many of these devices are Ethernet connected. When these control networks interact with public networks such as the Internet, the possible attack vectors which could be used to disable these control systems increase in number rapidly.

The architecture which is predominantly utilized in industry today makes use of off-the-shelf hardware and off-the-shelf software, such as the Linux Kernel, or Microsoft's Windows CE operating system. Research performed by Dr. Graham and Dr. Hieb [1] has shown that this standard leaves current RTU devices vulnerable to any attacks which can target this hardware and software stack. The

target architecture improves on this de-facto standard by providing strongly segmented memory spaces which cannot be interfered without detection. The architecture has been modified slightly in order to facilitate prototyping, but retains the principal properties of well-divided memory spaces.

Over the course of this project, we have made use of the OKL4 Microkernel based operating system. This operating system is available under an Open Source license for non-commercial purposes, and provides the strong memory segmentation which this architecture requires. More importantly, a variant of the Kernel used in this project has been subjected to mathematical verification [2]. As a result, this project can lay claim to some measurable degree of security verification.

Over the course of the summer semester, a great deal of development took place, moving the prototype from a very early stage of development, towards a device which can be tested for the desired security characteristics in a working environment. Among the challenges overcome were the inability to host more than 5 “cells” at once, the difficulty of configuring a foreign Network driver for use in the OKL4 Microkernel, ensuring IPC conflicts did not exist, as well as tracing and eliminating bugs in the Elfweaver tool.

The architecture of the prototype as-built during the course of the semester is identical to the Graham and Hieb [1] architecture with the addition of a Utility memory space which provides management of hardware devices, IPC based debug output, and an IPC performance testing thread. This space exists as a member of the Untrusted Computing Base, and interacts minimally with the Trusted components. Section 7 discusses future developments which could improve the separation between these components.

The hardware of the prototype, while functional for testing purposes, can not yet generate the voltage/current outputs required for actual control devices. The number of digital and analog inputs and outputs are comparable to many existing commercial RTUs. Sections 2 and 7 discuss the current and future limitations of this hardware suite.

The primary goal of this document is to provide a reference manual for individuals who may come along in the future to expand or continue this project, that any one following this path may avoid some of the pitfalls experienced.

Section 2 – Hardware Selection and Construction

Prior to beginning work on this project, Dr. Hieb and Dr. Graham the target architecture and OKL4 Microkernel based operating system were selected with certainty. A prototype possessing similar quantities and varieties of IO when compared to commercial examples was also desirable. Ease of development was a strong wish, but by no means a concrete requirement. The selection and configuration of prototype hardware sought to maximize these variables.

Section 2.1 – The Gumstix Platform

The Gumstix Platform is a simple, embedded Computer Module which includes an Intel Xscale PXA Microprocessor, 64 megabytes of RAM, and 16 megabytes of Flash. The module also includes several expansion connectors which are used in this project for GPIO and Network Connectivity. Very specifically, this project is making use of the Gumstix Verdex Pro XM4 COM. This module includes a PXA270 chip clocked at 400MHz. The PXA270 is an advanced 32-bit Microcontroller based on the ARMv5 architecture. Although dated, the processor compares favorably to existing RTU devices. Datasheets and Application manuals are commonly available from Marvell. [3]

The Gumstix Platform was selected for its expandability and compatibility with available OKL4 Distributions. The OKL4 Microkernel is currently targeted towards the Cell Phone and Mobile Device market. Although the PXA270 was designed principally as a Mobile Device chip, its specifications are not dissimilar from a number of RTUs already on the market. Furthermore, this chip is compatible with both Linux and the OKL4 Development Environment, allowing for robust testing in a real-world environment.

Along with the Gumstix, the Console-VX and Netpro-VX expansion boards were purchased. The Console-VX provides breakouts for all 3 UARTs included on the PXA270 Chip, as well as pin-headers for Audio, I2C, and several GPIOs. The Netpro-VX includes an SMC9118 Network PHY/MAC interface for network connectivity. Originally, we planned to pair the Netpro-VX with a breakout-VX, and use a Tweener board for UART access. Unfortunately, the tweener board interferes with the Netpro-VX. As a result, slightly fewer GPIOs are accessible. This is not a significant limitation, due to the selection of I2C devices for GPIO, as will be detailed in the following section.

The boot process of the Gumstix board is simple: A bootloader resides in flash memory, from 0x0 to 0x40000. The remainder of flash is utilized by the JFFS2 file system, pre-loaded with a Linux distribution. The bootloader used is uboot. Help regarding the Gumstix Implementation of Uboot can be found here: http://docwiki.gumstix.org/U-boot. Uboot executes a startup script, which loads a program image from Flash, MicroSD, or over the network into RAM, and then begins execution at that location. RAM begins at memory address 0xA2000000 and ends at address 0xA3F00000. The factory startup script loads a Linux image from Flash memory into RAM, and boots. In order to properly boot the Gumstix board, an image must be loaded into memory in either ARM ELF or Intel Hex format. This script has been modified for this project to load an image over the network, via TFTP, load it into RAM, and execute. The image loaded is in ARM ELF format. For the purpose of this project, the bootable elf format loaded onto the gumstix is `image.boot`.

The startup script currently running follows:

```
ipaddr = 192.168.0.2
serveriip = 192.168.0.1
tftpboot a2000000 image.boot
bootelf a2000000
```

This startup script has been written to flash, and overrides the default startup script. This script is determined by Environment Variables which may be interactively changed in the Uboot shell. In order to recover the Linux installation built into flash, execute the following at the Uboot shell:

```
setenv bootargs console=ttyS0,115200n8 root=1f01 rootfstype=jffs2
reboot=cold,hard
fsload a2000000 boot/uImage
bootm a2000000
```

The Linux Image burned into the flash by default is mostly functional. Unfortunately, the scripts which generate Hardware Device Nodes do not operate properly. With manual inspection of these scripts, the appropriate device nodes can be created interactively. This may be necessary to test Linux drivers and software for compatibility and use within the OKL4 Operating System. This has been used to verify the validity and operation of the I2C peripheral built into the PXA270 processor. As a result of this research, I2C operation was extended successfully into OKL4.

Section 2.2 – IO Component Selection

After careful research of some existing RTUs by major manufacturers, a list of needs was developed regarding the input/output scheme of the prototype. SixNet manufactures a variety of RTUs with approximately 12 Digital Inputs, 4 digital outputs, 2-8 analog inputs, and 0-2 analog outputs. Motorola manufactures more ruggedized, feature-rich models which contain 16 digial outpus, 8-16 digital inputs, 4-8 analog inputs, and 4-8 analog outputs. The existing RTUs interacted with devices which requires 0-9V analog, or 0-20mA analog, along with a range of possible digital signals. In order to generate this sort of I/O, we would have to build a signal conditioning board for the RTU. Furthermore, while the PXA270 has many, many GPIOs, it has neither analog input nor analog output. The GPIOs which do exist on the board have different specifications, and a misconfiguration or a misread datasheet could find the whole processor up in smoke.

In order to provide flexibility in the selection of components, as well as reduce possible avenues of catastrophic failure, an I2C-based IO system was devised. The I2C bus is an Inter-IC communication bus, with many thousands of available peripherals in many thousands of configurations. The bus has a maximum speed of 400KHz, and can support up to 127 devices. Using an I2C based input/output system for the project had multiple advantages. Changing the specifications of the I/O no longer required retooling of complex analog components, and the PXA270. Rather, a cheap commodity IC could be replaced and some minor code changes made. Furthermore, GPIO were conserved, and the risk of destroying the PXA270 was greatly reduced as a result of placing the I2C

chips between the PXA and the field components.

A benefit which was not realized until later on was the ability to separate IO in memory from the rest of the operating system. All of GPIO, including the IRQ lines, and the GPIO function registers, exist in the same memory space. As a result, only one OKL4 cell could have access to these functions. This is a disadvantage, as many hardware devices require interaction with these registers. The net result would be requiring the IO code to exist in the same execution space as the utility/hardware manager. This has many ramifications, in addition to complicating programming.

The I2C memory space, however, exists as its own page. As a result, the architecture of the prototype as-written can more closely match the target architecture in meeting segmentation goals, allowing for a more verifiably secure device.

Section 2.3 – IO Hardware Construction and Testing

All development and hardware build-out has been performed on a breadboard. Initial development of the I2C hardware was tested with an Arduino Microprocessing Environment. The Arduino is a simple, easy-to-use microcontroller development kit, with a simplified IDE. Although this device primarily targets the hobbyist market, it was useful in this case to get development started quickly. Without a complex understanding of the OKL4 operating system, and the PXA270 chip, the IO breakout board could be completed and tested. Furthermore, the code generated to test the devices on this breadboard could be made as cross-platform as possible, which greatly sped up the process of transferring the IO code to the PXA platform.

The project as-built has 8 Analog Inputs, 2 Analog Outputs, and 8 Digital IO spread across three chips. Without modification, all of these chips operate at the 3.3 volts provided by the PXA270 and the Console-VX breakout board. The Serial Data and Serial Clock lines of the I2C bus must be pulled to a high logic level. Nominally, 4.7 kOhm resistors are used. Steps must also be taken to minimize bus capacitance. With another layer of signal conditioning, these I/O can be easily converted into the 0-9v/0-20mA signals needed for interaction with actual control devices. The chips were selected for their speed, accuracy, and availability. Unfortunately, with the exception of the GPIO chip, this precluded prototype-ability. As a result, the DAC and ADC chips are soldered onto breakout boards. The DAC chips are in a SOT-23-6 Surface Mount Package, and the ADC chip is in a TSSOP-20 Surface Mount package. Moderately competent soldering skills are required.

The GPIO chip selected is the MCP23009. This chip provides 8 GPIO at up to 400KHz.

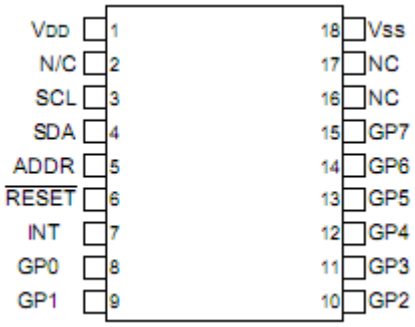


Figure 2: Pinout of the MSCP23009 Digital IO Chip.

Pins GP0-GP7 provide configurable digital input and output. Pin INT can provide an interrupt trigger for the target device based on configurable pin change or pin level trigger.

The DAC chip selected is the MCP4725. Each of these chips provides a single 12-bit DAC. The output is provided by a high-speed Resistor Ladder.

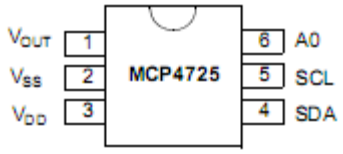


Figure 3: Pinout of the MCP4725 DAC Chip.

Based on a 12-bit serial input, the DAC generates a voltage on pin Vout. The first 2 address bits of this chip are burned in at the factory, based on a code which is laser imprinted on the chip. Magnification may be required to determine this address. The final bit may be set using the A0 pin.

The ADC chip selected is the AD7997. This is the most complex chip in the project, providing eight 10-bit Analog Inputs.

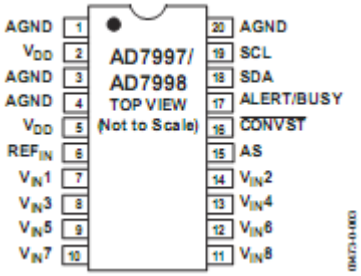


Figure 4: Pinout of the AD7997 ADC Chip.

Multiple configuration registers modify the function of this chip. Current usage places this chip in on-demand conversion mode. As a result, desired inputs require up to 2us to settle and go through the conversion process. During this time, there must be I2C bus silence. In practice, this is not a limitation.

Addressing of each chip is complicated by the fact that the AC7997 and the MCP23009 conflict in address space. This problem is solved by using a tuned resistor network to set the MCP23009 address pin. Based on an analog input derived from a voltage divider circuit applied to this pin, the appropriate address bit is set. In order to properly set this address bit, the resistors used must be measured, in order to assume values within tolerance.

On the Console-VX, Pin headers have been soldered onto the I2c header, as well as the ac97 header. On the I2C header, per the PXA datasheet, Pin 5 and Pin 8 are unusable. On the ac97 header, Pin 4 may not be used.

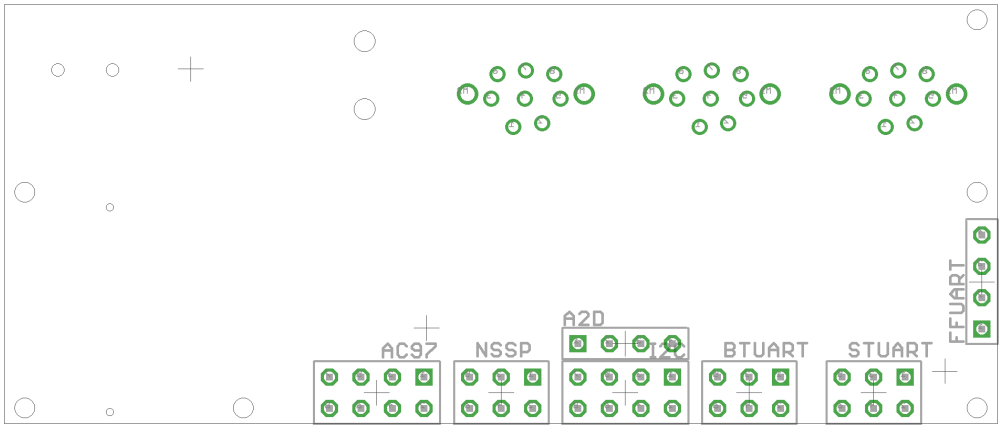


Figure 5: Pinout of the Underside of the Console-VX. Pin 1 is squared off.

Of the three UARTs broken out by the Console-VX, two are in use for this project. The FFUART is used for the Kernel debugger, while the STUART is used for the locally developed IPC-based console interface. Viewed from above, with the 3 UART connectors on the left side of board, the middle connector is the FFUART, while the leftmost connector is the STUART.

Section 3 – Overview of Development Environment

Following hardware selection, selection and configuration of the appropriate software development environment began the next set of challenges. Although the OKL4 developer's wiki acts as a repository of software and documentation, it contains little in the way of configuration information. Moreover, what information exists pertains more accurately to the previous versions of OKL4. As a result, much of the following information "fills the gaps" of available documentation.

Section 3.1 – The OKL4 Operating System

The OKL4 Microkernel is an operating system developed by OK Labs, a commercial spin-off of NICTA, which is itself a research facility founded by the Australian government, and several major Australian Universities. The OKL4 Operating System is a set of libraries and system functions built on a variant of the L4 kernel developed at NICTA called L4-Embedded, which is itself a branch of an L4 variant known as L4::Pistachio. The L4 Kernel itself was originally developed by a German computer scientist, Jochen Liedtke. During the 90's, he worked towards a better microkernel.

A Microkernel itself is an Operating System kernel which provides a bare minimum of services required to implement an operating system. This includes Memory management, threads, thread scheduling, and Interprocess Communication. This excludes device drivers, applications, file systems, virtual devices, network communications, and so on. Throughout history, most Kernels have been Monolithic Kernels, which include device drivers, file systems, and other services in the Kernel of the operating system, executing this code as trusted, privileged code. As a benefit, performance is increased, as the code has direct access to the necessary hardware and software faculties needed to perform their tasks. Unfortunately, this leads to Kernels which grow in size exponentially as new features are added. Furthermore, any kernel-mode program has the potential to crash the entire system. This is both a security threat, as well as a threat to stability, especially in a real-time, control system situation.

The primary reason for choosing a monolithic kernel is performance. The kernel controls memory and memory abstraction. As a result, any user-mode program must make a kernel system-call in order to access privileged system memory. In the past, the transitions from Kernel mode to User mode have been costly in terms of CPU power. [4] Fortunately, the L4 Microkernel solves most of these problems, resulting in very fast interprocess communication, making a Microkernel suitable for many more applications [5]. Unfortunately, there are still some caveats: among them, the inability to write hardware drivers which use DMA. Typical DMA controllers allow a hardware device to read and write to any location in system memory. This is dangerous, both in the case of a malicious driver, and in the case of a buggy driver. In the future, this is unlikely to be a problem, as IOMMUs are being included with newer systems.

The L4 Microkernel is the first microkernel with speedy enough Interprocess Communication to allow for user-mode abstractions of the required services in an embedded, time-sensitive (though not necessarily time-critical) application. [6] This is very important, but there is another property of the OKL4 operating system which makes its selection important for this project. A variant of the L4-Embedded kernel, called seL4, has been mathematically proven. Although this kernel has not been released to the general public, public statements have been made attesting to its minimal variation

from the Kernel used in OKL4. [2] As a result, we can be reasonable certain that the project as-built could be adapted to this verified kernel.

This project makes use of the OKL4 Version 3.0. The software libraries included with this version include headers for OKL4 Library calls, as well as direct L4 Kernel system calls. Many POSIX stubs exist, but these are thin abstractions for OKL4 Library calls. The OKL4 Library builds on top of the existing L4 system calls to make development easier. In some cases, however, there is a need for direct access to the L4 system calls. The source tree is invaluable in determining the operation of these OKL4 Library functions. The manual for the OKL4 Library is generally functional, although occasionally thin on the order in which processes must take place.

The OKL4 operating system provides strong memory segmentation. Virtual memory is assigned at compile time, and cannot be changed at run time with the exception of memory spaces set aside at compile time for run time assignment and configuration. These domains of segmentation are colloquially referred to as cells, although “cell” has no special meaning in OKL4. A cell consists of one or more threads, one or more virtual memory sections (each virtual memory section must be backed by physical memory), and may contain protection domains, zones, and their associated memory sections. A zone is an abstraction which maps a single memory section to multiple protection domains, and a protection domain is a memory segmentation abstraction with the ability to modify its (and the properties of other protection domains to which it has the appropriate capability) properties at run-time. Memory spaces are associated with their own Capability List (or 'clist'), which is a list of properties defining access, physical memory associations, and other special identifiers. They define which memory spaces may communicate with each other, as well as the association between physical memory and the memory space itself.

Section 3.2 – Initial Software Configuration

As of the writing of this document, all utilities and development tools used are a part of the 3.0 branch of the OKL4 Source. At the time of writing, OKL4 4.0 is available, but not as a full-source package. The availability of the source code has been critical in the understanding of the OKL4 Operating System, and the potential for disastrous changes in design methodologies has kept development of this prototype in the 3.0 version tree of the OKL4 Microkernel.

The software tools used for the development of this prototype include OKL4 Source Tree, the Xscale Processor SDK, and the ARM EABI GCC toolchain, all available from OK Labs (available at http://wiki.ok-labs.org/). This toolchain is available for 32-bit Linux systems only.

The SDK for the Xscale processor includes instructions for installing the SDK and compiler, as well as testing the sanity of the build environment with a simple “Hello World” program. Simply as a suggestion above and beyond the instructions included with this software package, it is immensely useful to add symlinks for Elfweaver and Skyeye to your binary path. Furthermore, on a multiple-developer system, it is necessary to place the SDK, compiler, and associated tools in a publicly accessible directory. In a sentence, unpack the SDK and compiler, add the correct environment variables, and the development environment is up and running. The utilities included with the Non-cross-compiling GCC included with many Linux distributions will disassemble ARM ELF files. Taking these steps, all example files should compile and execute flawlessly. For the skyeye simulator,

image.sim is the binary to use, while for the Gumstix platform, **image.boot** is appropriate.

Section 3.3 – Remote Compilation and Version Tracking

The development tools used for this project require a great amount of computing power, and completion speed is heavily dependent on the platform used. This is principally due to the massively complicated task undertaken by the Elfweaver program. As a result, development was moved to a multi-core server located in the Dahlem supercomputer lab. Development took place via remote terminals, and the resulting binary file was uploaded to a test machine in the ISR lab. The make script for the project was modified to automatically SCP the resulting file to the ISRL machine. This worked well for many weeks.

The computers in the lab were are not publicly reachable outside the network (in the future, this problem will be solved with creative management of shorewall configurations). Thus, the make script which uploaded the compiled file failed, being unable to reach the remote machine. The solution was a reverse SSH tunnel on the development computer inside the ISR lab. This tunnel would connect in an outgoing manner to the remote development machine, and traffic directed to an unused port on the remote computer's loop-back interface would instead be directed to port 22 of the development machine within the laboratory. The command used follows:

```
ssh -Nntf -R <RemotePort>:localhost:<LocalPort> <user>@<RemoteHost>
```

Subversion is useful in allowing the changes made to the code base to be tracked. Furthermore, multiple developers are able to modify the same code base at the same time. Their changes, when they do not conflict, are seamlessly merged, and when they do conflict, are brought to attention. Furthermore, this version control allows horrible design mistakes to be undone without much investment of time or brainpower. Subversion specifically has a large user-base, and is easy to use. In a larger development environment, or with a less localized development team, a version control system like Git may be more advantageous. More information than could ever be summarized by this document can be found here: http://svnbook.red-bean.com/.

Section 3.4 – Detailed Overview of Development Tools

The tools used to build the project are provided, pre-compiled, by OKLabs. With few exceptions, these tools are ready to use out of the box. These exceptions will be noted in the following sections.

Section 3.3.1 – OKL4 Software Developers Kit (SDK)

The SDK included with the OKL4 distribution is a platform-specific set of tools of libraries required to easily develop for the OKL4 operating system. The SDK contains Kernel files pre-compiled for use with the selected platform, as well as a binary distribution of Elfweaver, and several XML files which define system-wide characteristics. The most important of these XML files is the **machine.xml**.

There are several different Kernel files which ship with the SDK. The kernel used is determined by a variable in the project **makefile**. For the duration of this project, the “micro-debug” kernel has

been used. The debug kernel includes a serial kernel debug interface, which is invaluable in debugging threading and hardware issues. Other kernels include the "nano" series, which offers a stripped down interface into the OKL4 operating system (maintaining the fast IPC and Memory Space separation which makes OKL4 unique), for machines with limited space or processing power.

The `machine.xml` file defines memory spaces for the system. This defines elfweaver behavior at link-time. The included `machine.xml` is a fairly generic, and contains the bare-bones memory space information for Xscale systems. I2C and Timer Memory space were not included in the default XML file. Furthermore, only the default FFUART was included. The second UART used in the development of an IPC based serial output, the STUART was not included. The Elfweaver manual goes into great detail regarding the format of entries in this file: it's modification is trivial.

Also of note, the `machine.xml` file contains the main physical and virtual memory tables for the system. In systems with expanded memory, it could be useful to modify these entries. There is a bug in Elfweaver which will be detailed in Section 4.6 which may be linked to this property of the `machine.xml` file.

Using the "Hello World" example included with the SDK, an example source tree for a project might appear as follows:

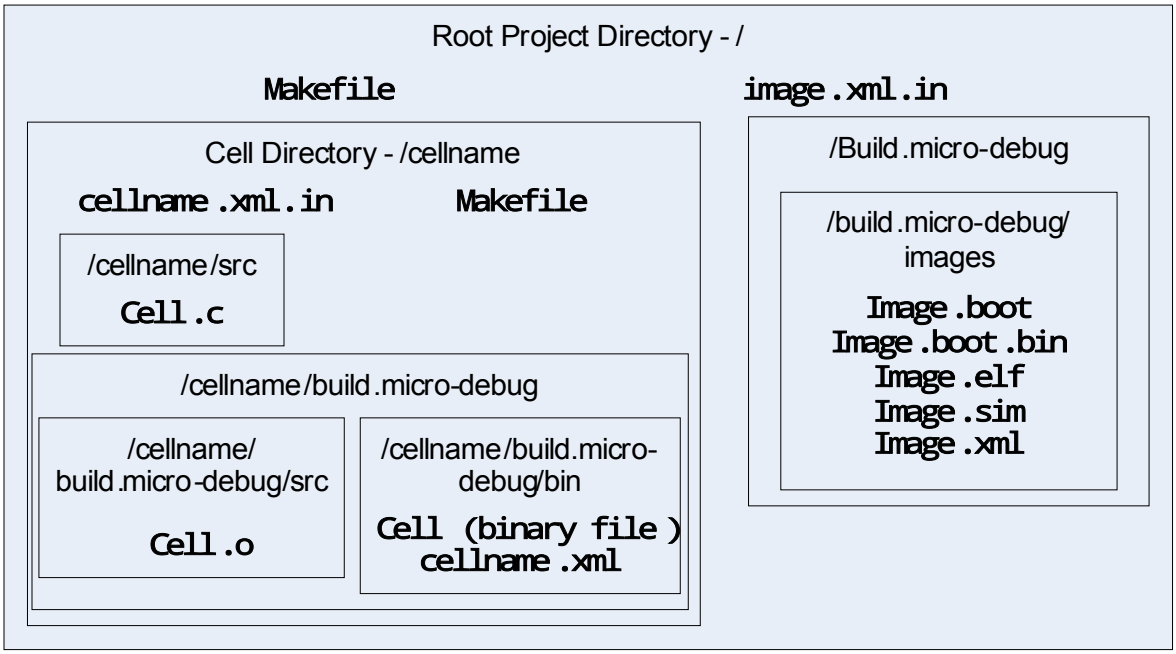


Figure 6: Directory Structure of an Example Project with Compiled Object Files.

Section 3.4.2 – Elfweaver

Elfweaver ties the OKL4 development system together. When writing code, the included EABI GCC compiler and linker ensures that each cell is compiled and linked into ARM object code correctly. Elfweaver takes the object code generated for each cell (each cell is compiled completely independently of the others), along with the included kernel object code, and uses the XML tree to modify the individual output files, and "weave" them together into a single bootable ELF. Elfweaver is

a massively reentrant and polymorphic piece of code written in Python, and contains multiple bugs and caveats.

Elfweaver uses the XML files which litter the development environment in order to determine the manner in which the resulting ELF files must be linked to the kernel.

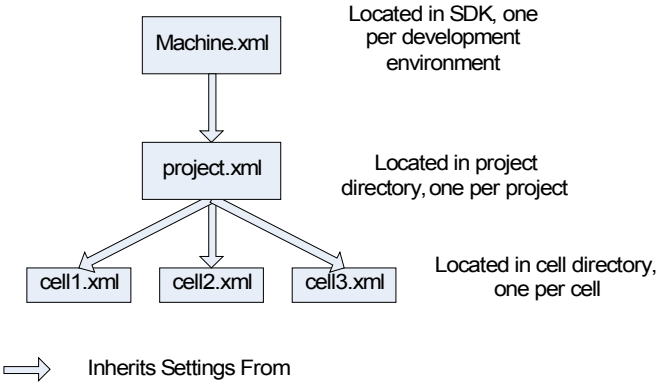


Figure 7: Heirarchy of XML Files.

This takes place in an inside-out order. First, the compiled output files from GCC and its linker are combined with the per-cell XML file, relocating the memory space of the program, and making note of required cell characteristics such as capabilities, threads, memory sections, and other attributes. Next, the cells are linked to the kernel, with the kernel getting assigned memory sections with low addresses, and cells coming in above that. Next, elfweaver takes note of the earlier XML file specifications, and statically writes kernel memory to inform the kernel thread of the capabilities and permissions of each memory space, as well as starting threads. This is how the final ELF file is generated.

Section 3.4.3 – OKL4 Resources

The source tree has been an invaluable resource in learning and understanding the operation of the OKL4 Library and System functions. The documentation for the practical usage of these functions is poorly written. Unfortunately, the source code is not significantly better. The source tree is an amalgam of different versions of kernels, utility cells, examples, unit tests, and half-finished code trees.

The source tree can potentially be compiled into Kernel and SDK images with a complex series of Python scripts glues together by the Scons Build system. This is not worth investing time in. The source tree has 4 main uses: System and Library Call references, Examples which may or may not be obsolete, Understanding hardware interfaces and bindings, and Elfweaver. Unfortunately, the source code which you are able to download from OK labs isn't the source code which ships with the SDK. There are slight differences in polish and interfaces. In general, these are easily discovered and deciphered. In rare cases, however, this is not the case. As mentioned above, Elfweaver is one of these cases. For all intents and purposes, however, one might as well simply use the version of elfweaver included with the source code. Its function is identical, down to the last bug. The build script is buggy, although running the source through an interpreter eliminates some non-critical errors, and allows for the use of versions of Python which aren't 2.5.

Although the source tree implies that there are many POSIX hooks built into the operating system, these are often thin shells of code. Where the hooks may be simply translated into a Microkernel interface, they are, and where they cannot, they are not. Comments throughout the code are sporadic, but generally trustworthy where they exist.

Additional resources exist by means of the Wiki (http://wiki.ok-labs.com), as well as a mailing list (available at the same URL as the wiki). The discussion on both the wiki and the mailing list pertain primarily to version 2.0 of the OKL4 operating system, but many archived issues provide insight into the working methods of the kernel. Also useful are the library, elfweaver, and operating system manuals, available http://wiki.ok-labs.com/downloads/release-3.0/okl4lib-ref-manual-3.0.pdf, http://wiki.ok-labs.com/downloads/release-3.0/elfweaver-ref-manual-3.0.pdf and http://wiki.ok-labs.com/downloads/release-3.0/okl4-ref-manual-3.0.pdf, respectively.

Section 3.4.4 – Bugs Discovered in the Build System

Two major bugs have been found in Elfweaver thus far: A limitation of 5 total cells, and difficulty in statically starting threads at compile-time. The threading issue has been solved by modifying the source code shipped with the Source tree. Unfortunately, this source code is different from the compiled version, adding difficulty to tracebacks which refer to lines and functions which do not match up with the available source.

The limitation of 5 total cells is related to the detection and assignment of free memory space. Memory space is assigned to threads on a first-come, first-served basis. If the list of free physical memory runs out before all threads are served, Elfweaver throws an exception and refuses to compile the program (and rightly so!). Unfortunately, inspection of the **machine.xml**, and the generated ELF files shows that physical memory isn't nearly fully assigned. Furthermore, this limitation seemingly has no bearing on how closesly cells are packed together in memory space: as soon as 5 or more cells are created, Elfweaver gives up. This problem has not been solved.

The threading issue was quite a bit simpler: The developers omitted a variable in a function call, causing all thread start locations to be linked to an incorrect section of memory. When the thread would start, the program would jump into useless memory, and the system would be lost to the world. With the addition of this variable in the function call, everything began to work properly.

```
threads.append(self.collect_thread(self.elf,                thread_el,
self.namespace,  image,    machine,    pools,    kernel,self.space,
"thread_start",  cell_create_thread = True))
```

The original code simply passes "**thread_start**" to the thread creation mechanism. The revised code fixes this, replacing "**thread_start**" with "**thread_el.start**":

```
threads.append(self.collect_thread(self.elf,        thread_el,
     self.namespace,  image,  machine,  pools,  kernel,  self.space,
     thread_el.start, thread_el.name, cell_create_thread = True))
```

Unfortunately, this required reversion to the Elfweaver source code included with the source distribution. Although this works properly, a Python interpreter must be installed. The included compilation to binary scripts do not function properly, and time has not been invested to fix them.

Section 4 – Software Development

Software Development has been this project's largest time investment. The OKL4 project has generated an impressive amount of documentation, but in many cases this documentation is lacking in information, or incorrectly describes an interface. Although all care has been taken to achieve Hieb and Graham's target architecture, there have been deviations in the creation of a utility cell.

Throughout the development of this project, each cell contained one memory space, and one thread. A utility cell was created which combined trivial processes which existed originally as their own cells. The one thread/one memory space paradigm proved useful in speeding development, as many of the provided examples followed this paradigm. With the provided interfaces to protection domains, zones, and other abstractions, this is not the only or best way of managing memory and permissions between cells and threads.

Development progressed incrementally with specific goals in mind. First, I2C development was moved from the Arduino to the OKL4/PXA270 environment. Next, a serial output cell was designed to provide more robust debug input and output. Then, a network layer was added. Finally, the cells created were reorganized to better conform to the architecture designed by Graham and Hieb. During this process, some basic performance testing was completed, as well as an initial research paper. Following this research paper, the mechanisms for performing IO were improved, and different cells wrapped together into a single multi-threaded cell. Finally, the chain of communication was tested fully by implementing a simple Ethernet server which activated a light on the development board.

Section 4.1 – IO Cell

The first cell to be created was a cell for I2C I/O. Although the hardware itself had been developed and debugged using the Arduino platform, it was an entirely different task to even start using the I2C device included with the PXA270. On the PXA270, devices are accessed through memory-mapped registers. The state of the device is altered by writing data to memory locations which are described in the programming manual and datasheet of the device. Unfortunately, this is made more difficult by OKL4 abstracting memory accesses through the Kernel. Physical Memory is not directly accessible to any thread or cell, it must be mapped to virtual memory.

OKL4 provides an abstraction with elfweaver to map this memory into a cell at compile-time. Unfortunately, the location that this memory will be mapped to is not determinable before compile time. As a result, we must determine how we can determine this virtual mapping at run time. There is no documentation on this process, and no examples are provided in the SDK. There are several possible abstractions for this memory mapping, so this is a non-trivial task. Fortunately, Dr. Jeff Hieb discovered an aged example in the source code which worked properly after some modification.

The “use device” XML tag in elfweaver simply performs a look up from the `machine.xml` file, and maps the appropriate memory sections as static memsections, and the associated IRQs into the thread/cell to which the device is assigned. As a result, an older example which made use of static memsections could be utilized for mapping the appropriate memsection into the cell.

```
    okl4_memsec_t           *               i2c_memsec;
    okl4_env_segment_t      *                 i2c_seg;
    okl4_static_memsec_t    *            i2c_static_memsec;
    okl4_static_memsec_attr_t         i2c_static_memsec_attr;
    okl4_virtmem_item_t     i2c_virtmem;
    int                             error;

    //                begin                execution

    okl4_init_thread();     //     this     sets     up     okl4     lib

    L4_KDB_SetThreadName(L4_Myself(),             "I2C");

    i2c_seg=okl4_env_get_segment("MAIN_I2C_MEM0");
    assert(i2c_seg);

    okl4_static_memsec_attr_init(&i2c_static_memsec_attr);

    okl4_static_memsec_attr_setsegment(&i2c_static_memsec_attr,
    i2c_seg);

    i2c_static_memsec=    malloc(    OKL4_STATIC_MEMSEC_SIZE_ATTR(
    &i2c_static_memsec_attr             )             );
    assert(i2c_static_memsec);

                                           okl4_static_memsec_init(
    i2c_static_memsec,&i2c_static_memsec_attr);

    i2c_memsec=okl4_static_memsec_getmemsec(i2c_static_memsec);

    i2c_virtmem=okl4_memsec_getrange(i2c_memsec);
```

After mapping the memory into the cell, it's a simple matter of applying pointer offsets in order to directly access this memory mapped IO.

```
//Need these offsets, since OKL4 Page-aligns everything to the nearest
//Smallest      page      size      (which      is      0x1000).

    IBMR = okl4_range_item_getbase(&i2c_virtmem)+ (okl4_word_t)0x680;
    IDBR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x688;

    ICR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x690;
    ISR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x698;
    ISAR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x6A0;
```

The kernel debugger is incredibly useful in debugging low-level problems, allowing the developer to step through a program with entrances to the debugger placed at strategic spots, or view physical and virtual memory. Additionally, the kernel debugger can be used to view the Thread Control Block of any running thread, allowing for inspection of priority levels, and blocking states, as well as which thread the cell is blocking for. The kernel debugger can be entered any time during

hard-locked, and must be reset. Unfortunately, after a great deal of trouble, and a great many hours spent staring at a data sheet, the memory addresses appeared to be changing as directed, but the input/output buffers and data lines weren't performing any of the work.

It took a great deal of head banging in order to solve this problem. Eventually, it was necessary to tear apart the Linux driver in order to discover which additional sections memory accesses were being performed. A section of memory which maps alternate IO functions and enables clock distribution was discovered in the Linux drivers. In order to conserve power, the PXA270 will only distribute clock signals to the components which specifically request this distribution. Furthermore, all GPIO is initially mapped to GPIO functionality, and must be mapped to "alternate functions" based on an extensive and obtuse table located in the programming guide.

The simplest way to enable the clock and alternate IO functions would require mapping these memory regions into the I2C cell. This was unattractive. In order to maintain the separation between cells which the target architecture requires, it would be unreasonable to include such large swaths of responsibility and functionality with the program performing the simple action of generating digital and analog IO signals. Furthermore, as additional devices came into use, they too would require clock distribution and the mapping of alternate functions. In order to ensure that these functions are enabled, a plan for a hardware manager cell was generated. This hardware generator would enable the functions required, then send a one-way message to the cells, informing them that they had been enabled. This began the project's first foray into IPC. First, however, these functions were mapped direct into the I2C cell for the completion of its testing.

After these additional functions were discovered, added, and properly manipulated, the existing I2C code operated as predicted. The I2C code generated utilizes a polling-mode architecture, in order to avoid the complexities of mapping IRQ functionality into the I2C cell. As the CPU of the PXA270 run 4 orders of magnitude faster than the I2C bus, it is unlikely that this will cause any problem in the prototyping stage. In section 7, we will see that the effect of the I2C bus is largely negligible. Roughly, the code writes data to an 8 bit deep send buffer, instructs the I2C hardware of the status of start and stop bits, then sets a flag which instructs the I2C hardware to send the data in this buffer. The program executes a polling loop which detects the change in transmit buffer state, at which point the program is free to move on to the next write.

Section 4.2 – Hardware Management Cell

With the necessary addition of a hardware management thread, the great expedition into the land of Microkernel IPC began. In 2007, Dr. Jeff Hieb documented [5] the capability of an OKL4 based microkernel to perform IPC in 64.59 microseconds. Although reasonably fast IPC is a requirement of this project, simply determining the library functions and mechanisms by which IPC operated in the 3.0 version proved challenging enough at first.

There are several modes of operating the OKL4 IPC library calls. Both sending and receiving calls can operating in blocking or non-blocking states. Using non-blocking calls, it would be further trivial to include a timeout functionality which operated as a hybrid-blocking call. Furthermore, when a receiving thread or cell successfully completes reception of an IPC transfer, a reply cap is generated, which is a temporary, non-deterministic capability which allows the thread performing the receiving

action to initiate a reply message to the sending memory space without requiring that the memory space posses compile-time capabilities to communicate with that space. A blocking receive command, known as `ipc_wait`, has the (arguable) benefit of waiting for any thread which posses the capability to communicate therewith.

For simplicity's sake, the hardware manager thread configures the hardware registers, then dispatches a message to the appropriate cell. This dispatch is blocking. The cell waiting for the arrival of this message enters a blocking wait as soon as all critical thread-initiation code is executed. Although a blocking wait allows any cell with capabilities to the receiving cell to send a message at any time, a system of magic numbers has been adequate up to this point, although a more complex abstraction may be necessary in the future. The initial design for the hardware manager thread involved a thread exit following the initialization of each hardware device. Unfortunately, the `exit()` system call in OKL4 is primitively fleshed out with a `while(1)` infinite loop. Rather than obfuscate the code with such a system call, this loop was added manually.

Section 4.3 – Serial Debug Cell

This primitive hardware manager cell allowed the memory space and functions of actual hardware devices to remain separate from the act of configuring and enabling these devices. This simplifies programming, enabling the avoidance of Mutex interfaces and multiple-access to identical memory regions. This also allowed the developers some practice at the IPC faculties included in the OKL4 library. At this point, with multiple interacting cells already in existence, and more on the way, a new manner of creating debug output became necessary.

Although the Kernel debugger is able to use the FFUART at the same time as another cell with a simple `<use device>` XML tag, multiple cells cannot share this serial port. Furthermore, the serial port implementation is somewhat primitive, and functions such as `getc()` are not implemented. As a result, only one cell can be debugged at a time using the kernel debug serial output, and no input can be generated. In order to (again) avoid the issues of device contention and mapping memory as read-write into multiple cells, and avoid mangled serial output, a serial cell would be created. This serial cell would accept input from any other cell. This input would consist of a magic number, followed by a command, followed by the cell's ASCII name. If the command is a direction to print data to the screen, the following data would consist of the data to be printed.

In order to keep this serial faculty simple, all transactions must fit inside of a single IPC call. On the Gumstix platform, this consists of 146 bytes. This is adequate for debugging purposes. Aside from additional complexity, there is no practical reason that this could not be extended indefinitely. The current implementation foolishly assumes that the data to be transmitted must be formatted as an `okl4_word_t` data type (32 bits), and includes complicated packing routines to fit char sized data into `okl4_word_t` data. This is completely unnecessary, but is included in the code at the moment, nonetheless.

Throughout the process of implementing this routing, the most difficulty was encountered in initializing and creating the proper structures and functions to implement `printf` and `scanf` functionality. The OKL4 operating system includes full `stdio.h` function compatibility. Unfortunately, `printf` is a complicated [7] piece of code to tame. Furthermore, in order to simplify execution on

both ends, it was necessary to implement **`printf`** and **`scanf`** functionality on both ends of the IPC call.

Functionally, there is a cell and a library portion to the added serial driver. The cell portion invokes the actual hardware of the STUART. A simple **`getc`** and **`putc`** method are created, which are mapped into a file **`struct`**, calling these function pointers when necessary. Recipient functions receive the formatted IPC, and print the cell name, followed by the message to be printed, or the prompt to be printed, depending on the command mode requested by the sending cell. In the case of a simple print request, the data is printed, and a confirmation word is sent in reply. In the case of a request for input, the data is returned with the appropriate confirmation word for processing by the recipient cell.

On the library side, a cell wishing to utilize this debug output need only include a header file. There are two initialization functions, one is required to properly configure the sending capabilities required to perform the IPC. The other configures an optional cell name which is transmitted in ASCII with every request to print or receive data. This is simplifies the process of determining which cell is sending which message, important when multiple cells may be attempting to send and receive data at nearly the same time. Fortunately, due to the single IPC/reply nature of this transaction, race conditions cannot exist. A cell wishing to transmit data while another is printing data is simply placed into a blocking state until the previous action is completed.

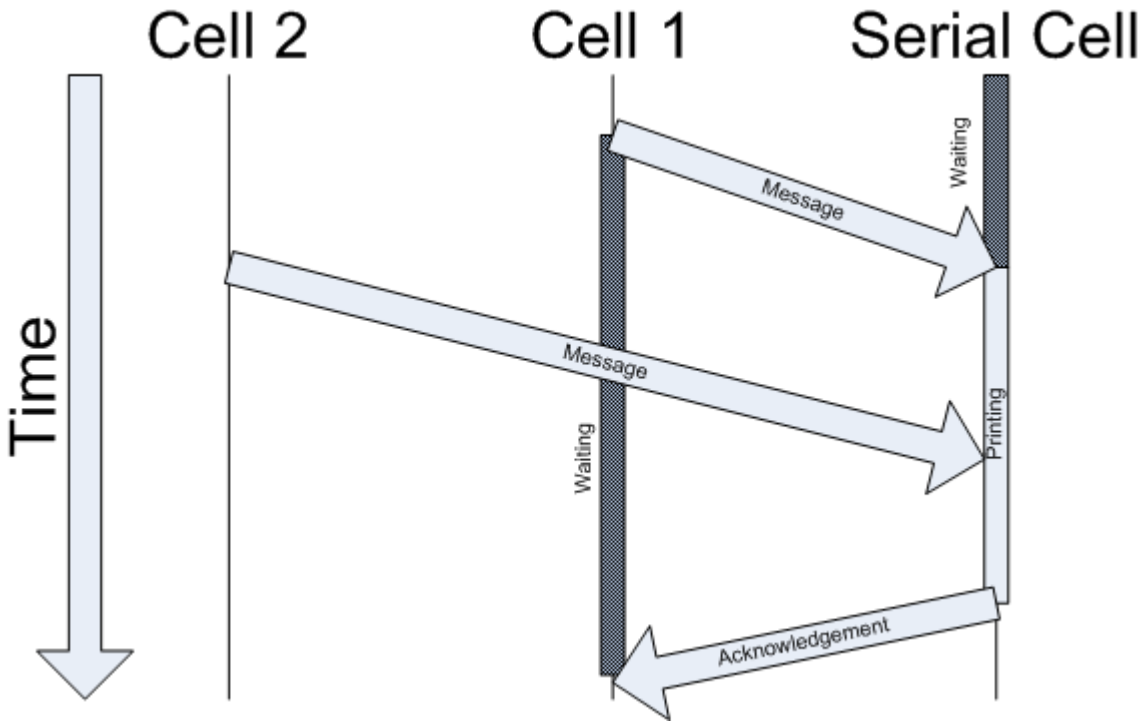


Figure 8: Timing Diagram demonstrating Serial Cell's resistance to interruption.

Section 4.4 – Net IO Cell

Configuring networking was a major undertaking. Most of the work on this section of the project was performed by Dr. Jeff Hieb. At first, tentative adventures into the Linux source code would be the source for this driver code. Unfortunately, the depth and breadth of the Linux influence on this driver kept the abstraction at a fairly high level. Much would would be required in order to adapt the Linux driver. Fortunately, the majority of this problem was easily tackled by borrowing driver source code from the uBoot project. This source code, as part of a project designed to simply bootstrap embedded systems, is very low level, yet very well documented. As a result, minimal adaptation was required in order to render the source code functional inside an OKL4 environment. Most simply was the mapping of memory. The network device is directly memory mapped, using a section of memory designed for use with additional RAM. As a result, the translation was fairly simple.

A few problems, however, were still encountered. Among them, the complexity of the driver. The uBoot driver is fully compatible with the hardware of the SMC9118 chip set. The SMC9118 includes multiple packet buffers, which are used in a round-robin fashion in order to maintain quality of service and data rate. The buffers are loaded from an even deeper hardware memory buffer. As this project does not require the ability to handle large amounts of data at this point, it was a challenge to remove this buffer faculty. There was also a non-terminating loop in the `eth_rx()` function. Simply removing a misplaced break call, and restructuring the loop solved the problem, although transient effect are as of yet unknown. Finally, many of the uBoot memory reads were not written as volatile, causing odd compiler optimization behavior. While seeking these troubles out took a good deal of time, it was not particularly difficult. Any issue of timing is likely caused by a missing "`volatile`" tag. If a large enough delay or debug statement inside the loop causes the loop to function as expected, then the problem is almost certainly a missing "`volatile`" keyword.

Less fortunately, this driver required the mapping of a hardware timer. Hardware timers are in short supply on the PXA270, and complicating this matter is the fact that the Operating System requires one of the timers. Unfortunately, it isn't plainly documented which timer is used for operating system purposes, or how. Furthermore, all the timers on the PXA270 are mapped into the same memory space. Any device wishing to use any timer had to use all the timers.

Timer Sharing is necessary with the introduction of the Net IO cell. The operating system uses timers to properly preempt threads, while the network driver uses timers to generate timeouts on processes which might otherwise take forever. Furthermore, as performance testing loomed on the horizon, the timer on the PXA proved to be the easiest option for quantitatively describing performance. Fortunately, all of these services (with the possible exception of the operating system) require read-only access to the timer function. Without knowing how the operating system makes use of the timer, manual inspection of physical memory confirmed the speed at which the timer operated, and the implementation of a simple timer function simply required a read-only mapping of memory into each cell. This memory is mapped as a static memsection using elfweaver XML calls. This multiple-mapping of read only memory poses no difficulty with simultaneous access. The timer function simply observes the current timer value. Unfortunately, there is no tracking of rollover-scenarios. As a result, there is a small probability of a wait-state lasting as long as 22 minutes longer than anticipated.

This timer faculty is a simple means of generating performance statistics, as well as a way to

implement error checking, ensuring that wait states do not last forever. Although this functionality has not yet been implemented in the interest of time, the ability to do so exists, nonetheless.

Simultaneous to the development and polishing of the network and timer faculties, some initial performance measurements were made.

Section 4.5 – Utility Cell Genesis

Unfortunately, by this point, the architecture of the prototype had diverged quite a bit from the target architecture (Figure 1). The addition of hardware manager, serial debugger and test cell brings the total number of separate memory spaces to 6, including the IO, Ethernet Access and Security cells. This causes trouble with the 5 cell limit caused by a bug in Elfweaver. Fortunately for the architecture of the project, the hardware manager, serial debugger, and test cell can all remain in the untrusted computing base. Furthermore, in order to separate them from the target architecture, they can be included in a memory space of their own. Although this represents a minor diversion from the target architecture, it does not compromise the overall security of the project. Further precautions against potential security problems will be described in section 7.

The change in architecture brings the total number of cells down to 4, which fits within the reluctant limit imposed by elfweaver. Simply by altering the source tree, and the make script for the individual cell, each cell can be transformed into a thread with a fair amount of ease. Threads which exist within cells behave differently from the single threaded cells utilized throughout the project to this point. Unless special action is taken, all threads within a cell have unfettered access to the virtual memory assigned to the thread. Although each thread is assigned its own heap and stack, it is able to explicitly reference memory outside of this heap and stack, unprotected. Although this function makes OKL4 ideal as a microvisor [8], it introduces potential security holes. There are many solutions to this problem, which will be described in Section 7.

In OKL4, there are two methods to starting threads within a cell. Threads can be started at run-time, by a cell's main thread, or they can be started at compile time, by using elfweaver calls. The benefit to run-time initialized threads is the ability to dynamically control properties such as protection domains, memory mappings, heap and stack size, and other properties of the thread. Capabilities for threads generated in this manner, however, cannot be determined at compile time. To this point, all capabilities have been passed into cells via elfweaver XML tags. Although it would be possible to pass these permissions at run time, such an implementation would be difficult. Although run time changes cannot easily be made to threads started with XML tags, capabilities are known at the time that elfweaver is executed.

This is the point at which the bugs described in Section 3.3 were encountered. Although these errors were fairly trivial, they took a great deal of time to trace through the elfweaver code. Once these were solved, as described above, threads started at compile-time operated as predicted.

Section 5 – Initial Performance Measurements

In order to submit a paper to the ISCA conference on Advanced Computer Communications, some performance measurements were required. Although the current implementation is primitive compared to the target device, some end-to-end performance measures could be made by simply passing IPC between cells. The IPC calls are the interesting portion of performance measurement. Jeff Hieb showed [5] that performance could be as good as 64.59 microseconds. Including the IO operation, how quickly could a response be generated once the desired action is parsed? At this time, a new Test cell was created, purely to pass messages to the IO cell. See Figure 9 for a simple explanation of the message passing sequence.

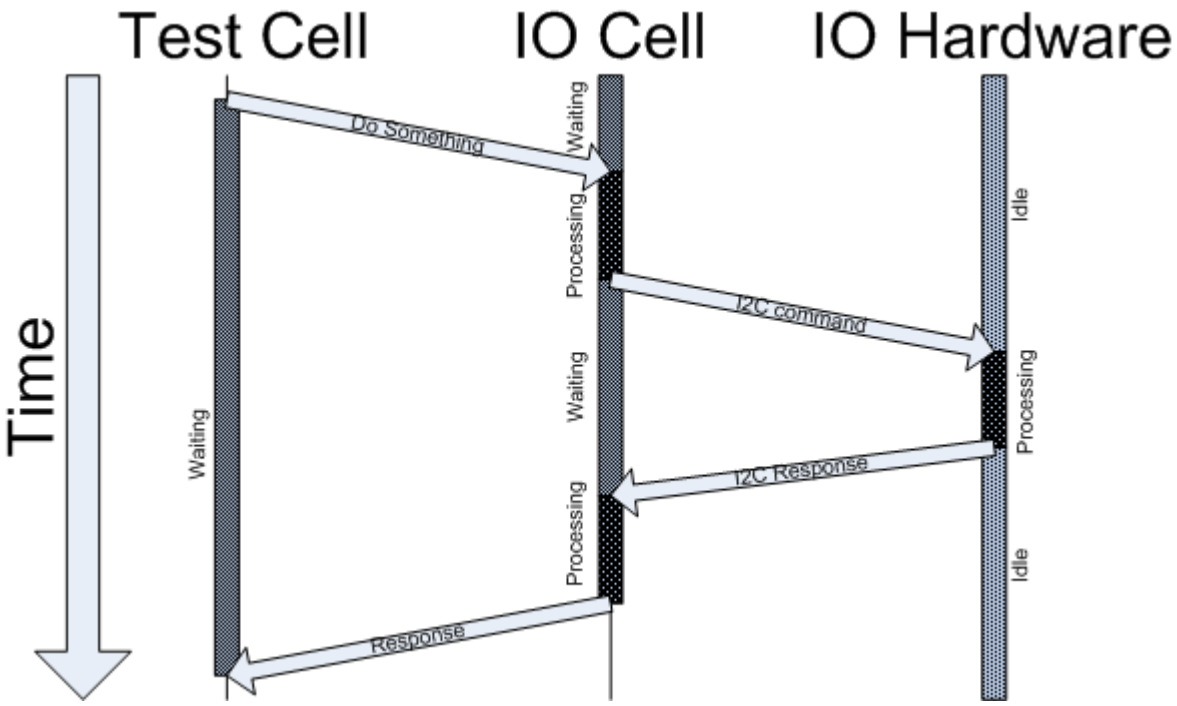


Figure 9: Example of IPC Speed Test.

The initial performance measurements utilizing this model were disappointing, falling orders of magnitude above the figures cited by Hieb. This indicated to us that there was a problem with the prototype software. The simple library calls made by the OKL4 communication faculties could not account for this error.

Table 1: Initial Performance Measurements

	Metric	ADC Performance		DAC Performance		GPIO Performance	
		Value	Units	Value	Units	Value	Units
IPC+I2C	Mean Time	11.342277	milliseconds	11.342283	milliseconds	11.342302	milliseconds
	St. Deviation	0.001536	milliseconds	0.001591	milliseconds	0.001553	milliseconds
	N	999		999		999	
	Max	11.343692	milliseconds	11.343692	milliseconds	11.344308	milliseconds
	Min	11.296308	milliseconds	11.294462	milliseconds	11.296000	milliseconds
I2C	Mean Time	0.503155	milliseconds	0.304515	milliseconds	0.411659	milliseconds
	St. Deviation	0.002334	milliseconds	0.002175	milliseconds	0.002312	milliseconds
	N	999		999		999	
	Max	0.519077	milliseconds	0.311692	milliseconds	0.422462	milliseconds
	Max	0.499692	milliseconds	0.300923	milliseconds	0.408308	milliseconds
	IPC Overhead	95.56%		97.32%		96.37%	

Before attempting to solve this problem, however, an architecture change was made to this test routine which resulted in a scenario much closer to the eventual end-produce, inserting a cell between the test cell and the IO cell, to simulate a security cell. Although this cell in the middle would do nothing but pass messages on, it would still be useful to model the delay imposed by multiple IPC passes. In order to avoid the 5 cell maximum problem, the hardware manager cell was drafted to act as the "man-in-the-middle." After the hardware manager initialized its hardware devices, it would sit in an IPC wait state, and simply pass messages on as shown below:

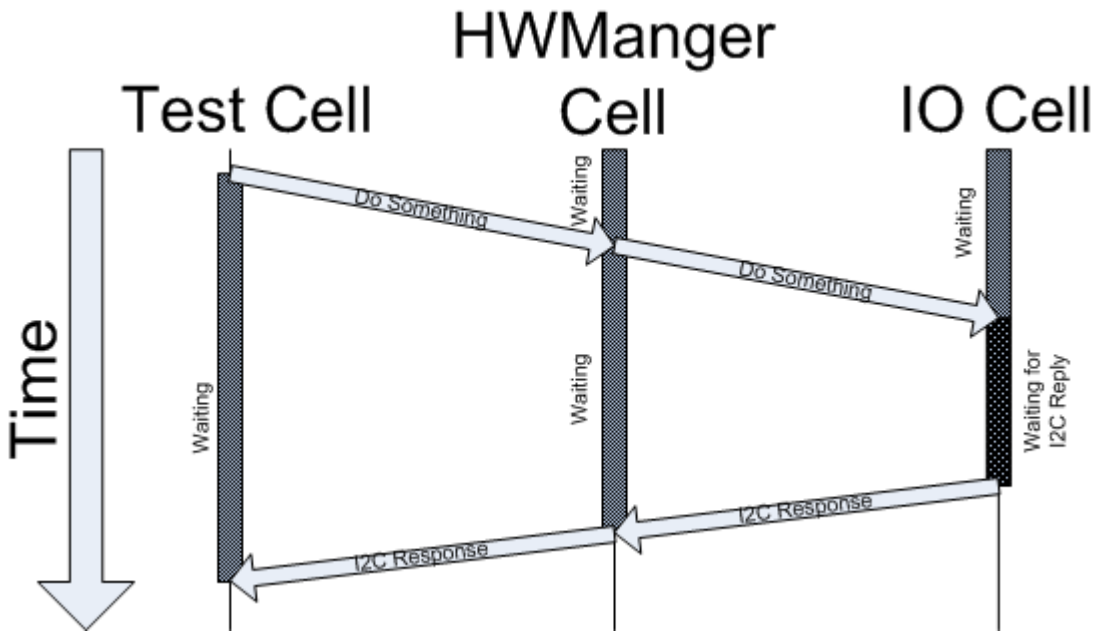


Figure 10: Example of Modified IPC Speed Test.

Amazingly, as soon as this change had been made, the amount of time required to perform IPC calls, even with the additional cell, dropped by 2 orders of magnitude. The hardware manager cell had been entering a **`while(1)`** loop, rather than calling a dummy **`exit()`** system-call. This while loop didn't preempt the hardware manager cell. As a result, each time slice assigned to the hardware

manager cell was utilized fully doing absolutely nothing. Although a method for preempting threads has yet to be discovered, for the moment this is not a problem.

The performance measurements made after this change can be seen below:

Table 2: Performance Measurements after Insertion of 3rd Cell.

	Metric	ADC Performance		DAC Performance		GPIO Performance	
		Value	Units	Value	Units	Value	Units
IPC+I2C	Mean Time	533.651190	microseconds	334.391622	microseconds	441.664434	microseconds
	St. Deviation	3.117140	microseconds	2.414563	microseconds	3.033070	microseconds
	N	999		999		999	
	Max	553.846154	microseconds	344.923077	microseconds	458.769231	microseconds
	Min	528.923077	microseconds	329.846154	microseconds	436.615385	microseconds
I2C	Mean Time	503.463771	microseconds	304.241780	microseconds	411.388311	microseconds
	St. Deviation	2.877504	microseconds	2.316978	microseconds	2.583055	microseconds
	N	999		999		999	
	Max	520.923077	microseconds	314.153846	microseconds	421.846154	microseconds
	Max	499.692308	microseconds	300.307692	microseconds	407.384615	microseconds
	IPC Overhead	5.66%		9.02%		6.86%	

This performance is more than adequate for the prototype. This is simply an initial measure of performance, and there are many changes which could be implemented to improve these numbers, which will be discussed in section 7.

Section 6 – Complete Communication Test

The final changes implemented before the close of the Summer 2010 semester implemented a simple Ethernet blink. Dr. Jeff Hieb had completed a simple Linux program which generated a raw Ethernet frame, formatted thus:

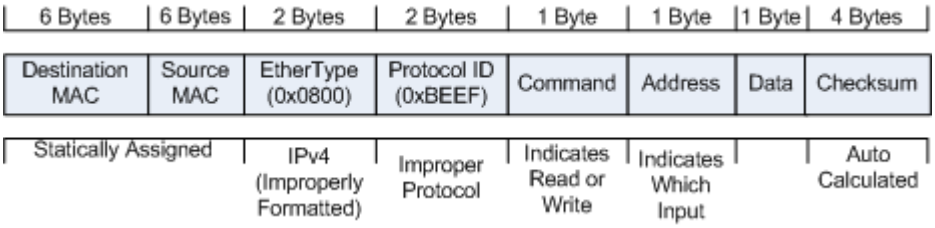


Figure 11: Ethernet frame used for testing.

Raw Ethernet packets simplified implementation, and sped development by avoiding the complexities of higher level protocols. The Gumstix Access Cell simply ignored all packets which did not match the Protocol number above. The implementation of a network blink proved the entire IO path, from Ethernet packet reception, through IPC, into the I2C IO.

Section 7 – Summary of Past and Projected Development

In summary, during the course of this semester, a successful set of utilities has been configured with an out-of-box OKL4 tool-chain, hardware has been selected and assembled to generate IO appropriate for an RTU, this hardware has been assembled and tested, and code for driving this hardware has been developed. Additional cells for required tasks such as hardware management, testing, and serial debug output have been programmed and tested, and introductory performance testing has been performed using this code. During this process, Dr. Graham, Dr. Hieb, and Brad Luyster have authored a conference paper which has been accepted to the ISCA conference on Advanced Computer Communications. Code for implementing a simple network driver has been discovered, modified and implemented into the prototype, and a simple Ethernet packet generator can test the entire communication string, from Ethernet packet to IO.

During the course of the semester, many small and large hurdles were overcome, although there is still quite a bit of room for improvement in much of the code. To begin, fixing the bugs in elfweaver regarding the 5 cell limit would be extremely helpful in future development, regardless of the final architecture of the project. Fixing these problems would allow for comparative testing between properties which are determined at compile time, and properties which are determined at run time. This is particularly useful in understanding how mechanisms like Protection Domains, Zones and Memory Sections actually work when applied to areas within a logical cell. It would be exceptionally useful to further isolate the code located in the utility cell using protection domains, should they prove to operate as described. This would allow for cleaner logical separation from portions of the untrusted computing base which interact with trusted computing base, and portions of the UTCB which do not.

There are many improvements to be made in the IO cell, and the I2C code. Particularly, at the moment this code is susceptible to race conditions should a cell be allowed permissions to the IO cell, and send an IPC message at an inappropriate time. This is simply solved by a variety of means. Furthermore, performance of this cell can be enhanced by increasing the speed of the I2C bus from the default 100kHz to the maximum 400kHz. At the moment, over 90% of each IO operation is in I2C overhead. This could be reduced by a factor of 4. Furthermore, there are many timeout conditions which could be caused by loops which never return. Timeouts can now be implemented with the addition of a timer device, and should be utilized.

The hardware manager is now running as a compile-time thread inside the utility cell. At the moment, this thread is still acting as part of a test sequence, as described in Section 4.5. Eventually, however, it will not. In order to avoid the performance troubles caused by an unending while loop, a method of preempting or ending the thread will need to be implemented. At the moment, with the thread statically initiated at compile time, there are no known library hooks to perform this action. However, simply lowering the priority of the thread lower than any other thread operating on the system would ensure that it never receives another time slice. This voluntary preemption mechanism will be useful for other threads as well, particularly those activated by hardware. Threads which receive IPC messages will be placed in a wait-state until they receive an IPC message, so they need not be preempted in this manner.

The serial debug cell would make great use of a good number of improvements. Most of all,

the ham-handed methods by which message are packed and unpacked into 32 big structures. Although unknown at the time, the IPC functions require only the messages passed be byte-aligned, not word-aligned. As a result, the complex pointer code which iterates through the passed strings in order to “maximize” the number of characters sent is absolutely needless. Only a few lines of code need be changed for the native type assumed by the cell to change to a char array. In addition to this, it would be remarkably useful if the serial debug input were able to generate an arbitrary size of output, without being susceptible to interference from other threads. Theoretically, using only reply caps should avoid this possibility. There are many simple solutions to this problem.

The Network Access cell is in a similar state of infancy. At this point, it is able to receive simple Ethernet frames, decode a message, and dispatch a frame in reply. At a minimum, it would be useful to implement UDP into such a device. In this manner, simpler programs could be written to interact with the device over the network. Although the implementation details of TCP may be beyond the scope of this project, some higher level protocol will prove very useful in the testing phases. Furthermore, at some point it will be useful for the device to decode and parse commands given over a basic control system protocol, such as Modbus. Again, a more complex protocol such as DNP3 may be beyond the scope of this project.

The hardware itself is largely comparable to existing RTUs in both number, speed, and accuracy. Unfortunately, most control device expect a signal which ranges from 0-9 volts or 0-20 milliamps. The output generated by the device at the moment isn't nearly so robust. In the future, it will be beneficial to modify the device with an signal conditioning layer, which alters the output signal to conform with these standards. Furthermore, opto-isolation of these components would be tremendously beneficial for the protection of the prototype hardware, and its more expensive components.

There are a few potential security flaws which should, at the very least, be investigated. First, many cells, including any using the hardware manager cell, enter a wait state upon start-up. Others repeatedly enter a wait state, while waiting for an appropriate IPC message. The security cell, too, will require interaction with cells located outside the Trusted Computing Base. As a result, it will be beneficial to test the segmentation provided by the supply cap functionality, and ensure that no cell has explicit capabilities to the security cell without also being a member of the trusted computing base. This is especially important if the design of the multi-threaded utility cell remains the same. Without the strict memory separation provided by the “cell” structure, it could be possible for a thread within the cell to interfere with the operation of a thread which possesses capabilities to the security cell. There is an alternate method to IPC known as “notification” which simplifies the operation of sending and receiving short messages. This may be useful in the context of communicating between the TCB and UTCB.

There are additional security threats possible in a multi-threaded context, as without explicit declaration, each thread within a cell possesses unfettered access to the whole of the memory assigned to the cell. In addition to a security threat, this is a prime vector for bugs to sweep through the code and cause havoc. It further remains to be seen whether declarations made in Elfweaver XML files are identical in functionality to those made at run time, regarding protection domains and memory segmentation and separation. Although these methods of attacking the problem likely have the same result, verification will be useful, given the bugs discovered in elfweaver to this point.

Further development will focus on fixing the flaws mentioned earlier, as well as adding robustness to the Network Access cell. At this point, development can begin to venture into the security cell, and the methods by which the security of the device can be maintained through this cell can be explored, as well as their speed and effectiveness. In order to test the chain of events from start to finish, a test platform will be configured with LabView software. A LabView simulation will be created which will both issue and receive the messages and IO interacting with the device. In this manner, the device can be tested for speed, accuracy, and security before leaving the lab, and before the signal conditioning hardware is complete.

Section 8 – References

[1] J. Hieb, J. Graham, and S. Patel, *Security Enhancements for Distributed Control Systems*, Boston, MA: Springer US, 2007.

[2] G. Klein, J. Andronick, and K. Elphinstone, "seL4: formal verification of an operating-system kernel," 2010, pp. 107-115.

[3] Intel, *Intel PXA27x Processor Family Developer's Manual*. Available: http://www.marvell.com/products/processors/applications/pxa_family/pxa_27x_dev_man.pdf

[4] J. Liedtke, "Improving IPC by kernel design," *ACM Symposium on Operating Systems Principles*, vol. 27, 1994, p. 175.

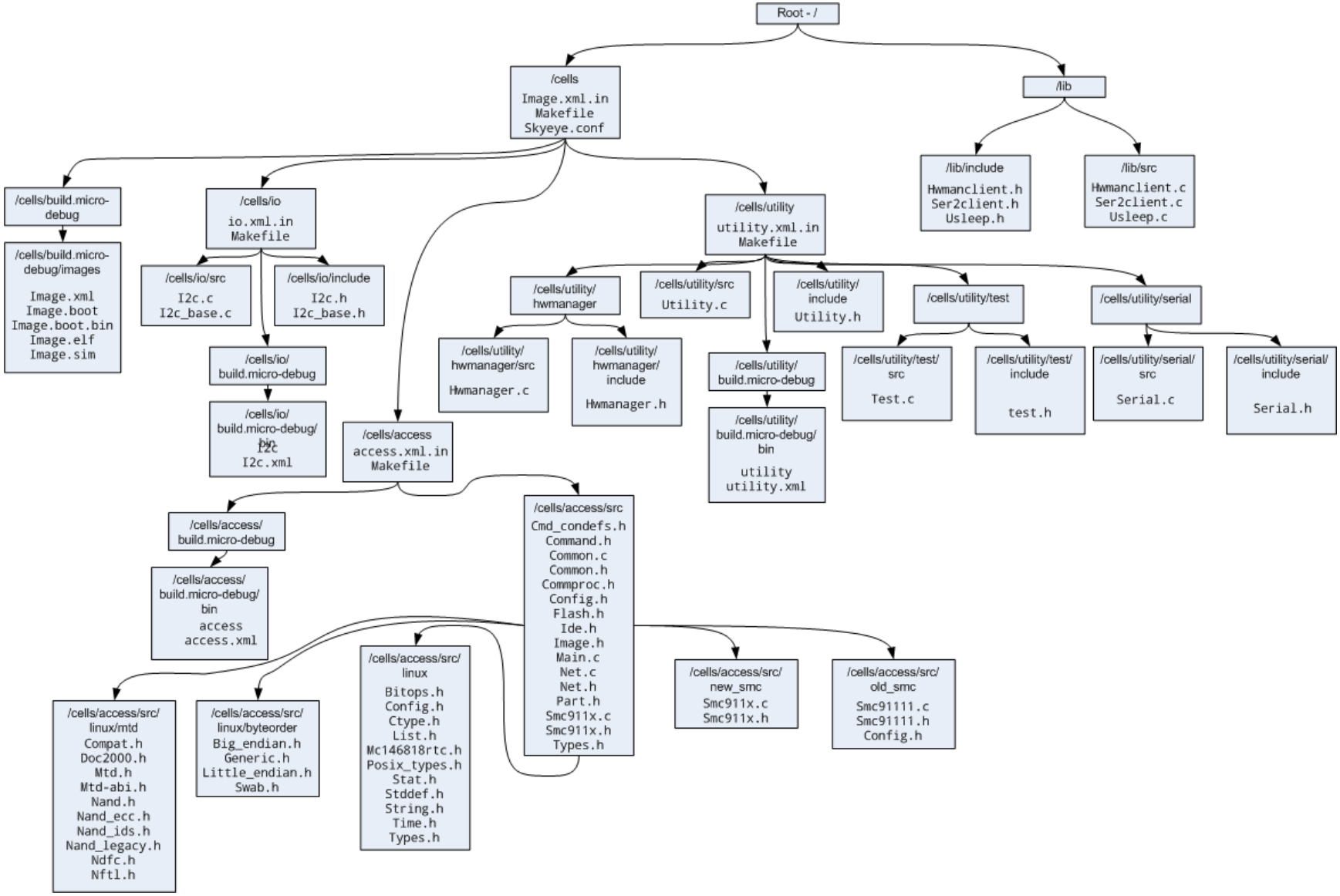
[5] J. Hieb and J. Graham, *Designing Security-Hardened Microkernels For Field Devices*, Boston, MA: Springer US, 2009.

[6] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of µ-kernel-based systems," *ACM Symposium on Operating Systems Principles*, vol. 31, 1997, p. 66.

[7] "Where the printf() Rubber Meets the Road," 2010. Available: http://hostilefork.com/2010/03/14/where-the-printf-rubber-meets-the-road/

[8] J. Liedtke, "On micro-kernel construction," *ACM Symposium on Operating Systems Principles*, vol. 29, 1995, p. 237.

Appendix A – Directory Structure of Project Source



Appendix B – Summary of Tables and Code Excerpts

The following sections will include the source code generated to this point in the project. Header files will not be included in order to maintain the brevity of this document.

Appendix B.1 – IO Cell

The IO cell interacts directly with the I2C hardware, as well as receiving IPC from the Security cell. At the moment, the Hardware Manager passes messages directly to the I2C cell, without any security processing. **`i2c_base.c`** contains low level I2C commands and interfacing instructions, while **`i2c.c`** contains higher level abstractions and accesses to IPC.

In order to test the performance of the I2C bus, simply uncomment the appropriate code in the switch-case structure.

Appendix B.1.1 – i2c_base.c

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need malloc, which is in stdlib
#include <stdlib.h>

// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <i2c_base.h>
#include <hwmanclient.h>

//i2c Register names
okl4_word_t IBMR, IDBR, ICR, ISR, ISAR;

/**
* Sends an I2C start
*
* Sends the start command to device address
*
* @param address
*   7 bit address, shifted left 1, with read/~write appended to the end.
*
* @return
*   0 if success, 1 otherwise
*
* @todo add timeout, and non-zero return value.
```

```
**/
int i2c_start(okl4_word_t address)
{


        inw(IDBR) = address & 0x000000ff;
        inw(ICR) |= _BV(START)| _BV(TRANSFER);
        while(( vinw(ISR) & _BV(ISR_ITE)) != 0x00000040)
        {
                if((vinw(ISR) & _BV(ISR_BED)) == 0x00000400)
                {
                        //did not get ack
                        return 1;
                }
        }
        inw(ISR) = 0xffffffff & _BV(ISR_ITE);
        inw(ICR) &= ~_BV(START);
        return 0;

}

/**
* TODO: Document This
*       Return from some errors properly
**/
int i2c_init(void)
{

        okl4_memsec_t         * i2c_memsec;
        okl4_env_segment_t    * i2c_seg;

        okl4_static_memsec_t  * i2c_static_memsec;
        okl4_static_memsec_attr_t i2c_static_memsec_attr;

        okl4_virtmem_item_t   i2c_virtmem;


        int error;
        // begin execution
        okl4_init_thread(); // this sets up okl4 lib, call for every thread.
                            //   may not need thsi due to weaver call
        L4_KDB_SetThreadName(L4_Myself(), "I2C");

        i2c_seg=okl4_env_get_segment("MAIN_I2C_MEM0");
        assert(i2c_seg);

        okl4_static_memsec_attr_init(&i2c_static_memsec_attr);
        okl4_static_memsec_attr_setsegment(&i2c_static_memsec_attr,i2c_seg);

        i2c_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&i2c_static_memsec_attr));
        assert(i2c_static_memsec);

        okl4_static_memsec_init(i2c_static_memsec,&i2c_static_memsec_attr);

        i2c_memsec=okl4_static_memsec_getmemsec(i2c_static_memsec);

        i2c_virtmem=okl4_memsec_getrange(i2c_memsec);


        //Need these offsets, since OKL4 Page-aligns everything to the nearest
        //Smallest page size (which is 0x1000).
        IBMR = okl4_range_item_getbase(&i2c_virtmem)+ (okl4_word_t)0x680;
        IDBR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x688;
        ICR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x690;
        ISR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x698;
        ISAR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x6A0;

        error = hwman_resource_block();
        assert(!error);
```

```
	i2c_reset();
	inw(ICR) |= _BV(SCLEN) | _BV(I2CEN);
	return 0;
}

/**
* TODO: Document This
*	Return from some errors properly
**/
int i2c_reset(void)
{

	inw(ICR) |= _BV(I2CRESET);
	inw(ISR) = 0x00000000;
	inw(ICR) &= ~_BV(I2CRESET);
	return 0;

}

/**
* TODO: Document This
**/
int i2c_send(okl4_word_t data)
{
	inw(IDBR) = data & 0x000000ff;
	inw(ICR) |= _BV(TRANSFER);
	while(((vinw(ISR) & _BV(ISR_ITE)) != 0x00000040))
	{
		if((vinw(ISR) & _BV(ISR_BED)) == 0x00000400)
		{
			//did not get ack
			return 1;
		}

	}
	inw(ISR) = 0xffffffff & _BV(ISR_ITE);
	return 0;
}

/**
* TODO: Document This
*	Return from some errors properly
**/
okl4_word_t i2c_get_ack(void)
{
	okl4_word_t data;
	inw(ICR) |= _BV(TRANSFER);
	while((vinw(ISR) & _BV(ISR_IRF)) != 0x080)
	{
		if((vinw(ISR) & _BV(ISR_BED)) == 0x00000400)
		{
			//error condition
		}
	}
	inw(ISR) = _BV(ISR_IRF);
	data = inw(IDBR);
	inw(ICR) &= ~_BV(STOP) & ~_BV(ACKNAK);
	return data;

}

/**
* TODO: Document This
*	Return from some errors properly
**/
okl4_word_t i2c_get_nak(void)
{
```

```
        okl4_word_t data;
        inw(ICR) |= _BV(TRANSFER) | _BV(ACKNAK);
        while((vinw(ISR) & _BV(ISR_IRF)) != 0x080)
        {
                if((vinw(ISR) & _BV(ISR_BED)) == 0x00000400)
                {
                        //error condition
                }
        }
        inw(ISR) = _BV(ISR_IRF);
        data = inw(IDBR);
        inw(ICR) &= ~_BV(STOP) & ~_BV(ACKNAK);
        return data;
}

/**
* TODO: Document This
*       Return from some errors properly
*       Implement some sort of timeout
**/
int i2c_stop(void)
{
        //Send a stop without data.  This gives us
        //Lots of flexibility in our code.
        inw(ICR) |= _BV(ABORT);
        while((vinw(ISR) & _BV(ISR_UB)) != 0)
        {
                //wait for stop
        }
        inw(ICR) &= ~_BV(ABORT);
        return 0;
}
```

Appendix B.1.2 – i2c.c

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need malloc, which is in stdlib
#include <stdlib.h>
// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>

#include <okl4/message.h>

#include <i2c.h>
#include <i2c_base.h>
#include <hwmanclient.h>
#include <ser2client.h>
#include <usleep.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
```

```
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <serial/serial.h>

void gpio_init(void)
{
        i2c_start((okl4_word_t)0x40);
        i2c_send((okl4_word_t)0x00);
        i2c_send((okl4_word_t)0xFE);
        i2c_stop();

}

okl4_word_t convert(void)
{

        okl4_word_t data1;
        okl4_word_t data2;

        i2c_start(MYADC+0);
        i2c_send(0x80);
        i2c_start(MYADC+1);
        data1 = i2c_get_ack();
        data2 = i2c_get_nak();
        i2c_stop();

        return data1<<8 | data2;
}


void adc_init(void)
{

        i2c_start((okl4_word_t)MYADC+0x0);
        i2c_send(0x02);
        i2c_send(0x00);
        i2c_send(0x18);
        i2c_stop();

}

void dac(okl4_word_t address, okl4_word_t value)
{

        okl4_word_t upper_b = (value>>8) & 0x0f;

        i2c_start(address+0);
        i2c_send(upper_b);
        i2c_send(value);
        i2c_stop();
}

void blink(void)
{
        i2c_start((okl4_word_t)0x40);
        i2c_send((okl4_word_t)0x09);
        i2c_send((okl4_word_t)0x01);
        i2c_stop();
        usleep(1000000);
        i2c_start((okl4_word_t)0x40);
        i2c_send((okl4_word_t)0x09);
        i2c_send((okl4_word_t)0x00);
        i2c_stop();
        usleep(1000000);
}

char getAllDigital(void)
{
        okl4_word_t retval;
```

```
        i2c_start(MYGPIO);
        i2c_send(0x09);
        i2c_start(MYGPIO+1);
        retval = i2c_get_nak();
        i2c_stop();

        return (char)retval;
}

char setAllDigital(void)
{
        i2c_start(MYGPIO);
        i2c_send(0x09);
        i2c_send(0xFF);
        i2c_stop();

        return 0x00;
}

char clearAllDigital(void)
{
        i2c_start(MYGPIO);
        i2c_send(0x09);
        i2c_send(0x00);
        i2c_stop();

        return 0x00;
}

char requestDigital(char selected)
{
        char retval;

        retval = getAllDigital();

        return retval & selected;
}

char setDigital(char selected, char state)
{
        i2c_start(MYGPIO);
        i2c_send(0x09);
        i2c_send((okl4_word_t)(selected & state));
        i2c_stop();

        return 0x00;
}

okl4_u16_t getAnalog(char selected)
{
        okl4_u8_t data1;
        okl4_u8_t data2;
        okl4_word_t channel;

        if(selected > 0x07)
        {
                return 0;
        } else {
                channel = (selected << 4) | 0x80;
                i2c_start(MYADC+0);
                i2c_send(channel);
                i2c_start(MYADC+1);
                data1 = (okl4_u8_t)i2c_get_ack();
                data2 = (okl4_u8_t)i2c_get_nak();
                i2c_stop();

                return data1<<8 | data2;
        }
```

```
}

char setAnalog(char selected, okl4_u16_t data)
{
	if(selected == 0x00)
	{
		dac(MYDAC1, (okl4_word_t)data);
	} else if(selected == 0x01)
	{
		dac(MYDAC2, (okl4_word_t)data);
	}
	return 0x00;
}

int scanForMessage(void)
{

	int error;

	okl4_word_t bytes;
	char buffer[MAX_CHARS];
	okl4_kcap_t client;

	okl4_u16_t holder;

	okl4_word_t time1, time2;

	char response[4];
	error = okl4_message_wait(buffer, MAX_CHARS, &bytes, &client);
	assert(!error);
	if(bytes > MAX_CHARS)
	{
		//Handle this error
	}
	else if(bytes > 4*sizeof(char))
	{
		//Handle this error
	}
	else
	{
		response[0] = buffer[0];
		switch(buffer[0])
		{
			case 0x00:
				//Request all Digital lines
				time1 = makeNote();
				response[1] = getAllDigital();
				time2 = makeNote();
				lprintf("%d ", (int)(time2 - time1));
				break;
			case 0x01:
				//set all digital lines
				response[1] = setAllDigital();
				break;
			case 0x02:
				//clear all digital lines
				response[1] = clearAllDigital();
			case 0x03:
				//request digital lines indicated by next byte
				response[1] = requestDigital(buffer[1]);
				break;
			case 0x04:
				//set digital lines indicated by second byte to state indicated by 3rd
byte
				response[1] = setDigital(buffer[1], buffer[2]);
				break;
			case 0x05:
				//request analog input indicated by next byte
```

```
                                time1 = makeNote();
                                holder = getAnalog(buffer[1]);
                                time2 = makeNote();
                                lprintf("%d ", (int)(time2 - time1));
                                response[1] = (char)(holder >> 8);
                                response[2] = (char)(holder);
                                break;
                        case 0x06:
                                //set analog output indicated by 2nd byte to values indicated by the top
12 bits of the next 2 bytes.
                                time1 = makeNote();
                                holder = 0x0000;
                                holder |= buffer[3] << 8;
                                holder |= buffer[4];
                                response[1] = setAnalog(buffer[1], holder);
                                time2 = makeNote();
                                lprintf("%d ", (int)(time2-time1));
                                break;
                        case 0x07:
                                //set mode of pins indicated by second byte to values of third byte (1=
input, 0 = output)
                                break;
                        case 0x08:
                                blink();
                                response[1] = (char)0xff;
                                break;
                }
        }

        error = okl4_message_reply(client, response, 4*sizeof(char));
        assert(!error);

        return 0;
}


/*
*  start the main program
*  much of this can be redone in functions
*  and #define and MACROs
*  but for now leave everything in
*  main so we can easily see what is going on
*/
int main(int argc, char** argv)
{

        i2c_init();

        sleep_init();
        gpio_init();
        adc_init();

        setCellName("i2c");
        ser2client_init();

        lprintf("Test\r\n");

        while(1)
        {
                scanForMessage();
        }

        L4_KDB_Enter("i2c test done");
}
```

Appendix B.1.3 – io.xml.in

```
<okl4 priority="255" clists="256" file="io" kernel_heap="0x400000" mutexes="256" name="i2c"
spaces="64">
        <!-- <use_device name="serial_dev"/> /-->
         <!-- <use_device name="rtc_dev"/> /-->
        <use_device name="i2c_dev"/>
        <heap size="0x100000"/>
        <memsection cache_policy="uncached" name="timer_vaddr" phys_addr="0x40A00000" size="0x1000"
attach="r" />
        <environment>
                <entry cap="/utility/hwman" key="HWMAN_CAP"/>
                <entry cap="/utility/serial" key="SERIALSERVER_CAP"/>
        </environment>
        <commandline/>
</okl4>
```

Appendix B.2 – Utility Cell

The Utility Cell is an amalgam of different threads assembled from earlier development. Each of these threads was developed and tested as its own cell. In the transition to thread, they have undergone minimal alteration. As a result, each thread is unaware of its ability to access memory to which it may not have full control. Although this is not a problem with the prototype at the moment, it may become a security issue in the future. The makefile for this cell is different from the makefile for all other cells, in order to maintain an orderly directory tree, placing each thread in its own subdirectory.

The `main()` thread of the Utility cell simply calls `L4_Yield()`, in order to give up its time-slice of CPU immediately.

Appendix B.2.1 – Hardware Manager

The hardware manager enables each device's clock distribution, as well as any other required settings (such as GPIO settings) which may reside outside the memory space of the device. Furthermore, the hardware manager has been drafted as a part of the testing chain, passing messages received between the test cell and the IO cell.

Appendix B.2.1.1 – hwmanager.c

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need mallloc, which is in stdlib
#include <stdlib.h>
// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
```

```
#include <okl4/irqset.h>
#include <okl4/message.h>

#include "../include/hwmanager.h"

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()


okl4_memsec_t *clock_memsec, *gpio_memsec;
okl4_env_segment_t *clock_seg, *gpio_seg;
okl4_static_memsec_t *clock_static_memsec, *gpio_static_memsec;
okl4_static_memsec_attr_t clock_static_memsec_attr, gpio_static_memsec_attr;
okl4_virtmem_item_t clock_virtmem, gpio_virtmem;

static void pxa_init_gpio(void)
{

        gpio_seg=okl4_env_get_segment("MAIN_GPIO_MEM0");
        assert(gpio_seg);

        okl4_static_memsec_attr_init(&gpio_static_memsec_attr);
        okl4_static_memsec_attr_setsegment(&gpio_static_memsec_attr,gpio_seg);

        gpio_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&gpio_static_memsec_attr));
        assert(gpio_static_memsec);

        okl4_static_memsec_init(gpio_static_memsec,&gpio_static_memsec_attr);

        gpio_memsec=okl4_static_memsec_getmemsec(gpio_static_memsec);

        gpio_virtmem=okl4_memsec_getrange(gpio_memsec);

}

static void pxa_init_clockman(void)
{

        clock_seg=okl4_env_get_segment("MAIN_CLOCK_MANAGER_MEM0");
        assert(clock_seg);

        okl4_static_memsec_attr_init(&clock_static_memsec_attr);
        okl4_static_memsec_attr_setsegment(&clock_static_memsec_attr,clock_seg);

        clock_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&clock_static_memsec_attr));
        assert(clock_static_memsec);

        okl4_static_memsec_init(clock_static_memsec,&clock_static_memsec_attr);

        clock_memsec=okl4_static_memsec_getmemsec(clock_static_memsec);

        clock_virtmem=okl4_memsec_getrange(clock_memsec);

}

static void enable_stuart_clock(void)
{

        okl4_word_t CKEN = okl4_range_item_getbase(&clock_virtmem) + (okl4_word_t)0x004;
        *(okl4_word_t*)CKEN |= 1<<5; //Magic.  Fix this.
}

static void enable_stuart_function(void)
{

        okl4_word_t GAFR1_L = okl4_range_item_getbase(&gpio_virtmem) + (okl4_word_t)0x05C;
        inw(GAFR1_L) |= _BV(RX_AF);
```

```
        inw(GAFR1_L) |= _BV(TX_AF);
}

static void enable_i2c_function(void)
{


        okl4_word_t GAFR3_U = okl4_range_item_getbase(&gpio_virtmem) + (okl4_word_t)0x070;
        inw(GAFR3_U) |= _BV(SDA_AF);
        inw(GAFR3_U) |= _BV(SCL_AF);

}

static void enable_i2c_clock(void)
{

        okl4_word_t CKEN = okl4_range_item_getbase(&clock_virtmem) + (okl4_word_t)0x004;
        *(okl4_word_t*)CKEN |= 1<<14; //Magic.  Fix this.

}

/**
* TODO: Error Handling
**/

static int init_sendcaps(void)
{
        i2c = okl4_env_get("I2C_CAP");
        serial = okl4_env_get("SERIAL_CAP");
        timer = okl4_env_get("TIMER_CAP");

        return 0;

}


static int send_confirmation(okl4_kcap_t *destination)
{
        int error;

        okl4_word_t reply;

        okl4_word_t message;
        message = 0xBEEF;//Magic Affirmative Message.
        error = okl4_message_call(*destination, &message, sizeof(okl4_word_t),
                                  &reply, sizeof(reply), NULL);
        assert(!error);

        if(reply == 0x0BAD)
        {
                return 1;
        } else {
                return 0;
        }

}
/*
*  start the main program
*  much of this can be redone in functions
*  and #define and MACROs
*  but for now leave everything in
*  main so we can easily see what is going on
*/
int hwman_main(int argc, char** argv)
{


        /* code added for timing testing */
```

```
        okl4_word_t bytes;
        char buffer[MAX_CHARS];
        okl4_kcap_t client;

        //okl4_u16_t holder;

        //okl4_word_t time1, time2;

        char response[4];

        /* end of that!*/

        okl4_word_t error;

        okl4_init_thread(); // this sets up okl4 lib, call for every thread.
                            //   may not need thsi due to weaver call
        L4_KDB_SetThreadName(L4_Myself(), "HWMan");

        error = init_sendcaps();
        assert(!error);


        pxa_init_gpio();
        pxa_init_clockman();

        //It is important to enable the serial first, as it receives
        //messages from other cells.  Don't want to confuse stuff.
        enable_stuart_function();
        enable_stuart_clock();
        error = send_confirmation(serial);
        assert(!error);

        enable_i2c_function();
        enable_i2c_clock();

        error = send_confirmation(i2c);
        assert(!error);


        /* code added for timing testing */

        while(1)
        {
        error = okl4_message_wait(buffer, MAX_CHARS, &bytes, &client);
        assert(!error);

        error = okl4_message_call(*i2c, buffer, 4*sizeof(char), response, 4*sizeof(char), NULL);
        assert(!error);

        error = okl4_message_reply(client, response, 4*sizeof(char));
        assert(!error);

        }
        /* end of that! */

        //If we let the main() routine exit, we get dropped into the debug
        //menu by the _exit() routine
        while(1);
}
```

Appendix B.2.2 – Serial Output

The serial output cell provides a simple method for displaying and accepting data from a terminal through IPC, allowing cells which do not have direct access to the serial port to implement

these faculties.

Appendix B.2.2.1 – serial.c

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need mallloc, which is in stdlib
#include <stdlib.h>

#include <string.h>

// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>
#include <okl4/message.h>

#include "../include/serial.h"
#include <hwmanclient.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <serial/serial.h>

okl4_word_t RBR,THR,DLL,IER,DLH,IIR,FCR,LCR,LSR;

static int setup_stuart(void)
{
        okl4_word_t DLLH, DLLL;
        okl4_word_t MYDLL = 48;//14745600 / (16*BAUDRATE);
        inw(LCR) |= _BV(LCR_DLAB);
        DLLH = (MYDLL & 0x0000FF00) >> 8;
        DLLL = (MYDLL & 0x000000FF);
        inw(DLH) |= DLLH;
        inw(DLL) |= DLLL;
        inw(LCR) &= ~_BV(LCR_DLAB);
        inw(IER) |= _BV(IER_UUE);
        inw(LCR) |= _BV(LCR_WLS) | _BV(LCR_WLS+1);
        return 0;
}

static int send_byte(okl4_word_t data)
{
        //printf("Input Data: %c Hex: %x\n", (char)data, (unsigned int)data);

        if((vinw(LSR) & _BV(LSR_TEMT)) != 0x40)
        {
                //Can't transmit, data still waiting!
                return 1;
        }
        inw(THR) = data & 0x000000FF;
        while((vinw(LSR) & _BV(LSR_TEMT))!= 0x40)
        {
        }
```

```
	return 0;

}


//Little Endian Unpack
static int unpackWord(okl4_word_t word, char *unpacked)
{

	unpacked[0] = (char)(word >> 24);
	unpacked[1] = (char)(word >> 16);
	unpacked[2] = (char)(word >> 8);
	unpacked[3] = (char)(word >>0);

	return 1;
}

/*
* Message Formatted Like So:
* | 4 bytes | up to 16 bytes | Remainder |
* | Command | Cell name      | Message   |
*/

static int printNameAndMessage(okl4_word_t buffer[], size_t size)
{

	int x,y;
	char cellName[16];
	char cellString[256];
	char unpacked[4];
	int stringholder = 0;

	for(x = 1; x < 5; x++)
	{
		unpackWord(buffer[x], unpacked);
		for(y = 0; y < 4; y++)
		{
			if(unpacked[y] != '\0')
			{
				cellName[stringholder] = unpacked[y];
				stringholder++;
			} else {
				break;
			}
		}
		if(unpacked[y] == '\0')
			break;
	}
	cellName[stringholder] = '\0';


	stringholder=0;
	x++;
	for(; x < size; x++)
	{
		unpackWord(buffer[x], unpacked);
		for(y = 0; y < 4; y++)
		{
			if(unpacked[y] != '\0')
			{
				cellString[stringholder] = unpacked[y];
				stringholder++;
			} else {
				break;
			}
		}
		if(unpacked[y] == '\0')
```

```
                        break;
        }
        cellString[stringholder] = '\0';


        fprintf(ser2out, "Cell %s: %s", cellName, cellString);

        return 0;

}


static size_t char_write(const void *data, long int position, size_t count, void *handle)
{

        //size_t i;
        const char *real_data = data;

        //for(i=0; i < count; i++)
        send_byte((okl4_word_t)*real_data);
        //printf("Real Data: %c\c n", *real_data);

        return 1;
}

static size_t char_get(void *data, long int position, size_t count, void *handle)
{

        char *localdata = data;

        //printf("In char_get!\n");

        *localdata = (char)get_byte();

        //printf("Local Data: %X", (unsigned int)*localdata);

        return 1;

}


static int getPrintCellName(okl4_kcap_t client)
{

        int error;
        okl4_word_t bytes;
        okl4_word_t buffer[MAX_CHARS];
        okl4_kcap_t localClient;
//      okl4_word_t messageType, reply, message;
        okl4_word_t reply;

        reply = 0xBA1; //Magic continue message.
        error = okl4_message_replywait(client, &reply, sizeof(okl4_word_t), buffer, MAX_CHARS, &bytes,
&localClient);
        assert(!error);
        if(bytes > MAX_CHARS)
        {
                //Error HAndle!
        }
        fprintf(ser2out, "Cell %s: ", (char*)buffer);

        //print buffer
        return 0;

}

static int sendLine(okl4_kcap_t client)
{
        int error;
```

```
        okl4_word_t bytes;
        char buffer[81];
        char *sendBuffer;
        okl4_word_t response;
        okl4_kcap_t localClient;
        okl4_word_t reply;

        //int i;

        reply = 0xBB0; //Magic continue, next message will contain your request.
        error = okl4_message_replywait(client, &reply, sizeof(okl4_word_t), &response, sizeof(response),
&bytes, &localClient);
        assert(!error);
        if(bytes > sizeof(response))
        {
                //Trouble!
        }
        if(response == 0xBB1) //Okay to send
        {
                error = getline(buffer, 81);
                sendBuffer = malloc(sizeof(char)*(strlen(buffer)+1));
                strcpy(sendBuffer, buffer);


                if(error > 81)
                {
                        //Trouble!
                }
                //Have to send strlen+1 so we properly send the null terminator.
                error = okl4_message_replywait(localClient, sendBuffer, sizeof(char)*(strlen(sendBuffer)
+1), &response, sizeof(response), &bytes, &localClient);
                assert(!error);
                free(sendBuffer);
                if(response == 0xBB2) //received okay
                {
                        return 0;
                } else {
                        return 1;
                }
        } else {
                return 1;
        }
}


static int waitForMessage(void)
{
        int error;
        okl4_word_t bytes;
        //ar printBuffer[MAX_CHARS+1];
        okl4_word_t buffer[MAX_CHARS];
        okl4_kcap_t client;
        okl4_word_t reply;


        //okl4_word_t messageType, reply, message;
        okl4_word_t messageType;

        error = okl4_message_wait(buffer, MAX_CHARS, &bytes, &client);
        assert(!error);
        if(bytes > MAX_CHARS)
        {
                //handle this error
        }

        else
        {
                messageType = buffer[0];
                if(messageType ==  0x00) //This is a request to send characters.
```

```
		{
			/*error = getPrintCellName(client);
			assert(!error);
			//Print Cell Name First.
			error = printMessage(client);
			assert(!error);*/
			error = printNameAndMessage(buffer, bytes/4); //Send the buffer along.
			assert(!error);
			reply = 0xBA0;
			error = okl4_message_reply(client, &reply, sizeof(reply));

		}
		if(messageType ==0x01) //This is a request to receive a line
		{
			fprintf(ser2out, "Request for Input from ");
			error = getPrintCellName(client);
			assert(!error);
			error = sendLine(client);
			assert(!error);
		}
	}

	return 0;
}

static int getline(char *s, int lim)
{

	int i;
	char c = fgetc(ser2in);

	for(i=0; i<(lim-1) && c!='\n'; ++i)
	{
		s[i]=c;
		c = fgetc(ser2in);
	}
	if(c == '\n') {
		s[i]=c;
		++i;
	}
	s[i]='\0';

	return i;
}

static okl4_word_t get_byte(void)
{

	okl4_word_t retVal;
	while((vinw(LSR) & _BV(LSR_DR)) != 0x00000001 )
	{

		//wait for a character

	//return inw(RBR);
	} /*else {
		return -1;
	}*/

	retVal = vinw(RBR);

	return retVal;
}

/*
*  start the main program
*  much of this can be redone in functions
*  and #define and MACROs
*  but for now leave everything in
```

```
*  main so we can easily see what is going on
*/
int serial_main(int argc, char** argv)
{


        int error;

        okl4_memsec_t         * stu_memsec;
        okl4_env_segment_t    * stu_seg;

        okl4_static_memsec_t  * stu_static_memsec;
        okl4_static_memsec_attr_t stu_static_memsec_attr;

        okl4_virtmem_item_t   stu_virtmem;

        okl4_init_thread(); // this sets up okl4 lib, call for every thread.
                            //   may not need thsi due to weaver call

        L4_KDB_SetThreadName(L4_Myself(), "Serial");


        stu_seg=okl4_env_get_segment("MAIN_STUART_MEM0");
        assert(stu_seg);


        okl4_static_memsec_attr_init(&stu_static_memsec_attr);
        okl4_static_memsec_attr_setsegment(&stu_static_memsec_attr,stu_seg);

        stu_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&stu_static_memsec_attr));
        assert(stu_static_memsec);

        okl4_static_memsec_init(stu_static_memsec,&stu_static_memsec_attr);

        stu_memsec=okl4_static_memsec_getmemsec(stu_static_memsec);

        stu_virtmem=okl4_memsec_getrange(stu_memsec);

        error = hwman_resource_block();
        assert(!error);


        RBR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x00;
        THR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x00;
        DLL = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x00;
        IER = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x04;
        DLH = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x04;
        IIR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x08;
        FCR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x08;
        LCR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x0C;
        LSR = okl4_range_item_getbase(&stu_virtmem) + (okl4_word_t)0x14;

        error = setup_stuart();
        assert(!error);

        fprintf(ser2out, "Serial is Alive:\r\n");

        while(1)
        {
                //char test[80];
                //int i;
                //int max=80;

                        //fprintf(ser2out, "Please Enter a string: \r\n");
                //i = getline(test, max);
                //fprintf(ser2out, "Got S: %s\r\n", test);

                waitForMessage();
        }
```

```
        L4_KDB_Enter("I am done!");
}
```

Appendix B.2.3 – Test Thread

The test thread makes IPC calls to the Hardware manager thread, which then passes these messages on to the IO cell. When performance testing was completed, each of these threads lay within its own cell. At the moment, the test and the hardware manager thread are contained by the same cell. No testing has been performed yet in order to determine the performance impact of this change.

In order to begin testing, uncomment the code located within the for loop, and alter the **`message[]`** array appropriate to the faculty which is being tested. This is described in better detail in the IO code. Uncommenting the code in the for loop of this thread enables testing of the complete communications path, including I2c.

Appendix B.2.3.1 – test.c

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need mallloc, which is in stdlib
#include <stdlib.h>

#include <string.h>

// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>
#include <okl4/message.h>


#include <serial/serial.h>
#include <usleep.h>

#include "../include/test.h"
// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <ser2client.h>

//int (*pt_test_main)(int, char**) = &test_main;

/*
*  start the main program
*  much of this can be redone in functions
*  and #define and MACROs
*  but for now leave everything in
```

```
*  main so we can easily see what is going on
*/
int test_main(int argc, char** argv)
{
        /* i2c test */

        char message[4];
        char response[4];
        //int error;
        okl4_kcap_t *i2c_server;

        //okl4_word_t time1, time2;

        int i;

        //okl4_word_t samples[1000];

        /* i2c test end */

        okl4_init_thread(); // this sets up okl4 lib, call for every thread.
                            //   may not need thsi due to weaver call

        L4_KDB_SetThreadName(L4_Myself(), "Test");

        sleep_init();
        usleep(150000);


        setCellName("test");
        ser2client_init();

        /* i2c test */

        lprintf("Sending Message.\r\n");


        i2c_server = okl4_env_get("HWMAN_CAP");

        message[0] = 0x06;
        message[1] = 0x00;
        message[2] = 0xBB;
        message[3] = 0xBB;

        for(i=0; i<1000; i++)
        {
        //      time1 = makeNote();

        okl4_message_call(*i2c_server, message, 4*sizeof(char), response, 4*sizeof(char), NULL);
        //      assert(!error);

        //      time2 = makeNote();

        //      lprintf("%d ", (int)(time2-time1));

        //      lprintf("Got Response: %x\r\n", response[0]);
        }


        /* i2c test end */

        L4_KDB_Enter("I am done!");

        return 0;
}
```

Appendix B.2.4 – utility.c

```
//  POSIX libray's we need for now printf
```

```
#include <stdio.h>
// we will need mallloc, which is in stdlib
#include <stdlib.h>
// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>
#include <okl4/message.h>

//#include "test/include/test.h"
//#include "hwmanager/include/hwmanager.h"
//#include "serial/include/serial.h"


#include <utility.h>
#include <l4/schedule.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

int main(int argc, char**  argv)
{

        int error;
        //okl4_word_t i;
        void *stacks;
        okl4_kthread_t threads;
        okl4_kcap_t caps;
        okl4_kthread_t *root_thread;

        okl4_kclist_t *root_kclist;
        okl4_kspace_t *root_kspace;
        struct okl4_utcb_item *utcb_item;
        struct okl4_kcap_item *kcap_item;*/


        okl4_init_thread();

        L4_KDB_SetThreadName(L4_Myself(), "Utility");

        while(1)
        {
                L4_Yield();
        }
}
```

Appendix B.2.5 – utility.xml.in

```
<okl4 priority="255" clists="256" file="utility" kernel_heap="0x400000" mutexes="256" name="utility"
spaces="32">
        <use_device name="clock_manager_dev"/>
        <use_device name="gpio_dev"/>
        <use_device name="stuart_dev"/>
        <heap size="0x100000"/>

        <thread name="serial" start="serial_main" priority="255"/>
```

```
        <thread name="test" start="test_main" priority="255"/>
        <thread name="hwman" start="hwman_main" priority="255"/>
        <memsection cache_policy="uncached" name="timer_vaddr" phys_addr="0x40A00000" size="0x1000"
attach="r" />

        <environment>
                <!-- Need both these caps because I am an idiot /-->
                <entry cap="/utility/serial" key="SERIAL_CAP"/>
                <entry cap="/i2c/main" key="I2C_CAP"/>
                <entry cap="/utility/serial" key="SERIALSERVER_CAP"/>
        </environment>
        <commandline/>
</okl4>
```

Appendix B.3 – Network Access

Only the **main.c** file is contained in this document. The remainder of the net code has been copied from the uBoot driver for the SMC9118 with only minor modification. This code has been generated primarily by Dr. Jeff Hieb.

Appendix B.3.1 – main.c

```
#include "config.h"
#include "common.h"
#include "asm/u-boot.h"
#include "net.h"


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


// okl4 stuff
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/irqset.h>
// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/thread.h>
#include <l4/misc.h>  // for L4_KDB_Enter()
#include <l4/kdebug.h>
#include <okl4/message.h>


// some base elements
okl4_virtmem_pool_t     *root_virtmem_pool;
okl4_physmem_segpool_t  *root_physseg_pool;
okl4_kspaceid_pool_t    *root_kspace_pool;
okl4_kclistid_pool_t    *root_kclist_pool;
okl4_kspace_t           *root_kspace;
okl4_kclist_t           *root_kclist;
okl4_kthread_t          *root_kthread;
okl4_pd_t               *root_pd;
//  timer memsection element
okl4_static_memsec_t * timer_static_memsec;
//  smc memsec base
```

```
okl4_static_memsec_t * smc_static_memsec;
unsigned int smc91111_okl4_base;

// Packet Buffer global
volatile uchar * NetRxPackets[PKTBUFSRX];
volatile uchar * NetRxPkt;
int NetRxPktLen;


// DEBUG GLOBALS
int use_smsc9118 = 0;

unsigned int timer_memsec_init(void);

unsigned int  timer_memsec_init(void)
{
okl4_env_segment_t * timer_seg;
okl4_static_memsec_attr_t timer_static_memsec_attr;
okl4_memsec_t * timer_memsec;
okl4_virtmem_item_t timer_virtmem;

timer_seg = okl4_env_get_segment("MAIN_TIMER_MEM0");
okl4_static_memsec_attr_init(&timer_static_memsec_attr);
okl4_static_memsec_attr_setsegment(&timer_static_memsec_attr,timer_seg);
timer_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&timer_static_memsec_attr));
okl4_static_memsec_init(timer_static_memsec,&timer_static_memsec_attr);
timer_memsec=okl4_static_memsec_getmemsec(timer_static_memsec);
timer_virtmem=okl4_memsec_getrange(timer_memsec);
return okl4_range_item_getbase(&timer_virtmem);
}

static int
i2c_blink(void)
{
        okl4_kcap_t *i2c;
        int error;
        char message[4];
        char reply[4];

        i2c = okl4_env_get("I2C_CAP");
        printf("Sending Message!\n");
        message[0] = 0x08;

        error = okl4_message_call(*i2c, message, 4, reply, 4, NULL);
        assert(!error);

        printf("Returned! 0x%x0x%x!\n", (unsigned int)reply[0], (unsigned int)reply[1]);

        if(reply[1] == 0xff)
                return 0;
        else
                return 1;

}

static void
get_root_weaved_objects(void)
{
    root_virtmem_pool = okl4_env_get("MAIN_VIRTMEM_POOL");
    assert(root_virtmem_pool);

    root_physseg_pool = okl4_env_get("MAIN_PHYSMEM_SEGPOOL");
    assert(root_physseg_pool);

    root_kspace_pool = okl4_env_get("MAIN_SPACE_ID_POOL");
    assert(root_kspace_pool);

    root_kclist_pool = okl4_env_get("MAIN_CLIST_ID_POOL");
    assert(root_kclist_pool);
```

```
    root_kspace = okl4_env_get("MAIN_KSPACE");
    assert(root_kspace);

    root_kclist = okl4_env_get("MAIN_KCLIST");
    assert(root_kclist);

    root_kthread = okl4_env_get("MAIN_KSPACE_THREAD_0");
    assert(root_kthread);

    root_pd = okl4_env_get("MAIN_PS");
//    assert(root_pd);   MAIN_PS seems to return null

}



// THIS WILL BE THE MAIN SERVER LOOP FOR THE RTU

int main(int argc, char **argv)
{

bd_t my_bd;
unsigned int t_base;
okl4_memsec_t * dev_memsec;
okl4_env_segment_t * cs_mem0_seg;
okl4_static_memsec_attr_t cs_mem0_static_memsec_attr;
okl4_virtmem_item_t dev_virtmem;
int packet_len;
int resp;
// 255 MAC: 00:0A:95:A5:47:3A
// Office Linux: 00:01:03:1C:76:C7

uchar send_packet[60];
//uchar recv_packet[60];

//Set all zeors in packet
memset(send_packet,0x00,60);
send_packet[0]=0x00;//DST MAC (6 bytes)
send_packet[1]=0x01;//Replace with Mac of Client
send_packet[2]=0x03;
send_packet[3]=0x1C;
send_packet[4]=0x76;
send_packet[5]=0xC7;
send_packet[6]=0x00;//SRC MAC (6 Bytes)
send_packet[7]=0x15;// Replace with gumstix mac
send_packet[8]=0xc9;
send_packet[9]=0x13;
send_packet[10]=0x80;
send_packet[11]=0x50;
send_packet[12]=0x08;// Type (2 Bytes)
send_packet[13]=0x00;
send_packet[14]=0xBE; // PROTOCOL ID
send_packet[15]=0xEF;
// 16 is command
// 17 is address
// 18 is data

get_root_weaved_objects();
// recieve packet buffer
NetRxPkt = malloc(1024);
// Could create many buffers, for now we will use one
// and over write
NetRxPackets[0]=NetRxPkt;
NetRxPackets[1]=NetRxPkt;
NetRxPackets[2]=NetRxPkt;
NetRxPackets[3]=NetRxPkt;

// set up memory mapping of SMC911x register access
```

```
cs_mem0_seg=okl4_env_get_segment("MAIN_CS_MEM1");
okl4_static_memsec_attr_init(&cs_mem0_static_memsec_attr);
okl4_static_memsec_attr_setsegment(&cs_mem0_static_memsec_attr,cs_mem0_seg);
smc_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&cs_mem0_static_memsec_attr));
okl4_static_memsec_init(smc_static_memsec,&cs_mem0_static_memsec_attr);
dev_memsec=okl4_static_memsec_getmemsec(smc_static_memsec);
dev_virtmem=okl4_memsec_getrange(dev_memsec);
smc91111_okl4_base = okl4_range_item_getbase(&dev_virtmem);


t_base= timer_memsec_init();
timer_base_init(t_base);
//Needed for SMC911x for now, I think we can just patch the code eventually
my_bd.bi_enetaddr[0] = 0x00;
my_bd.bi_enetaddr[1] = 0x15;
my_bd.bi_enetaddr[2] = 0xc9;
my_bd.bi_enetaddr[3] = 0x13;
my_bd.bi_enetaddr[4] = 0x80;
my_bd.bi_enetaddr[5] = 0x50;
eth_init(&my_bd);
packet_len=0;

// Start main server loop
// check for received packet, parse and respone
while(1)
{
        packet_len=eth_rx();
        if(packet_len != -1)
                printf("\n\nGot that packet! %d\n\n", packet_len);
        if (packet_len>0)
        {
                #ifdef DEBUG
                //printf("return from eth_rx, with packet lenght %i\n",packet_len);
                #endif
                if(NetRxPkt[14]==0xBE && NetRxPkt[15]==0xEF)
                {
                        #ifdef DEBUG
                        printf("received a protocol packet\n");
                        #endif
                        send_packet[16]=0x03; // prepare response package
                        switch (NetRxPkt[16])
                        {
                        case    0x01:   printf("Read ");
                                        if (NetRxPkt[17] == 0x01)
                                        {
                                        printf("analog input 1\n");
                                        send_packet[17]=0x01;
                                        send_packet[18]=0xFF; // need to get val from i2c
                                        }
                                        else if (NetRxPkt[17]==0x02)
                                        {
                                        printf("digital input 1\n");
                                        send_packet[17]=0x02;
                                        send_packet[18]=0x00;
                                        }
                                        else
                                        {
                                        printf("unknown code\n");
                                        send_packet[17]=0xFF;
                                        }
                                        break;
                        case    0x02:   printf("Write ");
                                        if (NetRxPkt[17]==0x03)
                                        {
                                        printf("analog output 1 %02x\n",NetRxPkt[18]);
                                        send_packet[17]=0x03;
                                        send_packet[18]=NetRxPkt[18];
                                        }
                                        else if (NetRxPkt[17]==0x04)
```

```
                                {
                                printf("digital output 1 %02x\n",NetRxPkt[18]);
                                send_packet[17]=0x04;
                                send_packet[18]=0x00;
                                }
                                break;
                        case    0x03:   printf("\nBlink!\n");
                                i2c_blink();

                                break;
                        default: break;
                        }
                        // reponse packet should be set up now
                        resp = eth_send(&send_packet,18);
                        #ifdef DEBUG
                        printf("response sent\n");
                        #endif
                        udelay(10000);
                }
        }

}
L4_KDB_Enter("JLH: end Reached");
}
```

Appendix B.3.2 – access.xml.in

```
<okl4 clists="256" file="access" kernel_heap="0x400000" mutexes="256" name="netDrv2" spaces="64">
        <use_device name="serial_dev"/>
        <use_device name="timer_dev"/>
        <!-- <use_device name="gpio_dev"/> /-->
         <use_device name="rtc_dev"/>
        <use_device name="cs_dev"/>
        <environment>
                <entry cap="/i2c/main" key="I2C_CAP"/>
        </environment>
        <heap size="0x100000"/>
        <commandline/>
</okl4>
```

Appendix B.4 – Libraries

These libraries do not exist as libraries in the traditional sense. They are not pre-compiled, and are not linked into cells on an as-needed basis. Rather, by including the header file for the appropriate library, the library is compiled into each cell or thread on an as-needed basis. Although this results in some lost CPU time in recompilations, the ease of this implementation in the makefile structure of this project compensates for this performance loss.

Appendix B.4.1 – hwmanclient.c

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

#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/kernel.h>
#include <okl4/message.h>

#include <hwmanclient.h>
```

```
int error;
okl4_word_t i, bytes, msglen;
okl4_kcap_t *hwman_server;

int hwman_resource_block(void)
{

        int error;
        okl4_word_t bytes;
        okl4_word_t buffer;
        okl4_kcap_t hwman;

        okl4_word_t reply;

        error = okl4_message_wait(&buffer, sizeof(okl4_word_t), &bytes, &hwman);
        assert(!error);

        if(buffer == 0xBEEF)
        {
                reply = 0x0000;
                error = okl4_message_reply(hwman, &reply, sizeof(okl4_word_t));
                assert(!error);
                return 0;
        } else {
                reply = 0x0BAD;
                error = okl4_message_reply(hwman, &reply, sizeof(okl4_word_t));
                assert(!error);
                return 1;
        }

}
```

Appendix B.4.2 – ser2client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/kernel.h>
#include <okl4/message.h>

#include <ser2client.h>

char *CELLNAME = NULL;

void ser2client_init(void)
{
        char defult[] = "DEFAULT_CELL_NAME\0";
        if(CELLNAME == NULL)
        {
                CELLNAME = malloc(strlen(defult)*sizeof(*CELLNAME));
                strcpy(CELLNAME, defult);
        }
        serial_server = okl4_env_get("SERIALSERVER_CAP");
}

void setCellName(char *cell_name)
{

        CELLNAME = malloc(strlen(cell_name)*sizeof(*CELLNAME));
        strcpy(CELLNAME, cell_name);
        //Error, need to ensure null termination.

}

int lscanf(char *str, ...)
```

```
{
	int retval;
	va_list args;
	char lstr[81];

	error = get_data(lstr);
	assert(!error);

	va_start(args, str);
	retval = vsscanf(lstr, str, args);
	va_end(args);

	return retval;
}

//Local printff
//Can print up to MAX_CHARS chars.
int lprintf(char *str, ...)
{

	char lstr[MAX_CHARS];
	int error;
	va_list args;

	va_start(args, str);

	error = vsnprintf(lstr, MAX_CHARS, str, args);
	if(error > MAX_CHARS)
	{
		//TODO: Handle Buffer Overflow


		va_end(args);
		return 1;
	} else {
		print_data(lstr);
		va_end(args);
		return 0;
	}

}



int packData(okl4_word_t *output, char *input, size_t inSize)
{

	int x,y;

	for(x = 0; x < inSize/4; x++)
	{
		for(y = 0; y < 4; y++)
		{
			output[x] |= (okl4_word_t)(input[y + x*4] << ((3-y)*8));
		}
	}
	if(inSize%4 == 0)
	{
		output[(inSize/4)+1] = 0x00000000;
	} else {
		for(y = 0; y < inSize%4; y++)
		{
			output[(inSize/4)] |= (okl4_word_t)(input[y+(inSize-inSize%4)] << ((3-y)*8));
		}
	}


	return 0;
}
```

```
int print_data(char *str)
{


        int error;


        int wl_name,wl_message;

        okl4_word_t reply;

        okl4_word_t *message;

        okl4_word_t *packedName;
        okl4_word_t *packedMessage;

        wl_name = sizeof(okl4_word_t)*((strlen(CELLNAME)/4)+1);
        wl_message = sizeof(okl4_word_t)*((strlen(str)/4)+1);

        message = malloc(wl_name+wl_message+4);

        packedName = malloc(wl_name);
        packedMessage = malloc(wl_message);

        memset(packedName, 0, wl_name);
        memset(packedMessage, 0, wl_message);

        packData(packedName, CELLNAME, strlen(CELLNAME));
        packData(packedMessage, str, strlen(str));

        *message = (okl4_word_t)0x00;
        memcpy((message+1),packedName,wl_name);
        memcpy((message+1+wl_name/4),packedMessage,wl_message);

        error  = okl4_message_call(*serial_server, message, wl_name+wl_message+4, &reply, sizeof(reply),
NULL);
        assert(!error);


        if(reply == 0xBA0) // Magic
        {
                return 0;
        }
        else
        {
                return 1;
        }

}


int get_data(char *data)
{
        int error;

        okl4_word_t reply;
        okl4_word_t bytes;

        okl4_word_t mode;

        char myData[81];

        mode = 0x01;
        error  = okl4_message_call(*serial_server, &mode, sizeof(mode), &reply, sizeof(reply), NULL);
        assert(!error);

        if(reply == 0xBA1) // Magic
        {
```

```
		//send name,
		error = okl4_message_call(*serial_server, CELLNAME, strlen(CELLNAME), &reply,
sizeof(reply), NULL);
		assert(!error);


		if(reply == 0xBB0)
		{
			//rx okay.  Wait for Message
			mode = 0xBB1;
			error = okl4_message_call(*serial_server, &mode, sizeof(mode), myData,
sizeof(myData), &bytes);
			assert(!error);
			if(bytes == 0)
			{
				//failure
				return 1;
			} else {
				strcpy(data,myData);
				mode = 0xBB2;
				error = okl4_message_send(*serial_server, &mode, sizeof(mode));
				assert(!error);
				return 0;
			}

		} else {
			return 1;
			//fail somehow
		}

	} else {
		return 1;
		//fail somehow
	}


}
```

Appendix B.4.3 – usleep.c

```
#include <stdlib.h>

#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <usleep.h>

okl4_word_t OSCR0;

int sleep_init(void)
{
```

```
        okl4_memsec_t         * timer_memsec;
        okl4_env_segment_t    * timer_seg;

        okl4_static_memsec_t  * timer_static_memsec;
        okl4_static_memsec_attr_t timer_static_memsec_attr;

        okl4_virtmem_item_t   timer_virtmem;

        timer_seg=okl4_env_get_segment("MAIN_TIMER_VADDR");
        assert(timer_seg);

        okl4_static_memsec_attr_init(&timer_static_memsec_attr);
        okl4_static_memsec_attr_setsegment(&timer_static_memsec_attr,timer_seg);

        timer_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&timer_static_memsec_attr));
        assert(timer_static_memsec);

        okl4_static_memsec_init(timer_static_memsec,&timer_static_memsec_attr);

        timer_memsec=okl4_static_memsec_getmemsec(timer_static_memsec);

        timer_virtmem=okl4_memsec_getrange(timer_memsec);

        OSCR0 = okl4_range_item_getbase(&timer_virtmem) +(okl4_word_t)0x10;

        return 0;
}

int usleep(long int useconds)
{
        okl4_word_t start_time;

        start_time = vinw(OSCR0);
        //Need Division since free-running counter runs at 3.25 MHz.
        while((vinw(OSCR0) - start_time)/3.25 < useconds)
        {
        }

        return 0;
}


okl4_word_t makeNote(void)
{

        return vinw(OSCR0);

}
```