A uClinux driver system for the NIOS2 processor

Prepared for Samuel Ginsberg

by Nicholas Thorne

22 October 2007

Plagiarism

I know that plagiarism is wrong. Plagiarism is using another's work and to pretend that it is ones own.

This thesis project is entirely my own work. Where necessary quotes and references have been provided and their sources acknowledged. The author-date system convention for referencing will be used for references where appropriate.

I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Name: Nicholas James Thorne

Signature: _____

Date: 22 October 2007

Ethics

Name of Principal Researcher/Student:	Nicholas James Thorne
Department:	Electrical Engineering
Degree:	B.sc (electrical and computer engineering)
Supervisor:	Samuel Ginsberg
Research Project Title:	SIG01 - Hand-held Embedded Linux Device

Overview of ethics issues:

Question 1: Is there a possibility that your research could cause harm to a third party (i.e. a person not involved in your project)?	**NO**
Question 2: Is your research making use of human subjects as sources of data? If your answer is YES, please complete Addendum 2.	**NO**
Question 3: Does your research involve the participation of or provision of services to communities? If your answer is YES, please complete Addendum 3.	**NO**
Question 4: If your research is sponsored, is there any potential for conflicts of interest? If your answer is YES, please complete Addendum 4.	**NO**

If you have answered YES to any of the above questions, please append a copy of your research proposal, as well as any interview schedules or questionnaires (Addendum 1) and please complete further addenda as appropriate.

I hereby undertake to carry out my research in such a way that:

- There is no apparent legal objection to the nature or the method of research
- The research will not compromise staff or students or the other responsibilities of the University
- The stated objective will be achieved, and the findings will have a high degree of validity
- Limitations and alternative interpretations will be considered
- The findings could be subject to peer review and publicly available
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

Signed by:

	Full name and signature	Date
Principal Researcher/Student:	Nicholas James Thorne _____	**16 October 2007**

This application is approved by:

Supervisor (if applicable):	Samuel Ginsberg _____	**16 October 2007**

HOD (or delegated nominee): Final authority for all assessments with NO to all questions and for all undergraduate research.	_____	**16 October 2007**

Abstract

Embedded systems are taking on more complicated tasks as the processors involved become more powerful. As the number of transistors per area of silicon increases in a fairly predictable fashion as forecast by Gordon E Moore in 1965 we have more resources available for features that speed up development and time to market. The extra computer resources being fitted into the same space allows some embedded projects to consider an operating system while still maintaining the required speed of response. Operating systems can greatly reduce the time to market of an embedded project and save the developers handling chores like race conditions, multitasking and hardware access.

The partitioning decision in any project as to which portion of the project to implement in hardware and which portion to implement in software has always been fundamental to embedded projects. The decision is usually based on the project's performance requirements and hardware used when software solutions are not fast enough. Software provides better versatility and flexibility and with the increase in performance in the computing field software can take on more and still have resources left over for improving code structure and development features.

Altera provide soft-core processors implemented on a Field Programmable Gate Array and the performance of these processors have become competitive with conventional processors. Using the Altera Cyclone 1 evaluation board and Cyclone 2 development board both running the NIOS2 soft-core processor; this thesis investigates the versatility and performance as well as the time-to-market which this soft-core processor technology brings to the embedded systems field.

Acknowledgements

Many thanks to:

Samuel Ginsberg for supervising this project and always being around when a little bit of help is needed.

The NIOS2 community (located at www.niosforum.com) for providing some great resources and support when needed.

Acronyms and conventions

Code:

As little code as possible has been placed in the report. All of the code written for this project can be found in appendix E. Where code has been used in the report it is written as in the example below:

```
1:    int main ()
2:    {
3:      return 0;
4:    }
```

Paths:

Paths within a directory structure will be displayed surrounded by <> when used. Usually these are generic paths based on the location in which particular pieces of software are installed.

Different operating environments use different conventions: Linux paths use a forward slash '/' when specifying paths while Windows uses a back slash '\'. In addition Windows paths are not case sensitive whereas Linux paths are. Where necessary it will be made clear which environment is being specified.

Linux commands:

Linux commands such as **/>ls** for listing directory contents and **/>chmod** for changing file permissions will be written with a preceding /> as is displayed as the prompt for the sash shell and in bold typeface. Note that Linux is case sensitive and so all commands are written exactly as they are used on the command line.

Acronyms:

API – Application Programming Interface
CMOS – Complementary Metal-Oxide Semiconductor
CPU – Central Processing Unit
DDR – Double Data Rate
DMIPS – Dhrystone Millions of Instructions Per Second
FPGA – Field Programmable Gate Array
HAL – Hardware Abstraction Layer
HDL – Hardware Definition Language
IDE – Integrated Development Environment
JTAG – Joint Test Action Group
LCD – Liquid Crystal Display
LVCMOS – Low Voltage CMOS
LVTTL – Low Voltage Transistor-Transistor Logic
MMU – Memory Management Unit
PDA – Personal Digital Assistant
PIO – Parallel Input Output
PLL – Phase Locked Loop
POSIX – Portable Operating System Interface
RAM – Random Access Memory
RISC – Reduced Instruction Set Computer
ROMFS – Read Only Memory File System
SOF – SRAM Object File
SOPC – System On Programmable Chip
SRAM – Static Random Access Memory
SSTL-2 – Stub Series Terminated Logic for 2.5V
SSTL-3 - Stub Series Terminated Logic for 3.3V

Table of Contents

List of figures

List of tables

1 Introduction

1.1 Terms of Reference

1.1.1 Original Brief

"The ARM9 microprocessor is a high performance embedded processor that finds wide application in devices such as PDAs and cellular telephones. The ARM9 is capable of running Linux and various Linux ports exist. In this project you will look at writing and integrating device drivers for a small ARM9 board.

Our board will have a miniature (low resolution) graphics LCD and a simple keypad.

If you're very daring we could also try modifying the board's hardware (e.g the memory system) and porting and running small open-source applications on the system."

As proposed by thesis supervisor Samuel Ginsberg.

1.1.2 Final Brief and deliverables

The processor used in the project is the NIOS2 soft-core processor instead of the ARM9. Both processors are 32bit RISC pipelined processors and the only significant difference is that the NIOS2 is soft-core, making it more customisable and suited to development.

The driver requirements remain the same; that is to write drivers for a small 16 button keypad and the PG12864F graphical LCD for a driver control system such as an operating system or in the case of NIOS2 the HAL provided by Altera.

1.1.3 Objectives

- To implement a driver abstraction layer. In most embedded projects this is in the form of a highly customisable operating system. Examples of operating systems are Linux (and more specifically uClinux), Microsoft Windows CE and MicroC/OS 2 to name just a few. Another option for the driver abstraction layer is the HAL provided by Altera in the NIOS2 IDE.
- To develop an efficient device driver for the PG12864F graphical LCD that runs on the chosen driver abstraction layer
- To develop an efficient device driver for a 16 key keypad that runs on the chosen driver abstraction layer
- To show that the structure is expandable to more complicated systems
- Determine whether the driver abstraction layer will provide a faster time to market and improved code re-use.
- To provide a report of the design and research process

1.2 Scope of the project

The design of the drivers is limited to character devices. Both the keypad and LCD fall into this category, along with many other simple hardware components such as serial devices.

The implementation of the drivers will be done for the uClinux kernel only, even though Altera's HAL is discussed in some detail. There are some useful comparisons to be made between these two systems.

1.3 Hardware introduction

The hardware used for this project was partially changed during the project. Initially the Cyclone 1 evaluation board, referred to as the 1c12 board was used and the drivers implemented in uClinux, due to the HAL being non-configurable on the evaluation board (more details on the 1c12 board in the hardware chapter).

The Cyclone 2 development board, referred to as the 2c35 board, was then tested and proved much more successful for developing HAL drivers. Unfortunately it arrived very late in the design process and not much time could be spared for testing and developing HAL drivers or further systems such as the real time kernel provided.

Additional hardware consists of a PG12864F LCD graphical display and a standard 16 key keypad shown below. The PG12864F LCD is a 128 x 64 pixel display that supports text and graphics modes.

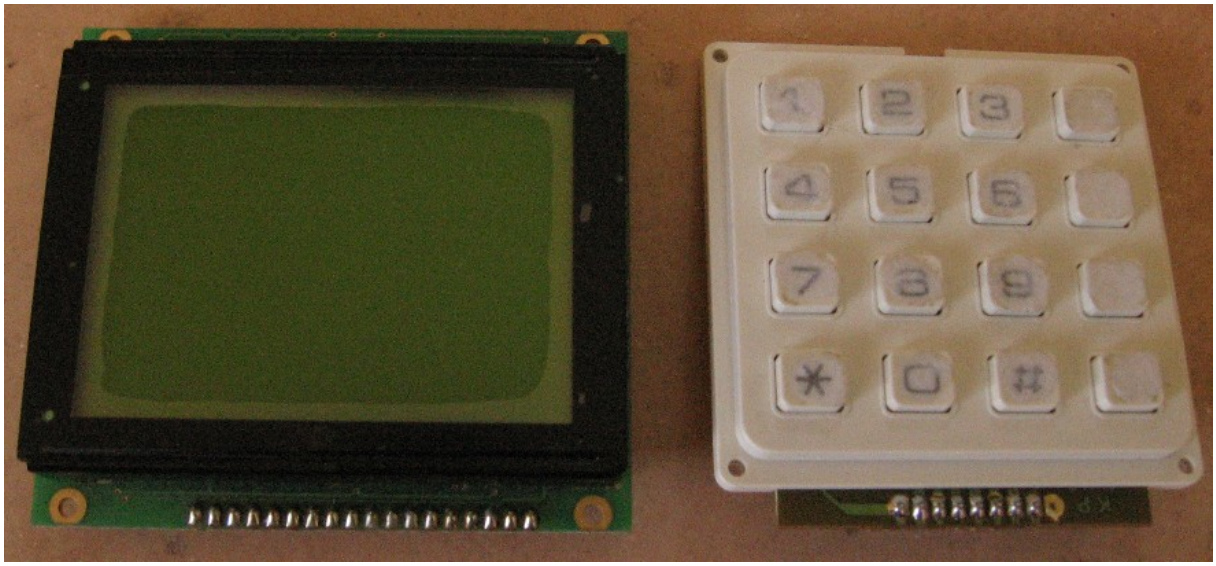

Figure 1.1 The PG12864F graphical LCD and the 16 key keypad used in the project

2 Literature review

As stated in the terms of reference section, drivers are required to be written as part of a coding structure such as an operating system. A number of options exist and this section contains a review of some of the literature associated with the NIOS2 processor and three of the options for implementing the driver support layer as outlined in the introduction section.

2.1 NIOS2 background

The processor used on both boards is the 32bit NIOS2 RISC processor which is a fully customisable, pipelined, soft-core microprocessor. It exists only as HDL code until it is fitted onto an FPGA, at which point it is implemented on fully general hardware. The NIOS2 core itself uses only a small portion of the total FPGA space which allows other standard components to be added alongside, as well as any custom hardware that is desired. This allows for the processor, components and even instructions to be customised. The NIOS2 currently has three processor cores to choose from: Nios2/f: fast, Nios2/e: economy and Nios2/s: standard.

Altera lists the NIOS2 features as follows [Altera, 2007]:

- Separate instruction and data caches (512 bytes to 64 Kbytes)
- Access to up to 2 Gbytes of external address space
- Optional tightly coupled memory for instructions and data
- Six-stage pipeline to achieve maximum DMIPS/MHz
- Optional Single-cycle hardware multiply and barrel shifter (depending on NIOS2 model)
- Optional hardware divide option (depending on NIOS2 model)

- Optional Dynamic branch prediction (depending on NIOS2 model)
- Up to 256 custom instructions and unlimited hardware accelerators
- JTAG debug module
- Optional JTAG debug module enhancements, including hardware breakpoints, data triggers, and real-time trace

The idea behind NIOS2 is that the soft-core feature of the processor should be completely transparent, therefore developing software for this processor should be no different from developing for a silicon wafer based processor.

2.2 Drivers as part of Altera's HAL

The discussion of implementing drivers into the HAL is discussed in the NIOS 2 software developer's handbook [Altera 2007]. The HAL provides a simple interface for device drivers to allow programs to communicate with the underlying hardware. The layered structure of the HAL as described in the NIOS2 software developer's handbook is shown below in Figure 2.1:

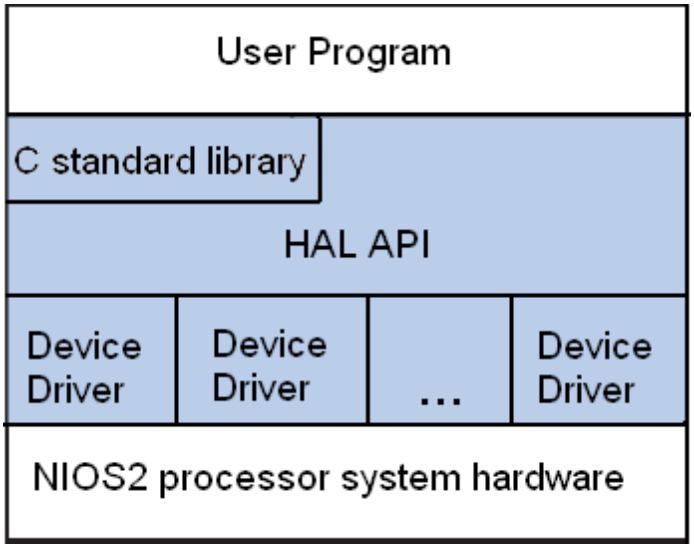

Figure 2.1 The layers for Altera's HAL

The process for creating a device driver is outlined in the “developing device drivers for the HAL” section of the software developer's handbook. Although

not the main implementation, the driver structure of the HAL will be looked at in some detail as an alternative system to a full operating system.

2.3 uClinux

uClinux started out as a port of Linux kernel 2 to micro-controllers, particularly those with no memory management units (MMUs). It was created by D Jeff Dionne and Kenneth Albanowski in 1998 using the Linux 2.0.33 kernel and since then it has kept up to date with the Linux kernel releases and is now considered a full operating system supporting kernel version 2.6 and a selection of user applications and tool chains [Wikipedia 2007][uClinux.org 2007].

Some of the desirable uClinux features are as follows [Michael Opdenacker, 2007]:

- Linux: Build-in IP connectivity, reliability, portability, file systems and free
- Lightweight: Full Linux 2.6 kernel occupies less than 300K and binaries are smaller when built with uClibc
- Execute In Place: Executables don't have to be loaded into RAM to run. This will cause the executables performance to decrease.
- Cheaper
- Faster: No cache flushes allow for faster context switches
- User access to hardware
- Full Linux API: uses familiar Linux system calls (with some minor exceptions) making it easier for developer's to learn
- Kernel Pre-emption
- Full multitasking
- Supported by many processors (including NIOS2)

uClinux is based on the Linux-2.6.x kernel which handles critical sections, scheduling, interrupt service routines, multitasking and disk access to mention a few of it's tasks. It starts out as a stripped down kernel as is expected for an embedded kernel, containing only a shell, http and ftp servers and a basic file system. It is customisable to allow you to build in many other features. The feature of interest to this project is the kernel support for loadable modules [uClinux.org, 2007].

2.4 MicroC/OS 2 Real Time Kernel

µC/OS2(as it is called by the author) is the real time kernel that comes with the Altera 2c35 board and looks on the surface to be very similar to uClinux. It is not based on a Linux kernel; however it's kernel implements similar functionality. If the 2c35 board had arrived at the start of the project this would probably have been the driver layer implemented instead of uClinux [microC/OS2, 2002].

3 Embedded Integrated Development Environment

3.1 Introduction

The IDE for any embedded project is fundamental to the project and is a fairly unique area of software development. Most programming projects involve compiling the system on the target that it is likely to run on in future, however in the case of embedded projects a cross-compiled environment must be set up. A cross-compile is the process of creating executable code for a platform other than the one on which the compiler is run[Wikipedia 2007]. In the case of developing for the NIOS2 processor there is an additional HDL layer to consider.

Due to the layered structure of embedded systems it is common to implement a tool-chain which effectively bridges the different coding techniques and languages that exist on different layers. The Figure (3.1) shows the layers that are common to most embedded projects and how they relate to this project. It also shows the tool-chain used to bridge the layers.

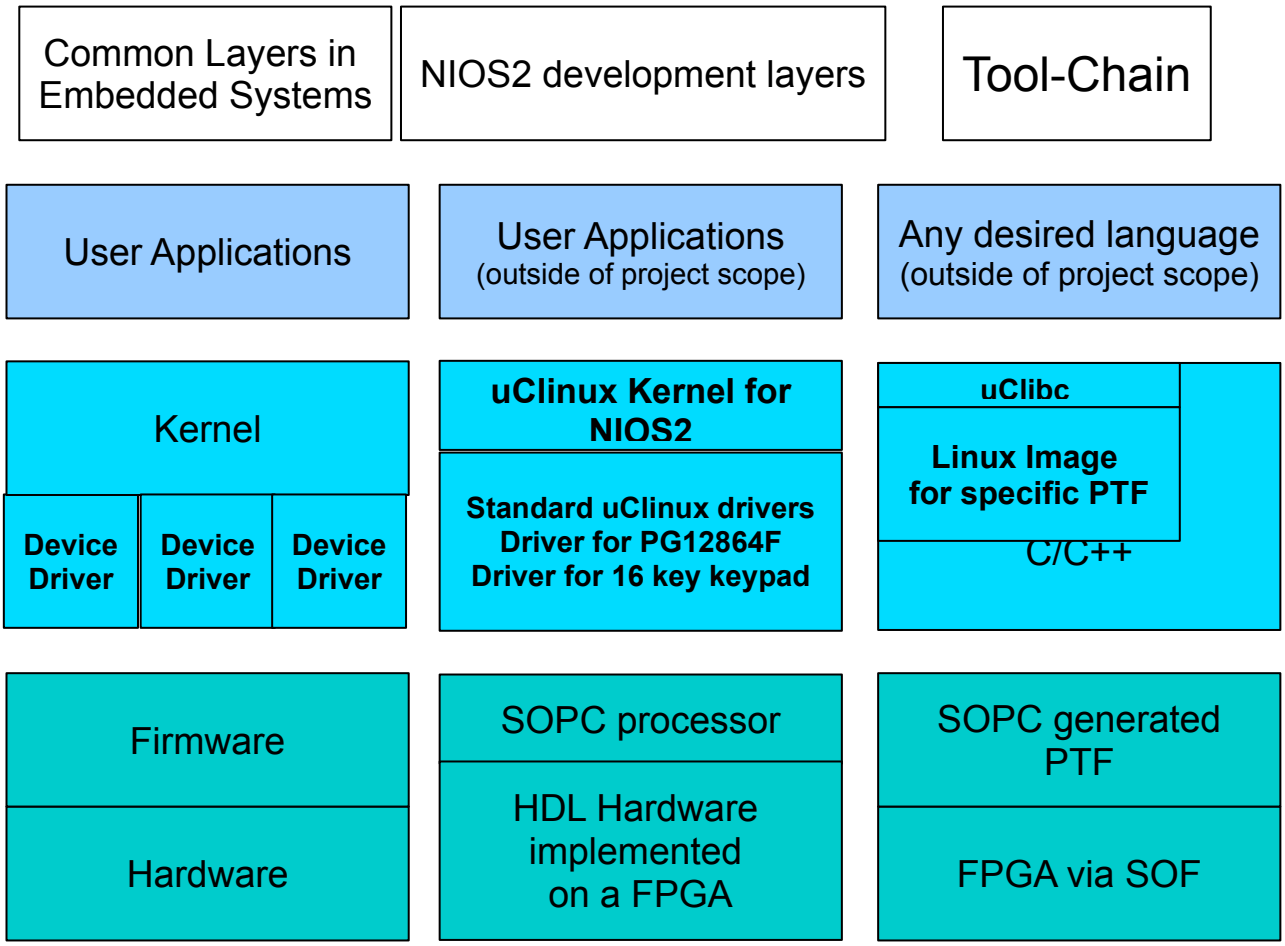

Figure 3.1 The layering structure and associated tool-chain for uClinux running on NIOS2

Key:

User Space

The top layer is where user applications are developed.

Kernel Space

The middle layer (referred to as Kernel space) is where the PG12864F LCD driver and the 16 key keypad will be implemented. The drivers are added to the kernel and are the inputs to this portion of the tool-chain. The full kernel is then compiled to target a specific PTF and the output is a uClinux image which can be downloaded to the board (see appendix C).

Hardware Space

The bottom layer (referred to as hardware space/layer) has the Linux image as its input and the SOF file is used to download this image to the board.

The purpose of this section and Figure 3.1 is to introduce the concepts of tool-chains and cross-compilation. Section 3.1.1 and 3.1.2 outline the specific details of the tool-chain used for implementing the drivers.

3.1.1 Altera packaged software tool-chain

The Altera IDE consists of a selection of software packages each of which plays a part in the development toolchain. They are as follows:

3.1.1.1 System on Programmable Chip builder

The SOPC builder allow us to add components to the NIOS2 processor and then wrap the whole package into a single HDL block. The extract from the SOPC builder below (Figure 3.2) shows the NIOS2 CPU (highlighted), as well as some of the components that are currently attached.

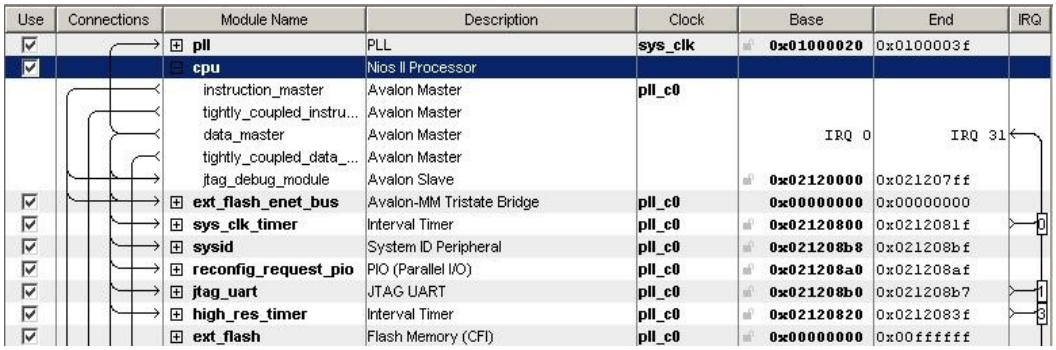

Figure 3.2 An extract from the SOPC builder

Components are then added and connected using the Avalon Switching Fabric [Altera, 2007]. There are many standard components written in HDL code that can be inserted which can be easily added to the system (examples include Ethernet cards and PCI buses). Parallel Input Output ports (PIO ports) are required to control the peripherals and these have been added into the SOPC builder. Figure 3.2 illustrates the standard

components attached to a 1c12 board, as well as the PIO ports added to control the PG12864F LCD and 16 button keypad.

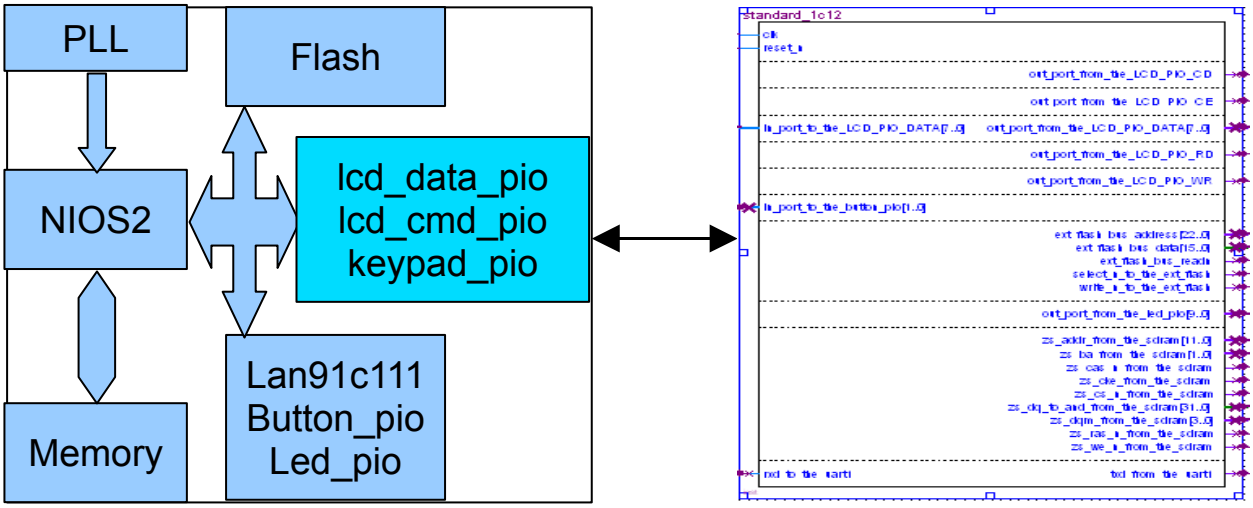

Figure 3.3 NIOS2 components generated into a single SOPC HDL entity

The file generated by this process is a .ptf file which is used to update the entity in Quartus2. In Figure 3.3 shows that there are additional ports to lcd_data_pio, lcd_cmd_pio and keypad_pio (for a full size image see Appendix E).

The .ptf file is used when building projects in the NIOS2 IDE to create the system.h file. An excerpt of the system.h is shown below:

```
/*
 * led_pio configuration
 *
 */

#define LED_PIO_NAME "/dev/led_pio"
#define LED_PIO_TYPE "altera_avalon_pio"
#define LED_PIO_BASE 0x02120870
#define LED_PIO_SPAN 16
#define LED_PIO_DO_TEST_BENCH_WIRING 0
#define LED_PIO_DRIVEN_SIM_VALUE 0
#define LED_PIO_HAS_TRI 0
#define LED_PIO_HAS_OUT 1
#define LED_PIO_HAS_IN 0
#define LED_PIO_CAPTURE 0
#define LED_PIO_DATA_WIDTH 10
#define LED_PIO_EDGE_TYPE "NONE"
#define LED_PIO_IRQ_TYPE "NONE"
#define LED_PIO_FREQ 85000000
#define ALT_MODULE_CLASS_led_pio altera_avalon_pio
```

Figure 3.4 definitions for the LED's found in the system.h file

Both the 1c12 board and the 2C35 board run the same NIOS2 CPU and most of the peripherals remain similar (uart, led_pio, flash, etc.) however the one distinct difference is that the 2C35 board has DDR RAM. When compiling the uClinux kernel we would like it to run from this space and this requires that the CPU's exception vector points to the DDR address space instead of the 64kb of on-chip RAM. The CPU must therefore be edited in the SOPC builder to affect this change as shown below.

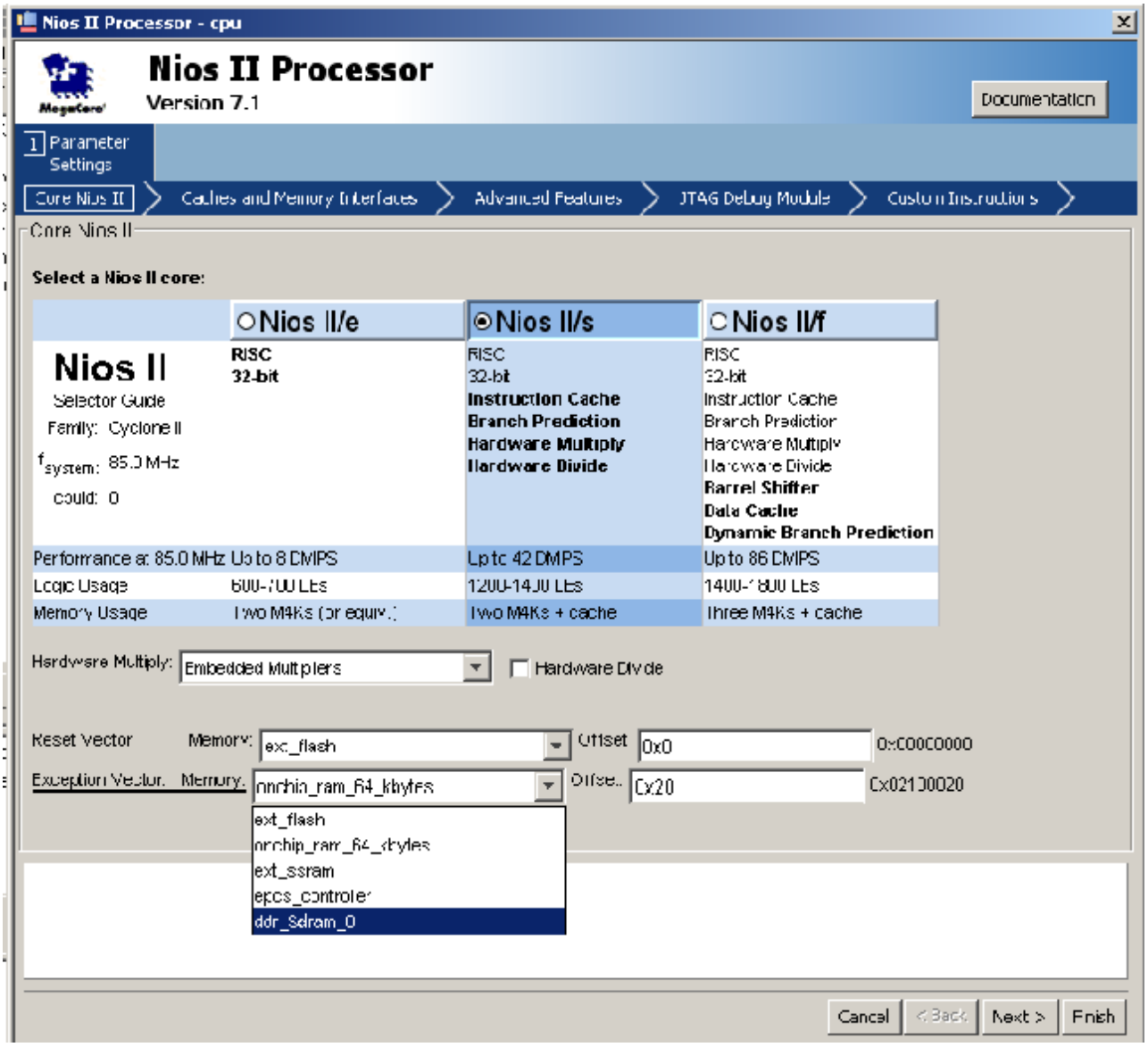

Figure 3.5 Changing the CPU's exception vector to target the DDR RAM instead of the on-chip RAM (full size image found in Appendix E)

3.1.1.2 Generating SRAM Object File using Quartus2

To program the FPGA we need to compile the Quartus2 project with the changes made in the SOPC builder. Pin assignments are made in Quartus2 and the SRAM Object File is generated when the project is compiled. See Appendix E for the generated files. Up to this point the tool chain for uClinux is exactly the same as the tool chain for the Altera packaged software.

3.1.1.3 NIOS2 IDE

Using the system generated above we can create a driver using the Altera provided HAL. The HAL instantiates and registers drivers automatically during system initialisation. Drivers are structured according to what the parser expects and placed within the HAL directory structure to allow the automated system to build them into the project [NIOS2 Software Developers Handbook 2007].

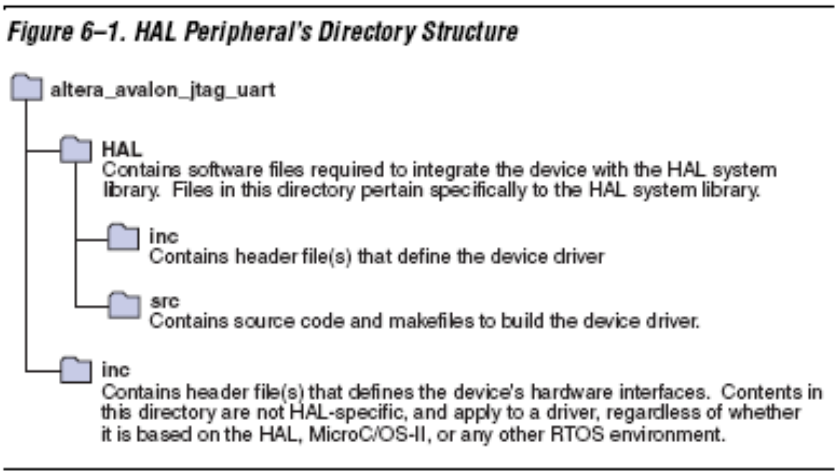

Figure 3.6 The required directory structure for implementing drivers using the HAL

This method of development seems to provide a very swift method of implementing drivers and controlling underlying hardware. A significant amount of time was spent trying to develop the necessary drivers for this structure using the 1c12 evaluation board. After modifying some of the drivers already present in the HAL and finding that these drivers were not being parsed in the way outlined in the documentation it was decided that the evaluation board did not provide features for modifying the HAL.

3.1.2 uClinux tool-chain

The initial steps in the uClinux tool-chain remain the same as those of Altera's HAL. They set up the hardware on which uClinux will run.

3.1.2.1 System on Programmable Chip builder

Construct a system .ptf as outlined in the Altera packaged software tool-chain section.

3.1.2.2 Generating SRAM Object File using Quartus2

Generate SRAM Object File(.sof) using Quartus2 as outlined in the Altera packaged software tool-chain section.

3.1.2.3 Host-side uClinux kernel

Implementation of the uClinux kernel requires it to be compiled on the host side and then ported to the board, as is common for many embedded operating systems. This requires a host system running Linux and also requires that the standard libraries are modified to target uClibc instead of glibc which normally runs on standard Linux based systems (see appendix A for more information on uClibc).

As with all Linux programming Makefiles are used to assist in compiling and linking projects. Makefiles are shell scripts that call the compiler commands. When compiling a kernel like uClinux a single makefile would be very complicated, so there are different makefiles for various subsections of the kernel. The makefiles form a hierarchical structure where a global make calls lower tier makefiles and can only complete after each of the makefiles completes its compilation.

As part of the tool-chain Fedora 5 was used as the host-side operating system for compiling the uClinux kernel and modified to use the uClibc library. The uClinux kernel was downloaded from www.uclinux.org along with the most recent patch.

The Nios Community Forum provide a gcc cross-compiler which forms the basis for the kernel space tool-chain (this forms a small part of the overall project tool-chain). Installing this tool-chain on the host-side allows the kernel and drivers to be cross-compiled, targeting the NIOS2 processor. The decision to run Fedora 5 on the host system was based on one of the requirements for installing the gcc tool-chain which requires version 3.4.6 or newer. The installation instructions are found on the Nios community forum Wiki but a brief summary is as follows [Nios Wiki, 2007]:

- Switch to the root or super-user on the host system's operating system
- Download the tool-chain binaries from the Nios Wiki and extract them
- Set up the Linux environment variable (which normally points to glibc) to target the directory where the binaries were placed
- Test to ensure that the installation was successful by checking that the nios2-linux-uclibc-gcc tool exists

The remainder of the IDE set up on the host system consists of extracting the kernel and configuring it.[Nios Wiki, 2007]:

- **/>make vendor_hwselect SYSPTF=<directory structure>/full_1c12.ptf**

  uClinux has been ported to many platforms; one of them being the NIOS2 processor. This means that we can inform the cross-compiler that our platform is a NIOS2 processor with any additional features that were added using the SOPC builder. In the same way as the .ptf file gets built into a system.h in Altera's HAL, we insert the .ptf file at the start of the

compilation process and while building the kernel a nios2_system.h is generated in the </uClinux/linux-2.6.x/include> directory.

- **/>make menuconfig**

  The menuconfig opens a menu based selection process which allows kernel features and applications to be added to the compile. The loadable module support is the main feature that is required for implementing and testing drivers, however browsing this menu structure gives an indication of how powerful the uClinux kernel is and the various programs and features that can be added in. Also of interest is the BusyBox application which contains many of the familiar Linux programs which have been modified to use very small coding footprints (see Appendix B for more details).

- **/>make romfs**

  At this stage we make the files associated with the romfs which is a small read-only file system originally designed for Linux. It is extremely simple and therefore has minimal overheads. There are 2 ways in which the overhead is reduced: Firstly it is a read-only file system which means that the disk cannot be "used", it gets built with the uClinux kernel image and changes to must be made before building the kernel. The second is that it stores the minimum number of features and notable exclusions are: no modification dates and no Linux style permissions [sourceforge.net, 2007].

- **/>make**

  This refers to the global makefile which calls up many other makefiles located in almost every directory. For example the char drivers directory, <uClinux/linux-2.6.x/drivers/char>, has a makefile which systematically

compiles each driver stored. In the case of the drivers that are being developed we edit the makefile that is in the <uClinux-dist/linux-2.6.x/drivers/misc> directory such that our added drivers are compiled into the project.

- **/>make linux image**

  This takes the already compiled kernel and converts it into a single image file. The image file can be found in the <uClinux-dist/images> directory and can be copied to the 1c12 board using the Quartus2 programmer and the .sof file generated by Quartus2. Note that the .sof file has a finite life since a free Quartus2 licence was used to generate it.

  For instructions on how to download and run the generated zImage on one of the boards please see Appendix C.

4 Hardware Layer

This section of the report is an in-depth discussion of the hardware developed in the project.

4.1 Selecting a NIOS2

One of the advantages of running a soft-core processor is that you can select between different processor models. The NIOS2 comes in 3 variations namely: economy, standard and fast. The details of each model are listed in table 4.1 below and figure 3.3 shows an extract from the SOPC builder.

	NIOS2/e	**NIOS2/s**	**NIOS2/f**
NIOS 2	RISC	RISC	RISC
Features	32bit	32bit	32bit
		Insruction Cache	Instruction Cache
		Branch Prediction	Branch Prediction
		Hardware Multiply	Hardware Multiply
		Hardware Divide	Hardware Divide
			Barrel Shifter
			Data Cache
			Dynamic Branch Prediction
Performance at 85MHz	Up to 8 DMIPS	Up to 42 DMIPS	Up to 86 DMIPS
Logic usage	600-700 LEs	1200 – 1400 LEs	1400 – 1800 LEs
Memory Usage	2 M4Ks	2 M4Ks + cache	3 M4Ks + cache

Table 4.1 Extract from the SOPC builder

The trade-offs between the different CPUs are clear from the table, increasing the speed of the CPU takes up more space on the FPGA and requires more memory. Any

of the above processors would have been suitable for the task of running uClinux and so the default NIOS2/s was chosen.

4.2 Physical Hardware

4.2.1 Cyclone 1 Evaluation Board (1c12 Board)

The 1c12 board (officially called the Cyclone 1) houses the FPGA unit on which the NIOS2 processor is implemented. Details of the FPGA unit (EP1C12F324) are found in table 4.3 reproduced directly from [Cyclone FPGA Family datasheet, 2003].

The board (as shown in figure 4.1) has a prototyping area in the top right hand corner to which the FPGA pins: proto_g1_io[7..0], proto_g2_io[7..0], proto_g3_io[7..0] and proto_g4_io[7..0] are linked. The external hardware is attached to this prototyping area. The pin mapping is covered in section 4.4.1.

Some of the physical attributes of the 1c12 board are listed in table 4.2 below. The maximum number of logic elements used by the NIOS2 standard core is 1400 so only a small fraction (11.6%) of the total FPGA space is used by the processor. Due to the fact that no other hardware was implemented on the FPGA it may have been better to use the NIOS2/f core, but in the end any of the three cores would have been able to run the operating system with resources to spare so it was decided to stay with the original decision.

Table 1. Cyclone Device Features					
Feature	EP1C3	EP1C4	EP1C6	**EP1C12**	EP1C20
Logic Elements	2,910	4,000	5,980	**12,060**	20,060
M4K RAM blocks (128x 36 bits)	13	17	20	**52**	64
Total RAM bits	59,904	78,336	92,160	**239,616**	294,912
Phase Locked Loops	1	2	2	**2**	2
Maximum user I/O pins	104	301	185	**249**	301

Table 4.2 Cyclone device features

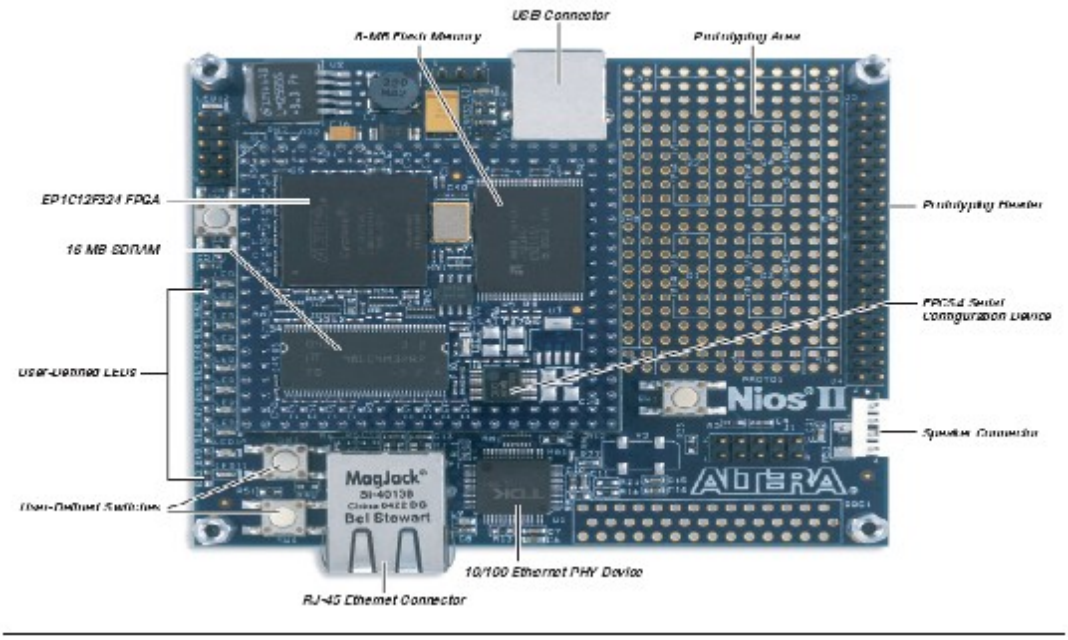

Figure 4.1 The NIOS2 Cyclone 1c12 evaluation board

4.2.2 NIOS2 2c35 Development Board

The Cyclone 2 board, referred to as the 2c35 board, houses and gives access to the EP2C35F672C6N FPGA unit, similar to the 1c12 board shown above. The board is shown below in figure 4.2 and, similarly to the 1c12 board, the prototyping area will be used to access the drivers. The important details pertaining to this board are found in the [Cyclone II Edition Reference Manual, 2007] including the FPGA pins used to access the external pin headers. The pin mapping will be covered in section 4.4.2.

Altera lists the 2c35 boards features as follows:

- Nios Development Board Cyclone II Edition A Cyclone II EP2C35F672C5 or EP2C35F672C5N FPGA with 33,216 logic elements (LE) and 483,840 bits of on-chip memory
- 16 MBytes of flash memory
- 2 MBytes of synchronous SRAM

- 32 MBytes of double data rate (DDR) SDRAM
- On-board logic for configuring the FPGA from flash memory
- On-board Ethernet MAC/PHY device and RJ45 connector
- Two 5.0 V-tolerant expansion/prototype headers each with access to 41 FPGA user I/O pins
- CompactFlash connector for Type I CompactFlash cards
- 32-bit PMC Connector capable of 33 MHz and 66 MHz operation
- Mictor connector for hardware and software debug
- RS-232 DB9 serial port
- Four push-button switches connected to FPGA user I/O pins
- Eight LEDs connected to FPGA user I/O pins
- Dual 7-segment LED display
- JTAG connectors to Altera devices via Altera download cables
- 50 MHz oscillator and zero-skew clock distribution circuitry
- Power-on reset circuitry

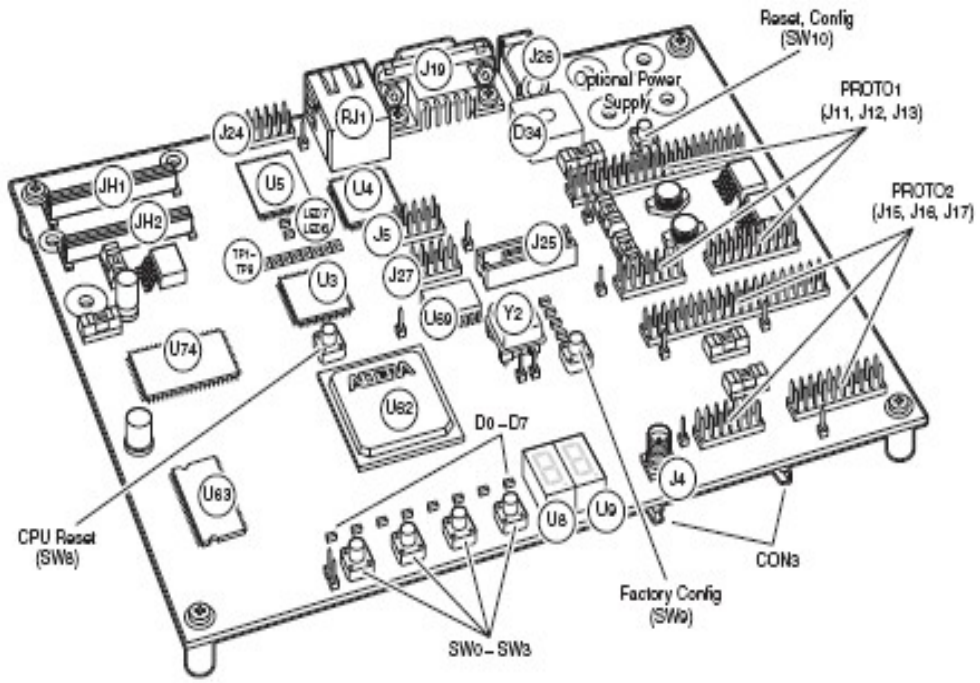

Figure 4.2 The 2c35 development board.

4.3 SOPC Hardware

As discussed in the IDE section the SOPC builder allows for a selection of HDL components to be bundled into a single HDL entity. Initial development was done by attempting to keep the hardware fully general (unmodified). The time spent trying to build drivers without modifying the SOPC symbol yielded no useful results. The SOPC builder was then included in the tool-chain and additional PIO ports were added to the basic configuration. The following memory mapped IO table is specified in the SOPC builder and is available to both the Altera HAL (<software\"project_name"\Debug\system_description\system.h> file) and uClinux (<uClinux/linux-2.6.x/include/nios2_system.h>). This table is specific to the development done on the 1c12 board. The addressing table for the 2c35 can be found in appendix D.

`#define na_dma`	`0x809208c0`
`#define na_dma_irq`	`0`

#define na_ext_flash	0000000000
#define na_ext_flash_size	0x00800000
#define na_ext_flash_end	0x00800000
#define na_epcs_controller	0x00900000
#define na_epcs_controller_size	0x00000800
#define na_epcs_controller_end	0x00900800
#define na_epcs_controller_irq	1
#define na_sys_clk_timer	0x80920800
#define na_sys_clk_timer_irq	2
#define na_jtag_uart	0x80920820
#define na_jtag_uart_irq	3
#define na_button_pio	0x80920830
#define na_button_pio_irq	4
#define na_led_pio	0x80920840
#define na_high_res_timer	0x80920860
#define na_high_res_timer_irq	5
#define na_uart1	0x809208a0
#define na_uart1_irq	6
#define na_sdram	0x01000000
#define na_sdram_size	0x01000000
#define na_sdram_end	0x02000000
#define na_sysid	0x80920828
#define na_sdram_pll	0x80800000
#define na_lcd_pio_cmd	0x80800020
#define na_lcd_pio_data	0x80800030
#define na_keypad_pio_row	0x80800040
#define na_keypad_pio_row_irq	7
#define na_keypad_pio_col	0x80800050

Table 4.3 An extract from the nios2_system.h when compiling the uClinux kernel for the 1c12 board

The final five entries are the components added to control the device drivers. The details entered into the SOPC builder are as follows:

	Bus width	Pin direction	Additional options
lcd_pio_cmd	4	Output	none
lcd_pio_data	8	Bi-directional	none
lcd_keypad_col	4	Output	none
lcd_keypad_row	4	Input	Rising edge trigger, generate irq

Table 4.4 Details of the added SOPC components

4.4 FPGA Hardware

The FPGA hardware changes are very minimal and consist of adding access pins to the memory mapped IO ports created by the SOPC builder. This was easily achieved on the Quartus2 project for the 1c12 board but proved more problematic when adding the pins to the 2c35 board.

4.4.1 Assigning the proto area on the 1c12 board

When the project for the standard 1c12 board is opened it is presented as a schematic diagram consisting of the system created by the SOPC builder and associated connections. After making the necessary changes to the SOPC builder the schematic file must be updated to reflect the added PIO ports. This is done by simply accessing the blocks options and selecting: “update symbol or block”. All that is required from there is to place primitive Quartus2 pins and attach them as shown in figure 4.3 below. It is unfortunate that when assigning these pins sequentially to proto_g(1/2/3/4)_io[7..0] they are not linked sequentially to the physical pins. This caused some of the physical hardware interfacing to be fairly untidy.

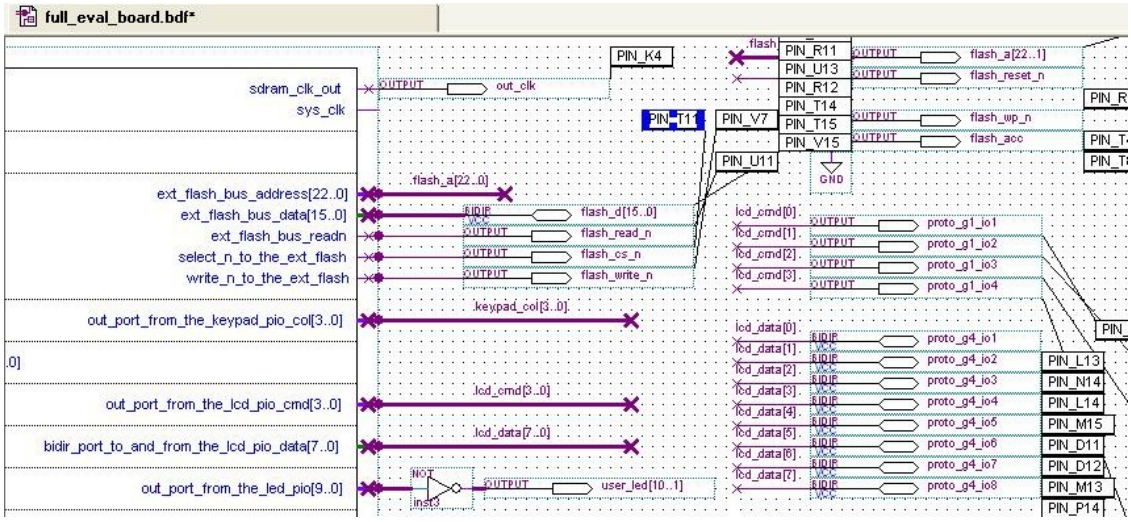

Figure 4.3 Placement of the Quartus2 primitive pins and connections to the SOPC added PIO ports

4.4.2 Assigning the proto area on the 2c35 Board

This proved to be significantly more difficult than using the 1c12 board. The SOPC builder generates a HDL file **<Final_project_2c35\NiosII_cycloneII_2c35_standard_sopc.vhd>** which defines all of the functionality and interconnections between the SOPC components. This file has a wrapper file **<Final_project_2c35\NiosII_cycloneII_2c35_standard.vhd>** which is the equivalent of the schematic provided for the 1c12 board. The problem is that it cannot be automatically updated, unlike the schematic of the 1c12 board. "If you modify and regenerate the SOPC Builder design, the port list of the SOPC Builder instance may change. You must manually edit the HDL wrapper file to rectify any discrepancies." [2c35 project readme.txt, unknown date]. This required a very quick refresher course in VHDL port mapping. Another problem with this method was that simply naming a pin "proto1_io[7..0]" did not assign it to the FPGA pins. This had to be done manually using the Pin assignment editor and required another crash course in Quartus2 PIN assignments.

4.4.3 Board logic levels

Both the 1c12 board and the 2c35 board support the following logic standards:

Table 8–1. I/O Standards Supported by Cyclone Devices Notes (1), (2)						
I/O Standard	Type	Input Voltage Level (V)	Output Voltage Level (V)	Input V_{REF} (V)	Output V_{CCIO} (V)	Termination V_{TT} (V)
3.3-V LVTTL/LVCMOS	Single-ended	3.3/2.5	3.3	N/A	3.3	N/A
2.5-V LVTTL/LVCMOS	Single-ended	3.3/2.5	2.5	N/A	2.5	N/A
1.8-V LVTTL/LVCMOS	Single-ended	3.3/2.5/1.8	1.8	N/A	1.8	N/A
1.5-V LVCMOS	Single-ended	3.3/2.5/1.8/1.5	1.5	N/A	1.5	N/A
PCI (3)	Single-ended	3.3	3.3	N/A	3.3	N/A
SSTL-3 Class I and II	Voltage-referenced	–0.3 to 3.9	3.3	1.5	3.3	1.5
SSTL-2 Class I and II	Voltage-referenced	–0.3 to 3.0	2.5	1.25	2.5	1.25
LVDS Compatibility	Differential	0 to 2.4	VOD = 0.25 to 0.55	N/A	2.5	N/A
RSDS Compatibility	Differential	0.1 to 1.4	VOD = 0.1 to 0.6	N/A	2.5	N/A
Differential SSTL - 2	Differential	N/A (4)	2.5	1.25	2.5	1.25

Figure 4.4 list of supported logic standards[Cyclone FPGA Family datasheet, 2003]

The default standard for the PIO pins is LVTTL 3.3V. This means that for interfacing with the PG12864F LCD some conversion circuitry is required.

4.4.3.1 Interfacing circuitry for the PG12864F

The voltage conversion circuitry consists of 2 sets of 8 octal buffers packaged on 2 74HCT541 chips. A schematic of the board is shown in appendix E, as well as a picture of the strip(or vero) board prototype.

4.4.3.2 Interfacing circuitry for the 16 key keypad

No voltage conversions were necessary so the keypad circuitry consists only of simple pull-up resistors and connectors. The schematics and images of the physical board are provided in appendix E.

4.4.4 PG12864F Graphical LCD

“The Powertip PG12864 is a 128x64 pixel, intelligent graphic LCD based on the Toshiba T6963C controller chip. It has a 128 word character generator ROM and a 4Kb RAM for text or graphics. It can function in either textual or

graphical mode." [Richard Armstrong, 2006] For the implementation of the drivers the textual mode will be used.

4.4.5 16 Key Keypad

The standard 16 key keypad has a pin for each row and each column. Reading the keypad is often done by polling the keypad for key presses. Usually the rows are read until a change is detected. Then the columns are cycled through to determine the exact key pressed. This system is highly inefficient due to the fact that it occupies the processor for the duration of polling. The idea behind the structure is to use the least possible number of pins to access the maximum number of buttons. See sections 5.4 for the implementation details.

5 Kernel Layer

Writing software for Linux (and most other protected mode operating systems) is divided into 2 disciplines; namely writing user space applications and writing kernel space (or layer) applications. These different areas of software development are defined by the sections of memory from which they run. They are strictly separated, each with its' own section of memory and under normal circumstances they cannot access each other's space. User space applications often swap memory out to the hard drive using a page swapping system to increase the visible size of the memory. This technique is very rarely applied to kernel memory because (amongst many other reasons) it contains the code necessary to manage the swapping procedure. It would be unfortunate to have the code necessary to get data swapped off the hard drive residing on the hard drive [Wikipedia, 2007][Linux Device Drivers, 2005].

The drivers that have been written as part of the deliverables for this project are kernel space applications.

The on-line book: “Linux Device Drivers” [Linux Device Drivers, 2005] provides very useful insight into the data structures involved in writing drivers.

5.1 Writing Linux Drivers

The concept of open source software and more specifically open source operating systems allows anyone with the necessary programming background to examine and modify the kernel. Drivers for Linux are written such that the underlying hardware is completely hidden from the application layers. Driver modules should allow users (usually other programmers) to transparently access hardware as if they are reading

and writing to a file. When opening a file for reading and/or writing Linux assigns a file descriptor to the opened file. The kernel manages a table of all open files and the file descriptor is an index to that table. Processes can access the required file by making a system call with the appropriate file descriptor as an argument. In Linux (and other Unix-like systems) drivers aim to appear no different from files and therefore character drivers must be written such that a file descriptor can be opened for the device (often referred to as a special file)[Wikipedia, 2007].

Drivers, commonly referred to in Linux as modules, can be loaded into the kernel at runtime. This feature is particularly important to this section of the project and the set up of these features for uClinux is discussed in chapter 3. Modules in the kernel can usually be classified into three categories: character module, block module or network module. Character modules usually deal with streams of individual bits and the keypad and LCD devices fall into this group.

5.2 Software development

This section discusses the strategy for developing the drivers required in the terms of reference. Development consists of writing software and then adding it and re-compiling the kernel. The result is an image which can be downloaded to the targeted board. A selection of image files have been provided in appendix E and these demonstrate some of the explanations below. For information on how to download an image please see appendix C.

5.2.1 Basic Linux modules

The most basic 'hello world' module is found on the [Nios Wiki, 2007] and must be added into the appropriate makefile and the Kconfig file must also be modified to include the new module. A basic shell for the PG12864F and keypad modules has been created and compiled into a Linux image and is provided in appendix E. Please see the [Nios Wiki, 2007] or [Linux Device Drivers {page 16}, 2005] for more details on compiling a basic module.

5.2.2 Adding modules to the kernel

The device control system for Linux has changed considerably over the various kernel versions. Originally kernels contained large amounts of redundant data stored in a static </dev> directory structure. Newer kernel versions allow devices to be assigned as they are needed to save space and a device manager controls the allocation process. The first device manager was devfs but this has largely been replaced by the udev device manager. The device management system must ensure that drivers can be accessed as files from the /dev directory and due to the change from static to dynamic systems each device is assigned a major and a minor number. These can be requested from the kernel or assigned statically within the driver [Wikipedia,]. The file: </uClinux-dist/linux-2.6.x/Documentation/devices.txt> contains details of which major and minor numbers have been used, which devices are using them and which are still free to be allocated. Major and minor numbers for the PG12864F LCD and the 16 key keypad were chosen from the 240-254 range which is set aside for experimental/local use on character devices.

The **/>modprobe "device name"** command registers the major and minor numbers with the kernel and calls the device init function.

The **/>rmmod "device name"** command de-registers the device and calls the device exit function.

Once major and minor numbers have been assigned a device file is opened using **/>mknod "device" c "major number" "minor number"** and placed in the </dev> directory.

5.2.3 IOCTL

Most drivers cannot function with only the ability to read to and write from them; additional hardware control needs to be done through the driver. A good example is the serial port which needs baud rate settings and a few other

parameters before data can be exchanged. The IOCTL data structure is often used to support these requirements. It is a data structure that consists of a collection of function pointers and allows the various hardware operation modes to be supported. An example of the IOCTL struct is shown below:

```
struct file_operations lcd_fops =
{
  .read = device_read,
  .write = device_write,
  .ioctl = device_ioctl,
  .open = device_open,
  .release = device_release,
};
```

Where each element of the struct points to one of the following functions:

```
device_read ()
device_write ()
device_ioctl ()
device_open ()
device_release ()
```

This structure is registered with the kernel during the modules initialisation (init function) and a pointer is added to the kernel devices table. When the kernel sees a call to that device it has a pointer to the structure which in turn points to a specific function. Figure 5.1 below shows an example of the interaction between user space applications and the underlying hardware using the IOCTL functionality. The steps for producing this result are as follows:

1. Add driver module to kernel using />**modprobe** and />**mknod.**
2. The driver module init function is called by />**modprobe** and within the init function are the calls to register the IOCTL.
3. When IOCTL is registered with the kernel a pointer to the IOCTL fops struct is added to the kernel's devices table.
4. User Applications can now request for the kernel to attempt to open the device and hand back a file descriptor. A pointer to the device_open () function in the driver code exists in the fops struct and is called at this point. If the open function returns successfully then the Kernel allocates a file descriptor and returns it to the user application.
5. The file descriptor is then used to allow user space programs to access the driver through normal file reads and writes. These calls are

submitted to the kernel which in turn calls the fops struct for the correct function pointers to access the drivers read and write functions.

6. Releasing the device calls the appropriate function as before and releases the file descriptor. It does not however remove the module from the kernel. If desired the driver can be removed using **/>rmmod** but this is not shown in figure 5.1, it would call the drivers exit function.

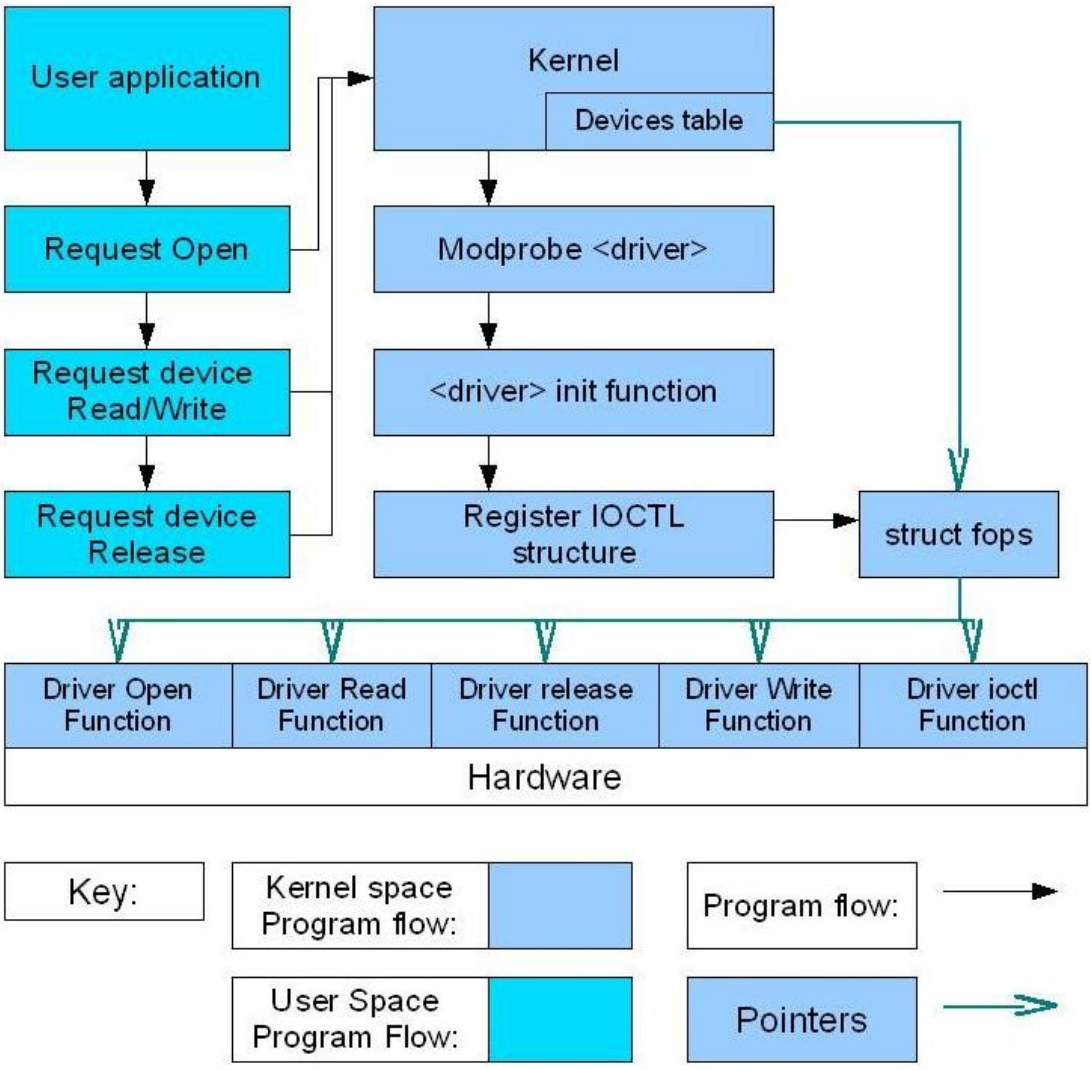

Figure 5.1 The interaction between user space applications and the hardware through the kernel and IOCTL structure

Of particular interest is the ioctl function call. This is not to be confused with the overall IOCTL structure. An ioctl function exists within the fops structure

and is used to control the device's settings. The keypad requires only data to be exchanged so the ioctl function remains empty, however for the LCD it would be useful to be able to switch between graphical and non-graphical modes. This would be taken care of using the ioctl function's argument.

1

5.3 PG12864F device driver

The implementation of the PG12864F device driver is fairly straight forward. A module shell is created such that the init function will run when the device is added to the kernel (see section 5.2.1) and the init function registers the fops structure with the kernel. The LCD is then accessed through the input and output pins assigned in the SOPC builder via memory mapped io. The control of these pins is through the pointer assignments shown below. It is important not to allow the compiler to optimise these addresses by caching them so the volatile reserved word is used (see [Wikipedia , 2007] for more information).

```
//assigning pointers to the memory-mapped-io locations
volatile unsigned int * UI_LCD_C = (unsigned int *)na_lcd_cmd_pio;
volatile unsigned int * UI_LCD_D = (unsigned int *)na_data_cmd_pio;

//Writing to the memory and therefore to the output pins
*UI_LCD_C = 0xA;  //Places hex value A on the lcd_cmd output ports
*UI_LCD_D = 0xFF; //Places hex value FF on the lcd_data output ports
```

The bit manipulations used to control the LCD were used by Richard Armstrong [Richard Armstrong, 2006] and referenced there to John P Beale. These individual bit manipulations are then wrapped so that bytes can be written to the LCD.

```
write_c_byte (unsigned int byte)
write_d_byte (unsigned int byte)
```

Higher level functions use the command and data writing functions to allow characters to be written to the display.

Code for the PG12864F device driver can be found in appendix E.

5.4 16 key keypad device driver

The keypad device driver is implemented in a similar way to the LCD driver. The init and exit core of the module is used to register the fops structure however for an efficient driver we also need to register the interrupts. The SOPC builder allowed interrupt numbers to be assigned to the input PIO ports when the .ptf file was generated. The interrupt numbers therefore appear in the nios2_system.h file and can be registered with the kernel using the following code:

```
register_chrdev_region (“device structure”, 1, “device name”);
unregister_chrdev_region (“device structure”, 1, “device name”);
```

Once an interrupt is captured the keypad is polled briefly to determine the exact key pressed. Applications in user space do not have access to interrupts and therefore keypresses are stored in a buffer until they are read from the driver.

Code for the 16 key keypad device driver can be found in appendix E.

6 Results

6.1 Implementing a driver abstraction layer

The initial implementation of a hardware abstraction layer was to write drivers for Altera's HAL which is already set up in the packaged Altera software. This was not strictly in line with the terms of reference and when this tactic failed on the 1c12 board it was a great excuse to push for a full operating system. Initial driver implementations were unsuccessful and then editing of the original drivers that make up the HAL failed as well. Since changes made to a working driver did not reflect in subsequent builds it was assumed that the 1c12 evaluation board does not support these features and a full development board was ordered. Since there was a 2-week waiting period for the new hardware an attempt was made to port uClinux to the 1c12 board. This was successful and uClinux was run as the driver abstraction layer from that point. When the 2c35 board arrived a significant amount of time had been spent on getting the drivers developed and therefore it was decided to stick to uClinux on the 2c35 board. Implementing this was more challenging, mostly due to the 2c35's DDR RAM, but was successful and proved to be a better solution than using Altera's HAL. The figure below shows uClinux running on the 2c35 board just after it has been booted.

Figure 6.1 uClinux running on the 2c35 board. Image shows boot sequence and command prompt with the PG12864F driver about to be added to the kernel

6.2 HAL driver implementation for 1c12 board

A substantial amount of time was spent attempting to implement the LCD and keypad drivers for the Altera HAL. Altera documentation [NIOS2 software developer's handbook, 2007] claims that the HAL is capable of instantiating, initialising and registering drivers automatically. This was the original tactic for

implementing a driver abstraction layer and developing the drivers as outlined in section 3.1.1. The relevant data structures were created and placed in the required directory structure. "Taking advantage of the automation provided by the HAL is mainly a process of placing files in the appropriate place in the HAL data structure" [NIOS2 software developer's handbook, 2007]. Despite many hours of creating driver shells and moving them in and out of directory structures the HAL parser did not appear to recognise the changes. After much experimentation it was decided to modify some of the existing drivers and the changes inflicted on the button driver did not register in any of the re-builds. The Altera HAL was abandoned as a driver abstraction layer for the 1c12 board.

6.3 HAL driver implementation for the 2c35 board

Due to the time spent on attempting to get the uClinux experiment to run no attempts were made to run the Altera HAL on the 2c35 board however changes made to the native LCD driver did appear when it was run.

6.4 Porting uClinux to 1c12 board

Porting uClinux to the 1c12 board was very successful. Both user space software and kernel space software can be written and executed. An image called 1c12_uClinux_test.zim can be found with the uClinux images in appendix E. It can be uploaded to the board and the user space application linux_test and the 2 drivers can be run.

6.5 Driver implementation for the 1c12 board

The implementation of the driver in the kernel space appeared to work well. The drivers were written, makefiles edited, Kconfig edited and the kernel built. The drivers were successfully added into the kernel using **/>modprobe** and the inti and exit functions ran without error. The devices were successfully registered using the major and minor numbers with the kernel. A file in the /dev directory

was created and the drivers can open, read, write and close this file. The process went exactly as expected and showed no signs of failure at any point however when testing began there was no sign of data coming through the prototyping pins. This functionality was then tested by writing to the on-board LEDs instead of the proto pins. When these gave no visible response the basic input output functionality was tested using the Nios IDE. It was found that the input and output pins were functioning. Repeated testing on the uClinux environment revealed that the LEDs were in fact working however none of the output ports were latching. If a write to the LEDs was placed in an infinite loop they responded as expected however placing a delay within this loop caused them to appear to remain off or come on for so brief a period of time that no response could be detected. This problem was very surprising and eventually led to the 1c12 board being abandoned. Some correspondence with one of the members of the Nios Forum did not shed any light on the problem (the correspondence can be found in appendix E). One possible solution would be to implement the latching on the FPGA hardware outside of the SOPC system. The code created for running the driver system can be found in appendix E.

6.6 Porting uClinux to 2c35 board

Porting uClinux to the 2c35 board caused a surprising number of new problems most of them surrounding the DDR RAM. Firstly the CPU exception pointer had to be redirected to target the ddr_sram_0 component in the SOPC builder. The Quartus2 builds also failed when trying to allocate DDR pins if the project was not cleaned before compilation. Once these simple but frustrating problems had been sorted out uClinux was ported to the 2c35 board successfully. An image called 2c35_uClinux_test.zim with a basic test program and kernel module has been complied and can be found in the zImages section of appendix E.

6.7 Driver implementation on the 2c35 board

The PG12864F and 16 key keypad drivers have been implemented and can be added to the kernel using **/>modprobe**. They function as expected calling the

init and exit functions when the module is created. The fops is registered within the inti function and associated to the driver's major and minor numbers. Using **/>mknod "desired path" c "major number" "minor number"** a file can be added to the /dev directory and from there the drivers can be opened like files.

7 Conclusions and Recommendations

7.1 Project goals

Implementation of a hardware abstraction layer was very successful. uClinux is a powerful tool that has already been applied to a number of commercial applications. The kernel is flexible as expected and required by embedded developers. It functions very much like a regular Linux operating system and is reasonably easy to get up and running. Particular thanks to the Nios community and the developers at www.uclinux.org for supporting this software.

The expandable nature of the operating system has been shown by adding the required drivers to the kernel. In addition to module support uClinux provides almost all of the applications that a Linux developer would expect to see on a regular Linux system using BusyBox and uClibc. Of course some of the functionality has been scaled down but if the BusyBox version of an application is insufficient for a developer's needs the full application can be integrated into the kernel. Between uClibc and BusyBox most of the native Linux features are supported by uClinux but using only a fraction of the space.

The device driver goals were less successful. It was hoped that a comprehensive PG12864F driver would be written however the limited time allowed only for a very basic driver. There is still much room for improvement on the PG12864F driver in particular. The keypad requires only read functionality with no additional parameters that can be set so the driver developed is significantly more comprehensive but could still be improved upon.

Based on the terms of reference most of the project goals were achieved. Not all of the desired functionality was implemented and much improvement could be made particularly on the PG12864F device driver.

7.2 Operating system advantages

There are many advantages gained by adding an operating system to an embedded project however the focus of this thesis project is on improvements in the time-to-market and code re-use. With the functionality provided by uClinux a full operating system, drivers and applications are implemented on two separate development boards (both running the same processor) by a single developer, with little previous experience in embedded Linux and over a relatively short period of time. This demonstrates the some advantages gained by using uClinux however more complicated projects would have to be investigated to determine whether a significant advantage exists when used in practice.

7.3 Further development

There is still significant amount of development that could be done on the PG12864F device driver. The graphical mode needs to be properly supported. This support would probably be in the form of an API outside of the driver itself. The driver would only provide the mechanics regarding writing to individual pixels but a user space API would provide some better access controls. For example functions like:

```
void plot_sin_wave (int phase, int amplitiude)
```

should be implemented in a user space API.

Further development on the PG12864F device driver would include full graphical support as well as passing arguments to the LCD using the IOCTL structure which has been registered. These arguments would allow the text and graphical modes to be properly separated perhaps even with completely separate read and write functions.

8 Summary

A driver abstraction layer was implemented using the uClinux kernel. A device driver for the PG12864F and a generic 16 key keypad have been developed and can be added to the kernel at runtime. The aim was to investigate improvements in the time-to-market of an embedded project.

9 Bibliography

Note that underscores are not easily visible in hyper-linked web-addresses.

ALTERA [Altera 2007 (A)]: All documentation from Altera can be found on their website however the relevant material has been placed in appendix E.

- NIOS2 software developer's handbook. [Online]. Available: Appendix E
- NIOS Cyclone 1 datasheet. [Online]. Available: Appendix E
- NIOS Development Board Cyclone II Edition Reference Manual. [Online]. Avalible: Appendix E
- Cyclone FPGA Family datasheet, 2003
- SOPC Builder's System Interconnect Fabric. [Online]. Available: http://www.altera.com/products/software/products/sopc/avalon/nio-avalon_bus.html

Wikipedia, [Online]. Available:

- http://en.wikipedia.org/wiki/cross_compiler [October 2007]
- http://en.wikipedia.org/wiki/BusyBox [October 2007]
- http://en.wikipedia.org/wiki/Executable_and_Linkable_Format [October 2007]
- http://en.wikipedia.org/wiki/File_descriptor [October 2007]
- http://en.wikipedia.org/wiki/Input/Output_standards [June 2007]
- http://en.wikipedia.org/wiki/Inter-process_communication [October 2007]
- http://en.wikipedia.org/wiki/Kernel_(computer_science) [October 2007]
- http://en.wikipedia.org/wiki/POSIX [October 2007]
- http://en.wikipedia.org/wiki/User_space [September 2007]
- http://en.wikipedia.org/wiki/Volatile_variable [August 2007]

Armstrong, R.,(2006). *A CORDIC Based Scientific Calculator Built on the Nios II*, University of Cape Town

Corbet, J., Kroah-Hartman, G,. Rubini, A. (2005). *Linux Device Drivers*. O'Reilly Media: United States of America

Die.Net, *sash (8) – Linux man page*. [Online]. Available: http://linux.die.net/man/8/sash [1999, Oct. 2]

Jotspot wiki, *Binary Toolchain*. [Online]. Available: http://nioswiki.jot.com/WikiHome/OperatingSystems/%C2%B5Clinux/BinaryToolchain [2007, Mar. 03]

Jotspot wiki, M*odule Programming*. [Online]. Available: http://nioswiki.jot.com/WikiHome/OperatingSystems/ModuleProgramming [2007, Mar. 03]

Jotspot wiki, *BusyBox*. [Online]. Available: http://nioswiki.jot.com/WikiHome/OperatingSystems/BusyBox [2007, Mar. 03]

Jotspot wiki, C*ompile Hello*. [Online]. Available: http://nioswiki.jot.com/WikiHome/OperatingSystems/CompileHello [2007, Mar. 03]

Jotspot wiki, *uClinux Distribution*. [Online]. Available: http://nioswiki.jot.com/WikiHome/OperatingSystems/UClinuxDist [2007, Mar. 03]

LinuxDevices.com, *uClinux: World's most popular embedded Linux distro?*. [Online] Available*:* http://www.linuxdevices.com/articles/AT3267251481.html [2002, Sept. 24]

Opdenacker, M., *Introduction to uClinux*. [Online]. Available: http://free-electrons.com [2006, Aug. 21]

Romfs.sourceforge.net, What is romfs?. [Online]. Available: http://romfs.sourceforge.net [2007, Jun. 26]

GNU C library, [Online]. Avalible: http://www.gnu.org/software/libc/#Overview

uClibc C library, [Online]. Avalible: http://www.uclibc.org/

uClinux Embedded Linux/Microcontroller Project, [Online]. Available: http://www.uclinux.org/

Quartus2 2c35 project readme.txt file, Available: within the 2c35 project directory

Appendices

Contents:

Appendix A – uClibc

uClibc is a library developed for embedded projects. It it much smaller than glibc which is the regular C library for Linux . uClibc claims that almost all applications supported by glibc will run on uClibc. It can run on standard Linux as well as MMU-less systems such as µClinux. The uClibc project started because glibc does not support MMU-less systems and because it is complies with many standards and runs on just about every operating system and architecture making it a very large library. The size of glibc alone makes it a poor choice for embedded systems.

uClibc space saving advantages:

“Some of the space savings in uClibc is obtained at the cost of performance, and some is due to sacrificing features. Much of it comes from aggressive re-factoring of code to eliminate redundancy. In regards to locale data, elimination of redundant data storage resulted in substantial space savings. The result is a libc that currently includes the features needed by nearly all applications and yet is considerably smaller than glibc. To compare "apples to apples", if you take uClibc and compile in locale data for about 170 UTF-8 locales, then uClibc will take up about 570k. If you take glibc and add in locale data for the same 170 UTF-8 locales, you will need over 30MB!!! The end result is a C library that will compile just about everything you throw at it, that looks like glibc to application programs when you compile, and is many times smaller.” [http://www.uclibc.org/]

uClinux strongly recommends the use of uClibc rather than glibc and for this project uClibc was used for all kernel space compilation.

Appendix B – BusyBox

BusyBox was originally written by Bruce Perens in 1996 and was designed to allow a complete boot-able system to exist on a 1.44mb floppy disk that could be used as a rescue disk or to install a larger operating system, in this case Bruce Perens wanted to use BusyBox to install Debian. BusyBox is a package of software tools that provide the same functionality as the GNU core utilities. Items such as **/>ls** (list directory)and **/>chmod** (modify permissions of file) through to vi (a small text editor) are included in BusyBox

In the **/>make menuconfig** menu a list of the possible BusyBox utilities can be found. Each con be added using this menu structure and there are over 200 utilities that a Linux user or programmer would expect to find on a regular Linux system. It can provide most of the utilities specified in the Single Unix Specification plus many others that a user would expect to see on a Linux system.

It consists of a single small executable or binary package that replicates most of the tools in GNU fileutils, shellutils, etc. The BusyBox utilities are generally more limited than the full-featured GNU counterparts however the functionality that is provided behaves very much like the GNU equivalent. The much smaller size that BusyBox occupies makes it ideal for running on embedded systems.

Appendix C – Downloading images to the target board

Software required for downloading the kernel is: Quartus2 and the NIOS2 IDE either the Linux or the Windows version.

Once a uClinux image is generated it is placed in the **<uClinux/images>** directory and called **<zImage>** with no file extension. It was decided that an extension should be used and so zImage is re-named zImage**.**zim.

Also it is important to note that when compiling the kernel a system PTF is used. The PTF is defined by the SOPC builder and is therefore specific to only 1 board. The naming convention has been to put the board title on the front of the file name. Example 2c35_module_template**.**zim is a zImage that can be downloaded to the 2c35 board. Accompanying each zImage is a readme.txt explaining the purpose and functionality of each zImage.

To download an image follow the following steps:

1. Open 2 NIOS2 command shells
2. In the first command shell navigate to the Quartus2 project directory (the directory containing the .sof file generated by Quartus2)
3. enter the command:

   />nios2-configure-sof NiosII_cycloneII_2c35_standard_time_limited.sof
4. wait until prompted to press i for information and q for quit.
5. Leave the first shells running (don't press quit or close it) and switch to the second shell.
6. In the second shell navigate to the directory containing the zImage and enter: **>/nios2-download -g "*.zim"**
7. Once download is complete start a NIOS terminal **/>nios2-terminal**

Appendix D – Addressing table for the SOPC system to be run on the 2c35 board

The extract is from the nios2_system.h file found in the kernel directory structure at </uClinux-dist/linux-2.6.x/include/nios2_system.h>.

#define na_dma	0x809208c0
#define na_dma_irq	0
#define na_ext_flash	00000000000
#define na_ext_flash_size	0x00800000
#define na_ext_flash_end	0x00800000
#define na_epcs_controller	0x00900000
#define na_epcs_controller_size	0x00000800
#define na_epcs_controller_end	0x00900800
#define na_epcs_controller_irq	1
#define na_sys_clk_timer	0x80920800
#define na_sys_clk_timer_irq	2
#define na_jtag_uart	0x80920820
#define na_jtag_uart_irq	3
#define na_button_pio	0x80920830
#define na_button_pio_irq	4
#define na_led_pio	0x80920840
#define na_high_res_timer	0x80920860
#define na_high_res_timer_irq	5

#define na_uart1	0x809208a0
#define na_uart1_irq	6
#define na_sdram	0x01000000
#define na_sdram_size	0x01000000
#define na_sdram_end	0x02000000
#define na_sysid	0x80920828
#define na_sdram_pll	0x80800000
#define na_lcd_pio_cmd	0x80800020
#define na_lcd_pio_data	0x80800030
#define na_keypad_pio_row	0x80800040
#define na_keypad_pio_row_irq	7
#define na_keypad_pio_col	0x80800050

Appendix E – Files located on the CD

Appendix D consists of all the files located on the CD. Below is the directory structure of the CD.

Documents

- **Datasheets**
- **Design schematics**
- **Images of connector boards**

Project Files

- **Final 1c12 project**
- **Final 2c35 project**

Code

- **Drivers**
- **User space applications**

Thesis report

- **Report**
- **Images**

zImages

- **1c12 board images**
- **2c35 board images**

Additional Files