Kimmo Nikkanen
Turku Polytechnic
Telecommunications
Embedded Systems

UCLINUX AS AN EMBEDDED SOLUTION

Bachelor's Thesis

ABSTRACT

Institute	Turku Polytechnic
Degree program	Telecommunications
Specialization	Embedded Systems
Author	Kimmo Nikkanen
Subject	uClinux as an embedded solution
Type of work	Bachelor's Thesis
Instructor	Jari Lahti, B.Sc
Supervisor	Jari-Pekka Paalassalo, Lic.Sc
Date	1 / 2003
Key Words	Embedded, uClinux, micro controller, Linux

This study is the conclusion of the work done by applying the embedded Linux distribution uClinux to Viola Systems Arctic platform. uClinux is the most widely used embedded Linux distribution at the moment, designed specially to run on microprocessors without the Memory Management Unit, MMU.

The study is designed to work as a Viola Systems internal instruction for the new developers. It introduces the basics of the uClinux kernel and the development platform. The work also points out some similarities and differences between the uClinux and the general purpose Linux.

The Arctic platform is specially designed for device connectivity and management in industrial usage. For these demands the uClinux and its tools provide a good foundation by being a totally royalty-free and open source solution, inheriting the strong networking support that the general purpose Linux provides.

TIIVISTELMÄ

Korkeakoulu	Turun ammattikorkeakoulu
Koulutusala	Tietoliikennetekniikka
Suuntautumisvaihtoehto	Sulautetut järjestelmät
Työn tekijä	Kimmo Nikkanen
Työn nimi	uClinux sulautettuna ratkaisuna
Työn laji	Insinöörityö
Ohjaaja	Insinööri Jari Lahti
Valvoja	Tekniikan lisensiaatti Jari-Pekka Paalassalo
Päivämäärä	1 / 2003
Avainsanat	Sulautettu, uClinux, mikro-kontrolleri, Linux

Tämän tutkimuksen tarkoitus on tehdä yhteenveto siitä työstä, joka on tehty sulautetun Linux-jakelupaketin, uClinuxin soveltamisesta Viola Systemsin Arctic -alustalle. uClinux on tällä hetkellä suosituin sulautettu Linux-jakelupaketti, joka on erityisesti suunniteltu prosessoreille, jotka eivät sisällä muistinhallintayksikköä.

Insinöörityö on suunniteltu toimimaan Viola Systemsin sisäisenä ohjeena uusille kehittäjille. Siinä esitellään lukijalle uClinux kernelin perusteita ja kehitysympäristöä. Työ kiinnittää huomiota myös uClinuxin ja yleiskäyttöisen Linuxin välisiin eroavaisuuksiin ja yhtymäkohtiin.

Arctic-kehitysalusta on suunniteltu erityisesti laitteiden liitäntään ja hallintaan teollisuusympäristössä. Näihin vaatimuksiin uClinux ja sen kehitystyökalut pystyvät tarjoamaan hyvän alustan, koska siitä ei makseta tekijänoikeuspalkkioita ja se perustuu avoimeen lähdekoodiin. Yleiskäyttöisen Linuxin tavoin, uClinux tarjoaa myös vahvan verkossa kommunikointi tuen.

TABLE OF CONTENTS

ABBREVIATION AND ACRONYMS

ANSI	American National Standards Institute
API	Application Program Interface
as	GNU assembler
bash	Bourne Again Shell
bFLT	Binary Flat Format
binutils	A GNU collection of binary utilities
BIOS	Basic Input/Output System
blkmem	The block memory device driver
brk	The brk() system call; moves the break point for the calling process
BSD	Berkeley Software Distributions; a UNIX Socket Library
CPU	Central Processing Unit
CRAMFS	Compressed RAM File System.
DCE	Data Communications Equipment
DIMM	Dual Inline Memory Module
DTE	Data Terminal Equipment
eCos	Embedded Configurable Operating System
EEPROM	Electrically Erasable Programmable Read-Only Memory
ELF	Execution and Linking Format
elf2flt	An elf to flat binary format conversion program
ext2	Second Extended File System
FIFO	First In, First Out; a queue type
Flash	A nonvolatile memory capable of retaining digital information
fork	The fork() system call; creates a new processes
FTP	File Transfer Protocol
GB	Giga Bytes
GCC	GNU C and C++ Compiler
GDB	GNU debugger
genromfs	Program to generate the ROM file system image
gid	Group Identifier
GNU	A recursive acronym for `GNU's Not Unix'
GOT	Global Offset Table

GPL	The GNU General Public License
GPRS	General Packet Radio Service
GRUB	Grand Unified Bootloader
GUI	General User Interface
HTTP	Hypertext Transfer Protocol
HW	Hardware
I/O	Input-Output
I2C	Inter-Integrated Circuit
IEEE	Institute of Electrical & Electronics Engineers
INET	Institutional Network; for sharing data and providing Internet
ioctl	The ictol() system call; for device input and output control
IP	Internet Protocol
IPC	Interprocess Communication
ISDN	Integrated Services Digital Network
JFFS/2	The Journaling Flash File system; two different versions
ld	GNU linker
LGPL	Lesser General Public License
LILO	Linux Loader
loadlin	Load Linux
lpr	Offline print command
ls	List directory contents command
m68k	Linux port for Motorola 680x0-based systems
MBR	Master Boot Record
MCF	Motorola Cold Fire
MCU	Micro Controller Unit
MHz	Mega Hertz
MIB	Management Information Base
MMU	Memory Management Unit
MP3	A loose acronym for MPEG Audio Layer 3
MPEG	Motion Picture Experts Group
msep-data	Separate Data; allows for the data and text segments to be separated and placed in different regions of memory
MTD	Memory Technology Device
NFS	Network File Systems
OS	Operating System
OSI	Open Systems Interconnection

PC	Personal Computer
PDA	Personal Digital Assistent
PHY	Physical Interface Circuit
PIC	Position Independent Code
PID	Process Identifier
POSIX	Portable Operating System Interface for Unix
PPCboot	PowerPC boot
ppp	Point-to-Point Protocol
RAM	Random Access Memory
ROM	Read-Only Memory
romfs	Read-only filesystem for Linux
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Message Protocol
SPI	Serial Peripheral Interface
SSH	Secure Shell
STL	Standard Template Library
SW	Software
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver Transmitter
uClibc/uC-libc	micro libc
uClinux	Micro-controller Linux
UDP	User Datagram Protocol
uid	User Identifier
UNIX	Uniplexed Information and Computing System
USB	Universal Serial Bus
vfork	The vfork() system call; to create processes and block parent
VFS	Virtual File System
WAN	Wide Area Network
XIP	Execute In Place

Chapter 1

INTRODUCTION

The barrier between embedded operating system and general purpose system is becoming more blur as embedded devices are evolving to meet the markets requirements. As more is required from the embedded devices the applications they run are becoming more sophisticated and dependent of more intelligent management. Still it is expected that they remain to be cost effective solutions for their limited mission.

For the companies providing solutions for embedded markets, a quick time to market and reliability of the platform are the key features of the development. The solutions aimed to industrial usage should be able to provide an network connectivity to the end device, supporting multiple of different networking protocols. For a total network solution, also the security and management of the end device asserts more requirements for the system. These and other demands increases the complexity level and more often require a services of an full operating system.

For these challenges the embedded Linux gives an advantage of being totally royalty free, open source and compact solution, providing an strong foundation for the applications to run on. It offers a strong networking support, such as network management protocols, e-mail and different security options. The advantages that the general purpose Linux offers can also be found from the embedded versions. The open nature of the Linux possibles quick modifications against the physical requirements, still providing an reliable and standardized base system for the developers. Linux also provides various of freely available open source applications, which can also be used with the embedded versions.

This thesis is the conclusion of the work done with applying the embedded Linux version, uClinux to the industrial micro controller and more specific to Viola Systems Arctic platform. It is intended to work as company's internal instruction for the developers by summarizing the information gained from this process.

The study mainly focuses on the uClinux kernel architecture and the development tools. This is done by pointing out some general solutions made with the still developed Arctic platform and by introducing some options that the developer can make. It also states out some similarities and differences against the general purpose Linux and briefly introduces the architectural concept of the Linux kernel, from the developers point of view.

Chapter 2

INTRODUCTION TO LINUX

Linux is a kernel developed by a Finnish Computer Science student Linus Torvalds, at the University of Helsinkin in 1991. The original goal with the Linux was just to create an free Unix like operating system based on the free Minix operating system. As Torvalds released the first version of Linux available on the internet it immediately began to gain response from the other programmers. Today Linux is one of the most talked operating system, having thousands of developers around the world improving the kernel itself and on developing Linux applications.

2.1 History of Linux

The idea behind Linux was to develop a Unix like operating system that could meet the demands of professional users. One major reason for developing Linux was that Unix was designed to meet industrial demands which made it expensive and without open source code. The solution for this seemed to be Andrew S. Tanenbaums small Unix-like operating system called Minix. It was originally developed to teach his students the inner workings of a real operating system, so the source code was totally available and free. [22.]

Linux is a low level core of an operating system, originally created for IBM-compatible personal computer based on the 32-bit Intel 80386 microprocessor. In September 1991 the Linux version 0.01 was released on the net and started to get attention from the developers around the net. As the amount of other programmers response kept increasing, the Linux kernel was decided to be licensed under the GNU General Public License, GPL thus ensuring that the source code would remain free for all copy and change. The history of Linux kernel can be seen from the table 1-1. [23.]

Table 1-1. History of Linux kernel.

Year	*Version*	*Event*
1991	0.01	Linus Torvalds writes Linux kernel and later on a same year announces the first official version of the kernel, version 0.02.
1992	0.99	Linux was decided to license under the GNU's GPL.
1994	1.0.0	First public release.
1995	1.2	First multiple platform support.
1996	2.0.0	Multiple hardware architectures, several new networking protocols.
1999	2.2	Dynamic kernel module support, some embedded systems supports.
2001	2.4	New file system supports, improved infrastructure to run as a server.
	2.5	Current Development Kernel

Nowadays Linux has thousands of developers around the world which ensures that Linux stays on the edge of the development and is modified to take advantage of it. It also gives new developers an advantage, as Linux provides free documentation and support over the network. Linus Torvalds is still in charge of the Linux kernel development. He is the one that accepts the modifications and additions to the kernel sources and merges all the new code into the kernel himself.

2.1.1 Version numbering

The Linux kernel has its own version numbering scheme. The first public version of the Linux kernel was released in 1994. This was Linux V1.0.0. Since then every released Linux software package has its own release number, each of them containing a series of numbers with three digits separated by a period. The version numbers are of the form a.b.c where a and b denotes the version and c for the current release. The middle number b denotes whether the kernel is a stable or under development. The odd number denotes for kernel release still under development and the even number that the release is stable. The stable version is not any more changed in a way that would change the behavior of the kernel core and the changes are only bug fixes for the current release. [1, p. 6-7.]

2.1.2 Linux supports

Almost all the common features supported by commercial versions of UNIX are included in Linux kernel. The Linux includes and supports features from both BSD Unix application program interface and Unix System V interface. Still the POSIX.1 kernel interface is considered to be the base standard for the kernel. Linux also supports many common features of other UNIX standards like true multitasking, virtual memory, shared libraries, proper memory management, TCP/IP networking plus some other features. Still unlike some other UNIX versions, Linux is very small. The entire system with only the basic setting can be set on a single floppy disk. [21.]

Linux is highly platform-independent operating system, which allows it to be used under various of platforms. It is also easily portable to other platforms as the hardware dependent code is isolated separate sections in the source tree. It has support for machines like Sun SPARC, Motorola 68000, PowerPC and of course Intel x86 based computers. It is also supported by 64-bit architectures. Linux is also highly compatible with many other operating systems, as it supports various of different file system types.

2.2 Benefits of Linux

From the beginning the idea was that the Linux code distributions would be made freely available. This feature is done under the GNU General Public License, GPL; which is designed to make sure that people have a freedom to distribute copies of the software and that they are free to change the software or use pieces of it in new free programs. Originally the GPL was started by Richard Stallman, who wanted his software tools to be available for everyone including the source code. The actual license holds rights to use and copy the software, distribute it further and to be able of changing the source code as long as the changes are made available. This is also known as the copyleft.

Linux includes also other important license concerning the developers under Linux. LGPL, Lesser General Public License is aiming for the same results as the GPL. It also requires the generated code under this license to be kept available to everyone. The Lesser GPL is specially designed to be used with some specially designated software packages, typically libraries. The real difference between these two licenses is whether the generated code is allowed to link with the open source code. The LGPL allows code to be linked with the open source code and

distributed the binaries with no restrictions, as long as the code is dynamically linked against the LGPL licensed code. The LGPL allows software developers to use the LGPL library and still protect their source code from the third party, providing also an advantage to the open source over competing non-free programs. When linking the libraries under the LGPL statically to the source code the distributor must at least provide the object data files of the concerned program so that the third party is able to relink the source with a different version of the LGPL library if they so choose. [18.]

It self Linux doesn't provide all the features that a full Unix operating system provides. However Linux community supports of free software that can be installed into one of the file systems supported by Linux. Also because of the related history with Unix, can many commercial applications developed for Unix run unchanged in binary from on Linux systems. All together Linux can run thousands of different application for different purposes, which can be customized to meet user needs. Customization is also one major advantage of the Linux kernel. Usually the user is able to fully customize the application programs and the kernel down to the smallest detail. This of course is also one disadvantage as Linux requires quite much knowledge of different Unix type commands and configuration methods in order to make it working at its best. Installation of different applications may also require knowledge of the basic settings of our computer.

The TCP/IP networking has been integrated to Linux from the beginning. It supports all the major networking protocols, making it an excellent networking operating system. Linux also includes supports for various of different networking hardware. Today Linux supports many different networking protocols Internet Packet Exchange/Sequenced Packet Exchange, AppleTalk Protocol, WAN Networking, ISDN and PPP. Linux like other Unix like systems, it also has great support for remote and distributed execution of applications. It is also possible to use Linux as interconnection device like a router or a firewall.

Linux as an operating system is very stable. It has been developed and tested by volunteer programmers around the world, which are submitting bug reports to main developers. All the suggested new features and bug fixes are thoroughly investigated and included to the release if necessary. The development is also continuous so when a new security holes or a bug is discovered it is fixed almost immediately to the kernel source. As the core Linux operating gets bug fixes, the user can always fetch the latest kernel version from the kernel.org pages. Linux is also designed to run without any unnecessary reboots. This enables that the booting of the kernel is only necessary when a hardware upgrade is done to the computer.

A multiuser system is able to serve multiple user at the same time allowing them to execute and run programs independently at a same time. Linux is a full multiuser system sharing the hardware like CPU and memory between the different user and applications. The operating system is able of giving different privileges to different users each of them with an access to a predefined set of system services and private data. By default every Linux installation has a specially privileged account for superuser with access to all services and resources. This allows to implement security on a very basic level between the different users and by operating system. The system also works as an outside protection as many viruses need the superuser privileges to do any damages. [1, p. 9.]

Linux also includes an on-line documentation of the commands, library routines, kernel system calls, file formats and device driver interfaces on the system in a for of manual pages. These are easy to use helpers which display the applied document on the screen. Network holds also a tremendous support in a form of other developers as a free technical support and with various of written documents.

2.3 Disadvantages of Linux

As Linux brings many advantages with the open source and the GPL licensing, this also has an downward effect. When designing code for commercial usage, one should notice that code based on the GPL should be kept available. Although Linux lowers the purchasing cost from the software side a lot of resources can be worn to personnels training to be able to use the platform. This factor also rise up as generally code made as open source might be rather poorly documented. Open source provides also an risk. The open source also provides an security problem as some hostile third parties have the access to the sources and there for easily seek for new wholes from the networking software.

The major disadvantage for regular CPU users is that Linux is basically designed from a programmer to a programmer. Beginning from the installation, Linux might be pretty complex, although new distributions are becoming more and more user friendly. The actual design with a Linux system was not to develop an user-friendly operating system, but more of an development platform for programmers around the world. Users moving from the other operating systems like from Windows might see it hard to use the new commands and vocabulary, especially when commands are fed in a command prompt. Linux has evolved during the time to more 'icon based' operating system, but still getting everything out from the

Linux requires quite much of studding and learning. Also the word Linux gives many regular CPU users an impression of hardly adjustable professionals tool which in commercial usage can generate an problem.

Linux operating system is also pretty young. From the first release in 1991, only few years for efficient development has past. This means that Linux and GNU cannot offer such a set of competitive packages in all software categories as Windows and Mac OS. The growth of Linux has been from the beginning significant so new software are released freely to Linux every day. As in installation also a new hardware installation requires more knowledge of the component it self, but also from the underlaying operating systems configuration. Linux distributions nowadays has an plug and play support, but it still quite minimal.

Chapter 3

LINUX KERNEL

The heart of the operating system is the kernel. By it self the kernel is useless and it only participates as one layer in the overall computer system between the physical hardware and the application programs. Within the kernel, Linux runs subsystems to keep track of the different files, provide an execution environment for the different applications that run on the system, assigns memory and other resources for various of processes take care of the networking functionality, handle the interprocess communication and possibles the low-level elements to interact with the hardware.

3.1 Booting sequence

The boot process details are architecture-specific, so this in only an example of a typical boot sequence of an x86 based computer with an separate boot loader. Roughly the boot process of Linux operating system can be divided into the three following stages:

1. The hardware is initialized and made ready for the operating system.
2. Kernel is loaded to the RAM by the boot loader from a specific storage.
3. The kernel continues to initialize further the hardware and starts to run the different applications.

When power is turned on to the system, the processor starts to execute the instructions for getting the hardware ready for the operating system. In a personal computer these instructions locate in the storage like ROM, flash memory or on a personal computer in a system BIOS ROM. The Basic Input/Output System, BIOS contains embedded code that initiates tasks to detect particular system parameters like the amount of memory, disk devices, the date and defines the order in which of these disk devices should be checked for bootable media. The first few bytes of the hard drive are usually turned into an instruction where to find and load the operating system located an the machine. These first few sectors are called the Master Boot

Record, MBR which starts the loading of the operating system. The MBR usually contains the boot loader which is invoked to continue to load the image of an operating system kernel into RAM. [26.]

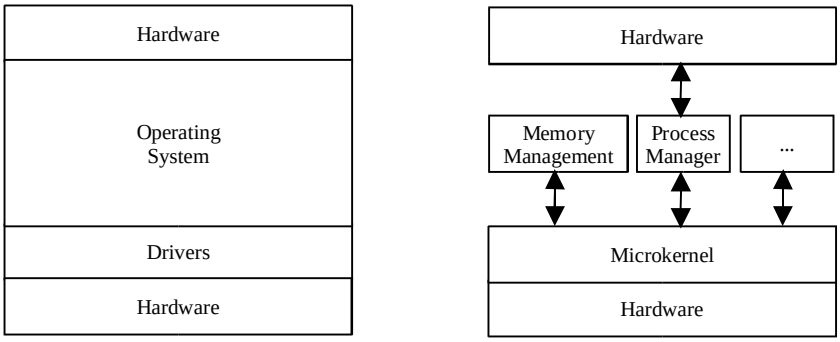
After the BIOS assigns the system to your MBR, the booting process of your system in a case of Linux continues with Linux bootsector (*arch/i386/boot/bootsect.S*), which is a specific bootloader or by letting some other instance for example loadlin to continue the booting process. From these the bootloader gives most intelligence to the boot process. It is a small program installed into the MBR or to the boot sector of the active partition, which allows to boot to any specified point on your computer, giving an option to have multiple number of kernels or other operating systems installed in. The Linux distributions to x86 based computers have two main stream bootloaders LInux LOader, LILO and GRand Unified Bootloader, GRUB. These two and other boot loaders differ in architectures but always when they have finished their job the kernel image is transferred from long-term storage usually hard drive to the memory. It is done by copying the boot sector of the corresponding partition into the RAM or directly copy the kernel image into RAM. The last thing that the bootloader does is to jump to setup.S from where the booting process continues. The memory space used by the loader after this can be used for other applications and it is not needed anymore. [24.]

By the bootloader, the compressed kernel image is now placed at physical address 0x100000 or 0x10000 depending whether the created compressed kernel image is in a form of zImage or big kernel, bzImage. The execution now continues with the setup.S to get the system data from the BIOS, and place it into the appropriate place in system memory. All tough the BIOS has initialized the hardware devices, Linux reinitializes these devices with its own manner. After this the setup, in a case of image loaded into low RAM, it moves to physical address 0x10000 and jumps to this address which is the head of the compressed kernel. This sets up the stack and calls the decompression routine. This is done so that the space reserved by the initialization of the hardware can be used as a temporary buffer following the kernel image in RAM. This process is invoked through a first *startup_32()* function. The uncompressed kernel is placed to the final position from where the execution continues with the second *startup_32()* function included in the head.S. This start the high-level environment initialization by creating the first process 0 to set up the Kernel mode stack setup. The function will also initialize interrupt descriptor table, save system parameters and detect the processor model. At the end it jumps to init/main.c and in there the *start_kernel()* function which completes the initialization of the Linux kernel by calling different external functions defined in the appropriate kernel subsystem. [1, p. 578-580.]

3.2 Linux Kernel Architecture

Linux kernel like most commercial Unix variants is monolithic, which means that it enables you to have all operating system services running with in the privileged mode of the processor. The privileged mode services are said to be running on a Kernel mode where the actual kernel code is loaded and where memory is allocated for kernel-level operations. In the contrast the applications run on a user mode and are isolated from the operating system. When the user mode application calls a system services through a limited set of interfaces, the processor traps the call and makes to kernel level service. Generally this makes the kernel simpler and faster because it does not have to switch between privileged to non-privileged mode. The problem with a true monolithic system is that the whole kernel has to be compiled together in a single executable, which makes expansion of the kernel more difficult. [8, p. 37-38.]

A microkernel architecture approaches this problem by implementing a smaller set of operations in a more limited form. It runs several system processes on the top of the microkernel, which handles the other operating system-layer functions like file, memory and process manager. This attends to be less hardware specific as many of the system specifics are pushed into user space but slower as it requires more message passing between the different layers. The microkernel version also attended to have more complex structure than in monolithic ones. [29.]



Picture 3-1. Monolithic kernel versus Microkernel. The structure of the Linux kernel can also been seen from the picture 3-3.

In 1991, when the Linux kernel was released, the the academic research was oriented to microkernel approach of operating systems. As Linus was designing the kernel structure he noticed the advantages of modularized approach where all operating system layers are relatively independent and the microkernel includes all the hardware dependencies. Linus Torvalds

introduced modules to the monolithic kernel structure, which still enabled the monolithic roots to exist. Modules brings the theoretical advantages of microkernel to the monolithic Linux kernel. This is done by allowing modules to be compiled as normal object files, but the actual linking is not done into the kernel image until the runtime. The loaded module or drivers act exactly as it would have been compiled directly to the kernel image, until they are unloaded. Linux can be said to be an hybrid monolithic kernel. This approach still enables the kernel to be compiled as an true monolithic kernel by not compiling anything as a module and by not enabling the *CONFIG_MODVERSIONS* support. [29.]

One other advantage that was implemented with the microkernel approach, was the easy portability. Originally the idea with Linux was not to be a portable operating system toward the hardware, as it was with the applications. In a true monolithic kernel approach the code includes also the low-level interaction with the hardware so the kernel appear to be specific to a particular architecture. A microkernel performs a much smaller set of operations, abstracting all the details so that a port to another platform would require only minimal changes. Torvalds noticed that in order for the code to be portable, the abstraction layer was not necessarily needed. Instead the Linux kernel structure is layered, so all the hardware dependencies can be implemented to the lowest level of the kernel. [30.]

3.2.1 Processes

Linux is a multitasking operating system which means that it allows many programs to run at once by switching the CPU between several tasks. When a program is executed it generates an instance which is called a process. Each of these processes is indicated with an unique process identifier called PID, which is assigned positive number usually between 2 and 32768. After this the Linux starts to recycle the lower unused PID's. The first PID is reserved for the init process which is in charge of managing other processes. Many times the PID is relevant in system administration who may have to debug or terminate processes by referencing the PID. The difference between programs and processes should be noticed. Several processes can execute the same program simultaneously, while the same process can execute several programs one after another. Each process also has a user, uid and group identifiers, gid. They are used to control the processes access to the files and devices in the system. [1, p. 68.]

The Kernel in multitasking environment is to one that is responsible for the management of different processes. The Multitasking is transparent to user processes. Each different process can act as it is the only process on the computer, with exclusive use of main memory and other hardware resources. The processes in monolithic systems are also written in such away that they do not have any knowledge of the underlaying physical hardware, so the kernel is responsible for giving a fair share of access to hardware resources.

In order for the kernel to handle the different processes, it has to have a clear picture of the current states of different processes. The kernel maintains information about each process in a process descriptor in a file *include/linux/sched.h* under the *task_struct.* It contains all the information related to a single process containing information from the processes priority, process identification, the current run-state of the process, address space, the amount of CPU time the task has been used and so on. [40.]

A process in multitasking environment can be in several stages during the execution. The stages in Linux are Running, Ready, Waiting, Stopped and Zombie. A task is ready when it can execute, but does not have high enough priority to do so. The running process is the one controlling the CPU at the time being. A waiting task waits for an event or for a resource to be released. When process is stopped when it receives a signal from the third party. This could be case for example when the process is being debugged, it can be in a stopped state. A zombie process is a halted process which still has a *task_struct* data structure and continues to reside along with its associated process in its own virtual address space. This process is still consuming the memory resources and is also known as dead process. [2, p. 39, 41.]

3.2.1.1 Scheduling algorithm under Linux

The most central subsystem in Linux kernel is the process scheduler. It uses a reasonably simple priority based scheduling algorithm to chooses which process will have the current access to the CPU. Switching from one process to another, in Linux, is done in a very small time-interval. This way it seems that the kernel can run several processes simultaneously. To determine which process will run next is done by the means of scheduler. Scheduler under Linux is based on time-sharing technique where CPU time is divided into turns called time slices or quantum's. If a process is not terminated when its time slice expires a timer interrupt is launched and a process switch may take a place.

The Linux scheduling algorithm priority based. Each task is assigned with a priority, which is the value used for determine which process is to be executed by the CPU. Linux introduces two kinds of priorities, static and dynamic. The most of given priorities are dynamic so the priority of the task can be changed during the application's execution. This way the kernel can adjust the priority of a process that have been denied to use of the CPU for a long time period and correspondingly it can lower the priority if a process that has been running a longer time. Linux can also assign a static priority to a process, which is the case with real-time processes. The static real-time processes are always with higher priority than the dynamic ones so the scheduler will start running the processes with dynamic priorities only when there is no real-time process in running state. [1, p. 281-282.]

The scheduler under the Linux kernel seeks for the process with highest priority in the queue of the run state processes. The real-time processes are always executed first. This is ensured by giving the normal process an priority of counter plus 1000. When the processes are assigned with an equal priority, the kernel uses round robin algorithm by choosing the process nearest to the front of the queue. If the kernel notices a process that runs on a higher priority than the executed one, the current process must be suspended and the higher priority process is executed. This is known as preemptive scheduling. When the kernel starts to execute a different process, it saves the values corresponding the current task in a stack and restores the values of the next process to the CPU's register and continues the execution of that process. The saved values are stored in the processes *task_struct* data structure. [33.]

All tough tasks are continuously interrupted as the Linux kernel switches between different processes to share the system resources, the interrupts still cannot happen at any time. Under Linux the interrupts can be disabled by a process. This for example is the case under the system call. This method is called a priority inversion and it disables the case that the higher priority process can take over the CPU at any time. This is also one major reason why Linux is not a real-time operating system.

3.2.2 Reentrant kernel

Linux kernel is reentrant, meaning that several processes can be executed, or at least block the kernel at the same time. The Linux provides this by introducing reentrant functions and a set of locking mechanism to prevent from several processes executing an non-reentrant function at the time. A reentrant kernel function is a function that can simultaneously executed by several

processes without fear of data corruption. This is done by using only local variables or protecting the data when global variables are used. It ensures that the function can be interrupted and resumed at any time. Kernel has also methods of protecting non-reentrant functions with disabling interrupts during critical regions, using a spin lock to control access to data structures or control the access through semaphores [1, p. 21-22.]

For programmers view the reentrant kernel should be noted when writing kernel modules. All the kernel code should be reentrant. For programmers point of view this means that when writing kernel level code, it should be capable of handle more than one simultaneous accesses. Also other shared data should be accessed by the code in such away that it won't corrupt the used data. An hardware interrupt in Linux kernel is also able to suspend the current process even if it exists in a kernel mode. This capability significantly increases the throughput of the device controller that issues the interrupt. [1, p. 21-22.]

3.2.3 Inter Process Communication

Processes in Linux communicate with each other and with the kernel via a mechanisms called Interprocess Communication, IPC. It enables the process to send and receive messages from other processes, share memory area with other processes and synchronize itself with other processes. Linux supports number of IPC mechanisms like pipes and signals but it also supports many of the Unix System V IPC mechanisms.

A Signal is a very short message allowing the process to communicate the occurrence of asynchronous events to other process. It is the oldest inter process communication method used by Unix systems. A signal could be generated by a software / hardware interrupt or by some error condition - for example process attempting access an invalid location in its virtual memory. Linux includes a set of defined signals with a prefix of signal that the kernel can generate or that can be generated by other processes with correct privileges in the system. A set of system calls are also introduced by the Linux kernel, which allow programmers to send and determine how the signal should be handled. The options for process to handle the signal are to block the signals or either choose to handle them them selfs or allow the kernel to handle them with a default action. Process is also in some cases allowed to ignore the signal. [32.]

When sending a signal to an other process, the kernel updates the descriptor of the destination process, which reads it when it enters to running stage. The process running can also choose to

pend for a signal by entering to suspended mode until the received signal raises an interrupt and wakes it up. The signal sending has also some restrictions where the normal process can only be send to a process with a same uid and gid or to processes in the same process group. Of course the kernel and the super user are able of sending signals to every other process. [32.]

As signals are only able of transferring information that was limited to a single number, pipes allow more useful data to be exchanged between the processes. The term pipe is used to describe the connection where the information flow can be done, between the processes. A command shell pipe can easily be done with a | separator,

ls | *lpr*

This command line pipes the output from the *ls* command listing into the standard input of the *lpr* command which prints the results on the default printer. Basically the idea with pipe command implementation is that both data structures point at the same temporary virtual file system inode, which is assigned itself to points at a physical page in memory. As the *ls* writes to the pipe, bytes are copied into the shared data page and the *lpr* copies these bytes again from the same shared data page. [5, p. 405 - 406.]

3.2.3.1 System V IPC

The System V IPC Mechanisms provides forms of IPC facilities originally introduced by Columbus Unix variant but later adopted by AT&T. The System V IPC provides tools for performing reliable, system-wide communication. It supports the three kinds of different mechanisms: message queues, semaphores and shared memory, which are all implemented via kernel system call. The user mode process can access these system calls by passing an unique identifier to the resource, which is used to identify an System V IPC object. When creating an object, it remains in the system memory until it is explicitly removed. The client process willing to access to the object has to obtain the objects identifier. Each object created has an unique IPC key associated with it, which the process has to obtain in order to construct a reference to the object. [32.]

A System V facility queue sends raw data between processes on the same machine. When an IPC message is sent, it is placed to the message queue where it will stay until the receiving part reads it. When a parent process constructs a message it places there an identifier of the destination queue, size of the text and specific type of the message agreed between the cooperating processes followed be the message it self. Before the message is placed in the

queue, the messages user and group identifiers are compared with the queue's mode to see if the message is allowed to be sent. If allowed, the messages is placed into a dynamically allocated memory, at the end of this message queue. When reading the message, the processes access rights to the queue are checked. The retrieved message may be chosen to either get the first message in the queue regardless of its type or select messages with particular types. If no match to this criteria is not found, the process will be added to the message queue's read wait queue to wait until the type of message is received. This can be done by linking all the messages to the next as an linked list. [1, p. 545 - 548.]

A semaphore is an object that manages one or more shared resources. They control access to shared data structures for multiple processes. The semaphore has an positive value if it is accessible and negative or zero if some other process is currently using it. When process wants to access the shared resource it would try to decease the semaphore's value and if the original value is positive it is allowed to use the resource. Otherwise, if the resource is reserved, the process will be suspended until the resource allocating process is done with it and increment the semaphore's value back to 1. Each semaphore object in system V IPC describe a semaphore array, which can hold several independent shared data structures. Each process with right privileges may make system calls that perform operations on them. When inquiring the resource protected by a semaphore, the process invokes the identifier of the semaphore with the corresponding IPC key. All the semaphores in the array are then tested for an access and if succeed the decrements are performed. The kernel applies the operations to the appropriate members of the semaphore array. After this it will check if it has suspended processes which may now apply their semaphore operations. [1, p. 540 - 541.]

The deadlock is a situation where two processes are each unknowingly waiting for resources held by the other. This could happen when a process dies without being able to release the semaphore. Linux protects these situations with an fail-safe mechanism by maintaining lists of adjustments to the semaphore arrays. If the process unexpectedly dies, it is able to return all the values from the starting point. [2, p. 58.]

Shared memory allows one ore more processes to access a common data structures by placing them in to a memory segment shared by all of them. The pages of the shared memory is referenced by page table entries in each of the sharing processes page table. Access to the shared memory is controlled via keys and access right checkings. Once the process have access, there is no control of how the processes are using it so to do this they can only relay on other methods like semaphores. The object creator has the ability to define the access rights to the

region and also if it is assigned with high enough privileges, it can lock the shared memory into physical memory. The process willing to use the shared memory space can do this by using a specific system call. When the process no longer needs the shared region it will detach from it and the processes page frame is updated to invalidate the area of virtual memory. Still if other processes are using the region, the detachment will only affect to the current process until the last process detaches from it. [32.]

3.2.4 Threads under Linux

When creating a new process or performing an context switch, in order to change to an another process, a lot of overhead is created to the CPU. Linux provides an mechanism to lower this overhead by introducing threads. They allow multiple strands of execution in a single program. Each process in Linux is basically an address space and an one thread of control. When a new thread is created in a process, the new thread of execution gets its own stack but the execution will still happen in a same address space, which means that they share the access to the code and to the global data. An good example of thread using is an process using input output processing with separate threads where the global data can be read and write by either one of the threads. The drawback with the threads is that designing and debugging multi thread programs is much harder. Still in a single CPU machine a well designed thread can reduce the amount of work that the operating system has to perform, this because less context related data must be saved. For this reason threads are often known as light weight processes as in a contrary the normal processes are known as heavyweight processes. The major part of Linux distributions support the IEEE POSIX 1003.1c threads or Pthreads, which are very portable and supported also by most Unix vendors. [38.]

Linux introduces also kernel threads, which are designed to run only in kernel mode and do not interact with the user. These are some critical tasks that require to be executed immediately like swapping out unused page frames or flushing disk caches. The Kernel threads are designed to run only in the kernel mode by each of these threads executing an single kernel function. This also enables that the thread does not have to make the context switch between user and the kernel mode. Although this reduces the little of the overhead, the over all time beaten is rather small. In Linux the kernel threads are used in a very limited way to execute a few kernel functions. [1, p. 94.]

3.2.5 Memory management in general purpose Linux

The different processes all require part of available memory from the system. Most often a computer offers some continuous space of physical RAM and a separate memory management unit MMU that distributes this memory over the different tasks. The MMU maps virtual pages to physical page frames, which allows processes to have their own virtual address spaces. Virtual memory allows this address space to be larger than the available physical memory and protects this space from other processes.

The physical memory, RAM in Linux is divided into a two parts. The first block is permanently assigned to kernel image. This contains the kernel code and the static data structures. The remaining part is called dynamic memory. This part is used not only by the processes, but also by the kernel. In the processes case the block is used for request of generic memory areas. The kernel uses this block for dynamic kernel structures like buffers and file descriptors. One of the major issue concerning the whole system is how efficiently the dynamic memory area is managed by the operating system. [9.]

The main idea behind the virtual memory is to have a secondary memory big enough to store the complete program. Also there has to be a way that changes made to a program in main physical memory are also reflected in to the original one. The available physical memory, RAM is divided into fixed-length partitions called page frames. In the same way the virtual memory is partitioned into equal size pages, each piece being the same size as the page frames. When the program points to address that is not located in the physical memory, the contents of the main memory is stored in to the secondary memory, the memory area is located and loaded to the main memory and before continuing and the address maps are changed. This technique is called paging and done by the operating system so to user it is invisible. A memory map is the one that relates the virtual memory addresses to physical ones. [7, p.324.]

3.2.6 File systems

Linux environment very file oriented operating system. Almost every service and device can be accessed trough a file, which means that devices like printer, serial ports and disks can be used as a regular file, which also makes programmers job more easier. Linux also provides an access rights settings for files and folders, which can be set separately for read, write and execute. This allows protection for system files and can limit the access for certain devices. The files under

the Linux system are all placed hierarchically in directories where the root directory contains all of the system files under the subdirectories. The Linux source tree denote directory names containing information about the files beneath it. Usually Linux contains at least directories for executable programs *./bin*, directory for physical devices *./dev*, a directory for configuration files *./etc* and a specific directory for libraries *./lib*.

For accessing the devices and services that the files represent are done through the system calls, which are explained later in this section. The file system that the user sees is actually a user-level view of the physical organization of the hard drive. When interacting with these low-level components the actual operation is to done by the Kernel Mode, trough a system call. It provides methods for file-system calls like *open*, *close* and *ioctl*, which passes the control information to a device driver.

Linux also provides an higher level interface services and devices. This is done through a standard libraries, which include a set of functions that can be used used instead of the actual system calls. They are much more flexible than the actual system calls, providing for example a buffered output. This way the programmers does not have to take care varying size of data blocks and the overhead of the system call is minimized. [5, p. 93.]

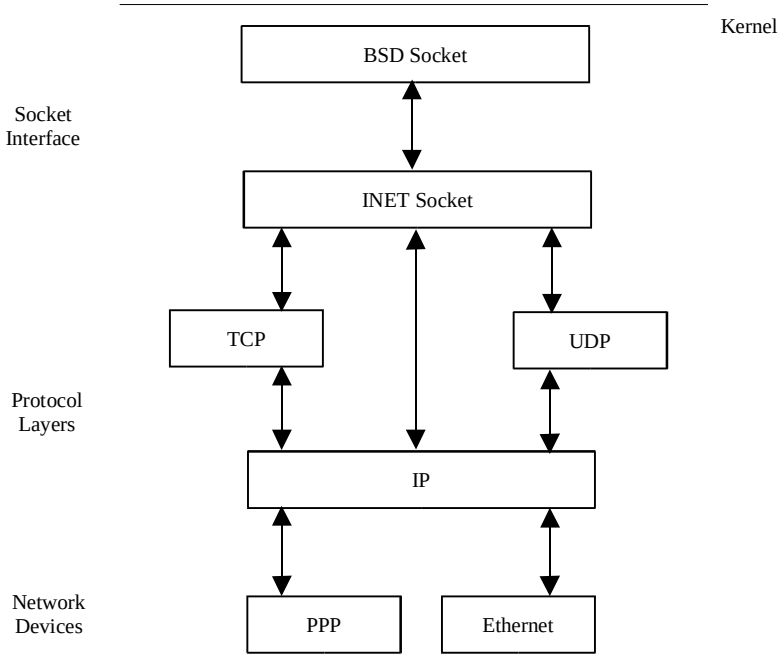
One major advantage of Linux is that it supports various of different file systems, which makes it exceedingly flexible and allow easy access to other operating systems file systems. Nowadays Linux supports various of different file systems from different platforms, including msdos and vfat from windows, Mac OS file system and even file system from Amiga. Linux allows these file formats to be transparently mounted to Linux system through a common Virtual File System, VFS. The VFS is actually a kernel software layer providing all system calls related to a standard Unix file system. Supported file systems are invoked trough this software layer so that all file systems appear identical to the rest of the Linux kernel and to the programs running in the Linux system. The function of VFS is totally transparent to the end user, after the user has mounted the accessed disk or partition as part of the source tree file system to Linux. After the file system operation is made, the kernel actually calls functions defined in VFS-interface, which takes care of non-file system dependent operations and forward the call to appropriate file systems function. [1, p. 328-329.]

The VFS has also to take care of the modifications that the user might do to the files on a mounted file system. The write and read commands to the different file systems are done through specified device driver for the different file systems. As the real file systems reads data

from the physical disk, the block device driver is used to read physical blocks from the device that they control. The files read from this block are then saved into a buffer cache shared by all of the file systems and the Linux kernel. Here it is buffered with an identifier according to their block number and a unique identifier for the device that read it. This enables the data access to the same data without needing to retrieve it each time separately. [32.]

3.2.7 Networking interface

Linux has an strong networking support, supporting many different networking protocols including the most common TCP/UDP, IP, and Ethernet. The networking messaging system is structured with several different layers, each of the layers using the services of another, with Internet Protocol, IP being the heart of it. When an application generates network traffic, it sends packets through the socket layers to a transport layer (TCP or UDP) which forward them to IP layer. In the IP layer the kernel resolves route to the host. If the packet is meant to a other host, it the IP layer sends it to a ethernet device and out over the physical medium. The structure of the Linux network layers can be seen from the picture 3-2. [32.]



Picture 3-2. The structured networking system of Linux.[32.]

Two main types of socket implementations are provided in Linux by BSD and INET sockets. The BSD sockets are implemented using the INET socket layer and provide the interface that the user programs can use. The purpose of the BSD socket is to provide greater portability to the user applications by providing communication details to a common interface. The BSD socket interface is now the de facto industry standard programming interface. The socket interface basically works as an pipe, but provides possibility to communicate over the network. Socket is a communication mechanism developed between a client and a server system where a client or clients connect to the defined server. If the client is accepted by the server, a socket between these two nodes is created enabling a two-way data communication. Linux supports several different socket domains with most common UNIX for internal and INET for TCP/IP networking communication. There are also several socket types representing the type of service that supports the connection. The possible values include stream sockets which are supported by the TCP protocol of the Internet address family and a datagram socket, supported by the UDP protocol. [5, p. 488, 491.]

3.3 Programming under Linux

The source code of Linux is released under the terms of GPL. It was designed to ensure that the companies and individuals cannot make changes to the code unless they are willing to share it. This license also means that the developed code under these rules are also to be available and open for the competitors. The other license concerning the development done under the Linux is LGPL. The idea is that company may release only the binaries of the code developed to run under the Linux, if it is dynamically linked to the libraries, like glibc under this license. So if the libraries require modification in order to run the code, the changes and the original source of the library has to available. The actual source code can still be released as a binary if the linking is done dynamically. Usually the easiest way to release both of these version is to patch our changes against your modifications and release them. Still it should always be remembered that Linux would not be available in this amplitude if it would not use these licenses. [4, p. xxiv.]

Like most UNIX like operating systems, also Linux attempts to be IEEE POSIX compliant. It specifies the common application programming interface, API on which the user applications can be build on. These APIs are general an supported by many other operating systems and makes an easier to write source code that can be compiled on different POSIX compliant systems. The POSIX also defines rules for the Linux application and kernel by defining how the application obtains the basic services. The API definition also enables that it does not set any

rules on internal design for the kernel. [5, p. 13-14.]

By it self the Linux kernel cannot do much. For using Linux as an fully working operating system, a set of utilities has to compiled with it. Richard Stallman from the free software foundation had developed Unix like tools set under and licensed as open source, under the GPL license. The tools created with ANSI C included all the standard UNIX commands and utilities, but the main goal for Stallman was also to create a free Unix-like system, called GNU. At the same time Linus had made Linux freely available which eventually was merged with the tool set to make up the bulk of a Linux distribution. The Linux version 0.02 included the ability to work with bash, GNUs Bourne Again Shell and GCC, the GNU C compiler. Linux has also many other packages merged to it after a time. The networking utilities and daemons came from Berkeley UNIX and X window system that is a standard GUI for UNIX systems came from MIT. [34.]

Both Ansi-C and Ansi-C++ are supported by the Gnu C Compiler, GCC. The GCC can be used for many different platforms and also for cross platform development and is included to most standard Linux distributions. Standard Linux distributions also included an ability to use script languages as command language interpreters. Linux supports also a large amount of different programming languages like Perl, Java and Lisp that can be included to the Linux system.

3.3.1 Libraries

The programs running in the Linux are stored on a disk as an executables. These executables will perform the functions defined in the source code. Many functions in the executable are actually an service routines, which are defined in special files called libraries. To include the library to the executable, can either be statically copied in the executable file or be linked at the run time. With an static linking a copy of the library's code is linked into every program that uses it.

The disadvantage with static libraries is that programs using the same libraries may end up with multiple copies of the same functions in the memory. This of course is very memory and disk space consuming. As an solution for Linux, like many other Unix like kernels provide the ability to use shared libraries. When a program uses a shared library, the executable will not contain the actual library but an reference to it. When the program is loaded in to the memory a program called program interpreter takes care of locating the library and makes the requested

code available. This will do a slight performance penalty when a shared executable starts up and the references are solved, but the references need be resolved only once so the performance penalty is quite small. Shared libraries provide also an advantage when a library is updated. In this case updating the library will affect to all of the executables using it. The shared libraries can be recognized from the extension *.so* where the so stands for shared object. [5, p. 22.]

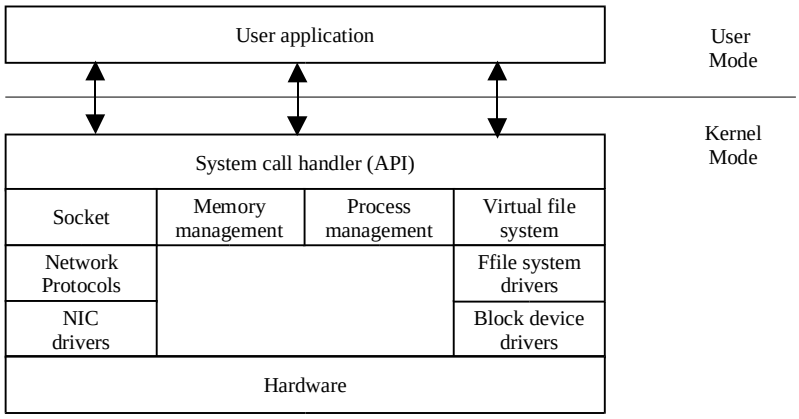
3.3.2 System call

When running a Linux system, the main task for kernel is to provide an execution environment for different applications to run on. These applications attend to do different tasks, each of them requiring for system resources from the system. Instead of putting code to manage the hardware controllers in the system into every application, the interfacing code is kept and compiled with the kernel. For allowing different processes for accessing the physical hardware the kernel provides an well-defined set of internal programming interfaces, which hides completely the complexity of communicating with the physical hardware. This way the kernel provides an easy interface for programmers to use, when using the hardware devices and increases the system security by handling the requests by pre-defined format. Services for accessing the hardware is CPU dependent and usually the hardware introduces at least two different execution modes for the system: a non-privileged or the user mode and the privileged or the kernel mode.

When an application wants to access a hardware device, it does this via a specific request to the operating system. This request is evaluated by the kernel and if accepted, the kernel interacts with the hardware instead of the user program. After the service provided by the kernel is done, returns the program back to user mode. This service provided by the kernel is called a system call. Most of the time the process runs on the top of the kernel, on the user mode and switches to kernel mode when requesting a service provided by the kernel.

When a system call takes a place, the process fills the registers with a group of parameters that identifies that specific call and then executes the hardware-dependent CPU instructions to switch from user mode to kernel mode. Under Linux the system calls are invoked by the means of interrupt 0x80 from where the process jumps to system call handler. Since the kernel implements multiple system calls, it has to be aware of what service the process is requesting. All of the system calls must pass a specific parameter called system call number to identify the call to inform the kernel which call is invoked. The function *system_call()* is the one that implements the system call handler. It saves the system call number and the contents of most

registers in the Kernel Mode stack invokes a corresponding system call service routine and finally exits the handler by the means of *ret_from_sys_call()* function. [1, p. 234-235.]



Picture 3-3. System call with Linux kernel.

3.3.3 Device drivers

To fulfill the demands of the kernel calls that the different applications do through system calls are provided by the device drivers. The device drivers provides an actual interface to the physical hardware and provides an ability to present an uniform interface to applications. Writing device drivers is basically a same kind of a task than writing an regular application. The device driver should be able to map the different user needs to the underlaying hardware. As Linux provides the whole kernel code freely available, the device driver designer should look and reuse the already done code, as they provide usually convenient solutions for different problems. As the device driver will function at the kernel level, a certain issues should be taken under concern. The device driver code should never have busy wait loops. If the device driver is designed to run a loop in a kernel mode, the kernel will appear to hang the whole system. Also the driver should only implement an mapping from the applications to the physical device. The issue of how to use the hardware is up to the application. The driver should also be reentrant, as many different user applications may use the device simultaneously. [6, p. 2-5.]

Linux supports three types of peripheral drivers; character, block and network device drivers. Although other classes of driver exist in Linux, these are the biggest ones. A block device driver is accessed by the user application through an system buffer. These drivers serves the data to an end physical device in a form of fixed size block typically 512 or 1024 bytes, which buffers the data before it is written to the device. This allows random access to a device. The

other way is an direct access to an device. In character devices read and written can be done directly without any buffering. These drivers usually provides an *read*, *write*, *open* and *close* system calls which allows to invoke the device. The third type is a network interfaces provides any transaction made toward the network. These devices are able to exchange the files with an other host locally or over the network. This physical or software interface is responsible of sending and receiving data packets. [6, p. 2-5.]

3.4 Compiling the Linux Kernel

As pointed out earlier Linux provides support for multiple different configuration and processor choises. When compiling the Kernel, the user can configure the choices that are to be supported by it. This is done by enabling / disabling certain options which build up a makefile from which the kernel is compiled from. To configuration to Linux kernel is done by making a configuration script that eventualy builds the kernel. This can be executed with the command "*make config*". The "*make config*" command needs a bash to work. Which is a command language interpreter that can execute commands from a file. Linux also provides some helpful machanism to do the configuration in more 'user friendly' way. These alternatives are

make oldconfig	- when you want to carry your existing confiration to a new version. Here only non-existing configarations are asked . Old configurations are red from the *./.config* file
make menuconfig	- text based colored configuration with possibility to mark options
make xconfig	- is a X-window based option for configuration

All of these options will lead to a same conclusion. The choices done by the user will effect to the size of the kernel, depending from included drivers and support functions that are included. This is why all the options should always be selected to prior to the platform and processor that you are using.

Next state is to check and the dependencies, with the command "*make dep*". This will check each C file in the source and figure out to which header file it depends on. With older versions of the Kernel, after the dependencies, you should clean up before actually building the Kernel. With "*make clean*" command you can remove all the object files and some other files that are

left behind. [28.]

In order to reboot your kernel, the user will have to have a copy of the kernel image in a proper place. The Linux kernel compilation is done with the "*make bzImage*" command, which eventually will generate an compressed image of the Kernel. The compressed kernel image resides in *./arch/i386/boot/bzImage*. In older versions of Linux the compression was done with the command "*make zImage*". This command is still available , but cannot be used because it can not handle the code size of new Kernels. The detailed steps taken in a kernel compilation are introduced in appendix A.

Chapter 4

EMBEDDED SYSTEMS

By definition 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 table 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.

4.1 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 an stand alone approach 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. 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 today 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 todays 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 an 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. [12.]

4.2 Embedded computing

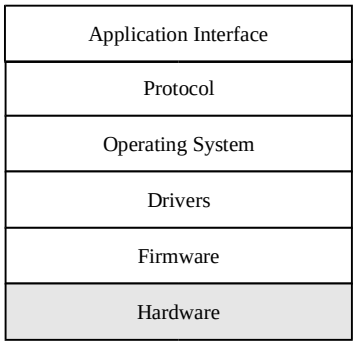
As components like processor and memory has become less expensive are embedded devices integrated in several different kinds of devices. This also means that we are able to use more processing power and memory on these devices, so they are becoming much more complex, providing more functionalities. The 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. An more intelligent devices requires more intelligent operating system to manage all the several functions that they provide. The operating systems for these devices are many times derivated from a general purpose operating system , so the barrier between embedded and general purpose operating system can be inconstant.

The concept of operating system is not that clear. Usually the operating 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 API for the programmer. Operating system also provides an resource management for different applications running on that device by handling interrupts and allocating memory for applications. Operating system also gives an advantage to the programmer, as it provides some kind pf protection against programming failures for low level devices and gives an foundation where programs can run on. [8, p. 3-4.]

4.2.1 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 size that the general purpose OS takes. 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 tough embedded devices are designed to be beneficial solutions for specific task the barrier between these two systems 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 started to develop operating system specially designed for these embedded devices demands. 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 an substantial efforts from major companies have been made to enter these markets, also with non Open Source distributions. Still regardless of the embedded OS all embedded systems are an entirety of several different layers of software providing drivers for the hardware and an interface to the end user. All of the embedded system can be inspected with the following different blocks.



Picture 4-1. Embedded system structure.

The Linux is most popular UNIX-like clone 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 a embedded devices and there exists many companies providing commercial and non-commercial embedded Linux solutions. When obtaining an embedded Linux distribution it usually includes an toolchain, some ported applications, ported libraries

and of course the actual kernel. The toolchain enables you to build applications or to create the image for an embedded device. Most of these are based on a standard GNU toolchain, 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 that solution that can be compiled to image. [11, p. 4.]

The biggest problem with Linux in commercial usage is that the 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 GPL. Some companies providing Linux distributions have even provided specially designed tools to check which parts of the software have GPL violations. One example of these companies is Lineo's GPL Compliance Tool. More problematic license in embedded devices is 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 against the libraries. This means that the source code has to be available to all parts. [39.]

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 solutions, more 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 start to eliminate the unnecessary functionalities that are not required from 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 against of the demands of the embedded device. Advantage if this system is that the Linux applications can be ported rather easily to this platform. Examples of these kind of solution are AXIS and BlueCat Linux. [12, p. 12.]

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 also an real-time extension to the kernel which enables to 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.

4.2.2 Managing embedded devices

Most common physical interfaces in embedded devices are serial and ethernet interface. These are also in many times the only interfaces which allow physical management to be done to the device without any user interfaces like display or keyboard. From these the serial is basically 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 an 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 acting as of connectivity interface to end device, but in embedded devices it is still quite rare.

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 various of different applications that can be used to connect and control the devices. The next chapters introduces 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 to a embedded device. The telnet daemon runs on a embedded device allowing to transmit the keystrokes from the remote host to the target embedded device and displays the resulting screen to back to the remote host. SSH is a more secure version for remote connectivity. The idea is the same as in telnet, but provide 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 a embedded device. This way we can easily place files in a 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 a 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 an web configurator, which makes it an powerful tool. The GUI of the HTTP page can be used as a virtual interface to control the device itself and the end system that it is connected to. 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 an 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 an small burden to the network. As it is well known, it also has an extensive range of tool support. The agent of a SNMP protocol is running as an server in each of the monitored or managed device. It provides an interface to each of these nodes by providing an data structure called a management information base, MIB. SNMP provides also an ability to easily manage all the devices distributed to the field by running a 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 card to act as an proxy toward the end system by holding a specially designed MIB for that task. The agent end is also able to send asynchronous traps to the end station when a predefined event occurs. [15.]

4.3 Hardware

The basic architecture in 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 solution, 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 an 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 in table PC's. [4, p. 60.] The processors in embedded devices has two major alignments. Ones holding the Memory Management Unit, MMU and others that are designed without it. The advantages gained from the MMU is 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 an cheaper solution for designing the 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 device in industrial surrounding so a failure in embedded device might mean that the controlled machine is damaged. This requires an thorough testing from the hardware and the software side. The cost efficiency is also an key feature, as the devices are usually distributed to a field, a small change in the price affects significantly to the end purchasing price.

4.3.1 Example hardware setting

As an hardware example we take a closer look to Viola Systems Arctic development board and more closely its processor module. Arctic board is a multi board layout based on Motorola's MCF5272 Coldfire Processor. It is designed to three separate boards, which all have different functionality. The different modules in the Arctic are:

Connector board (I/O board)
Carrier board
Processor module

The three board setting allows that customer specific tailoring can easily be made without designing the whole hardware layout again.

From these modules, the connector board is most modular. It can vary among the different customer requirements. Therefore it is designed to be easily changed, but also still offering some basic I/O board layout options. The changes in the board are mainly made to the input / outputs, which allow different number of digital and analog interfaces to be implemented. Connector board has also an optional temperature sensor availability. The connector board is attached to the carrier board via and Viola DIMM connector or with an SPI bus.

The carrier board varies between the different product releases, which can be made with different assemble options. The default interfaces in the Arctic are two external and internal serial interfaces, physical ethernet connector and internal and external USB's. The Carrier board has also an option to be compiled with an external GPRS module, connected to the processors internal serial interface and an external wireless local area network and blue tooth chip, connected true an internal USB. The board has also an option to get the power from an external battery and to give an output voltage level of 7 to 24 voltages through an chopper.

The main technical features of Arctic processor module and other boards are listed in the next paragraphs. The main components and the architecture of the processor module can be seen from the appendix B.

The processor in Arctic is the Motorola's MCF5272 microprocessor. It belongs to a ColdFire Family and was first released in 1994, developed specially to meet the embedded markets. The goal of the architecture was to design a low-cost processor with high performance, debug support and code compatibility with m68k. The Linux/m68k is a port of Linux for the Motorola's m68k processor family with full source compatible with other Linux platforms. The MCF5272 microprocessor is a MMU-less, 32-bit processor with main features of:

- Includes two internal universal asynchronous/synchronous receiver transmitters (UART's).
- Ethernet module with on board transmitter and receiver FIFO's.
- Internal Universal serial bus (USB) module.
- Queued serial peripheral interface (SPI)

The operating frequency of the processor can go up to 66 MHz, however on the Arctic processor module the frequency is set to 48 MHz operating clock, which allows the same oscillator to be used with the external USB and UART chips.

The processor module provides 8MB of SDRAM and 4MB Flash ROM. These are configured to use 16 bit wide bus. The Flash ROM is connected to processors programmable chip select

output that is active after to reset. The Flash ROM device on the Arctic is an Fujitsu MBM29DL323E 32-bit bottom boot sector device with an size of 4MB that is divided to 71 different sectors. The memory layout of the Flash is shown in the table 4-1.

Table 4-1. The memory layout of the Flash chip on Arctic.

Address range	***Function***
0xFFC00000-0xFFC01FFF	The start up code for the boot. Initializes the chip select signals, SDRAM and serial interface. At the end copies the actual boot loader to the correct place in SDRAM and jumps to there.
0xFFC02000-0xFFC03FFF	Empty.
0xFFC04000-0xFFC05FFF	Environment settings.
0xFFC06000-	
0xFFC0FFFF	Empty
0xFFC10000-0xFFC1FFFF	Boot loader code. Copied to the SDRAM during the initialization.
0xFFC40000-	
0xFFC4FFFF	Space for system image.
0xFFFF0000-0xFFFFFFFF	Space for system image.

The Arctic has also an external physical serial Flash chip, AT45DB011 and an EEPROM chip, which both are connected to the processor through an the SPI channel. The EEPROM is designed to store the manufacturing information and board settings, as the serial Flash can be used to store variables that are also to be saved after the reset of the board.

Motorola's MCF527 microprocessor provides an internal USB and two full-duplex (can send and receive data in both direction simultaneously) UART's with on-chip baud generators. In Arctic these lines are used for internal interfaces, controlling the different modules on the carrier board. As external physical connectors, Arctic provides the ability to use an ethernet, serial and an additional USB. The ethernet Controller on MCF5272 is an on board Ethernet controller operating at 10M bit/sec or 100M bits/sec requiring an external physical interface circuit, PHY. As an external UART the processor module provides an dual UART serial controller chip TL16C752BPT. This chip, with the connectors, is used as an serial interface with channel A providing full DTE serial port and channel B an full DCE serial port capabilities. As an external USB interface the Arctic provides an ability to use external ISP1161 circuit, interfacing the processor system bus. This USB host can be used to control in-device equipments.

Other external chips that the processor module provides, are an on-board temperature sensor, LM75, connected to the processor via the I2C bus. It also includes an a real-time clock, provided with an external DS1302 chip. It is used to maintain the time also when the processor has no power available.

The Serial Peripheral Interface, SPI provides an direct connection to a large amount of peripherals. In Arctic processor module it is used for connecting the EEPROM and serial FLASH to the processor. The interface has also an optional place for another temperature sensor. For connecting the carrier board to the processor module is done with a standardized 144 pin Viola DIMM System Connector. This bus will integrate the external 16-bit data bus, 23-bit address bus, chip select, interrupts, SPI, ethernet, and the serial and the USB interfaces.
The microprocessor has 32-bit internal address bus, from where 23-bits are used for Arctic's address bus and 8 bits used for external chip selects. This setting allows direct addressing up to 2^{23} (2 GB).

4.4 Programming in embedded devices

In spite of the usage of embedded device, the design coal is to try to put more computing power using cheaper CPU's. The amount of memory is often limited to few kilobytes so the decision of what are the different applications integrated with the embedded OS are carefully to be taught through. The embedded device, at least on industrial usage, can stand alone for a long period of time so the software and hardware on the device should never fail. This undresses the device from mechanical parts that are more sensitive for failures and require more complex drivers and well tested software to be added with the OS.

Unlike software designed to general purpose PC, the embedded software cannot be used in other embedded devices unless it is modified. This is because embedded devices hardware layout of the boards may differ significantly from each other. Different solutions of operating systems often consists an specific tools for compiling applications specially for that processor or board. Still addresses in memory locations and other offsets of the physical interfaces may vary and needs to be modified. Compilers many times have also options for building the code to several specific processor type.

The code generated for the embedded device is compiled in the host machine, where the development tools exist. This process is called cross-compilation. It means that the host

machine, regularly a normal table PC, holds the resulting tool chain that will eventually the create binaries for the target embedded machine. The main idea is that the compilation can be done in a machine with more capacity than the target platform. After the compilation the whole package including the operating system kernel and the applications are concatenated to a single package which can be loaded to the embedded machine.

When developing software to embedded devices, thorough testing is required. In worts cases the stand alone device can freeze up totally by badly designed and tested software. This is why generated executable should always be first tested in a host machine, to minimize the existence of errors. Usually some modification might be needed after wards, before it can runs on a target machine. The generated code size is limited by the physical memory of the device. Also processor might do some restrictions by not having enough processing power to run several complex programs at the same time. Programming languages used in embedded devices is usually done with assembly as a low level language and with an C as an application level language. Both of these languages allow programmers to access directly to hardware and not not have any really complex data structures.

Usually the tool chain provides also some debugging tools, to easy up developers job. The commercial packages might also include some development studious with GUI included debuggers with multiple options. Example of this is CodeWarriors development studio. Many Linux distributions still relay on GNU tool, and its GNU debugger.

Chapter 5

UCLINUX

The uClinux or micro-controller Linux is the popular variant of mainstream Linux, specially designed for deeply embedded microprocessors without the memory management unit. The MMU-less is quite common factor for low-cost microprocessors, where component prices are crucial. The uClinux is an royalty free and open source solution, aimed to be compatible with general purpose Linux.

This part of the work explains the general scope of the uClinux. It provides an overview of the uClinuc platform and its development tools. The examples and solutions are based on the Arctic platform and to the MCF5272 microprocessor, but can easily be adjusted to other solutions.

5.1 History of uClinux

uClinux project was started in 1997, by a goal to derivate 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 in to a Memory Management Unit -less network controllers ,that could handle the communication between the network and communication system. 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. [17.]

After this start decided Jeff Dionne and Michael Durrant from Lineo to design and build a line of embedded controllers known as uCsimm and uCdimm. At this same time Gerg Ungerer from the same company ported uClinux onto the Motorola's ColdFire platform and designed several systems using this as the base platform. The early focus an cross-platform development of uClinux soon led also ports for other platforms. [17.]

The interest for these small processors were crowing rapidly and led to a number of other software development. One of these was uC-libc library which was designed to replace Linux's libc and glibc libraries into an tiny package. Other improvements were done by SnapGear by adding the binary flat format, bFLT support and by RidgeRun with ELF shared library. [17.]

Originally the uClinux development were 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 late year 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 kernel, but there still continues to be strong interest to the code based on version 2.0 and still changes are also made to this uClinux tree. [17.]

5.2 uClinux architecture

The uClinux is the most popular form of embedded Linux. It is intended for microcontrollers without Memory Management Units, MMUs. Nowadays the kernel supports multiple of different CPU platforms including ColdFire, Axis ETRAX, ARM, Atari 68k and many others. It has grown exponentially as more and more MMU-less chips receive their own ports. The main difference compared to the Linux is the MMU-less and the fact that it is designed to be very compact solution. Like Linux, uClinux also has a strong networking support including a full TCP/IP stack and supports a wide range of different networking protocols. It also has a support for various of different file systems, including the ones specially designed for the embedded solutions. The compatibility of uClinux has been tried to be kept as close to regular Linux as possible. This means that the different applications developed under the GPL may also be adopted to support MMU-less version of Linux, usually with rather small changes. This of course is limited through the physical hardware of the embedded system, due the lack or capability of certain peripherals.

The MMU-less brings some changes to the kernel. All the architecture-generic memory management subsystems for reliance to the hardware are removed from the kernel source tree and the functions are driven from the kernel software itself. This is done in the folder */nommu*, which replaces the directory */mm*. Some other subsystems are also modified to meet the demands of MMU-less processor. Also program loaders with position independent code have been added and a new flat binary object code format has been created. Other program loaders

like ELF, are also modified to support absolute references.

The main advantage that the uClinux kernel offers, compared to Linux kernel, is the size. When compiling the kernel with only the compulsory options with support for processor, file systems and needed character devices the kernel size can be stripped down to a size in about 400 Kb. Still when the compression is done at the boot time, it requires a space of almost one megabyte. With this the smallest realistic size of memory needed is about 2MB, which of course includes the needed applications. [4, p. xxv.] With uClinux the size of the loadable image can be fitted in a footprint of 500 to 900 Kb. From here the actual uClinux kernel takes about < 512 Kb, the kernel including the basic set of networking tools about < 900 Kb and the MCF5272 with default settings goes in 1.2 Mb. [36.]

The uClinux kernel like the regular Linux kernel can be downloaded free without any royalty fees. The kernel also belongs to the GNU GPL, like also all the applications coming with the full distribution package. The package includes also some libraries, licensed under the LGPL. The kernel and a the tool set can be obtained from www.uclinux.org. From here the developer can get the whole distribution package including the uClinux kernel, some libraries and a set of useful ready ported applications. The page also offers a full tool chain that possibles the kernel and user applications compiling. This is also released under the GPL. As bug fixes and new features are added to the distribution, they are immediately after the basic testing released in uClinux pages. For example the uClinux kernel is almost in sync with the basic Linux kernel, and patches can be obtained against the Linux kernel.

5.2.1 uClinux Libraries

uClinux uses an stripped version of the standard C library, which was originally developed along with the uClinux kernel. It is based on the Linux-8086 C Library, but stripped down to a much more compact package. All tough the idea of developing uClibc was to design a space optimized C library for MMU-less microcontrollers such as Dragonball, coldfire and ARM, it still supports standard Linux architectures. uClinux also strictly provides the standard Linux libC APIs, so the developers can migrate applications from POSIX based operating systems to uClinux. It is freely available under the LPGL license. [35.]

Under the uClinux distribution the developer can choose between two libc libraries, uC-libc and uClibc depending from the developers needs. The uClibc is a derivative of uC-libc designed to overcome problems that occurred with uC-libc by making all the API's standard and by filling in many of the missing routines. In general uClibc tries to provide a glibc like library, which also means that most documentation written for functions in glibc also apply to uClibc functions. It also can be compiled as a shared library on most platforms with MMU support. All tough uClibc seems to provide a better platform, many times the older uC-libc provides an more convenient solutions as it may be a little smaller and provides an more tested support for shared libraries. Still as the uClibc provides standard compliant library, it may be easier to use if the goal is to port new applications to the platform. [41.]

The uClinux package includes also an set of other useful libraries that some of the applications might require. The other ported libraries for the uClinux kernel that can be chosen to be compiled in are:

libgmp	- GNU library for arbitrary precision arithmetic.
libg	- Includes gtermcap, which is designed to send control strings to terminal.
libpcap	- Provides a system-independent interface for user-level packet capturing.
zlib	- A general purpose data compression library.

The distribution has also a set of libraries that are always compiled with the package:

libatm	- An experimental software for Asynchronous Transfer Mode.
libjpeg	- Library for JPEG image compression.
libm	- A math library compiled with the libc.
libnet	- A generic networking API that provides access to several protocols.
libpam	- Interface library for Pluggable Authentication Modules for Linux.
libpng	- A Portable Network Graphics reference library. Requires zlib

5.2.2 Source tree

When uncompressing the source distribution, the package will create an tree structure to the current working directory, all under the uClinux-dist directory. The directory will contain all the sources that are needed to compile an target image with our options. The source tree has the following structure:

bin	- Utilities for uClinux platform to create the flash.bin.
Documentation	- Some details of uClinux/ColdFire.
tools	- uClinux/ColdFire tools for compiler and other build tools.
user	- User applications and sources for them.
freeswan	- A free IPSec program providing security functions, authentication and encryption. Requires some additional libraries.
lib	- All the application libraries.
linux-2.4.x and linux	- Patched uClinux kernel sources.
uClibc	- Newer libc version.
config	- Configuration for setting up the uClinux files system. Used to drive the vendor configuration.
romfs	- ROM based file system structure. Includes user application binaries and device nodes. Created after the compilation.
vendors	- Vendor specific build instructions. Includes sub-directories for all the support platforms.
Images	- After the building process this will include the final build binaries of the kernel, ROM file system and the companied image. Created after the compilation.

The directories in the romfs folder contains the same tree that will eventually be created to the target machine. The folders will also contain the configurations files. The file structure is basically the same as in standard Linux:

bin	- System program binaries.
dev	- Files that provide an interface to a physical device.
etc	- System configuration.
home	- User home directories.
lib	- Includes the shared libraries, when supported.
mnt	- The mount points.
proc	- The mount point stub of the virtual proc file system.
tmp	- For temporary files.
usr	- Additional utilities and applications
var	- The variable files.

Most of the changes are made to the architecture of memory management subsystem, that has been modified by removing the parts that refer to the MMU hardware. This includes the modification of the subsystems by rewriting and removing the parts interacting the MMU-unit. All tough this seems like a large modification, the kernel and the user space software hacking is rather small. The lack of memory management and memory protection are important issues when implementing new code under the uClinux platform. With bad software design you can completely freeze an MMU-less processor.

5.3 Setting up the uClinux environment

Compiling the uClinux kernel is the same kind of task than compiling the normal Linux kernel, which was explained earlier, in the LINUX KERNEL section. The configuration supports also X and menu based menu options. The dependencies are checked with an exact same way. The actual image is simply done with the command "*make*", which will eventually place to created image under the created folder images. This will also create an folder *romfs*, which includes the whole source tree as it is in the image.

The main *Makefile* under the root of the source tree is the one that is called when make commands are executed under the uClinux environment. When making the configuration the uClinux will prompt an main menu screen to the user. From there the user can choose the basic settings, the platform model, C library version, whether the user wants to modify the kernel settings, what user applications are to be compiled in and also reset back to basic settings or

create new basic settings. The basic configuration is done trough an set of architecture based configuration files, which are included into a *vendor/PRODUCT/MODEL* folder. This folder contains the following parts:

config.linux-2.4.x	- Includes the default the kernel configuration.
config.vendor-2.4.x	- Includes the default set of user applications to be build with the image.
rc	- system startup script transferred to romfs.
config.arch	- Architecture specific set ups for compiling the image.
config.modules	- If modules enabled, this will include the modules configuration.
config.uClibc	- Used to set the basic configuration for the uClibc.
inittab	- Transferred to *romfs/etc/inittab*. Is a script for init process.
Makefile	- Builds the system. Includes the instructions how to build the romfs and the image. This file will also define the source tree which will be build in the romfs and install some files to the source tree like the *rc* and *inittab.*

The model specific folder might also include the default configuration files example for *pppd* or *motd*. Also user might want to set an own default http pages that are also convenient to place under this folder. The eventual make command instructions are comping from the model specific Makefile. As a default, the M5272 based platforms will build under the image file the *image.bin, image.elf, imagez.bin, linux.bin* and *romfs.img.*

As in Linux the uClinux also provides mechanism to clean up the system from unnecessary object files and from other created folders and files. The lighter version of the cleaning methods is the "*make clean*" command execution. This will remove the folders *ROMFSDIR* and *IMAGDIR* and also erase the *config.tk* if xconfig has been used and all the old entries from a previous build. More thorough option is the "*make mrproper*", which will clean all the created configuration files. After this command the system will be in its original state and all the configurations will be needed to do again.

uClinux environment also provides some useful make commands that can be used to compile certain parts of the distribution:

make user_only	- Scout trough the user tree and do what it needs.
make romfs	- Makes only the rom file system.
make image	- Runs genromfs and creates images.
make linux	- Compiles the Linux kernel.
make lib_only	- Compiles only the libraries.
make user_clean	- Cleans up the user folder by removing all the object.

5.3.1 Updating the uClinux kernel

Updating the uClinux kernel can be done directly from the Linux kernel just by patching it against the corresponding revision. The uClinux community provides new kernel patch versions almost in synchronization with the new Linux kernel appearance. The uClinux offers also regular patches to the development kernels.

The uClinux patch for the kernel is applied to bring the kernel convenient to the MMU-less platform. The stages of preparing the new kernel is to unpack it to the root of the source tree and apply the patch to the created folder containing the Linux sources. The phases the kernel installation are,

cd /opt/uClinux
tar -zvxf linux-2.X.X.tar.gz
gzip -d uClinux-2.X.X-ucX.diff.gz
cd linux
patch -p1 < ../uClinux-2.X.X-ucX.diff

To enable the platform to recognize the new kernel, it should be named with linux-2.X.x.

5.3.2 Hardware dependency under the uClinux

As in Linux, uClinux also provides support for multiple different processors. In the regular Linux kernel the separation between hardware-dependent and -independent source code are made as clear as possible. This is also the case with the uClinux kernel. This makes the adding

of a new processor support an straight forward task. The kernel source tree includes two folders which both, *arch* and *include* directories hold the platform dependent code under the specific files and folders. In uClinux there exists also a specific vendor folder that holds building instructions how to create the model specific settings.

When adding a new vendor and board to the uClinux distribution the developer has to make the kernel aware of the new option. The first step is to modify the file *Boards.mk* under the */arch/PROCESSOR_TYPE* folder, in the Arctics case the *m68knmmu*. This will include the definitions for platforms and boards, and should have the following lines,

```
ifdef CONFIG_VIOLA
BOARD := Arctic
endif
```

Next the file config.in should have the following lines, under the specific processor option, to inform to configuration to notice the new option.

```
if [ "$CONFIG_M5272" = "y" ]; then
        .....
bool 'Viola Arctic Board support' CONFIG_VIOLA
```

The platform folder in this same directory holds all the processor options defined to this architecture. To here under the processor model, in this case the 5272, should be created an model specific folder, including the files *crt0_ram.S* and *ram.ld*, which are the architecture specific*.ld* file and the startup C asm code. The model for these files can be checked under the platforms with same processor. The last step is to create an branch under the vendor folder for the product and type of the created platform. This file will contain the files described earlier.

5.4 Development tools

In order to build and compile applications for uClinux or an image for your system, a set of development tools has to be installed to host machine. To set up an development environment for uClinux, the user has to set up the kernel and libraries that are used when compiling the binary. In order to compile uClinux kernel a set of development tools are required, which provides an foundation for development. These tools are used for cross-compiling the applications and the kernel to the target platform.

The version of uClinux development tool is m68k-elf-DATE toolchain, which can compile several different types of binaries. These tools provide a foundation for the development and can be obtained as an pre-build binary tool package or compiled by the developer by patching the GNU tools and then compiling them with m68k-elf option. At the time being the toolchain is based on gcc-2.95.3. The whole toolchain includes the following components:

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.
genromfs	- Tool for creating romfs images. Based on sourceforges romfs project, genromfs-0.5.1.

By default the script used to install these tools will install them to */usr/local* directory. If the tools are build from a scratch, all the tools have to be patched and configured in order to get them to work with the uClinux kernel. [19.]

Important issue when 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 other for the user applications. For this uClinux provides the ability to use GNU debugger, GDB, which allows user to debug programs written in C, C++ and with some other languages.

The GBD has number of patches available for uClinux and Motorola's processors. When running the GDB, typically it runs on a host machine which is connected through a serial interface of network to the embedded target device. The embedded device runs a GBD-stub or gdbserver, which the host machine can connect to. The problem with embedded systems is that the space for applications is always limited. Under uClinux the debugging information is stripped away when making the conversion to flat memory model. When using the debugger option with the compiler an *.gbd* file is created which holds the symbol information of the debugged application and which can be loaded on a host machines debugger.

The GBD-stub is used for kernel debugging. It communicates with the host using its own remote protocol. It also allows ability to inquiry register info, set breakpoints and some other debugging commands. The gdbserver is a user application which can be included and compiled to run on the embedded target machine. It provides user application debugging with features

similar to GDB-stub. To host machine it communicates true a pthread call. For other general developing debugging tools that Linux provides are normal print commands included to the source code and an command called *strace* which allows user to trace the system calls and signals when programming is executed under this command. [10, p. 7.]

5.5 Runtime linker and loader

At the runtime loaders and linkers are the ones responsible of running the applications. They also attach the abstract names, for the programmers point of view, to more reasonable ones. In systems with MMU the applications can use the advantage of virtual memory and so the instructions and the read-only data can be shared among the applications. Without MMU, the flat memory model makes it necessary for the application to be assigned with memory allocation different at each runtime. The linker/loader locate the application and install it to the system's memory space in such a way that they can work from the place that they are installed in.

Basically the linker loader perform three different kinds of tasks:

- Program loading, by copying the program to main memory for execution.
- Relocation, assigning proper address for the code and data. Can be done either loader or linker.
- Symbol resolution, assigning addresses pointing to the subprograms

[3, p. 3.]

Generated from the source, the relocatable or object module can be inspected from the three different segments, text/code, data and bss. The text file containing executable instructions, data holding the initial values to be used in variables and uninitialized variables in the bss segment. The data and bss containing only global variables. The variables defined in a function are defined in a stack. The different segments are generated from the source file as follows,

```
extern int x = 8;                    //.data
int y = 9;                           //.data
int z;                               //.bss
int func(){                          //.text
        return x*y;                  //.text
}
```

The linker loader will merge these segments to an loadable object file. It will generate an symbol table including all the internal and external symbols that the program contains and an segment table according to the sizes and the locations of each segment. After this it is able to generate an relocated code to the output file. The process will generate a small amount of code included to the output file, in order to get everything working. The runtime loader-linker can handle these stages to generate an executable. For this toolchain of uClinux uses a little modified Execution and Linking Format, ELF. [3, p. 4-5.]

Originally the ELF was developed by Unix System laboratories and widely used with Unix like operating systems. The ELF format is flexible and can be adopted to many operating systems, so it is widely used. It has capabilities of dynamic-linking, dynamic-loading, runtime control to the program and ability of creating shared libraries. The ELF defines both the linking and the execution of the file. The ELF file layout contains four major parts:

- ELF header, gives an information about the file
- Program header, Part of execution view.
- Section headers, contents of the file. Part of linking view.
- The data

[23.]

The uClinux ELF toolchain is modified in such a way that when creating binaries, the linker *ld* is actually a script. The linker will look for the *-elf2flt* argument and if this is present it first runs the real ELF link command and the runs elf2flt to create the actual executable that uClinux uses. This is called a Binary Flat Format, bFLT which is simple and lightweight executable format based on a.out format. The bFLT has two major revisions from which version 4 is used with m68k elf2flt converter. In the conversion to flat format symbols and debug information is stripped out and a much simpler header is constructed. This is why Flat format is much smaller than the equivalent ELF format.

The uClinux provides the memory as a single address space. This means that the stack is physically contiguous with the static data, the stack must have an fixed size in order to not overwrite the static data and code space. Since the MMU-less processor does not provide any memory protection, the stack size has to be chosen at the program linking time. Having a fixed size stack with the applications usually means that they waste at least some part of the stack. The m68k bFLT converter specifies the stack size in bytes. This is normally set to 4096, but it can be changed by giving an argument *-s* to *m68k-elf-elf2flt*. [16.]

5.5.1 Flat file relocations

The linkers tasks is to merge the created object files generated from the source into a single executable object file that the loader can execute. To run the program, the loader has to bring the objects into the main memory. Here lays a problem, because after the compilation each created object represents a separate address space without a knowledge what exactly will be their real address during the execution. With a machine with flat memory model all the objects are merged into an one address space. This makes the linkers task more difficult as it is responsible of making sure that all the object modules can communicate with each other.
[7, p. 418.]

With elf2flt a list of relocations is specified at the time the flat file format is generated. The uClinux provides two different relocation methods to overcome this problem.

(1) Code relocation fix ups the address references in a program once it is loaded into RAM. The operation uses a specific relocation table generated by the elf2flt. This table is appended at the run time to the end of the data segment, including each of location-dependent address within the program. Each relevant text or data segment containing a memory address in the relocation table is first added with the load offset of where the actual binary is loaded in the memory. This is called a relocation constant which is equal to the starting address of the object. The actual relocation process is transparent to the user program.
Once the modified addresses has been calculated the outcome is rather simple binary which can be fully relocated anywhere from the memory. In order to modify the parts of the code, the whole binary has to be loaded the RAM memory so that new relocations can be assigned. The relocation also means that multiple copies of text and data segments are required for each instance. [42.]

(2) The m68k-elf tools also provide an option to generate position independent code, PIC. This can be achieved with an compiler option -fpic/fPIC. The option differ with *-fpic* to be suitable for use in a shared library and with *-fPIC* option suitable for dynamic linking and avoiding any limit on the size of global offset table.
The position independence is achieved by making the code addressing relative. Data access can be achieved with Global Offset Table, GOT. The data segment will start with a GOT that is a 16 bit look-up table containing 32-bit address pointers. The addresses in the table will need to be relocated to point to the actual location of the data section. The GOT is terminated with a -1 (0xffffffff). [42.]

The problem with the second approach is that the data segment has to follow the text segment. This occurs because, the compilation is made against an origin of 0 and that the breaches can be safely made. The uClinux provides an ability to separate the data and text segments in a different ares in the memory and so by overcome this problem. This is done with an *-msep-data* options, given to the compiler. This improves the PIC by enabling it to take advantage of XIP, eXecute In Place functionality to increase the payload of the Flash. The code compiled with an *-msep-data* option needs less relocations and so by saves the memory, but usually is a little bit slower. Still the main advantage of msep-data is the ability to do XIP. [19.]

The Execute in Place option allows text and data segments to be in separated memory regions. When the application is compiled to support XIP and their header is flagged with XIP option they will be loaded and executed without duplications of the text segment. The advantage of this is that the text segment of the code can locate in the flash / ROM memory and for each running program the stack, bss and data segments are loaded to the RAM, where the code can be modified. The XIP option comes especially useful when the application have large program bodies with many executable instances running in the system. The problem with XIP is that compression cannot be used. If the data is in compressed format, they cannot be used directly in place. [42.]

5.6 Creating the image

In the full uClinux distribution, the image is simply done with the "*make*" command. The command will first change the directory to the kernel directory and starts to build the individual components. The actual kernel building process is basically the same that is explained in the Appendix A, but with uClinux kernels own building tools. After the compilation the process will continue by combining all the subsystem objects and archive files using the sections from the architecture specific.*ld* file and the startup C asm code, *crt0_ram.S,* to create a Linux file. The compiler script will list the symbols from the Linux file with the specified patterns and sorts the output to the *System.map,* which can be used in the debugging phase to show where each function is located in the memory. [31.]

The Linux file is then translated into an binary file, *linux.bin,* which will hold the binary code of the kernel.

```
m68k-elf-objcopy -O binary /.../uClinux-dist/linux-2.4.x/linux  \
/.../uClinux-dist/images/linux.bin
```

The ROM file system is generated based on the settings, located in the vendor file. The file includes the configuration files the vendor specific settings for applications, kernel and building. The makefile of the vendor folder includes the instructions, how to build the actual image for that processor / vendor. The makefile includes tags for which different images are constructed the structure of the source tree with *ROMFS_DIRS* and tags for building the actual file system and images.

```
ROMFSIMG = $(IMAGEDIR)/romfs.img
IMAGE    = $(IMAGEDIR)/image.bin
ELFIMAGE = $(IMAGEDIR)/image.elf
```

At the end of the compilation process the vendor makefile first generates the actual tree including the devices and the binaries from chosen applications to the top of the development tree. From this folder the process generates the ROM file system image for the system.

```
genromfs -v -V "ROMdisk" -f /.../uClinux-dist/images/romfs.img -d \
/opt/uClinux- dist/romfs
```

This ROM file-system binary is then concatenated with linux.bin in order to create the complete binary that can be loaded as it is to the target device.

```
cat /.../uClinux-dist/images/linux.bin /.../uClinux-dist/images/romfs.img >
/.../uClinux-dist/images/image.bin
/.../uClinux-dist/tools/cksum -b -o 2 /.../uClinux-dist/images/image.bin
>>/.../uClinux-dist/images/image.bin
```

The image is then placed to the image folder, which eventually also contains the file system image and other files that are created in the process.

5.7 Booting the device

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

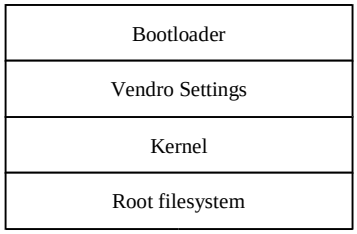
An more safe and flexible option is to place a small stand alone piece of code called a bootloader, to the start offset of the flash. The bootloader can handle the initial settings of the board like the basic hardware setup and allows to download the image to the board via. Many times it can also handle some environment settings, which ables the user to write some simple configurations without needing a writable file system for the Flash. The uClinux can be used with various of different bootloaders for specialized tasks in different stages of development. Some bootloaders for uClinux are CoLilo, My Right Boot, Motorola's dBug and PPCboot, which is the one used with the Arctic platform. The PowerPC boot, PPCboot enables loading the image or images through an serial or ethernet, with the Trivial File Transfer Protocol, TFTP protocol. It also enables the usage of environment variables, booting of an compressed and decompressed images and booting kernel from a JFFS2 partition.

In uClinux the root file system and kernel are compiled as separate entirety. In regular Linux distribution the kernel is embedded as an part of the file system. This setting requires an sophisticated bootloader in order to place the required parts at the boot to the RAM. The uClinux allows the file system and kernel partitions to be placed in to the devices memory in several different ways. The setting depends whether the image is compressed and from where the actual execution takes place. When doing the development the most convenient solutions is to download the image directly to the fixed offset of RAM and execute it from there, but when the image is intended to solid state storage, it should be placed to the systems Flash. As the Flash memory is usually more expensive and the access to 32-bit wide RAM is faster than to the 16-bit Flash. In to the Flash the image can be placed either compressed or as it is. Usually the case of placing the kernel + root file system as it is to Flash is used with deeply embedded devices with this being the only available memory by making the root file system read-only. It is also possible to make part of your file system read-only and only place the read / write partitions to RAM. [37.]

The amount of used Flash can also be saved by first decompressing the kernel and root file system and at power up, uncompressing it to Flash. This requires that the bootloader used is capable of handling the uncompressing. With this solution the footprint of the Flash is considerably smaller compared to image by it self.

-rw-r--r-- 1 root root 1.6M Oct 16 19:06 image.bin
-rw-r--r-- 1 root root 712k Oct 16 19:06 image.bin.gz

Typically the Flash partition under the uClinux holds the bootloader and some other configurations at the beginning and the rest of the Flash reserved to the Linux image. A typical flash partition could look like in the picture below.



Picture 5-1. Typical Flash partition arrangement of uClinux platform.

The solutions of placing the kernel and the root file system to the flash, has several options with their advantages and disadvantages. In Arctic, at the time being, the kernel and root file system are concatenated together and placed as an compressed image to the flash. From there, at the boot, the image is decompressed and placed to the RAM. For configuration and file saving the Arctic provides an separate serial flash chip. From the root file system the most important and changed folders like */etc* can be linked to point to the serial flash.

An other option is to place the kernel and the root file system as separate files to the storage. This way the updating of one of these pieces is easier and do not require of compiling and loading the whole image again. It also possibles that the root file system is kept in the flash as it is, so all of the modifications will automatically be saved. The kernel in this option can be kept in a compressed form in the flash and from there decompressed to the RAM. This option requires some hacking as the *crt0_ram.S* for Coldfire always tries to relocate an attached romfs after the .bss section. The solution for this is either to modify the *crt0_ram.S* or just to create an pseude-romfs image file that the code can locate. [25.]

The boot sequence in Arctic is like the normal boot sequence in any uClinux platform. With the Arctic, the bootloader gives user a few options from where and how the image is executed. If the execution is done directly from the RAM, at location 0x20000, the command *go 20000* can be used to start the execution from a defined place and start to uncompress the kernel. To copy and execute the image directly from the Flash the ppcboot has to have a copy of the kernel in a defined place in the Flash. To copy the kernel image to the Flash the image has to be copied first to the RAM through the serial or tftp. In the PPCboot, this is done with the *cp.b 20000 ffc40000 $(filesize)* command. If the image was compiled with the option of kernel run from ROM, the execution can simply be done with the *go ffc40000*. Other wise the kernel

image has to be uncompressed to a defined place in the RAM by *bootm ffc40000*. From here the boot sequence continues by decompressing the image into the memory. It will place the *kernel.text* and *kernel.data*, which together forms the linux.bin, at the predefined place into the memory. The romfs will be placed to the end of the *kernel.data* and it will form the *kernel.bss* section.

Like in the regular Linux, from here the initialization will continue to detect and initialize the hardware. It sets up the interrupts and loads the necessary drivers for these devices. After the hardware setup the kernel will execute the init, which will then read the system inittab file. Next the init executes the start script which holds programs that are executed only once, at the boot. To finalize the boot process, the uClinux runs the rc script, which contains the commands that need to be run for the device to work. The script will normally mount the necessary file systems and assigns the network addresses.

The location of the configuration scripts are the same as in Linux. The */etc* folder contains all the necessary scripts that are necessary to boot up the device. It also includes all the other general configuration files that the different applications like *ppp, chat* and the *cron* uses. These files are in Arctic are saved to the serial flash so that they can easily be modified and after wards saved also after booting the device.

5.8 Root filesystem

The uClinux supports various of different file systems, including the ones specially designed for the embedded systems. Usually the hardware platform used with the uClinux is a combination of different physical memory storages, which all require an different file system for efficient usage. The Arctic at this stage is designed to boot from the Flash from where the kernel and the ROM filesystem is then uncompressed into a RAM. Arctic includes also an specific storage for the data that is to be saved after the reset of the board. This Serial Flash is set as an JFFS section, which is specifically designed for efficient use of Flash devices. All the file systems used in Arctic are listed next,

/proc> cat filesystems
nodev rootfs
nodev bdev
nodev proc
nodev sockfs
nodev pipefs
ext2
nodev ramfs
jffs
romfs

The user application in uClinux are compiled into an ROM file system, which the kernel mounts at the boot time. This file system is a quite simple, space efficient read only file system mainly used for initial RAM disks of installation disks. The file structure of the ROM file system is defined in the vendor specific makefile. All the executables created from the user applications are placed in the */bin* file. The other folders created are basically following the same structure as in regular Linux distribution.

The *rc* script in the vendor specific file is executed in the boot time. This script contains information of mounting the RAM file system to the end systems uClinux. The RAM file system is a placement for the regular table PC's hard drive to store information like log files that the applications produce. The size of the RAM file system is defined under the Customize Vendor / User Settings in Compulsory Configuration. The options for the system size are 64, 128 and 256kbytes. The mount point for the file system is */var*, but uClinux also has the */tmp* directory in the root connected via a symbolic link to point to the */var/tmp*. The file system is uncompressed and mount at the rc script. A Scratch from the *rc* script can be like:

 # expand the ramdisk
/sbin/expand /ramfs.img /dev/ram0

 # mount ramdisk
/bin/mount -t ext2 /dev/ram0 /var
mkdir /var/tmp
mkdir /var/log
mkdir /var/run
mkdir /var/lock

The ramdisk is mounted to the ext2 file system, which is uncompressed with the expand utility to the */dev/ram0* block device. Same way the Arctic uses the serial flash, which is also mounted in the *rc*,but only to the block device mtdblock0.

mount -t jffs /dev/mtdblock0 /mnt/jffs

The second extended file system, ext2 is the most widely used file system in Linux systems. It is an efficient file system with fast performance and a support for file linking. It is designed to keep data in fixed size blocks, which is set when the file system is created. The files in the ext2 are presented with a single inode, which each have a single number identifying it. All the inodes for the file system are kept in a inode tables so the directories in the ext2 are simply special files which contain pointers to the inodes of their directory entries. [14.]

The proc file system provides some useful information about the current state of the kernel and the kernel modules. The proc is a virtual file system providing the ability to peer into the kernel's view of the system. It is also mounted from the *rc* script to the */proc* folder, which includes the information in a form of plain text.

The uClinux also contains some virtual or pseudo file systems that does not require a block device for mounting. These are used by default in the image. It includes the pipefs, which provides the entry point into the pipe system call, the socket file system enabling the socket layers and bdev defining a disk boot device.

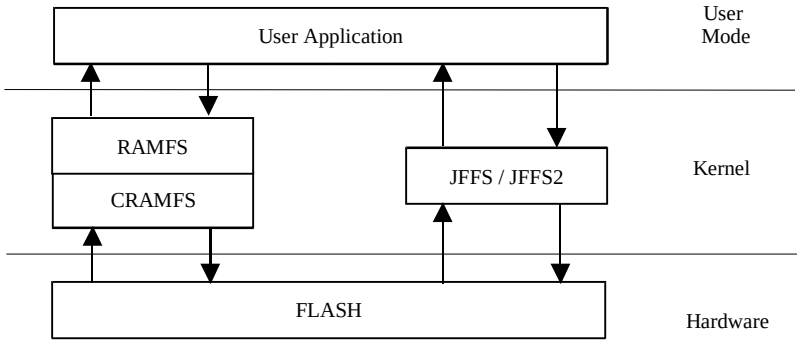
5.8.1 Serial Flash file system

The previous file systems used in Arctic are a default set of M5272 configuration file systems. The external serial Flash chip that the Arctic provides is mounted with the JFFS. In the future the more convenient solutions might be the JFFS2 though it provides some benefits against the older model. Still the JFFS brings some advantages compared to CRAMFS. This chapter inspects all of these three solutions, including their advantages and disadvantages.

The uClinux provides three main types of file systems for use in Flash devices, CRAMFS, JFFS and JFFS2. From these the JFFS and JFFS2 belong to same evolution tree originally designed by Axis Communications AB. The CRAMFS was originally developed by Linus Torvalds and was added to kernel source of 2.4 series. It was designed to be an simple read-only file system which uses compression in each page of the file individually when storing it and decompressing

it on the fly when reading it. This enables an random access to data. The only problem with this file system is that in order to access data it must be first transmitted to RAM. These kinds of pseudo-file system are designed to emulate an block device with fixed amount of bytes in sector and then using standard file system on that. [13.]

To overcome the translation layer problem that comes with the CRAMFS (see. picture 5-2) an more efficient file system, designed particularly to Flash device was developed. The Journaling Flash File System is implemented directly to the flash device so it does not need any translation layer between.



Picture 5-2. CRAMFS versus JFFS/JFFS2.

The JFFS is a log-structured file system with build-in wear leveling, where nodes or log entry's are stored sequentially to a log locating in a flash. The actual data and meta data written is placed to the storage in a location defined by the log. The log holds information like version number, which tells the chronological sequence of the nodes, meta data like uid, gid and the offset of the data within the file. The design of JFFS is implemented in such away that it guarantees that the data written to the storage will not get corrupted. The storage will always contain the latest data written even if power fail occurs during the write. In such a case the data is a combination of the old and new data that were able to already be written. [43.]

When the file system is mounted, the whole log is scanned to determine how the directory hierarchy is to be created. Also the new blocks are re-arranged in such away that they can be accessed and deleted blocks marked to the garbage collector. When changes are made to the file structure ,the information in the log is updated in all times when the file system is mounted. The changes to the files may make part of the used space as obsoleted. These dirty spaces are then marked for the JFFS's garbage collector to be recycled. [43.]

The storing of data goes strictly linearly through the whole storage. The data can easily be appended until it reaches the end of the media. After this the data can only be written to the part that already contains this dirty space and which are allowed to be overwritten. At this time the garbage collection is trigged by the threshold to make some free space if possible. The triggering will happen either by the means of a kernel thread or when process attempts to write to the storage and finds that it is out of free space. The collector proceeds linearly from the head node of the flash toward the tail. Blocks from the head are copied to the tail and after wards erased. The free space that may occur from this process is again marked to be clean. The head tag is then moved to the next reserved block and the writing can again start from the beginning of the flash. Idea is that every block is erased and written the exact same amount of times, which gives an perfect wear leveling to the flash. The only problem with this method is that some unnecessary erasing and writing is done that uses the flash. This and some other limitations were the key features that started an new project to develop a more evolved version of the file system. [44.]

The journaling flash file system got an new revision based on the design concepts of the first version. The JFFS2 was developed by RedHat and it started with a project of including an compression to JFFS, but because of some limitations of the first version it was decided that the code would be re-written to overcome all the problems facing the first release. The development was launched to eCos embedded operating system, so for this reason it was released with dual license, GPL and Red Hat eCos Public License. Officially it was included to 2.4.10 series of kernel. [43.]

The compression in JFFS2 is based quick compression algorithm on zlib based file compression that compresses all the files before it is written to storage. When data is written from the storage it is decompressed on the fly so that the whole process is invisible for the end user. The second version also provides an more efficient, non-sequential garbage collection. It treaded all the blocks individually allowing the garbage collector to make a decision which block will be erased next. The erase blocks in the log structure is stored in one of the data structures depending on the current contents of the block. To determine which block is to be erased next based on the jiffies counter. The counter is based on formula jiffies%100, where to non-zero number indicates that the block is erased from *dirty_list*, and the remaining 1 in 100 times the pick is done from the *clean_list* containing only valid nodes. This ensures that the data is also moved around the media and wear leveling is achieved. [43.]

With both JFFS and JFFS2 one major flaw is the amount of space that is required by the garbage collector. At the time five full erase blocks are required in order to perform a new write for the user space. The compression that the JFFS2 uses also causes unnecessary overhead to files that are already compressed.

5.8.2 Kernel block drivers

The uClinux kernel supports three different types of block devices. These drivers enables to read and write a fixed size blocks to the host file system such as an disk. The drivers are used as the lowest level of accessing the physical device. Above these the Linux uses some root file system, which handles the arrangement of the blocks.

The common way to start up a uClinux based system is by include the block memory device, blkmem and the ram disk drivers to the system. The RAM disk driver is designed to allow a portion of the RAM memory to be used as a block device allowing user to read and write to it. The file system can be either in its native format or compressed. This file system does not have any direct support for the Flash device and is only really useful for storing the root file system. In Arctic the RAM disk is mounted at the boot to the */var* folder. The another common file system in uClinux is the blkmem driver, which is the oldest driver and specially designed for the uClinux. It supports only few different Flash devices as well as the root file systems in RAM. [16.]

A more convenient solutions for the Flash device to is is the Memory Technology device, MTD driver. It provides an generic interface to memory devices providing a hardware abstraction layer, HAL which allows different file systems to mount them selfs on various of different memory devices. The mtdblock provides some security against overlapping by allowing these functionalities to be done to the actual flash device only when the request is done to another block. This is because the Linux's buffer cache is smaller than typical erasable sector in flash and is done with a local cache. [43.] In Arctic this driver is used with the JFFS allowing it to be mounted on any raw random access device. It is specially designed for the Flash memories, but can also be used with the RAM.

The uClinux has an lack of using blkmem and MTD device drivers active at the same time. This is for example the case with the Arctic board as the serial Flash uses the JFFS file system and the blkmem driver is used with the root file system. The problem is that both block device

drivers use the same major number, *BLKMEM_MAJOR 31* and *MTD_BLOCK_MAJOR 31*. As a solution, the major number from the other can be changed to some unused devices number, which makes the both drivers usable.

5.9 Changes in programming interfaces

The lack of MMU in uClinux environment is the biggest change compared to the Linux. This causes the lack of memory protection and a virtual memory model. This of course affects to the development done under the uClinux by forcing some changes to be adopted to some system calls and the fact that the programmer is responsible of memory management.

The *fork()* system call is used to duplicate the current process by creating a new entry in the process table. This can be handy if the program handles more than one function at the time. The created child process is all most identical to the parent executing the same code but with its own data space, environment and descriptors. The fork command is implemented by using copy-on-write pages. When either one of the process tries to write on the page frame, a private copy of the page is created for this process. The new physical page is mapped into the original logical address space. Without MMU the process cannot be completely and securely clone a process, nor does it have access to copy-on-write page.

The uClinux implements BSDs *vfork()* in order the offer the functionality of fork. The process created by this system call shares all their memory space including the stack. To prevent the parent to override the data needed by the child process the parent is suspended until child exists. [10, p. 8.]

Memory allocation malloc is normally implemented at the low level using the brk system call. Under Unix environment each created process owns a specific memory region called a heap. This is used to for processes dynamic memory request. The size of the memory region can be increased / decreased directly by the *brk()* system call. Under uClinux this system call cannot increase the memory region unless it goes trough some changes.

The process is allowed to allocate memory from the global pool of free memory. The simplest memory allocation method under uClinux is *mmap()* and *munmap()* to return the memory to the memory pool. The choice of different methods for memory allocation depends also from the libc version. Both uC-libc and uClibc support a simple allocator, malloc-simple, which uses the mmap / munmap and lets the kernel actually to handle the allocation. The problem with this call

is that the based allocation is about 56 bytes, which can cause small memory request to crow quite high. The main problem lies an fact that efficient and fast memory allocation generally also add code size to small application. [27.]

5.9.1 Memory management

The virtual memory system provides an layer between the applications memory request and the Memory Management Unit, MMU. The main advantage of this system is that the memory that the application refers is separated from the physical memory and that it provides an higher level memory protection. The virtual memory request is allocated with cooperation of MMU and kernel. The MMU provides a level of protection for applications that run on the platform. Working without the MMU means that all of the program memory is literally mapped against the physical memory. This is called a flat memory architecture. An Invalid memory pointer in user application may trigger an address error which can completely freeze or corrupt an MMU-less processor . The code implemented to MMU-less platform has to be working properly, which of course means thorough testing.

Dynamic memory allocation in a flat memory model can also cause fragmentation which also can starve the system. In a ideal case the physical memory is used as continuous memory areas and the allocation should fail only when the number of free page frame is too small. All tough the all together memory would be large enough, there might not be a continuous piece of memory available. This is also a problem in a operating system with virtual memory. One way to overcome this is to request memory allocation through preallocated buffer pool.

Linux and other Unix like operating systems usually provides an method called swapping. Here the kernel uses a part of disk as an extension of RAM in order for the process to run by the illusion that they have all the physical memory available all tough some of their pages are stored away and retrieved again when needed. In uClinux this method is not possible because it can ensure that the pages would be loaded to the same location in RAM. In general in embedded environments it also would not be acceptable to suspend an application in order to use more RAM than is physically available. [1, p. 456.]

5.10 User applications

User applications exists under the uClinux-tree, in folder *./user*. As a default, the environment has quite many ported applications for different purposes. If the user wants to add or port new applications, they should be placed under this directory and modify some files in order to see the new application in the menus.

The major concern, when developing and running large, sophisticated programs under uClinux is the lack of MMU-unit. This of course changes the memory management so that heavily memory dependent applications may not run properly under the uClinux environment. The parts in the code using memory allocations and de-allocations require redevelopment. Also memory swapping and paging needs to be removed.

5.10.1 Adding user applications

First thing when adding new user applications to uClinux environment is to create a new subdirectory for your new application under the user folder. This is where you place your source files and create a makefile that the uClinux needs. The building options that the uClinux uses for cross-compiling are set in a *config.arch*, which can be just represented at your application makefile. Example of the your user application makefile could be the following,

```
EXEC = applicationName
OBJS = applicationName.o
all: $(EXEC)
$(EXEC): $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
romfs:
        $(ROMFSINST) /bin/$(EXEC)
clean:
        rm -f $(EXEC) *.elf *.gdb *.o
```

In here we call the binary file that is compiled from the source. Applications are the ones responsible for copying files to the correct place in the ROM files system under the romfs target.

The next thing is to add your application to the makefile that consist the building instructions for all of the user level applications. This is done by adding there a line,

dir_$(CONFIG_USER_APPLDIR_APPLBINARY) += applicationName

This way the application is added to the list of directories that can be build with the image.

For making the program as one option to choose, when making the kernel and customizing the vendor/user settings is to add your application to the *./config/config.in* file. The line

bool 'applicationName ' CONFIG_USER_APPLDIR_APPLBINARY

should be placed under heading where you want your application to appear during the configuration. Usually, for user program , the proper place is under 'Miscellaneous Applications'.

User can also add some useful text describing for what the program is used for. This text is placed to the *./config/Configure.help* file. Note that when using menuconfig, the line should not be more than 70 characters and that as in normal shell script the lines beging with *'#'* character interpreted as comment.

CONFIG_USER_APPLDIR_APPLBINARY

"Something usefull here"

By default m68k-elf-gcc cross compiler will generate m68020 code. The developer has to choose a right target processors option as an argument to prevent unnecessary stops. When compiling the application, the interpreter can be set to create the following kinds of binaries.

Fully Relocated binaries

m68k-elf-gcc -Wl, -elf2flt -m5200 -o FILE FILE.c -lc

PIC Separated-Data Binaries

m68k-elf-gcc -Wl, -elf2flt -m5200 -msep-data -o FILE FILE.c -lc

Compressed binaries

*m68k-elf-gcc -Wl, -elf2flt -m5200 [-msep-data]-o FILE FILE.c -lc *

elf2flt -z -o FILE FILE.c

C++ code

m68k-elf-g++ -Wl, -elf2flt -m5200 -o FILE FILE.cpp -lstdc++ -lc -lgcc

The m68k-elf tool chains gcc-2.95.3 compilers ability to generate compact C++ idioms can be quite space consuming. When using C++ with the tool chain a special note should be taken to see how it should be used. Especially run time checking and exceptions can be quite memory consuming. Also any usage of STL or libstdc++ generate code that is heavy compared to C. An simple example of the binary size with C and C++ are shown in the table below.

Table 5-1. Size of generated C and C++ binary with m68k gcc-2.95.3 based compiler.

C	***C++***
#include <stdio.h> main(){printf("HelloWorld");}	#include <iostream> main(){std::cout<<"Hello World";}
Binary size 17k	Binary size 95K

5.10.2 Porting applications

Porting applications to uClinux is a same kind of tasks than in regular Linux. As Linux already has a create deal of applications developed under the GPL, they can be added with a rather minor modifications to run under the uClinux kernel. The main steps for porting is to modify the applications Makefiles and configuration files to ensure that the compilation is done with the uClinux tool chain. The other thing that should be noted is the lack of the MMU and the system calls related to this. For efficient and user friendly programming it is usually convenient to comment all the changes made to the source code, so that the other developers seeking your code can easily see what and how the change is made.

Most problems when compiling the newly ported application usually is generated from the libraries. The lack of library forces the user to also port the whole library to the system and when linking the code with these libraries some unexpected problems might occur. Also some differences might occur from the API's offered by the C libraries. From here the newer uClibc has a better support for Linux like API's.

5.11 Future of uClinux

Today uClinux community continues to provide the patches for latest kernels as they appear. This development process has always been rapid and includes also the releases of the development kernels. The community also releases the latest uClibc version through the uClinux support pages. The distribution also evolves to provide support for new MMU-less

microprocessors architectures, as they appear.

General purpose Linux distribution has also merged the uClinux 2.5.x patches to the main kernel source tree, beginning from the version linux-2.5.46. This enables that the developers for system architectures lacking a MMU, can fetch and set the development platform from the main Linux release. The distribution contains the m68knommu and v850 architecture support, binfmt_flat loader, 5272 ethernet driver and uclinux MTD romfs map driver. [20.]

The Linux kernel already provides the soft real-time support with the preemptive scheduler, but the uClinux kernel has also had some efforts for hard real-time support. There exists two different efforts available to give uClinux real time support - Real-Time Linux and Real-Time Application interface. From these efforts have been done with the Real-Time Application Interface approach to provide the real-time subsystems for use on MMU-less processors.

Chapter 6

CONCLUSION

The purpose of the graduation assignment was to explore the architecture and implementation of the uClinux kernel and its development tools. It presents this concept with the helps of Viola Systems Arctic hardware platform, by inspecting the different solutions and future options that can be made with it. The study mainly focuses on the software design and programming tools of the uClinux platform. It explains how to set up the uClinux development tools and the building process of creating an loadable image. As uClinux is a straight derivate of the general purpose Linux, the work briefly explains the Linux kernel architecture, from a programmers point of view.

The problem with Linux is that it is quite complex. With embedded solution this is even more relevant, as they usually do not provide any graphical user interface. To fully understand the different options that the uClinux platform provides, will require basic knowledge of Unix based systems and some programming skills. The uClinux architecture is in a large scale quite clearly designed, but still leaves some details too complicated, at least for commercial usage. This is for example the case with adding an new application to the platform, which was introduced in the chapter 5, under Adding user applications.

The uClinux is quite poorly documented. The interfaces changed due the lack of MMU and derivation of the different pieces of the kernel have made some changes to the basic functionality. This also means that all the Linux documentation do not work with the uClinux. Generally this requires more attention from the developers. This fact has also affected to this work, as the information about the uClinux platform was scattered around the Internet. For this issue, the uClinux forum gave a lot of help.

The uClinux platform and the tools provide an good, royalty free foundation for the development. Traditionally Linux provides an stable platform, with an excellent networking support. This is also the case with the uClinux, which makes it an good solution as an operating system providing device connectivity and management. The uClinux development platform

also provides an good set of tools, based on well known GNU development tools. These factors generally makes the development more faster and easier.

For as it is, the platform is mainly designed for the developers and requires work to be done in order to prepare it for commercial usage. The uClinux platform offers a lot of kernel options and user application choices that the end user can make, which can make the image compilation to fail. This is also an issue that should be restricted from the end user, at least concerning the supported platform image. Otherwise it will make the platform support an impossible task. It is also necessary to document the uClinux development platform and the usage of the tools.

For the time being, the Arctic platform is still under development. The basic architecture is set, but it can still face some modifications. This is at least an issue with the serial Flash file system and how the root file system and the kernel are placed into the Flash.

For consumers using embedded solutions, also an true real-time operating system might be required for performing some critical tasks. There has been some development related toward the hard real-time support, but yet an fully working and tested solution is not available. The embedded solutions generally have less hardware to deal with, which enables the system to have better interrupt latencies than any typical personal computer and the need for hard real-time support is not that significant. This factor of course also lowers the level of applying the real-time support for the kernel.

The uClinux will require development to be done relating to the software and kernel upgrades. It is essential for an industrial machine providing network connectivity to have the latest security hole updates at least against the major security risks. User having multiple Arctic boards distributed in the field, should easily be able to update the software image of the board without physically visiting the boards. This is also the case with configuration the present software in the board.

REFERENCES

Literature

1. Bovet, Daniel P - Cesati Marco. 2001. *Understanding the Linux Kernel*. Sebastopol: O'Reilly & Associates, Inc.
2. Labrosse, Jean J. MicroC. 1998. *OS II: The Real Time Kernel, 2nd Edition*. Gilroy: R&D Books.
3. Levine, John R. 1999. *Linkers & Loaders*. 1999. San Francisco: Morgan Kaufmann Publishers.
4. Lombardo, John. 2002. *Embedded Linux*. Indiana: New Riders Publishing.
5. Matthew, Neil - Stones Richard. 2000. *Beginning Linux Programming, 2nd edition*. Birmingham: Wrox Press Ltd.
6. Rubini, Alessandro - Corbet, Jonathan. 2001. *Linux Device Drivers, 2nd Edition.* Sebastopol: O'Reilly & Associates, Inc.
7. Tanenbaum, Andrew S. 1990. *Structured Computer Organization, third edition*. Upper Saddle River: Prentice Hall, Inc.
8. Tanenbaum, Andrew S - Woodhull Albert S. *Operating Systems: Design and Implementation, 2nd Edition*. Upper Saddle River: Prentice Hall, Inc.

Internet

9. Aivazian, Tigran. 2001. *Linux Kernel Internals*. http://www.sulgi.net/pdsdata/data_file/data_menu/Linux-Kernel-Internals.pdf. 29.10.2002
10. Arcturus Networks, Inc. 2001. *uClinux WHITE PAPER OVERVIEW*. http://www.arcturusnetworks.com/Docs/UCLINUXWP.pdf. 5.11.2002.
11. Blue Mug, Inc. 2002. *Embedded Linux Survey*. http://www.bluemug.com/research/els/els.pdf. 16.10.2002.
12. Bruyninckx, Herman. 2001. *Real-Time and Embedded HOWTO*. http://people.mech.kuleuven.ac.be/~bruyninc/rthowto/rtHOWTO.pdf. 15.10.2002.

13.Brake, Cliff - Sutherland, Jeff. 2001. *Flash Filesystems for Embedded Linux Systems*. http://www.linuxdevices.com/cgi-bin/printerfriendly.cgi?id=AT7478621147. 19.11.2002.

14.Card, Rémy - Ts'o, Theodore - Tweedie, Stephen. *Design and Implementation of the Second Extended Filesystem*. http://e2fsprogs.sourceforge.net/ext2intro.html. 24.11.2002.

15.Cohen, Yoram. *SNMP - Simple Network Managment Protocol*. http://www2.rad.com/networks/1995/snmp/snmp.htm. 23.10.2002.

16.deBlaquiere, Joe. *Supporting New hardware Environments with uClinux*. http://www.redhat.com/embedded/technologies/resources/deblaquiere.pdf. 20.11.2002.

17.Drabik, John. 2002. *uClinux: World's most popular embedded Linux distro?*. http://www.linuxdevices.com/articles/AT3267251481.html. 2.11.2002.

18.Free Software Foundation, Inc. 1999. *GNU Lesser General Public License*. http://www.gnu.org/copyleft/lesser.html. 21.9.2002.

19.*Gcc-2.95.3 m68k-elf for uClinux*. http://www.beyondlogic.org/uClinux/gcc-2.95.3.pdf. 17.11.2002.

20.Gillham, Miles. 2002. *uClinux and Linux Set To Merge*. http://www.snapgear.com/tb20021115.html. 2.12.2002.

21.Griffin, Terry. 1998. *Standards and more standards ...* . http://www.computerbits.com/archive/1998/1100/lnx9811.html. 12.9.2002.

22.Hasan, Ragiib. 1999. *History of Linux* . http://ragib.hypermart.net/linux/. 12.9.2002.

23.Hongjiu, Lu. *ELF: From Programmer's Perspective*. http://citeseer.nj.nec.com/lu95elf.html. 20.11.2002.

24.Keane, Justin C. 2002. *Linux Bootloaders (LILO vs. GRUB)*. http://www.madirish.net/printer_friendly.php?section=5&article=95. 1.10.2002.

25.Kuhn, Bernhard. 2001. *uClinux flash device mini howto*. http://www.uclinux.org/~bkuhn/Platforms/Coldfire/tarifa/20011119/README. 20.11.2002.

26.*Linux booting*. http://www.aitel.hist.no/fag/plx-e/01-install/20-linux-booting.pdf. 1.10.2002.

27.McCullough, David. 2002. *Why is Malloc Different Under uClinux?*. http://www.linuxdevices.com/articles/AT7777470166.html. 23.11.2002.

28.Nigel, Dick. 2000. *Embedded Coldfire - Taking Linux on Board*. http://e-www.motorola.com/brdata/PDFDB/docs/AN2005.pdf. 21.10.2002.

29.O'Reilly & Associates. 1999. *Microkernels*. http://www.sindominio.net/biblioweb/telematica/open-sources-html/node86.html. 3.10.2002.

30.O'Reilly & Associates. 1999. *The story of the Linux kernel*. http://www.linuxworld.com/linuxworld/lw-1999-03/lw-03-opensources.html. 5.10.2002.

31.Peacocku, Craig. 2002. *uClinux - Understanding the build tools*. http://www.beyondlogic.org/uClinux/builduC.htm. 17.11.2002.

32. Rusling, David A. *The Linux Kernel*. http://www.tldp.org/LDP/tlk/tlk.html. 12.10.2002.
33. Schlapbach, Andreas. 2000. *Linux Process Scheduling.* http://iamexwiwww.unibe.ch/studenten/schlpbch/linuxScheduling/LinuxScheduling.html#toc2. 10.10.2002.
34. Stallman, Richard. 2002. *Linux and the GNU Project.* http://www.gnu.org/gnu/linux-and-gnu.html. 20.10.2002.
35. *uClibc -- a C library for embedded systems*. http://www.uclibc.org/. 7.11.2002.
36. *uClinux -- Linux on Microcontrollers*. http://www.linuxdevices.com/links/LK8053710489.html. 22.10.2002.
37. Ungerer, Greg. 2002. *Using Flash Memory with uClinux.* http://www.realtime-info.be/vpr/layout/display/pr.asp?PRID=3615. 20.11.2002.
38. Walton, Sean. 1996. *Linux Threads Frequently Asked Questions (FAQ).* http://linas.org/linux/threads-faq.html. 15.10.2002.
39. Webb, Warren. 2002. *Pick and Place, Linux grabs the embedded markets*. http://www.e-insite.net/ednmag/contents/images/253780.pdf. 22.10.2002.
40. West, Rich. 2002. *The Linux Kernel: Process Management*. http://www.cs.bu.edu/fac/richwest/cs591_spring_2002/notes/linux_process_mgt.PDF. 10.10.2002.
41. *What is the difference between uC-libc and uClibc.* http://www.ucdot.org/article.pl?sid=02/08/21/1124218. 7.11.2002.
42. Wilshire, Phil. *SDCS uClinux Training ... eXecute In Place.* http://www.ucdot.org/article.pl?sid=02/08/28/0434210&mode=thread. 22.10.2002.
43. Woodhouse, David. 2001. *The Journalling Flash File System.* http://sources.redhat.com/jffs2/jffs2-html/. 19.11.2002.
44. Woodhouse, David. 2001. *The Journaling Flash File System.* http://sources.redhat.com/jffs2/jffs2-slides-transformed.pdf. 19.11.2002.

Appendix A

LINUX BOOTING SEQUENCE

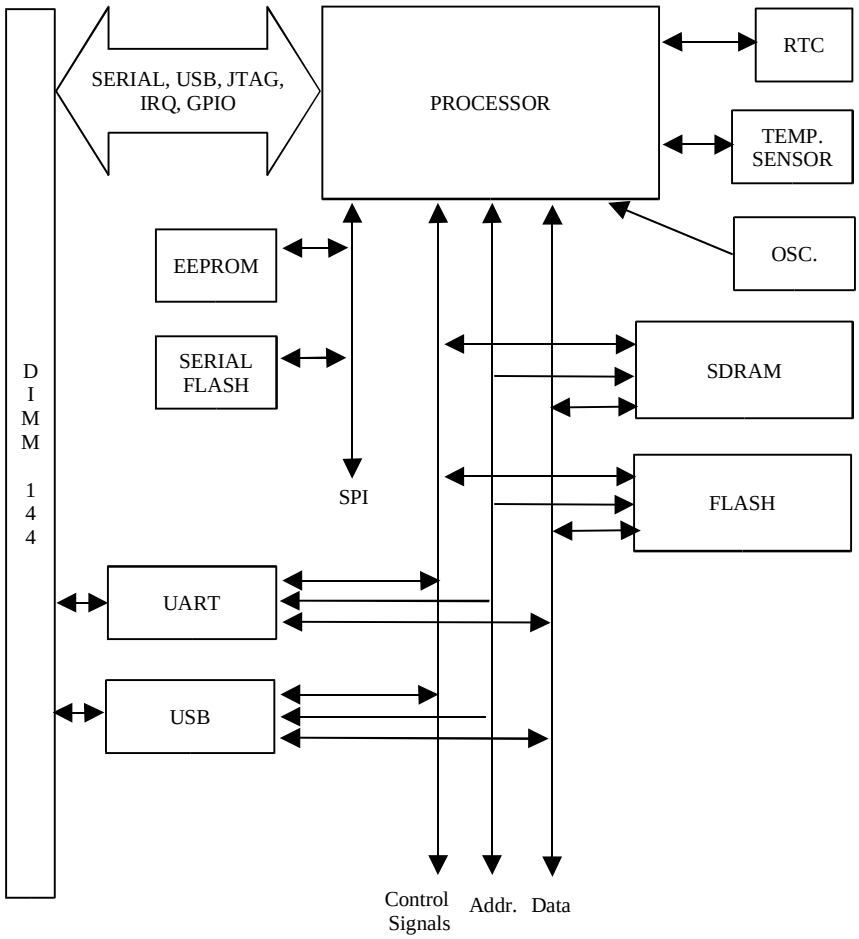
This explains more carefully the steps taken to compile the compressed x86 based Linux kernel image. The resulting image is placed in x86 based architecture in the */arch/i386/boot/* folder. The kernel compilation options "*make bzImage*", will create bzip compressed kernel image and "*make zImage*", an compressed kernel image with gnu zip. The difference between these two options is that the old *zImage* uncompresses the kernel into low memory and *bzImage* uncompresses the kernel into high memory. This is due the historical reasons, when the image was small enough to fit into the lower part of memory. The 'b' in *bzImage* stands for big *zImage*, it allows kernel bigger then 512K.

The steps taken to build the image:

1. The sources of the Linux operating system are compiled.
2. Created object and archive files from the phase 1 are linked together into a *vmlinux*, which is an uncompressed kernel.
3. The *System.map* is created with the kernel compile from the *vmlinux*. *System.map* contains the address of all symbols exported by the kernel.
4. The bootsect.S in the *arch/i386/boot* is preprocessed depending from the target image compression. After this the assembler code compiled and converted into *bbootsect*.
4. The *setup.S* is preprocessed also depending from the target image either to a *bsetup.S* or *setup.S* in a case of zImage. The result is then converted into *bsetup*.
5. The *vmlinux* is now compressed into a temporary *$tmppiggy.gz*.
6. The *head.S* and *misc.c* are compiled into a object files and *$tmppiggy.gz* linked into *piggy.o*.
7. Now these three object files are linked together either to *bvmlinux* or into *vmlinux* in the case of *zImage*.
8. The final stage is to convert /strip the created *bvmlinux* or *vmlinux .out* files and concatenate the *bootsect, setup* and *vmlinux.out* files into a *zImage*. In the case of *bzImage* the concatenated files are the corresponding ones.

Appendix B

ARCTIC PROCESSOR MODULE



The Arcitc processor module. Based on Motorola's MCF5272 microprocessor.