SMARTLiFE

A POINT OF INTELLIGENCE FOR WIRELESS SENSOR NETWORKS IN UBIQUITOUS ENVIRONMENT

by

Anwar Haq, M.Sc. Physics, 1995
Government College, University of the Punjab, Pakistan

A project
Presented to Ryerson University
In partial fulfillment of the
requirement for the degree of
Master of Engineering
In the program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2005

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

Anwar Haq

I further authorize, Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Anwar Haq

Ryerson University requires the signatures of all persons using or photocopying this project. Please sign below, and give address and date.

SMARTLiFE

A point of Intelligence for wireless sensor networks in ubiquitous environment

Anwar Haq,
Master of Engineering, 2005,
Electrical and Computer Engineering,
Ryerson University

Abstract

As home becomes more technologically advanced nowadays people not only need to protect their homes and families from theft or fire, but from carbon monoxide, excessive heat or low temperatures, flooding as well as monitoring their loved ones while they are away.
In this research Project, we present the design case for an intelligent embedded system (Hardware and Software) called "SmartLife". We build a working prototype for SmartLife which is made up of tiny sensors, mobile devices, appliances and personal computers from diverse sources. Considering diversity in smart home environment the architecture must be open and flexible to embrace a variety of entities without any special favor towards particular participants or target domains.

SmartLife will be the point of intelligence in the smart home environment (complete pervasive environment) which not only communicates with wireless sensors network (monitor & control) but also provides a secure state of mind to elderly homeowners.

The work introduces the basics of uClinux kernel as well as the differences between uClinux and the general purpose operating system Linux. It also introduces FPGA based soft-core CPU NIOS-II as an embedded platform for our research project. Our uClinux based architecture provides an integrated and comprehensive framework for building pervasive applications. We describe the design and implementation of our architecture as well as building SmartLife application within it.

Keywords:

NIOS-II, FPGA, Embedded Systems, RTOS, Microprocessor, uClinux, Micro controller, Ubiquitous & Pervasive Computing.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Gul N. Khan, for his support during the course of this project. I would like to thank Marc Leeman (Barco) and Scott (Psyent Corporation) for various discussions on porting uClinux and U-boot loader on NIOS-II stratix development board.

This project would not have been possible without the support and encouragement of my wife Mariam Haq for her exceptional patience as I spent countless hours porting, writing and testing Smartlife.

This project exists because of my parents support and encouragement. Last but not the least I am very thankful to my sister Misbah and younger brother Ikram for their support and encouragement during this research project.

Finally, I am grateful to Canadian Microelectronics Corporation (CMC) and Ryerson University for providing research facilities for this project.

Table of Contents

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

1.1 Overview

In the future, computation will be human-centered. It will be freely available everywhere, like batteries and power sockets, or oxygen in the air we breathe. Computation will enter the human world, handling our goals and needs and helping us to do more while doing less. We will not need to carry our own devices around with us. Instead, configurable generic devices, either handheld or embedded in the environment, will bring computation to us, whenever we need it and wherever we might be. As we interact with these "anonymous" devices, they will adopt our information personalities [1]. They will respect our desires for privacy and security. We won't have to type, click, or learn new computer jargon. Instead, we'll communicate naturally, using speech and gestures that describe our intent ("send this to Hari" or "print that picture on the nearest color printer"), and leave it to the computer to carry out our will as shown in figure 1.1.

New systems will boost our productivity. They will help us to automate repetitive human tasks, control a wealth of physical devices in the environment, find the information we need (when we need it, without forcing our eyes to examine thousands of search-engine hits), and enable us to work together with other people through space and time [1].

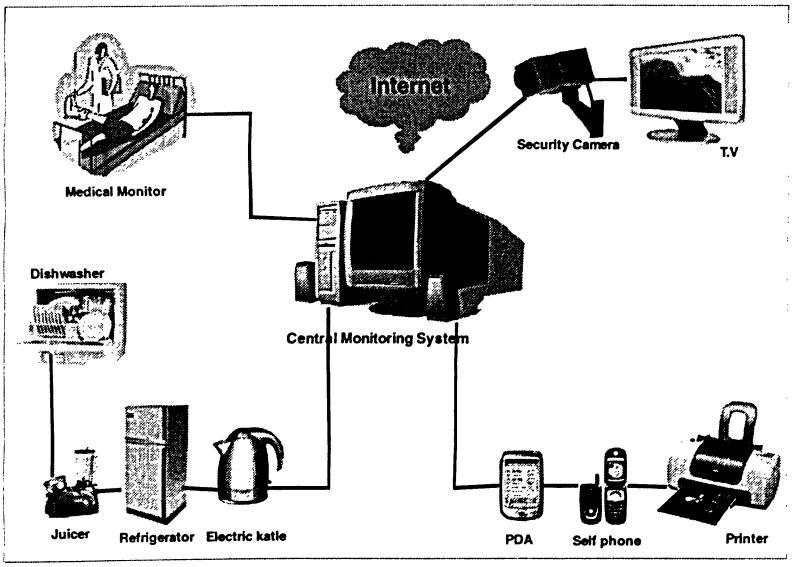

Figure 1.1: Smart Home Environment

Challenges:

The SmartLife system must master many technical challenges to support highly dynamic and varied human activities. It must be;

- Pervasive: It must be everywhere, with every portal reaching into the same information base.
- Embedded: It must live in our world, sensing and affecting it.
- Nomadic: It must allow users and computations to move around freely, according to their needs.
- Adaptable: It must provide flexibility and spontaneity, in response to changes in user requirements and operating conditions.
- Powerful, yet efficient: It must free itself from constraints imposed by bounded hardware resources, addressing instead system constraints imposed by user demands and available power or communication bandwidth.
- Intentional: It must enable people to name services and software objects by intent, for example, "the nearest printer," as opposed to by address.
- Eternal: It must never shut down or reboot, components may come and go in response to demand, errors, and upgrades, but SmartLife system as a whole must be available all the time.

The research mainly focuses on the architectural design of SmartLife using uClinux kernel with FPGA based soft-core CPU NIOS-II primarily accommodated by stratix FPGA (EP1S40F780C).

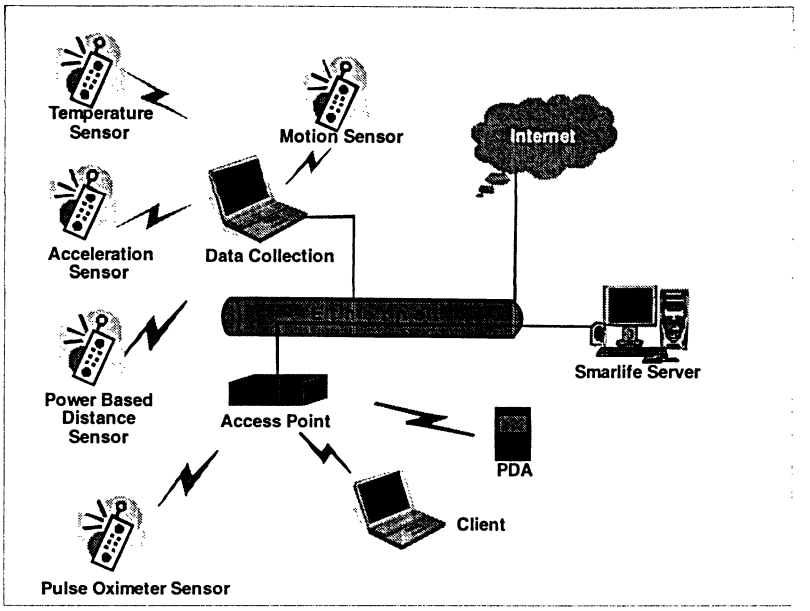

Figure 1.2: General Architecture of SmartLife

It also states out some differences against the general purpose Linux as well as introduces the architectural concept of uClinux kernel.

1.2 Project Organization

This project is organized into six chapters. This chapter provides an introduction and outlines the project organization. Chapter 2 surveys embedded systems. This chapter also provides some details about the FPGA based hardware platform used for our research. It also explains a software based CPU architecture called NIOS-II. Chapter 3 describes an operating system uClinux and its architecture. Chapter 4 describes all phases of porting uClinux on an FPGA based hardware platform along with a boot loader that's used to boot uclinux. Chapter 5 shows some experimental work based on an application architectural design using uClinux based an FPGA platform for wireless sensor networks to monitor and control critically ill patients at home. Chapter 6 concludes the project by stating some directions for the future work.

CHAPTER 2
EMBEDDED SYSTEMS

An embedded system is a device embedded in a larger system, enabling the end system to interact with the physical world. They are designed as low cost and reliable autonomous systems for special purpose. Compared to desktop PC, both of them manage the interface between application programs and the hardware, but the interface between the user and the system is totally different. Usually embedded devices have a minimum user interface, which could only include a single button, a touch screen or no interface at all. They usually act as stand-alone devices that are designed for that particular task which could be from mobile phone to a controller between the industrial machine and network.

2.1 Embedded systems today

Today embedded systems control lots of different tasks deploying more powerful and complex devices. They are integrated and plugged in many kinds of machines, but also taking a stand alone role as they have been released in devices like PDA's, mobile phones and MP3 players. Many times these devices affect in our everyday life in forms which are totally transparent to the end user. One major sector of embedded devices from the beginning has been the devices in industry. With an external controller an industry machine with a proper interface can easily be connected to local area network or to internet without making any modifications to the system it self [37]. The embedded controller acts as a modem like device to the end system. The collected information can then be displayed via a table PC in the main controller room and the necessary settings to the end machine can be done from there. More advanced embedded devices can also include analog or digital inputs and outputs for controlling and process the collected information before it is forwarded on. The embedded systems can roughly be divided into high-end embedded systems and deeply embedded systems.

- The high-end embedded system classification is used when a general purpose OS is stripped down and left with specific modules for specific purpose. As the different parts of embedded solutions are becoming less expensive, more and more devices classified to this

category. Examples of these devices are router, personal digital assistant; PDA and today's mobile phones.

- Deeply embedded systems are designed for particular application. They need to be very compact, integrated with only few basic functions. Many times these are designed with a minimal operating systems and hardware layout designed for the specific purpose. One key word for these kinds of devices is transparency as their functions is invisible to the end user. Examples of these devices are small controllers and devices in our every day life like microwave, where they are embedded in.

2.2 Embedded computing

As components like processor and memory has become less expensive are being embedded devices integrated in several different kinds of devices. This also means that we are able to use more processing power and memory on these devices, so they are becoming much more complex, providing more functionalities. The different resources running on the device requires to be controlled by more intelligent operating system. In many simple 8-bit controllers, the tasks can be handled in a simple server loop, just by polling the different interfaces on the device. A more intelligent device requires more intelligent operating system to manage all the several functions that they provide. The operating systems for these devices are many times derivates from a general purpose operating system, so the barrier between embedded and general purpose operating system can be inconsistent [37].
Usually the operating system (OS) is supposed to fulfill the tasks of providing an extension to the hardware. This means that it provides layering by providing the lower level drivers for the physical chips and an application programming interface (API) for the programmer. Operating system also provides a resource management for different applications running on that device by handling interrupts and allocating memory for applications. It also provides an advantage to the programmer, as it supports some kind of protection against programming failures for low level devices and gives a foundation where programs execute.

2.2.1 Different solutions

The main difference with general purpose operating system and embedded operating system is that the footprint of the embedded version should be only a part of the general purpose OS. The difference can be seen when exploring the systems mission. The limitation of memory, processor and interfaces limit the tasks that a real embedded computer can attempt. As the system operates on a narrow, pre-designed area it provides tasks that are useful for that purpose. All complex embedded devices are designed to be beneficial solutions for specific tasks. The barrier between general purpose and embedded operating system is nowadays becoming more and more invisible, as the amount of memory and process power is increasing, which also allows the systems mission to be crown.

Several companies and communities have developed operating system specially designed for these embedded devices. Most of these solutions have chosen UNIX –like approach to develop the system, as it holds clear solution for small simple device often without almost any external interfaces. As the market of embedded devices are growing a substantial efforts from major companies have been made to enter these markets, also with non Open Source distributions. All of the embedded system can be inspected with the following different blocks.

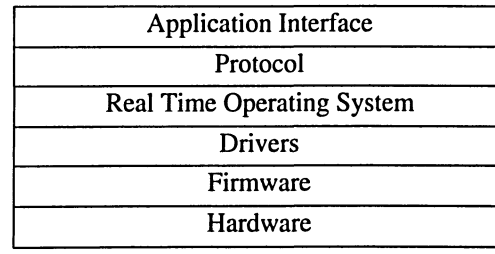

Figure 2.1: Embedded System Structure

The Linux is the most popular UNIX-like open source operating system used today. The advantages that Linux provides are that it is royalty free solution with easy configuration and strong networking support. It has also been ported to run on various embedded device and there

exists many companies providing commercial and non-commercial embedded Linux solutions. When obtaining an embedded Linux distribution it usually includes a tool chain, some ported applications, ported libraries and of course the actual kernel. The tool chain enables you to build applications or to create the image for an embedded device. Most of these are based on a standard GNU tool chain, but there also exists some packages that require proprietary tools. The applications included to the package are a set of ported applications designed specially for the solution that can be compiled to image.

The biggest problem with Linux in commercial usage is that the general public license (GPL) rules forces companies to keep the source code available to everyone, including the competitors. The license still allows applications and device drivers to be private if they remain separate from the Linux kernel and do not contain any other parts from programs developed under the general public license (GPL). Some companies providing Linux distributions have even provided specially designed tools to check which parts of the software have general purpose license (GPL) violations. One example of these companies is Lineo's general public license (GPL) Compliance Tool. More problematic license in embedded devices is lesser general public license (LGPL). In embedded device the whole application set is build in a single compact executable. To save space, the package is in many cases linked statically with the libraries. This means that the source code has to be available to all parts.

By dropping unnecessary modules and drivers from the general purpose Linux kernel, it can be compiled in the size of ~800Kb. Most embedded Linux solutions are using this approach, but when adding some real-time support or supporting MMU-less solution, additional patching is required. The open source efforts against embedded Linux systems can be inspected from different angles depending how they are developed. One approach is to eliminate the unnecessary functionalities that are not required by the embedded device. One example of this stripped, small footprint operating system is uClinux. As the Linux it self can be compiled to rather small package as it is, the standard Linux kernel itself can be patched to meet of the demands of the embedded device. Main advantage of this system is that the Linux applications can be ported easily to the platform. Examples of this kind of solution are AXIS and BlueCat Linux.

Embedded Linux is also distributed by some bigger commercial companies like Lineo, MontaVista and Red Hat. The main key behind these companies is that they provide support and professional services, a thorough documentation of their software development kits, maybe some specialized tools and systemization services. Some of these vendors, like Monta Vista provides a real-time extension to the kernel which enables the embedded device to have real-time performance. Two embedded solutions uClinux and ELKS are also focused on MMU-less processors. From these the uClinux provides an advantage by offering an active developer community and providing free support. It has also been ported to several different processor's and development boards. The uClinux kernel and tools are also totally free so for embedded solution it does not create any extra costs.

2.2.2 Managing embedded devices

Most common physical interfaces in embedded devices are serial, parallel, A/D, D/A and ethernet interface. These are also the only interfaces which allow physical management to be done to the device without any user interfaces like display or keyboard. The serial communication can be used to create the end connection to the controlled machine, but at the development stage with a proper serial communication program it can also be used to connect to the embedded device. Here the host machine works as a terminal emulator for the embedded device. Serial is also often used to move the systems image to the device or to set all the basic configuration to the board. The USB has nowadays also taken a place of connectivity interface to the end device, but in embedded devices it is not so common.

The physical ethernet with the connectivity application enables the device to be connected to the network. The TCP/IP's OSI model layer 4 has two protocols UDP and TCP providing the boundary between the user applications and host-layer protocols. Embedded controllers might provide several of different applications that can be used to connect and control the devices. Chapters 2 and 3 introduce the most used networking programs provided by the uClinux platform.

Client-server programs in embedded devices are usually run as servers or daemons, to enable the connection to be made to the device. Two most known and used program in this category are Terminal Emulation, Telnet and Secure Shell (SSH). Telnet provides the ability to remotely access an embedded device. The telnet daemon runs on the embedded device allowing transmitting the keystrokes from the remote host to the target embedded device and displays the resulting screen back to the remote host. SSH is a more secure version for remote connectivity [2]. The idea is the same as in telnet, but provides secure encrypted communications between the devices. For each new connection a new daemon is created which handles the exchange, encryption, authentication, command execution, and data exchange.

The File transfer protocol (FTP) is designed for file up- and downloading. In embedded devices it can often be used in both directions by running both a daemon and a client in an embedded device [2]. This way we can easily place files in an embedded system and also send for example log information back to the host machine. The Simple Mail Transfer Protocol (SMTP) is usually in embedded device implemented as client software. It allows mail to be sent to the remote host when an event triggers it. Usually it is used for sending scheduled information to a known host.

The HTTP can be used as a web configurator, which makes it a powerful tool. The GUI of the HTTP page can be used as a virtual interface to control the device itself and the end system connected to it. Usually this enables only the basic set configuration to be done, but still this is extremely handy in embedded Linux as it regularly provides only a console based connection used with the predefined set of commands. The simple network message protocol (SNMP) is based on asynchronous request / response commands. It is a widely accepted protocol, providing the management of different types of networks with a simple design that causes only a small burden to the network. It also has an extensive range of tool support. The agent of a simple network message protocol (SNMP) executes as a server in each of the monitored or managed device. The simple network message protocol (SNMP) provides an ability to easily manage all the devices distributed to the field by running special management software running on the management station. The agent on the embedded device responds to the control stations query or setting and acts on it. The protocol also enables the embedded device to act as a proxy toward the end system by holding a specially designed management information base (MIB) for that task.

The agent end is also able to send asynchronous traps to the end station when a predefined event occurs.

2.3 Hardware

The basic architecture of embedded device is usually the same when it is designed to provide network connectivity to the end system. They usually hold and serial interface to connect the embedded device to the end system, an ethernet plug for network connectivity and of course a Micro Controller Unit (MCU) core and some kind of system memory. The different hardware solutions usually concern the processor type, size and type of the memory and the interfaces that the device provides. When designing the hardware, the design goals can meet the customer markets by competing with better solution than the existing one or by getting the product to market faster than the competitor.

The best substitute for devices without a hard drive is flash memory which can be designed to emulate a drive. Flash also gives an advantage of being less power consuming, faster and space efficient than standard storages. The processors in embedded devices have two major alignments with and without memory management unit (MMU). The advantages gained from the memory management unit (MMU) are that it provides memory protection against the applications, but with precautions it is possible to run an MMU-less version on a smaller device for compact embedded applications. Of course it also gives a cheaper solution for designing the embedded controller.

The features that are required from the embedded devices are a possibility for long term autonomy, cost efficiency, low power consumption and general reliability. The device is to achieve reliability from the software and hardware. They are used to sense the outside world and control the devices in house surrounding so a failure in embedded device might mean that the controlled device is damaged. This requires a thorough testing of the hardware and the software. The cost efficiency is also a key feature, as the embedded devices are usually distributed and a small change in the price affects significantly to the purchasing. The target embedded system developed in this project is based on NIOS-II soft CPU core from Altera.

2.4 NIOS-II Stratix Development Board

The Nios-II development Kit, startix professional edition [35] as shown in figure 2.2 includes the following features:

- Stratix EP1S40F780 FPGA device
- MAX® EPM7128AE CPLD configuration control logic
- SRAM (1 Mbyte in two banks of 512 Kbytes, 16-bit wide)
- SDR SDRAM (16 Mbytes, 32-bit wide)
- Flash (8 Mbytes)
- CompactFlash connector header for Type I CompactFlash cards (40 available user I/O pins)
- 10/100 Ethernet physical layer/media access control (PHY/MAC)
- Ethernet connector (RJ-45)
- Two serial connectors (RS-232 DB9 port)
- Two 5-V-tolerant expansion/prototype headers (2 x 41 available user I/O pins)
- Two JTAG connectors
- 50-MHz crystal (socket), external clock input
- Mictor connector for debugging
- Four user-defined push-button switches
- Eight user-defined LEDs
- Dual 7-segment LED display
- Power-on reset circuitry

Stratix devices contain a two-dimensional row- and column-based architecture to implement custom logic. A series of column and row interconnects of varying length and speed provides signal interconnects between logic array blocks (LABs), memory block structures, and DSP blocks [35]. The logic array consists of LABs, with 10 logic elements (LEs) in each LAB. An LE is a small unit of logic providing efficient implementation of user logic functions. LABs are grouped into rows and columns across the device.

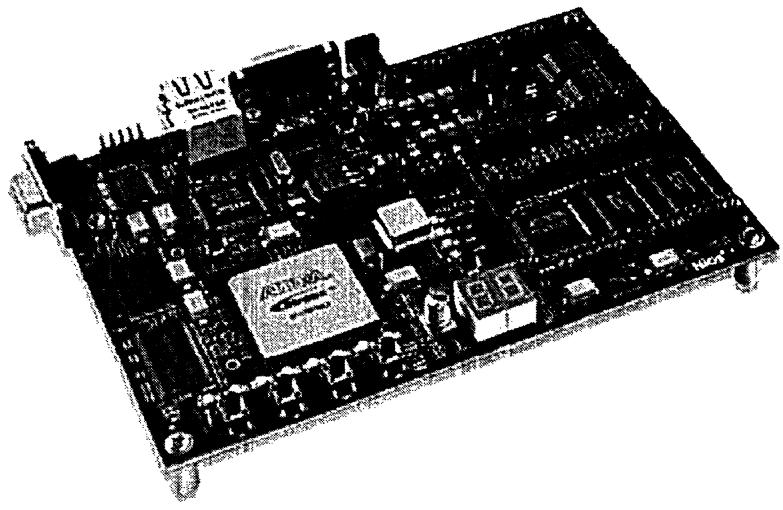

Figure 2.2: NIOS-II Development Board (Reproduced from Ref. [35])

M512 RAM blocks are simple dual-port memory blocks with 512 bits plus parity (576 bits). These blocks provide dedicated simple dual-port or single-port memory up to 18-bits wide at up to 318 MHz. M512 blocks are grouped into columns across the device in between certain LABs. M4K RAM blocks are true dual-port memory blocks with 4K bits plus parity (4,608 bits). These blocks provide dedicated true dual-port, simple dual-port, or single-port memory up to 36-bits wide at up to 291 MHz. These blocks are grouped into columns across the device in between certain LABs [35]. M-RAM blocks are true dual-port memory blocks with 512K bits plus parity (589,824 bits). These blocks provide dedicated true dual-port, simple dual-port, or single-port memory up to 144-bits wide at up to 269 MHz. Several M-RAM blocks are located individually or in pairs within the device's logic array.

Digital signal processing (DSP) blocks can implement up to either eight full-precision 9 × 9-bit multipliers, four full-precision 18 × 18-bit multipliers, or one full-precision 36 × 36-bit multiplier with add or subtract features. These blocks also contain 18-bit input shift registers for digital signal processing applications, including FIR and infinite impulse response (IIR) filters. DSP blocks are grouped into two columns in each device [35]

Each Stratix device I/O pin is fed by an I/O element (IOE) located at the end of LAB rows and columns around the periphery of the device. I/O pins support numerous single-ended and differential I/O standards.

Each IOE contains a bidirectional I/O buffer and six registers for registering input, output, and output-enable signals [35]. When used with dedicated clocks, these registers provide exceptional performance and interface support with external memory devices such as DDR SDRAM, FCRAM, ZBT, and QDR SRAM devices. High-speed serial interface channels support transfers at up to 840 Mbps using LVDS, LVPECL, 3.3-V PCML, or Hyper Transport technology I/O standards. The number of M512 RAM, M4K RAM, and DSP blocks varies by device along with row and column numbers and M-RAM blocks [35]

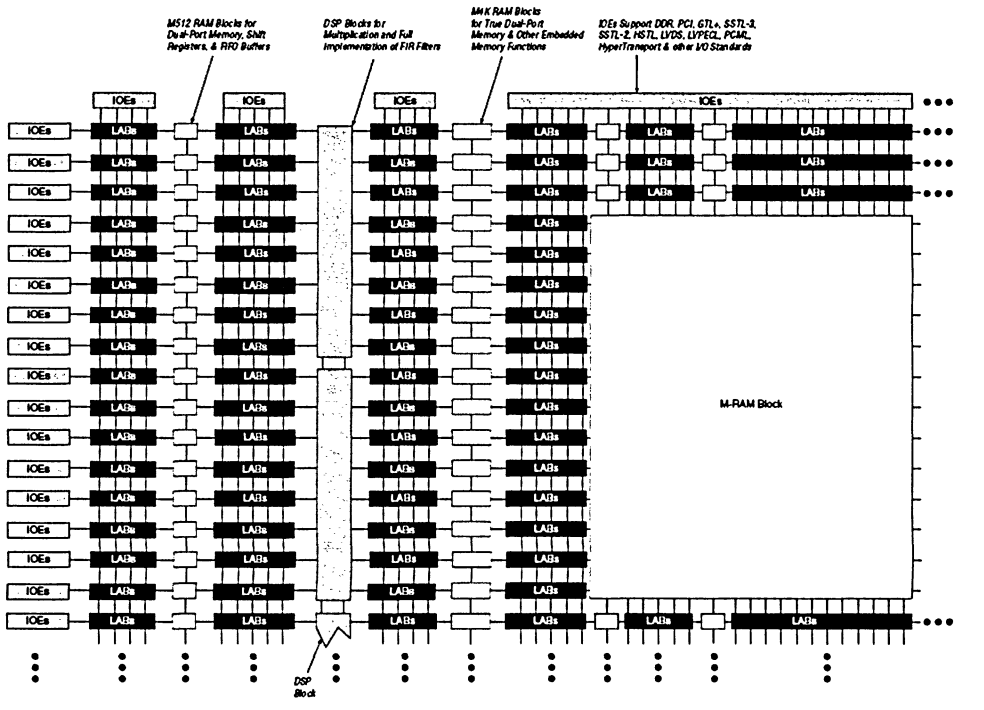

Figure: 2.3: An overview of stratix device (Reproduced from Ref. [35])

The nios-II development board provides a hardware platform for developing embedded systems. The board features s stratix EP1S40F780C5 device with 41,250 logic elements (LEs) and 3,423,744 bits of on-chip memory. It comes pre-programmed with a nios-II processor reference design shown in figure 2.4.

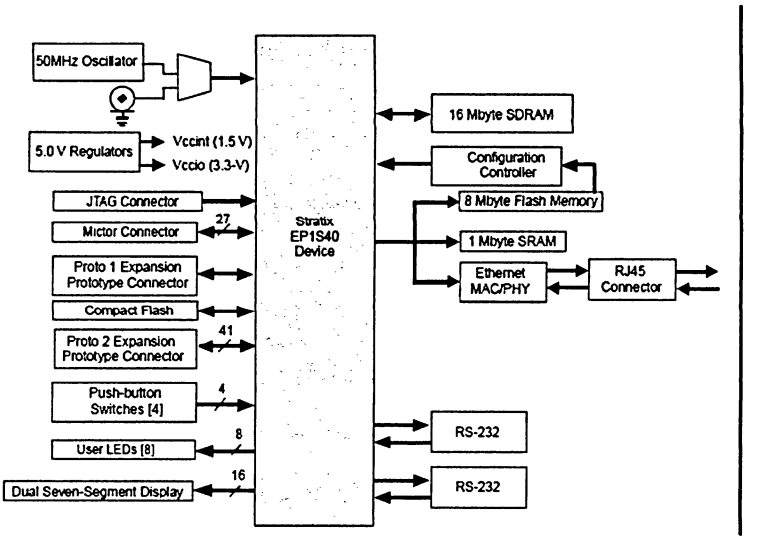

Figure: 2.4: Stratix EP1S40 Block diagram (Reproduced from Ref. [35])

Table 2.1 lists the Stratix device features [35].

Table 2.1: Stratix 1S40 device features

Logic Elements	**41,250**
M512 RAM blocks (32 x 18 bits)	**384**
M4K RAM blocks (128 x 36)	**183**
M-RAM blocks (4K x 144 bits)	**4**
Total RAM bits	**3,423,744**
DSP blocks	**14**
Embedded multipliers	**112**
PLLs	**12**
Maximum user I/O pins	**822**

The development board provides two separate methods for configuring the Stratix device.

- Using the Quartus II software running on a host computer, a designer configures the device directly via an Altera download cable connected to the Stratix JTAG header (J24).

- When power is applied to the board, a configuration controller device (U3) attempts to configure the Stratix device with hardware configuration data stored in flash memory [35].

2.5 NIOS-II Processor

The nios II processor is a general-purpose RISC processor core, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- 32 external interrupt sources
- Single-instruction 32 × 32 multiply and divide producing a 32-bit result
- Dedicated instructions for computing 64-bit and 128-bit products of multiplication
- Single-instruction barrel shifter access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals
- Hardware-assisted debug module enabling processor start, stop step and trace under integrated development environment (IDE) control software development environment based on the GNU C/C++ tool chain and Eclipse IDE
- Instruction set architecture (ISA) compatible across all Nios II processor systems performance beyond 150 DMIPS

A Nios II processor system is equivalent to a microcontroller or "computer on a chip" that includes a CPU and a combination of peripherals and memory on a single chip. The term "Nios II processor system" refers to a Nios II processor core, a set of on-chip peripherals, onchip memory, and interfaces to off-chip memory, all implemented on a single Altera chip as shown in figure 2.5. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model [35].

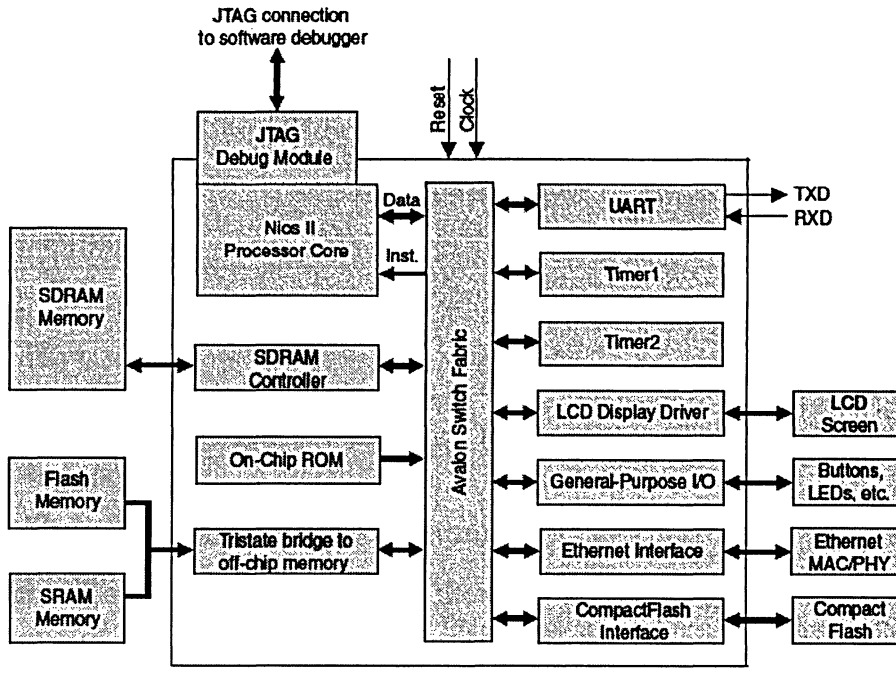

Figure 2.5: Example of a Nios II Processor System (Reproduced from Ref. [35])

The Nios II architecture describes an instruction set architecture (ISA). The ISA in turn necessitates a set of functional units that implement the instructions. A Nios II processor core is a hardware design that implements the Nios II instruction set and supports the functional units described in this document. The processor core does not include peripherals or the connection logic to the outside world. It includes only the circuits required to implement the Nios II architecture. Figure 2.6 shows a block diagram of the Nios II processor core [35].

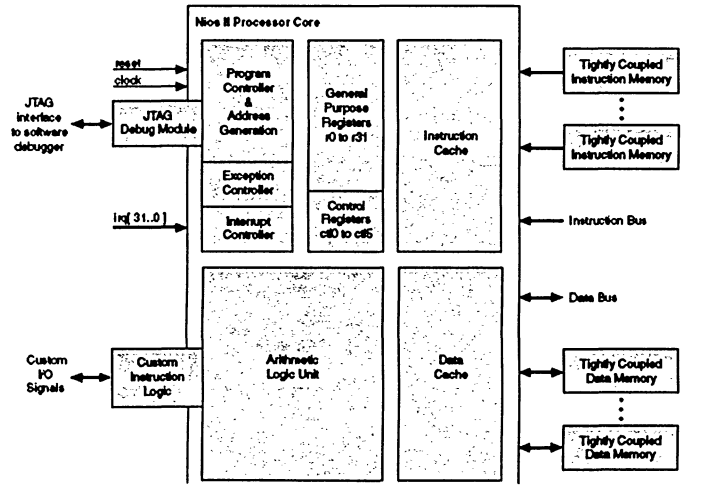

Figure 2.6: Nios II processor core block diagram (Reproduced from Ref. [36])

The Nios II architecture defines the following user-visible functional units:

- Register file
- Arithmetic logic unit
- Interface to custom instruction logic
- Exception controller
- Interrupt controller
- Instruction bus
- Data bus
- Instruction and data cache memories
- Tightly coupled memory interfaces for instructions and data
- JTAG debug module

The functional units of the Nios II architecture form the foundation for the Nios II instruction set. However, this does not indicate that any unit is implemented in hardware. The Nios II architecture describes an instruction set, not a particular hardware implementation. A functional unit can be implemented in hardware, emulated in software, or omitted entirely [35].

CHAPTER 3
uCLINUX (MICRO-CONTROLLER LINUX)

Pronounced "you-see-linux", the name uClinux comes from combining the greek letter "mu" and the english capital "C". "Mu" stands for "micro", and the "C" is for "controller".

Micro-Controller Linux (uClinux) is an open source project that adds support to Linux that enable it to run on microprocessors without Memory Management Units (MMU).These types of processors have traditionally made up the bulk of processors used in embedded systems [20].
This research project will cover the basic architecture of uClinux, and in particular the design and code changes required to deal with not having any memory management. The kernel, device driver, library and application level changes will be discussed and explained.

3.1 History of uClinux

uClinux project was started in 1997 with a goal to derive a version of Linux kernel 2.0 for low cost micro controllers. It was Jeff Dionne, Kenneth Albanowski and a group of other developers who discussed about this possibility to embed the Linux into a Memory Management Unit less network controllers that could handle the communication between the network and communication system [20]. The first release of this small footprint operating system was released with Motorola 68000 processor, which was based on MC68328 DragonBall Integrated Microprocessor that was deployed in a SCADA controller in 1997 / 98. The first release for public open source community was released as an alternative operating system for Palm Pilot in February 1998 [20].

Jeff Dionne and Michael Durrant from Lineo started to design and build a line of embedded controllers known as uCsimm and uCdimm. At the same time Gerg Ungerer from the same company ported uClinux onto Motorola's ColdFire platform and designed several systems with this base platform. The early focus of a cross-platform development of uClinux soon led to port it for other platforms. The interest for these small processors was growing rapidly and led to a

number of other software development. One of these was uC-libc library which was designed to replace Linux libc and glibc libraries into a tiny package. Other improvements were done by SnapGear by adding the binary flat format, bFLT support and by RidgeRun with ELF shared library.

Originally the uClinux development was based on Linux kernel version 2.0.33. The kernel release 2.2 did have only minor changes affecting to MMU-less devices. It turned out that the drivers for the MMU-less design from this version could rather easily be ported to version 2.0 and the needed changes could be done in this way. In the late year of 2000 the Linux kernel 2.4 was released and the changes made to this revision were major enough to port it to MMU-less platform. Nowadays most of the development is done with the 2.4/2.6 kernel, but there still is a strong interest to the code based on version 2.0 and changes are also made to this uClinux tree.

uClinux systems are true Linux systems though, the kernel support for running without an MMU is in an add on to Linux, it is not a different code base. So the uClinux kernel support is no more than a patch against standard Linux kernel sources [28]. Although the micro-controller market contains 4bit to 64bit CPU architectures, uClinux is targeted at the classic 32bit (and even 64bit) microprocessors. There is no support for 16bit or less CPU's [20].

The differentiation between a micro-controller and a standard CPU is blurry [28]. A simplistic definition is any CPU that may be used in an embedded system could be considered a micro-controller. Better is any CPU that integrates a number of system peripherals with the CPU core is a micro-controller. Historically these types of CPUs are low cost or specialized for certain types of functions, and they are not as full featured as their real computer counterparts. They did not have features like memory management units. In recent years though the trend is to include MMUs, even on ultra low cost specialized CPU's. In any case uClinux is all about supporting CPUs that do not have MMUs [24].

Interestingly because uClinux is a set of additional patches for standard Linux sources all the existing CPU support for processors with MMUs is still present. The one kernel sources tree supports both processors with and without MMUs.

3.2 uClinux Architecture

The uClinux kernel is just the Linux kernel with support added for processors without MMUs. For the most part, you get the full Linux kernel feature set when you use uClinux. The Linux kernel API (in this case the system call set) is unchanged from standard Linux. Architecture implementation differences still apply, but in the same way as for all ports of Linux to non x86 architectures [22].

The uClinux system is fully multi-tasking, with the usual process and process control model. All file systems and related operations are identical. All networking support and even the device driver interfaces are unchanged for uClinux. uClinux even supports dynamic kernel loadable modules [20]. Obviously some changes are required to the memory management sub-system of the kernel. Outside of architecture support this is the bulk of the uClinux patch. There is no notion of virtual memory (VM), and no form of memory protection between processes, between the kernel and processes or hardware device register sets. That is a fact of life without an MMU [20].

Not withstanding the different memory subsystem, uClinux maintains the classic separation of user and kernel space. Each has its own stack, just as on a VM system, and if the hardware supports it the kernel maintains different privilege levels for each. Although clearly it doesn't mean much when there is no memory protection. Where hardware does not support privilege levels, or different mode stack pointers these are emulated in software.

A common question is whether uClinux needs less memory than a VM Linux system. In general the answer is no, however, most uClinux systems are small by design, keeping their setup to a minimum. Practical uClinux systems can be built in as small as 1MB of RAM [20].

3.3 Supported Architectures

The range of CPU architectures and specific CPUs that uClinux supports is truly amazing. At the very least the list is [21]:

- Motorola 68k family (68x302, 68306, 68x328, 68332, 68360)
- Motorola ColdFire (5206, 5206e, 5249, 5272, 5282, 5307, 5407)
- ARM (silicon from Atmel, NetSilicon, Aplio, TI, Samsung, Conexant, etc.)
- Intel i960
- Sparc LEON
- MIPS (Brecis, ...)
- NEC v850 family
- Hitachi H8/300
- Xilinx Microblaze (FPGA Soft core CPU)
- Altera NIOS and NIOS-II (FPGA Soft core CPU)
- AXIS ETRAX
- Analog Devices Blackfin

There is more in development includeing [21]:

- Hitachi Super SH2
- Motorola MCORE
- OpenCORES OpenRISC (FPGA)

There is probably more, the uClinux community is very active!

3.4 uClinux Kernel Internals

The key difference between standard Linux and uClinux is the lack of hardware assisted memory management support. That implies no on demand loading, and that applications must wholly fit in RAM (or at least RAM and flash/ROM if executing in place). No current uClinux systems support swapping to any form of secondary storage [25].

The underlying memory allocation system of Linux is used "as is". The management of free and used areas of memory can be identical, it does not matter that virtual page mappings exist on top of used memory or not. The only change in this area is to allow the Linux allocator to keep regions of larger sizes available for allocation. When a memory allocation is requested in

uClinux the kernel allocator needs to find a single contiguous chunk of RAM big enough to satisfy the request. It is not possible to virtually map a set of pages together to construct a larger region, so uClinux needs these larger allocation regions to satisfy large requests [25].

For the most part the virtual mapping support code is just stubbed out for uClinux. Virtual and physical addresses are treated identically. Most kernel data structures associated with virtual memory support are left intact, and the internal function interfaces left unchanged. The changes made within the 2.6 series kernels to support uClinux are clean and reasonably small, and demonstrate low overall impact by adding MMU-less [24].

There are some interesting side effects of not having virtual memory in other parts of the kernel. It is worth going over those here [25]:

- No easy way to implement real *fork()*
- No way to dynamically grow an applications stack
- No way to dynamically grow a heap (effects *sbrk()* system call)
- Memory fragmentation problems

Fork is more of problem that it would first seem. A true fork creates a mirror image of the current process memory space, and then each of the parent and child get to execute in their own memory space. What one does has no effect on the other. Problem is that we have no notion of a virtual address space and when applications are running in uClinux they are all sharing the same address space with each other (the kernel and usually peripheral devices as well). Absolute addresses are used when pointers are created or when call return addresses are pushed onto a stack. One cannot just copy the process memory image to another location, all these absolute addresses will now be wrong – pointing back into the parent's memory region. There is also no way to "fix" these absolute addresses as you copy, we just cannot tell what is really a pointer and what is random data.

For efficiency sake in uClinux we use the *vfork()* system call in place of *fork()*. With *vfork()* both parent and child share the memory region of the process. The semantics are that the child process

runs to either *exec()* or *exit()* completion, the parent sleeps until then and resumes normal scheduled execution. The child process must be extremely careful to leave the parent memory region in a consistent state. *vfork()* has been around for years, originating from BSD UNIX. The reasoning behind it was that most programs *fork()* then do an *exec()* soon after, effectively tearing down the copy of the memory space that was just copied in the *fork()*. Without virtual memory we have no page mapping and there is no way to set markers when the application stack becomes full. In uClinux fixed size stacks are allocated for each process at *exec()* time. The stack size is stored as part of the binary program file header and it can be set on a program by program basis to minimize the wasted memory.

Moreover, we cannot dynamically grow a process heap in the conventional way without page table mappings in place. There is no simple way to implement the convention *sbrk()* system call that grows the heap contiguously. It is straight forward to allocate more memory, just not easy to make it contiguous with the current heap allocation. It turns out that this is relatively easy to work around the library code. The trick is to use *mmap()* to allocate memory instead of *sbrk()*. Using *mmap()* means the kernel will keep track of the application allocated memory regions (which can be anywhere in the system address space) . When the process exists it is simple to walk the list of associated *mmap* regions and free them back to the kernel free memory pool.

Lastly memory fragmentation is generally more of a problem under uClinux. When the kernel or a process tries to allocate a chunk of memory it must be fulfilled with a single contiguous chunk; It means that a single region of the right size needs to be found – separate smaller pages cannot be virtually mapped together to form the desired region size.

3.5 Libraries

There is one good reason one would not use glibc in uClinux systems, it is rather large and in practice no one uses it [27]. The preferred library for use in uClinux systems is uClibc. It is a descendant of the original uClinux library uClibc which is a collection of lightweight, standards compliant, functions providing about 95% coverage of the glibc function set. As a general rule anything that compiles and works on glibc will compile and work on uClibc. uClibc can be used

on both MMU and MMU-less systems. uClibc can be used as a shared or static library, and offers many advanced features including threading.

Some uClinux supported architectures support shared libraries. Currently they are only supported on MC68k and ColdFire based systems. There has been at least one implementation of shared library support for ARM based uClinux system, but this has never been made available as GPL open source [27].

Fundamentally two changes need to be made to a C library to support uClinux. Firstly *vfork()* needs to be implemented, and secondly the *malloc()* family of functions needs to be changed to use *mmap()* as the system call to get and free memory. Generally these are simple tasks. Many other libraries have also been ported to uClinux, The list includes openssl, libpcap, zlib, libjpeg, libpng, and many others [27].

3.6 Applications

Applications are loaded and run the same way under uClinux as Linux. Applications are made up of the same fundamental parts in uClinux too and they each have a code portion (sometimes called the text segment) an initialized data section (often called the data segment), an un-initialized data section (called the bss) and a stack.

One notion that is supported on many uClinux target architectures is the ability to leave the code section of an application in fixed random access storage (e.g. Flash or ROM memory) and execute the instructions from that memory space. This is called "execute in place" (XIP) and can provide great memory savings. In this case, the entire code section of the program must be stored in one contiguous chunk. Not many filesystems actually do this. uClinux also supports the more typical notion of loading a programs code and data into RAM and executing it from there.

3.7 FLAT Files

uClinux uses a new application binary file format called the flat file. The reasoning for a new file format is two fold. Primarily to simplify the loading and executing process for an application.

Secondly a small and lightweight binary format to build a small footprint system. On a virtual memory system applications are absolutely linked to load and execute in their own virtual memory space. Addresses within the code and data are fixed in that virtual address space. Generally we don't have fixed addresses in uClinux. An application may be loaded and run anywhere in RAM, or when XIP at some location in flash/ROM. We will not know in advance the memory address the code actually resides in. There are basically two different methods used in uClinux to deal with the unknown address problem [29].

3.7.1. Relocation

Relocation entries are stored in the flat binary. When an application program is being loaded to execute, the kernel flat loader (binfmt_flat) patches the code and data with the relocations. It uses the addresses range allocated for this application as the relocation address. Obviously for this method an applications code must be loaded into RAM, it cannot be executed XIP in flash/ROM [29].

3.7.2. Position Independent Code (PIC)

During compiling the application, the compiler can generate position independent code that is code that has no absolute address references. It is not enough though to just have the code position independent as we also need position independent data section. This is typically achieved through the use of a global offset table, where a table of address offsets is created for each address and all the accesses are indexed through a base register.

PIC code often tends to be a little slower, due to the indirect access required. But it has the advantage of sharing of code regions and for XIP. Please note that every instance of a running application still has to have its own data segment and stack in RAM. The code segment can only be shared or left and used in place in the flash/ROM [29].

It would be fair to say that the PIC method is more popular in uClinux systems. However it cannot be supported by all the architectures, and it does require a compiler capable of generating

PIC code and data. Relocation is simpler to implement and it is often supported first on a new uClinux architecture port [29]. Relocation and PIC are not mutually exclusive; both can and often are supported on a system. The kernel loader can determine from the flat format file header whether the program can be run XIP or not.

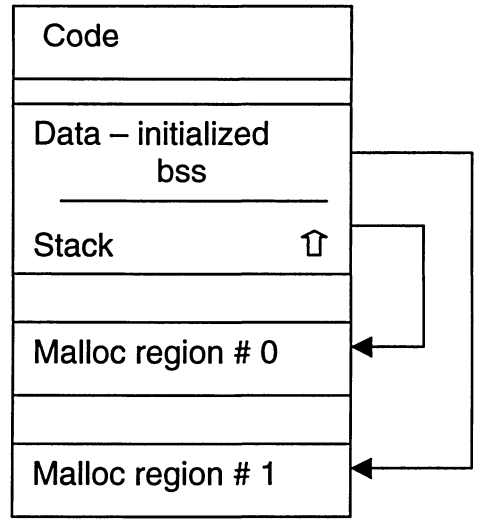

Figure 3.1: Processes memory mapping

Figure 3.1 shows a simplistic representation of what a processes memory mapping might look like. Note that typically the data and stack regions are allocated as a single chunk of memory and notice that this is dislocated from the programs code section. The code section may well be in flash/ROM or some other place in RAM, which is typical for XIP. It is also possible for a relocation load that the code, data and stack are allocated as a single chunk and contiguous [29]. Also note the malloced regions (that is what is conventially referred to as the heap) is allocated from whatever free memory the kernel has available.

Although it would be possible to support the executable and linking format (ELF) applications on uClinux it has never been done. It would require relocating the code and data at load time – unlike on a VM Linux kernel where it is already fully linked. So the Linux kernel ELF loader could not be used as it is. The other advantage of flat format files is that they are extremely small. The header is 40 bytes, and no padding is used within the file [29].

3.8 Application Ports

The great thing about building a system on top of standard Linux and preserving the API is that one can port any application to uClinux that exists for Linux. The set of ported applications for uClinux is simply huge. Here is a short list of ported application packages:

TOOLS/UTILITIES: sash shell, minix shell, busybox, tinylogin, agetty, python, vi (clone), tip

NETWORKING: net-tools, ping, ipfwadm/iptables, tftp, ftp, dhcpcd, traceroute, tcpdump, ssh, ntp, wget, iproute2

SERVERS: init, inetd, pppd, pptpd, diald, boa (web), telnetd, tftpd, ftpd,dhcpd, samba, squid, snmpd, zebra, Freeswan (IPsec), dnsmasq, gdbserver, sshd

FILESYSTEM: mount/umount (including NFS), smbmount/smbumount, e2fsprogs, fdisk, reiserfs tools,

MISCELLANEOUS: mp3play, microwindows, mtd-utils, netflash, hotplug tools

This is but a sampling of the packages ported. The uClinux-dist distribution contains over 150 application packages currently that can run on uClinux.

3.9 Tools

Like any other Linux system, uClinux systems are built using the standard GNU tools. Exact versions vary between architectures but currently many of the main-stream stable targets are using [26]:

- binutils-2.14
- gcc-2.95.3
- gdb-5.0

On many targets the uClinux community has patched these tools to improve position independent code and data support, and support for shared libraries. This is certainly true for MC68k and ARM tool chains used for uClinux. Moves are under way to update to more recent

gcc versions (specifically 3.3) and to integrate many of the uClinux specific patches back into the gcc source base [26].

The GNU debugger (GDB) and its simulator capabilities can be used to run uClinux for some architecture. For example the ARMulator simulator extension of GDB can run uClinux in its own right. Makes a great development tool to get up to speed on uClinux on ARM platforms, or to develop without real hardware.

Now a day many of the embedded processors contain jtag, bdm or on-chip debug modules. Generally these can be driven by simple hardware dongles to parallel ports or similar on a PC. Many are supported through servers or with patches by GDB. Many offer advanced debug features like the ability to start and stop the CPU, set break points, dump and change memory. Many also allow programming flash memory in the circuit. All these features make debug easier on these embedded platforms. GDB can also be used to debug uClinux applications. Normally this is done via a network debug arrangement, running the gdbserver stub on the uClinux target system [26].

Another of the key tools required for uClinux development is the *elf2flt* converter. *Elf2flt* converts a uClinux application that has been compiled as an ELF format object (as is normally done) to a uClinux flat format file. The conversion is actually reasonably strait forward. For those unfamiliar with developing for deeply embedded targets the usual setup is to cross compile for target from a host development PC. This is true for uClinux, where the target system is almost never used as the development system. Most developers choose a Linux PC as their development system. It has been done on PowerPC based laptops as well. And for the truly disturbed one can even develop uClinux systems (compiling from source and all) under Windows using Cygwin [26].

CHAPTER 4
PORTING uCLINUX ON NIOS-II DEVELOPMENT BOARD

4.1 Development Tools

Installing development tools to cross-compile the uClinux kernel for the Nios-II development board is necessary. Cross compiling is the act of building source code on one system, the build host, into executables or libraries to be executed on a different host, the *native host*. The build host and native host may differ in operating system and/or processor type. Cross-compiling is a relatively new thing if one compares it to the history of computers and software [26].

These tools provide a foundation for the development and can be obtained as a pre-build binary tool package or compiled by the developer by patching the GNU tools and then compiling them. The whole tool chain includes the following components is installed under /opt/nios2_toolchain directory:

binutils -	a collection of binary tools (*ld*, *as*, etc.). Based on the GNU binutils-2.10.
gcc -	C/C++ compiler. Based on GNU gcc-2.95.3.
elf2flt -	An elf to flat converter.

All the tools have to be patched and configured in order to get them to work with the uClinux kernel. An important issue in developing code is the ability to track down the instances that might effect on programs execution. Under the uClinux, two different kinds of debugging is required, one for kernel source code and the other for user applications. In this way, uClinux provides the ability to use GNU debugger (GDB), which allows user to debug programs written in C/C++ and with some other languages. Apart from cross compiler tool chain we need a hardware design file for Nios-II development board as well as utility programs to download and debug uClinux kernel [26].

4.2 Building a uCLinux Kernel Image

A typical uCLinux kernel can be built with the following steps:

1. The kernel source file (for example, linux 2.6.11.tar.bz2) can be unzipped if necessary using the following bunzip2 command:
bunzip2 linux-2.6.11.tar.bz2

2. The source code is extracted from the tar image using the tar command. An example follows:
tar –xvf linux-2.6.11.tar

3. Create a new directory (for example, /usr/src/linux-2-6-11) and copy the extracted source files to it.

4. The new kernel is now ready to be built. Drill down into the linux-2-6-11 directory by typing the following: cd /usr/src/linux-2-6-11

5. Download the uclinux and nios2 patch files from www.niosforum.org and patch the linux-2.6.11 stock kernel to a Nios-II kernel.

6. uClinux kernel compilation steps:

make ARCH=nios2nommu CROSS_COMPILE=nios2-linux-uclibc- hwselect SYSPTF=std_1s40.ptf

make ARCH=nios2nommu CROSS_COMPILE=nios2-linux-uclibc-

make ARCH=nios2nommu CROSS_COMPILE=nios2-linux-uclibc- menuconfig

Configure the Linux kernel to be as large or small as required, depending on the number of drivers and support functions that are needed on the final application.

7. Upload the uClinux kernel:

nios2-download -g vmlinux

The –g option starts after uploading.

4.3 Booting the uCLinux kernel

With storage cases of flash and RAM, the processor is able to address directly to bits stored in them. In a simplest case the execution of a uClinux kernel is achieved by placing the startup code to the flash in to the processors startup address. With these kind of setting the uClinux kernel is in charge of doing the hardware setup and placing the necessary segments into the RAM.

A more safe and flexible option is to place a small stand alone piece of code called a boot loader, to the start offset of the flash. The boot loader can handle the initial settings of the board including the basic hardware setup and allows downloading the image to the board via serial or ethernet. It can also handle some environment settings, which enables the user to write some simple configurations without needing a writable file system for the Flash.

The uClinux can be used with different boot loaders for specialized tasks in different stages of the development. Some of the boot loaders for uClinux are CoLilo, My Right Boot, Motorola's dBug and U-boot, which is used with the Nios-II platform. The Universal boot, U-boot enables the loading of image or images through a serial or ethernet, with the Trivial File Transfer Protocol (TFTP) protocol. It also enables the usage of environment variables, booting of compressed and decompressed images and booting kernel from a JFFS2 partition.

Das U-Boot is a GPL'ed cross-platform boot loader shepherded by project leader Wolfgang Denk and backed by an active developer and user community. U-Boot provides out-of-the-box support for hundreds of embedded boards and a wide variety of CPUs including PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze and x86. One can easily configure U-Boot to strike the right balance between a rich feature set and a small binary footprint. U-Boot has its origins in the 8xxROM project, a boot loader for 8xx PowerPC systems by Magnus Damm. When bringing that project to Sourceforge in 2000 the current project leader, Wolfgang Denk, renamed the project PPCBoot since Sourceforge did not allow project names to begin with a digit.

The openness and utility of PPCBoot fanned the flames of its popularity, driving developers to port PPCBoot to new architectures. By September 2002 PPCBoot supported four different ARM

processors and the name PPCBoot was becoming quaint. In November 2002 the PPCBoot team retired the project, which led directly to the surfacing of "Das U-Boot".

4.4 Building Applications

When the kernel is complete and the root file system is configured, applications can be built to run on the NIOS-II development board. These applications could range from telnet to the apache web server. The development tools containing the NIOS-II cross compiler usually contain application source files that can be built for a NIOS-II target. A couple of these software packages are discussed below. Note that when the applications are built, the system must be configured to run them correctly.

4.4.1 BusyBox

BusyBox provides basic utilities for embedded systems and is ideal for providing minimalist replacements for most of the common utilities one would have on the desktop system (for example, sh, tar, ls, etc.). It provides a good starting point for developing an effective and efficient embedded Linux file system.

The following steps show an example of building BusyBox for the NIOS-II system:

1. Download the BusyBox source code from http://www.busybox.net/
2. Extract the BusyBox files using the tar command.
3. Modify the config.h file to define or undefined the applications to make new line during the build.
4. If necessary, change the CROSS field in the main BusyBox Makefile to point to the NIOS-II cross compiler.
5. Type **make clean**
6. Type **make install**

7. The utilities for the system are now built and can be copied from the _install directory of BusyBox to the location of the root file system to be mounted on the target board (e.g., /opt/nios2development/).

4.4.2 Boa Web Server

A computer that delivers web pages is a *web server*. Every web server has an IP address and possibly a domain name. Installing server software and connecting the machine to the Internet can turn any computer into a Web server.

Boa is a single-task HTTP server. Unlike traditional web servers, it does not fork for each incoming connection, and it does not fork many copies of itself to handle multiple connections. Boa internally multiplexes all of the ongoing HTTP connections and forks only for common gateway interface (CGI) programs, which must be separate processes.

The following steps show how to build Boa for the NIOS-II system:

1. Download the Boa source code from http://www.boa.org/
2. Extract the Boa files using the tar command.
3. Type **make clean**.
4. Type **./configure**
5. If necessary, change the CC field in the main Boa Makefile to point to the NIOS-II cross compiler.
6. Type **make all**.

A Boa executable file should be located in the main Boa directory. Copy this file and relevant other files to the root file system for the target.

4.5 Deployment of uCLinux

The Linux version used on the host was Debian Linux r3.0. The serial port (COM1) was configured to operate at 115200 baud rate with no parity or stop bits and 8 data bits using the Kermit application. The serial port of the host Linux PC was connected to the terminal port (RS232) of the NIOS-II development board using a standard 9-way connector.

After building the uClinux kernel image we used a utility program **mkimage** that comes with U-Boot to add a tiny header containing the load and execute address for the image as given below (all on one line):

> **./mkimage** -A nios2 -O linux -T kernel -C gzip -a 0x01000000 -e 0x01000000 -n "Nios-II Linux Kernel Image" -d vmlinux.gz uImage

This command appends a small header containing the load and executes address 0x01000000 to the kernel image and creates a new file called uImage. The header also contains a CRC32 checksum, checked later during the image load.

The following output shows the uClinux kernel being downloaded and executed using U-Boot loader.

```
DK1S40 ==> tftpboot 1100000 uImage
Using MAC Address 00:07:FFFFFFED:0C:04:FFFFFFBD
TFTP from server 10.0.0.1; our IP address is 10.0.0.51
Filename 'uImage'.
Load address: 0x1100000
Loading: ##################################################################
     ##################################################################
     ##########
done
Bytes transferred = 715524 (aeb04 hex)

DK1S40 ==> bootm 1100000
```

```
## Booting image at 01100000 ...
   Image Name:   Linux Kernel Image
   Image Type:   Nios-II Linux Kernel Image (gzip compressed)
   Data Size:    715460 Bytes = 698.7 kB
   Load Address: 01000000
   Entry Point:  01000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Linux version 2.6.11-uc0-barco1 (root@debian-dell-p3) (gcc version 3.4.3 (Barco Control
Rooms)) #2 Sat Jul 2 11:40:34 EDT 2005
uClinux/Nios II
Altera Nios II support (C) 2004 Microtronix Datacom Ltd.
Built 1 zonelists
Kernel command line: CONSOLE=/dev/ttyS0 noinitrd ip=bootp root=/dev/nfs rw
PID hash table entries: 128 (order: 7, 2048 bytes)
Console: colour dummy device 80x25
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory available: 14464k/16384k RAM, 0k/0k ROM (1524k kernel code, 215k data)
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
NET: Registered protocol family 16
devfs: 2004-01-31 Richard Gooch (rgooch@atnf.csiro.au)
devfs: devfs_debug: 0x0
devfs: boot_options: 0x1
NIOS serial driver version 0.0
ttyS0 (irq = 4) is a builtin NIOS UART
io scheduler noop registered
io scheduler cfq registered
smc_probe: 50000 Khz Nios
SMSC LAN91C111 Driver (v2.1), (Linux Kernel 2.6)
```

```
eth0: SMC91C11xFD(rev:1) at 0x82110300 IRQ:6 MEMSIZE:8192b NOWAIT:0 ADDR:
00:07:ed:0c:04:bd
smc_probe: 50000 Khz Nios
NET: Registered protocol family 2
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
NET: Registered protocol family 1
NET: Registered protocol family 17
eth0:PHY 100BaseT
eth0:PHY Full Duplex
Sending BOOTP requests . OK
IP-Config: Got BOOTP answer from 10.0.0.1, my address is 10.0.0.51
IP-Config: Complete:
     device=eth0, addr=10.0.0.51, mask=255.0.0.0, gw=10.0.0.1,
    host=nios2, domain=nios2.net, nis-domain=(none),
    bootserver=10.0.0.1, rootserver=10.0.0.1, rootpath=/opt/nios2development
Looking up port of RPC 100003/2 on 10.0.0.1
Looking up port of RPC 100005/1 on 10.0.0.1
VFS: Mounted root (nfs filesystem).
Mounted devfs on /dev
Freeing unused kernel memory: 68k freed (0x1190000 - 0x11a0000)
init started:  BusyBox v1.00-pre8 (2005.08.19-23:48+0000) multi-call binary
Bummer, could not run '/etc/init.d/rcS': No such file or directory
Please press Enter to activate this console.
BusyBox v1.00-pre8 (2005.08.19-23:48+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.
#
```

CHAPTER 5
SMARTLIFE ARCHITECTURE AND SIMULATION RESULTS

SmartLife is an FPGA based point of Intelligence that monitor the critically ill patients at home as well as provide assistance in their daily life. People with asthma, emphysema chronic bronchitis, occupational lung disease, lung cancer, cystic fibrosis, or congestive heart failure may need oxygen therapy at home [30].

5.1 Overnight monitoring patients using pulse oximeter

It has long been recognized that the physiological response of the patient to a stress or disease will largely determine the outcome. It is important, therefore, to monitor the physiological responses of patients since this not only allows the assessment of physiological reserve but will also give a baseline against which the effectiveness of any applied treatment can be judged. Pulse oximetry is one of the most widely used tools to determine a patient's cardio respiratory stability. Over the last 40 years, it has often replaced arterial blood gas analysis because the arterial oxygen saturation (SaO_2) frequently gives a sufficient amount of information about a person's respiratory patterns. The reduced supply of oxygen over demand for oxygen results in cell injury and organ dysfunction [31].

Pulse oximetry is a well-established tool routinely used in many settings of modern medicine to determine a patient's arterial oxygen saturation and heart rate. A descriptive statistics of normal oxygen values in different age groups are shown in the following table.

Table 5.1: Statistics of normal oxygen

Age Group, Yr	Patient, No.	Low Sat (SD), %	Sat 10 (SD). %	Sat 50 (SD),%
All ages	350	90.4(3.1)	94.7(1.6)	96.5(1.5)
< 1	30	90.7(2.6)	95.2(1.0)	96.4(1.2)
1-10	180	90.1(3.6)	95.1(1.5)	96.8(1.4)
10-20	46	90.4(2.7)	94.5(1.8)	96.5(1.6)
20-30	12	92.0(3.4)	94.8(1.1)	96.3(1.0)
30-40	24	91.5(2.2)	94.8(1.3)	96.3(1.1)
40-50	25	91.1(2.0)	94.2(1.7)	96.0(1.3)
50-60	16	90.4(1.9)	93.6(1.6)	95.8(1.7)
≥ 60	17	89.3(2.8)	92.8(2.3)	95.1(2.0)

Low Sat = lowest oxygen saturation during the night; Sat 10 = saturation below which the patient spent 10% of the time; Sat 50 = median saturation during the night (Reproduced from Ref. [38])

An overnight oximetry data as shown in figure 5.1 shows a breathing pattern of an obstructive apneas and hypopneas.

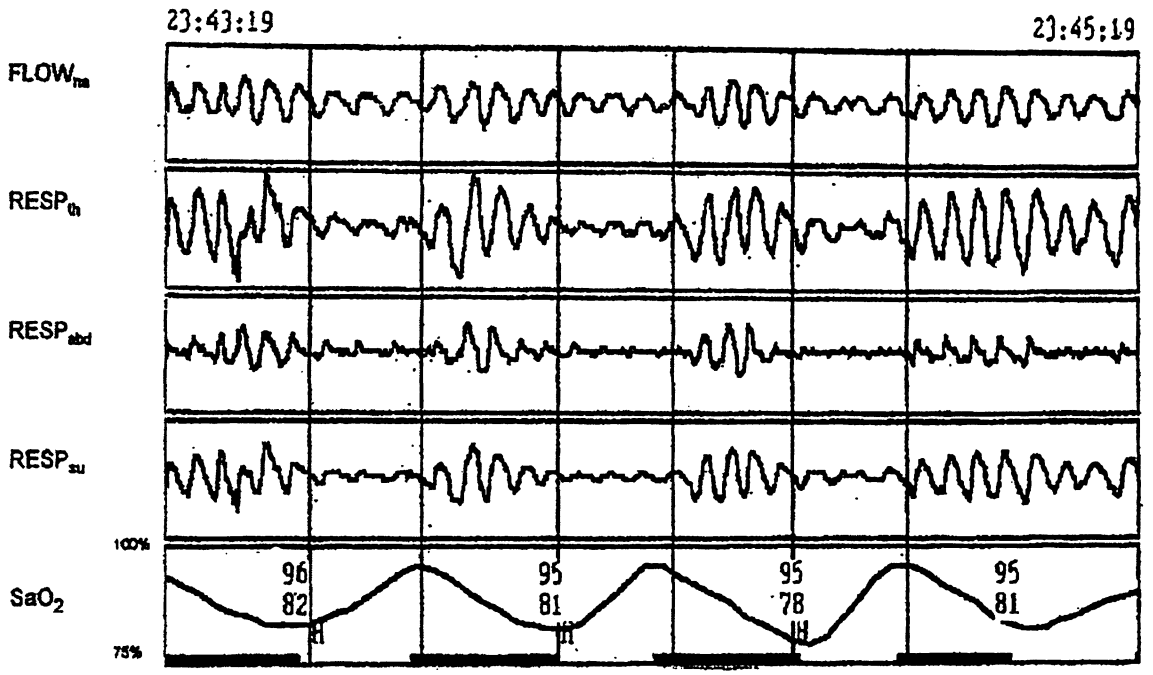

Figure 5.1 Respiratory patterns: A 3-min time period for a patient with OSA syndrome showing obstructive hypopneas with a more regular up-and-down waveform of the pulse oximetry curve (Reproduced from Ref. [38])

5.1.1 Wireless pulse oximeter

Advantages of wireless technology are obvious in situations where monitoring a patient from a distance leads to improve efficiency [34]. As shown in figure 5.2 the oximeter auto magically turns itself on. After a few seconds, the "perfusion display" LED starts blinking in sync with your pulse. The color of the blinking LED is green, yellow or red, indicating whether the unit is detecting good, marginal or inadequate pulse amplitude (If the indication is yellow or red, simply reposition the clip or change to a different finger). After a few heartbeats, the two numeric LED displays light up. The top number -- labeled "%SpO2" -- shows the percentage of oxygen saturation of your arterial blood, normally a figure between 95% and 100% at sea level, and progressively less at higher altitudes. The bottom number -- labeled with a little heart symbol -- shows your pulse rate in beats per minute.

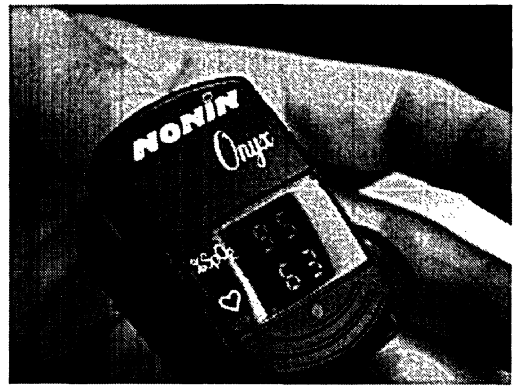

Figure 5.2: Pulse Oximeter (Reproduced from Ref. [34])

The figure 5.3 below shows the block diagram of a wireless Oximeter sensor. The sensor itself, a way to perform rudimentary processing on its output to measure blood oxygen saturation where as the controller convert the sensor output into a digital form and pass this data onto a suitable form the transceiver, and of course, the wireless transmitter. Most medical equipment uses the Industrial, Scientific and Medical (ISM) band, which operates at 2.4GHz.

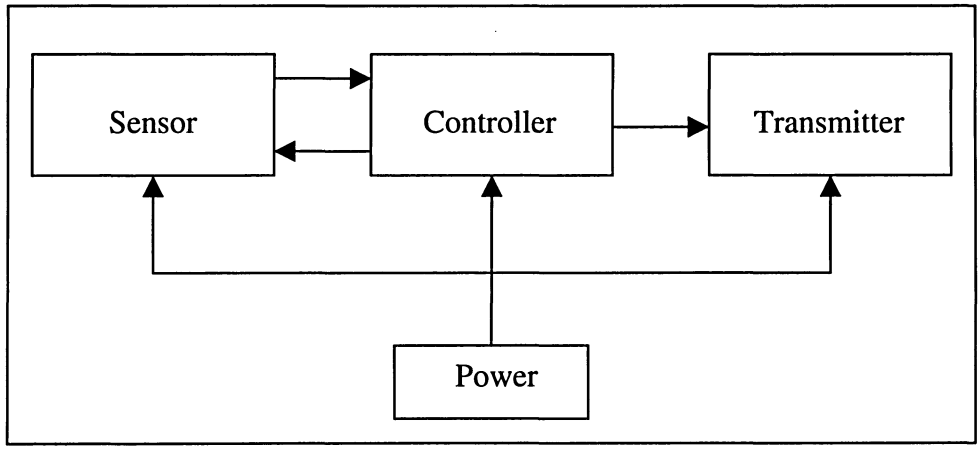

Figure 5.3: Block diagram of Oximeter Sensor

The most common non-invasive method used to measure blood oxygen saturation is known as pulse oximetry. This technique is based upon the different red and infrared light absorption characteristics of oxygenated (HbO2) and deoxygenated (Hb) hemoglobin. The figure below illustrates the differences in absorption of both.

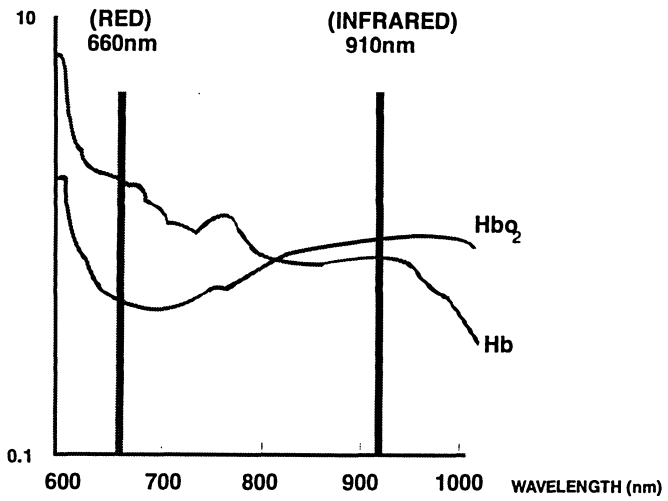

Figure 5.4: Light absorption characteristics of the two types of hemoglobin

Both red and infrared light are shone through a part of the body that is translucent, and has good blood flow (typically the ear, finger or toe). A photo detector at the opposite end determines the strength of the resulting red and infrared signals.

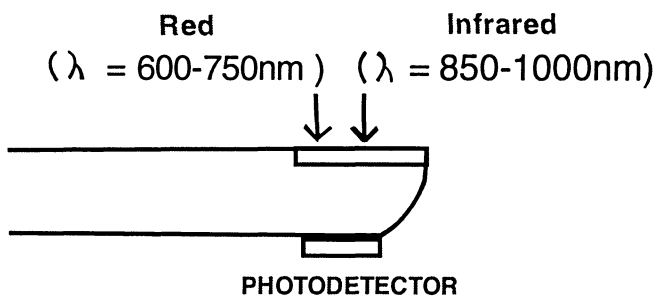

Figure 5.5: Illustration of the principle of a pulse oximeter

The ratio of red to infrared is calculated, and from this the percent of oxygen saturation (or SpO2 value) can be determined. Most oximeters are pulse oximeter, which means they can compensate for the fact that the pattern of saturation level at any instant will be a waveform, the peaks occurring at every heartbeat or 'pulse' [46].

Pulse oximetry provides a rapid, simple, continuous, and noninvasive way to measure blood oxygen saturation levels it's perhaps the fastest and most objective early warning indicator available to gauge the presence and degree of hypoxemia Pulse oximetry may alert you to hypoxemia within seconds widening your window of opportunity for intervention. Baseline

oxygen saturation levels vary depending on patient's cardiopulmonary status, but in general, the normal saturation for a patient at sea level is 95% SpO2 or above. The level at which a person starts to become noticeably impaired is approximately 90% SpO2, and a reading close to 80% SpO2 indicates severe hypoxia. However, there are limitations to this technique, which can affect the accuracy of the reading [47].

5.2 Long Term Oxygen Therapy in Adults

Domiciliary oxygen therapy is an effective but potentially an expensive therapy. Previous studies have demonstrated reduced mortality with the use of domiciliary oxygen therapy in patients with chronic obstructive pulmonary disease (COPD) [30]. There is also evidence that it improves right heart failure caused by corpulmonale, enhances neurophysiological function, and increases exercise tolerance in the performance of day to day activity. Supplementary oxygen is unlikely to contribute usefully to the relief of dyspnoea, heart failure or angina in absence of hypoxaemia [31]. The other conditions likely to benefit from supplemental oxygen therapy include cyanotic congenital heart disease, severe congestive cardiac failure, interstitial lung disease, advanced lung cancer, bronchiectasis or any illness with chronic hypoxaemia [31].

5.2.1 Oxygen Delivery Systems

The various oxygen delivery systems are,

- Cylinders
- Concentrators
- Liquid oxygen
- Conservation devices

5.2.2 Cylinders

The cylinders contain pure oxygen in a compressed state. They deliver 100% oxygen at the outlet and are preferred for intermittent use at home. Cylinders need to be protected from heat, as it causes an increase in their pressure. When oxygen is used for more than 8 hours a day, it is more convenient to use an oxygen concentrator. The use of portable oxygen cylinders may improve exercise tolerance, quality of life and the ability to do simple tasks. Portable cylinders can be

refilled at home from a source of liquid oxygen using a special valve, but not from large gas cylinder or an oxygen concentrator [33].

5.2.3 Oxygen Concentrators

Oxygen concentrators are electrically driven devices that entrain room air. A molecular sieve removes nitrogen and delivers oxygen at the outlet. They do not store oxygen and hence must run all the time for which oxygen is needed. The concentrators should be placed in a well-ventilated area with adequate tubing and with multiple outlets to increase patient freedom. These units deliver up to 90%-95% oxygen at the outlet at a flow rate of 2L/min. The percentage of oxygen falls with the increase in flow rate. A back up oxygen cylinder is desirable in case of concentrator breakdown or power failure [33].

Figure 5.6: Oxygen Concentrator (Reproduced from Ref. [33])

5.2.4 Liquid Oxygen Systems

Liquid oxygen systems provide the most flexible source of home oxygen. Oxygen has a boiling point of -183°C and 1 liter of liquid oxygen provides 860 liters of oxygen. The liquid oxygen containers are insulated and are at a relatively low pressure. Frost bite or burns can occur by contact with the container or the tubing. The reservoir is used to fill light portable cylinders containing 1 litre of liquid oxygen, which can last up to 8 hours when the oxygen is delivered at

2 litres/min [33].

5.2.5 Conservation Devices

Conservation devices are introduced between the oxygen delivery source and the patient. They ensure that oxygen is delivered only during inspiration and not wasted during expiration. These are useful cost and time conserving devices, especially for portable units. Conservation devices switch on the flow by sensing negative pressure at the nares via the nasal cannula. These may not trigger if the patient breathes by his mouth.

5.2.6 Devices for delivering oxygen

Nasal prongs are generally the best way of delivering long-term oxygen as it is convenient whilst eating and talking. Extra-soft nasal prongs may be used for continuous oxygen therapy. Facemasks can be used when there is no danger of carbon dioxide retention as the fraction of inspired oxygen (FiO2) delivered by variable performing masks varies with changing breathing pattern. Transtracheal oxygen delivery has the advantage of allowing substantially lower flow rates. However, care of this relatively invasive delivery system is demanding.

5.3 Application Architecture

Due to limited research resources we build a software simulator for wireless sensors network environment using Java language based on its platform independence as well as rich class libraries. SmartLife (A point of Intelligence) application is written in C language which will run on uClinux based NIOS-II development board. Figure 5.7 shows a block diagram of SmartLife.

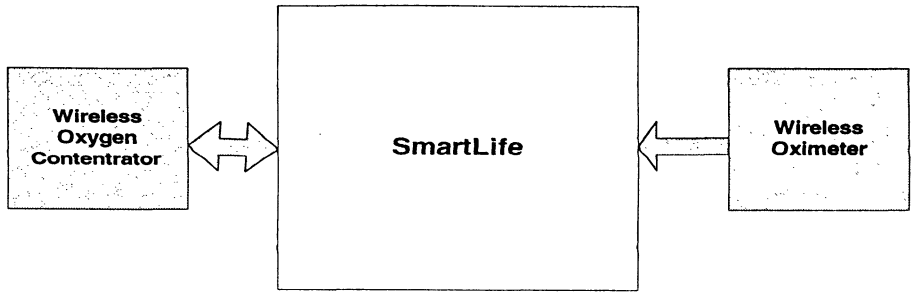

Figure 5.7: Block diagram of SmartLife

Upon power on, the oximeter transmits an 'announcement' message, and then waits for a response. If none is forthcoming after 2 seconds, it will try again. The acknowledgement consists of a message identifier, as well as an ID for the oximeter, which is then used for all the subsequent messages.

Sampling of the output from the wireless sensor then begins, and transmissions occur in a 'burst' every 20ms. Once per second, or 50 transmitted readings, the oximeter will begin to watch for a 'refresh request' message sent by the SmartLife and will respond with an acknowledgement. If this message is not received after another 50 readings, then it is assumed that the communications have been broken, and will re-transmit data otherwise it return to the beginning and transmit the 'announcement' message as shown in figure 5.8.

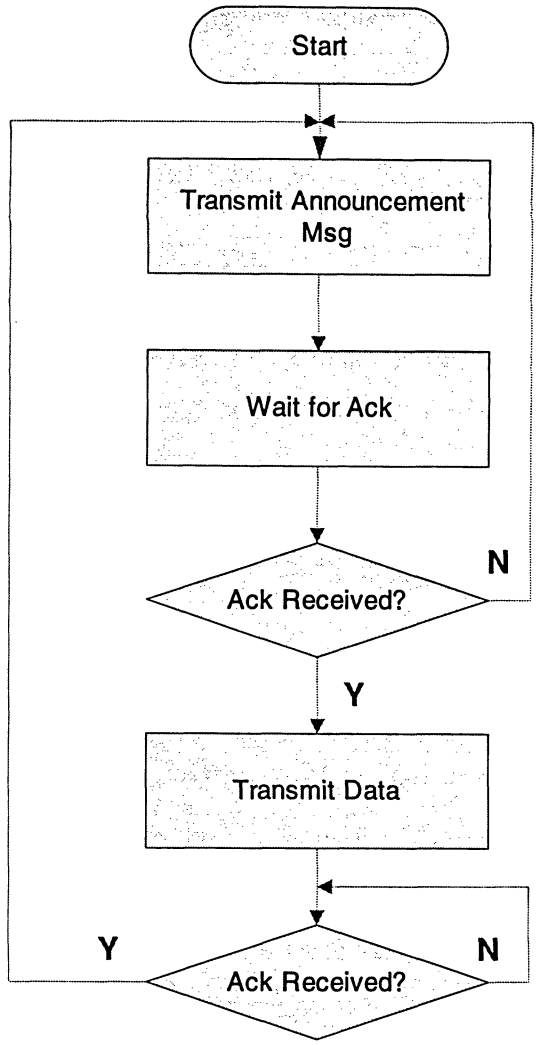

Figure 5.8: Oximeter Data Flow Chart

Oxygen concentrators are stationary, electrical units that take in nitrogen from the air around the patient, provides a streaming flow of oxygen. Oxygen concentrators are less expensive than liquid oxygen and are the most cost-effective source of oxygen therapy. Almost 80% of US home oxygen patients use oxygen concentrators in their daily lives. A concentrator does not store oxygen like the aluminum cylinders of the past. It produces and distributes Oxygen on a continuous basis. It takes the regular air, which normally contains 21% oxygen, and removes the nitrogen. The resulting air is 95.5 percent pure oxygen. Actuality, concentrators produce oxygen that is generally between 87 - 95 percent pure. US Medicare requires the purity to be greater than 85 percent.

Oxygen concentrators are very reliable and easy to maintain. Keeping the intake filter clean and clear of dust is being the most important. This should be done weekly. There are also some very important safety issues. Oxygen is the most dangerous in the presence of fire. There should be no smoking, no flammable materials, and no heat sources near a working oxygen concentrator. Oxygen concentrators have some limitations; they can't produce high flow oxygen or a supply of portable oxygen. Light weight cylinders, liquid oxygen systems and oxygen conservation devices are excellent portable supply systems and each should be discussed with the patient's physician to determine which is the best. Supplemental oxygen is only available by prescription and must meet certain requirements to be covered by Medicare or insurance. An oxygen concentrator should have an oxygen sensing device which monitors the level of oxygen produced by the concentrator and warns both visual and audible alarms when its level falls too low.

Upon power on, a typical oxygen concentrator transmits an 'announcement' message and when waits for a response. If none is forthcoming after 2 seconds, it will try again. The acknowledgement consists of a message identifier, as well as an ID for the oxygen concentrator, which is then used for all subsequent messages. Sampling of the output from the wireless sensor then begins, and transmissions occur as a 'burst' every 20ms. Once every second, or 50 transmitted readings, the concentrator will begin to watch for a 'refresh request' message sent by the SmartLife and will respond with an acknowledgement. If this message is not received after another 50 readings, then it is assumed that the communications have been broken and will re-transmit data. Otherwise it returns to the beginning, transmitting the 'announcement' message as shown in figure 5.9.

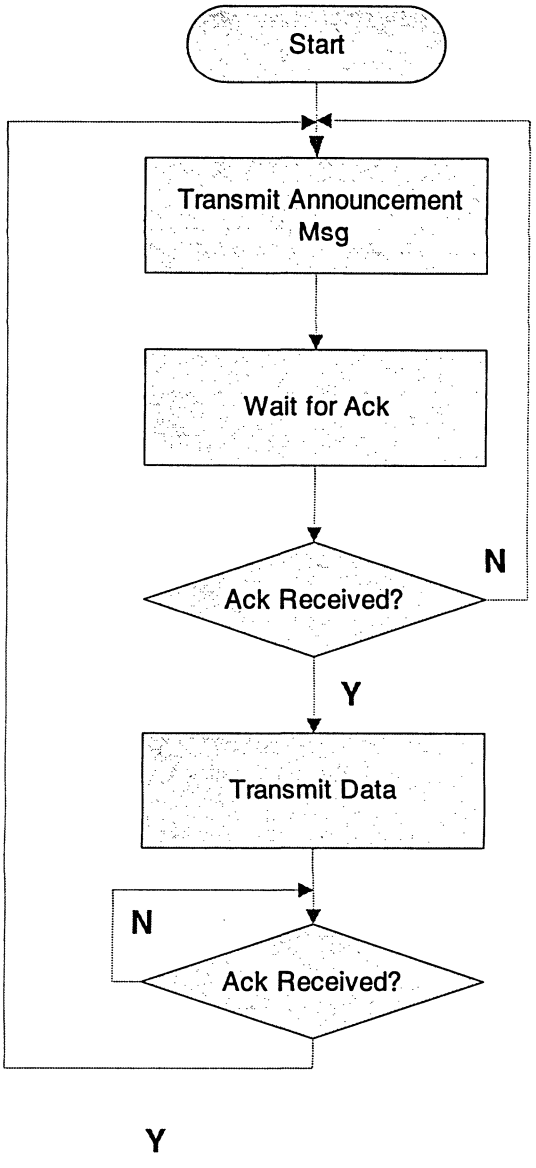

Figure 5.9: Oxygen Concentrator Data Flow Chart

Upon power on, the SmartLife server waits for an announcement message, once it received a message it sends an acknowledgement and then waits for an incoming data. If none is forthcoming after 2 seconds, it will try again. Sampling of the data from the wireless sensor then begins, and receiving occur as a 'burst' every 20ms. SmartLife will respond to the incoming data appropriately. If this data is received from wireless oximeter sensor and its reading is less than 90 then it will set an oxygen concentrator to deliver oxygen at the rate of 3lpm. If this data is

received from wireless oxygen concentrator and its reading is less than 85 then it will turn on the alarm to make sure that oxygen concentrator is not malfunctioning as shown in figure 5.10.

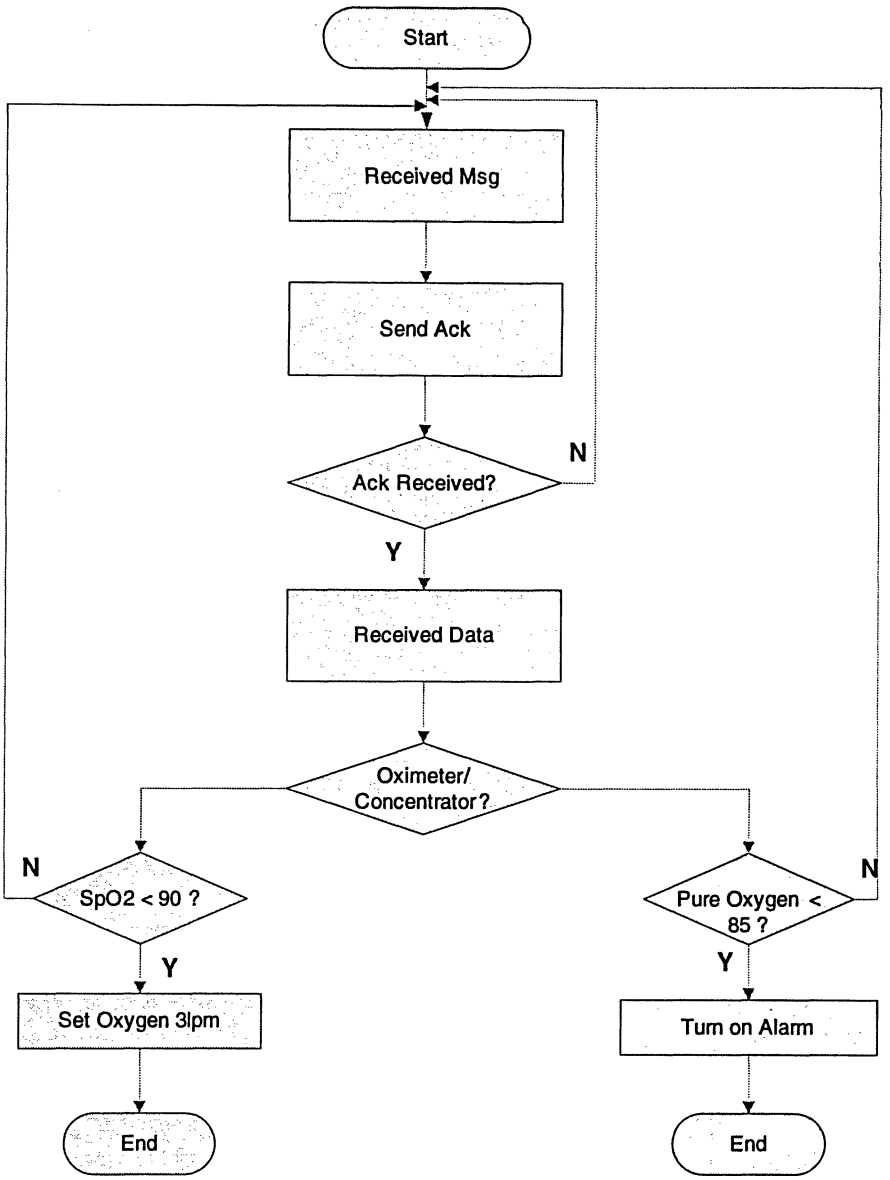

Figure 5.10: SmartLife Data Flow Chart

5.4 Simulation Results

A server running on uClinux shows sending and receiving data from different wireless sensor devices as seen in figure 5.11. A simulator shows sending and receiving data from SmartLife server as seen in figure 5.12.

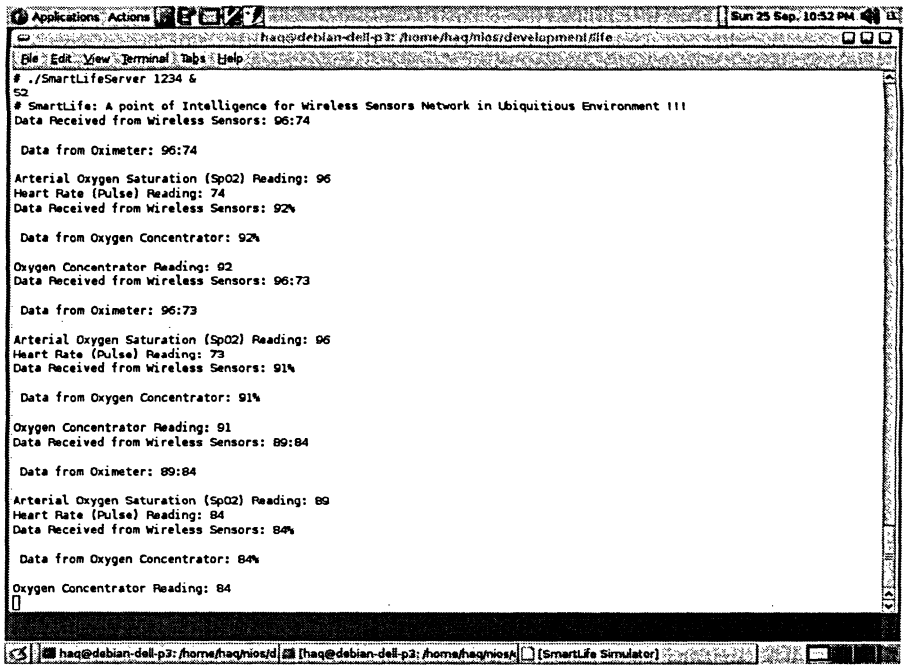

Figure 5.11: SmartLife server

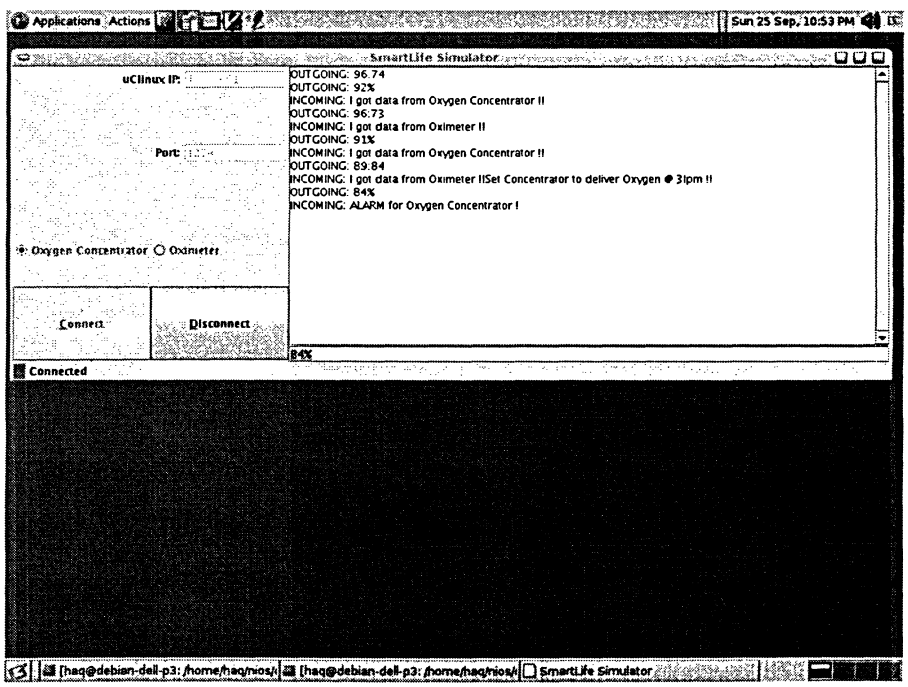

Figure 5.12: SmartLife wireless simulator

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

The research has been focused on uClinux based Embedded System architecture which provides an integrated and comprehensive framework for building pervasive applications. We describe the design and implementation of our architecture as well as building an application using uClinux.

Major contributions include the investigation, deployment, design and development of a wireless sensor networks application for monitoring critically ill patients in smart home environment as well as porting uClinux on an FPGA based hardware platform along with universal boot loader called U-Boot. In this research project we simulated an environment in which a patient at home can be monitored remotely, which is not only cost effective but it is also very convenient.

We focused on providing an embedded system that can be used for monitoring as well as controlling critically ill patients that need oxygen therapy for long period of time. Recently Microtronix company partner with Altera to provide an embedded operating system called uClinux for NIOS-II based FPGA development boards.

Future Work:

Current uClinux kernel is based on linux kernel 2.6.11 which is made pre-emptive i.e., a higher-priority task can now interrupt the lower-priority task even during the processing of a system call. But there are still regions of kernel code wherein the task is not pre-emptive.

This research work will be more challenging if we use real wireless sensors instead of simulator application. Research in this field is far from complete; in fact, it's still in its infant stage. This research project insight into a new approach of an FPGA based Ubiquitous computing environment. We addressed future challenges of pervasive computing where network sensors will be part of our daily life to assist us.

CHAPTER 7
REFERENCES

[1] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, John Anderson, "Wireless Sensor Networks for Habitat Monitoring," WSNA, Atlanta, GA, Sept. 2002, pp. 88-97.

[2] Ke Cui, Zhenwei Wu, "Research and implementation of remote monitoring system based on real-time uClinux," Services Systems and Services Management, 2005. Proceedings of ICSSSM '05. 2005 International Conference, vol. 2, June 13-15, 2005, pp.1182-1187.

[3] A. Weaver, J. Luo, and X.Zhang, "Monitoring and control using the Internet and Java," IEEE Int. Conf. on Indu. Elec. (IECON'99), 1999, pp. 1152-1158.

[4] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A Taxonomy of Wireless Micro-Sensor Network Models," MC2R, vol. 6, no. 2, Apr. 2002, pp. 28–36.

[5] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Emerging Challenges: Mobile Networking for Smart Dust," J. Commun. and Networks, vol. 2, no. 3, Sept. 2000, pp. 188–96.

[6] P. Juang Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, Daniel Rubenstein, "Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet," Proc. ASPLOS X, San Jose, CA, Oct. 2002,pp.96 –107.

[7] Florian Michahelles, Peter Matter, Albrecht Schmidt, Bernt Schiele, "Applying Wearable Sensors to Avalanche Rescue," Computers and Graphics, vol. 27, no. 6, 2003, pp. 839–847.

[8] H. Baldus, K. Klabunde, and G. Muesch. "Reliable Set- Up of Medical Body-Sensor Networks," Proc. EWSN 2004, Berlin, Germany, Jan. 19-21, 2004, pp. 353-363.

[9] C. Kappler and G. Riegel, "A Real-World, Simple Wireless Sensor Network for Monitoring Electrical Energy Consumption," Proc. EWSN 2004, Berlin, Germany, Jan. 19-21, 2004, pp. 339-352.

[10] S. Antifakos, F. Michahelles, and B. Schiele, "Proactive Instructions for Furniture Assembly," Proc. Ubicomp 2002, Gothenburg, Sweden, Sept. 2002, pp. 351 – 360.

[11] Liang Cheng, Yuecheng Zhang, Tian Lin, Qing Ye, " Integration of Wireless Sensor Networks, Wireless Local Area Networks and the Internet". Proc. IEEE Sensing and Control ,Taipei, Taiwan, Mar. 21-23, 2004, pp.462-467.

[12] William M. Merrill, Fredric Newberg, Kathy Sohrabi, William Kaiser, Greg Pottie,

"Collaborative Networking Requirements for Unattended Ground Sensor Systems," Proc. IEEE Aerospace Conf., vol. 5, Mar. 8-15, 2003, pp. 5_2153- 5_2165.

[13] G. Simon, A. Ledezczi, and M. Maroti. "Sensor Network- Based Countersniper System," Conference On Embedded Networked Sensor Systems, Baltimore, MD, Nov. 2004, pp.1-12.

[14] Bovet, Daniel P - Cesati Marco. 2001. Understanding the Linux Kernel. Sebastopol: O'Reilly & Associates, Inc.

[15] Labrosse, Jean J. MicroC. 1998. OS II: The Real Time Kernel, 2nd Edition. Gilroy: R&D Books.

[16] Levine, John R., Linkers & Loaders. 1999. San Francisco: Morgan Kaufmann Publishers.

[17] Matthew, Neil-Stones Richard, 2000. Beginning Linux Programming, 2nd edition. Birmingham: Wrox Press Ltd.

[18] Rubini, Alessandro - Corbet, Jonathan. 2001. Linux Device Drivers, 2nd Edition Sebastopol: O'Reilly & Associates, Inc.

[19] Tanenbaum, Andrew S-Woodhull Albert S. Operating Systems: Design and Implementation, 2nd Edition. Upper Saddle River: Prentice Hall, Inc. 1997.

[20] Arcturus Networks, Inc. 2001. uClinux WHITE PAPER OVERVIEW. http://www.arcturusnetworks.com/Docs/UCLINUXWP.pdf.

[21] deBlaquiere, Joe. Supporting New hardware Environments with uClinux. http://www.redhat.com/embedded/technologies/resources/deblaquiere.pdf.

[22] Drabik, John. 2002. uClinux: World's most popular embedded Linux distro?. http://www.linuxdevices.com/articles/AT3267251481.html.

[23] Free Software Foundation, Inc. 1999. GNU Lesser General Public License. http://www.gnu.org/copyleft/lesser.html.

[24] Gillham, Miles, 2002. uClinux and Linux Set To Merge. http://www.snapgear.com/

[25] McCullough, David. 2002. Why is Malloc Different Under uClinux?. http://www.linuxdevices.com/articles/AT7777470166.html.

[26] Peacocku, Craig, 2002, uClinux - Understanding the build tools. http://www.beyondlogic.org/uClinux/builduC.htm.

[27] uClibc -- a C library for embedded systems. http://www.uclibc.org/

[28] uClinux—Linux on Microcontrollers. http://www.linuxdevices.com/links/LK8053710489.html.

[29] Ungerer, Greg, 2002. Using Flash Memory with uClinux. http://www.realtimeinfo.

[30] Iven H Young, Alan J Crockett and Christine F McDonald, "Adult domiciliary oxygen therapy. Position Statement of the Thoracic Society of Australia and New Zealand". MJA (Medical Journal Association) 1998, vol. 168, pp. 21-25.

[31] Recommendations for long term oxygen therapy (LTOT). Report of European Society of Pneumology Task Group. Eur Respir J 1989, vol. 2, pp.160-65.

[32] Fletcher EC, Luckett RA, Good-night White S, Miller CC, Qian W, Costarangos-Galarza C. A double-blind trial of nocturnal supplemental oxygen for sleep desaturation in patients with chronic obstructive pulmonary disease and a day time PO2 above 60 mm Hg. Am Rev Respir Dis 1992, vol.145, pp. 1070-76.

[33] http://www.airsep.com/medical

[34] http://www.nonin.com/products/9500.asp

[35] http://www.altera.com/products/devkits/altera/kit-nios_1S40.html

[36] http://www.altera.com/literature/lit-nio2.jsp

[37] Karim Yaghmour, Building Embedded Linux Systems, O'Reilly 2003.

[38] Nikolaus Netzer, Arn H. Eliasson, Cordula Netzer and David A. Kristo, "Overnight Pulse Oximetry for Sleep-Disordered Breathing in Adults: A Review". Chest 2001, vol. 120, pp. 625-633.