Getting Started With µClinux Development

Embedded Artists

Embedded Artists AB
Södra Promenaden 51
SE-211 38 Malmö
Sweden

info@EmbeddedArtists.com
http://www.EmbeddedArtists.com

Feedback

We appreciate any feedback you may have for improvements on this document. Please send your comments to support@EmbeddedArtists.com.

Table of Contents

1 Introduction

This book is designed to help you get acquainted with µClinux, the Linux distribution for processors without a Memory Management Unit (MMU). The book has a practical approach with lots of step-by-step guides. The guides have been designed around the Embedded Artists LPC2468 OEM Board and LPC2478 OEM Board with appropriate Base Board and for the Embedded Artists µClinux port for the LPC24xx microcontroller.

The development environment used for the exercises is based on a Debian Etch Linux distribution which is distributed as a VMware image that can be run in, for example, the VMware Player available for Windows PCs as well as Linux PCs, i.e., a virtualization approach has been chosen for the Debian Etch distribution.

It is not necessary to have expert knowledge about using Linux in order to understand the content of this book or to do the exercises since one part of the book will cover the basics of using Linux.

Besides describing µClinux the Universal boot loader (U-boot) will also be covered in this book since without a boot loader it will be difficult to get µClinux up-and-running. There should be enough information in this book to get you working with µClinux on an Embedded System.

1.1 What is an Embedded System?

The term Embedded Systems is generally defined to mean a computer system designed to solve one or a very few specific functions. These functions may need to be performed during long periods of time without interruption or even interaction with a person. Because of this they must in general be reliable and stable, maybe even meet real-time and safety critical requirements. It is not acceptable to have to reboot an embedded system every day or even on a weekly or monthly basis, they may have to be able to run continually for several years. The computer system is *embedded* in a sense where it is put into a device in a way where the device is not perceived as being a computer system.

This term is quite general and applicable for a lot of different devices found in the everyday world today. There are in fact a lot more embedded systems around you than you would probably imagine, ranging from simple sensors such as a thermostat, thermometer or motion sensor to a TV, washing machine, mobile phone or parts in a modern car or airplane. A modern car today can have 60 or more embedded computers controlling everything from fuel injection, the power windows, airbags and brakes.

The opposite of an Embedded System is a general purpose computer system that can be used to perform many types of tasks and run many types of applications. The Personal Computer (PC) found in many homes today is a general purpose computer system. It can be used for word processing, photo editing, software development, web browsing, entertainment (play games, listen to music, watch movies), heavy computational tasks and much more.

The mobile phone is on the list of embedded devices, but the question is if this is still true. When the mobile phone could only be used to make a phone call and maybe send and receive text messages the definition would apply for a mobile phone, but the phones on the market today can be used for a lot more; take photographs, play music, browse the web, send and receive e-mails, navigation (using GPS), and lots more, i.e., it is more of a general purpose computer system. This last paragraph just want to point out that sometimes it might be difficult to say if a device is really considered to be an embedded system in the true original meaning of the term. Nevertheless it is a device with a computer system probably in the need for an operating system.

1.2 The Operating System

Why would an Operating System (OS) be needed in an embedded system? First of all it is not sure it would be needed at all. This depends on the situation and what the embedded system is supposed to do and which problem it must solve. For the simplest device only performing one task such as regularly reading a sensor value, an operating system would probably not be needed, while a more complex device where several sensors should be read, a display regularly updated, and data sent onto a wireless network an operating system would probably be a necessity.

The responsibility of the operating system is to manage the resources, such as processing power (access to the processor), memory, and other input- and output devices attached to the computer system. It lies as a layer between the applications and the actual hardware making sure access is handled in a fair and controlled way. Many different types of operating systems exist and below is a list of a couple of different types to know about.

- **Multitasking or single-tasking OS** – In a multitasking OS multiple tasks (also called processes) can be performed simultaneously although the computer system only has one CPU. It is the operating systems responsibility to divide the access time to the CPU into smaller parts, time slices, and to schedule the processes so that they are given one or more time slices at a time. How much time that is given to a process is dependent on a specific scheduling algorithm.

  In a single tasking OS several processes may exist, but only one will run at any given time and the next process will not start to run until the first has stopped executing.
- **Real-Time Operating System (RTOS)** – This is an operating system usually found in embedded systems. The applications running in this OS need to react on input and deliver an output in real-time, sometimes with requirements on guaranteeing that deadlines are met and that the behaviour is deterministic. The RTOS is usually a multitasking operating system allowing several applications to run simultaneously.
- **Multi-user or single-user** – As the name implies a multi-user operating system allows for several users to have concurrent access to the computer while a single-user system only allows one user access. A server is typically running a multi-user operating system while a mobile phone usually only needs a single-user operating system.

1.3 Choosing Linux

In the annual survey conducted in the year 2007 by Embedded System Design, see ref [1], the participants were asked which operating system *type* they would be likely to use in their next embedded project. The majority would chose a commercial OS, but the number of developers choosing an open-source OS is high; 27 % as shown in Figure 1. Looking at the trend it reveals that the number of developers choosing a commercial OS is dropping.

According to the same survey 21 % of the participants were already using Linux as the open-source OS in their embedded project and 31 % were likely to use it in the next 6 to 12 months. The reason for choosing Linux was mostly because of low cost, adaptability/extensibility of the OS and personal control of its features and migration.

The survey shows that Linux, a multi-user and multitasking operating system, has become popular to use also in embedded systems, not only desktops and servers and that low cost is the prominent factor when choosing an open-source operating system for an embedded project.

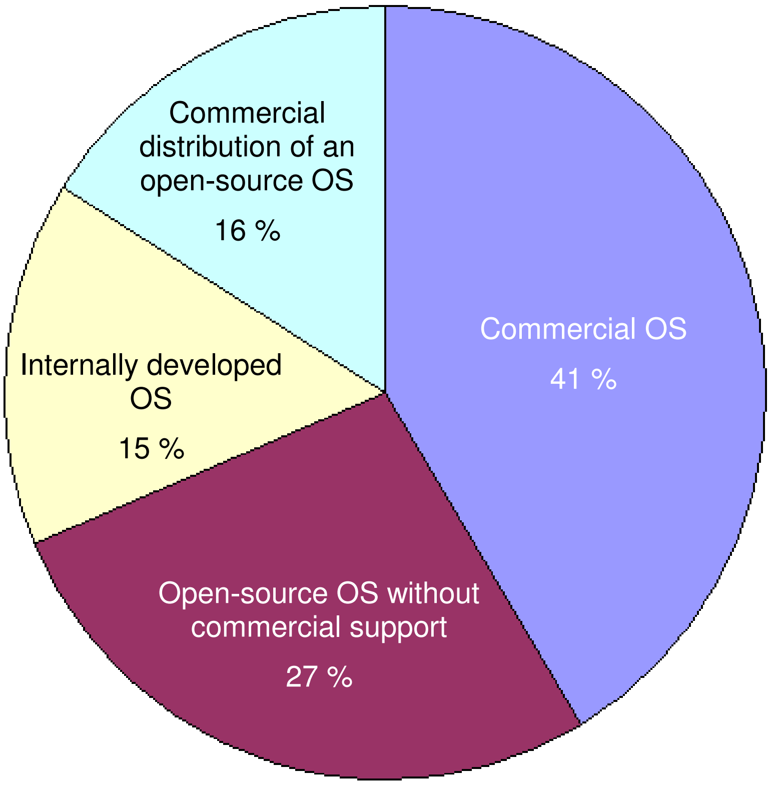


Figure 1 Survey of which type of operating system to use in an embedded project

Low cost can of course not be the only factor when choosing an OS for an embedded project. There are a lot of other factors to take into consideration as well. First of all the cost might not be as low as expected. If Linux hasn't already been ported to the hardware being used in the project a significant amount of time must be put into this. If no one in the organization has any experience working with Linux a lot of training and support might be needed which takes time and comes with a cost. Choosing a commercial OS would most likely also need training and support, but that might be included in the price for the OS.

Another aspect to take into consideration is if the software licenses (GPL and usually LGPL) for the Linux kernel and libraries can be used together with the business model and commercial terms that must apply for the product being developed.

If the project must meet real-time requirements, further investigations must be performed to see if Linux will meet those requirements, especially if it is *hard* real-time requirements that must be met. Improvements have been made to the Linux kernel related to real-time capabilities, for example, the RT preempt patch. There are also projects that with different kind of solutions add real-time capabilities to an embedded project using Linux as its base OS. One solution is to have a dual-kernel approach, read more about this in section 2.5 .

The benefits of using Linux are, however, many. Linux has a very large community of developers; it is a well-tested, well proven and stable operating system. If a critical error is discovered a patch will usually be available on short notice. Linux has got a vast number of software libraries as well as applications freely available which can get a head start on a project where you would otherwise spend a lot of time developing this functionality (or money to buy the functionality).

If the project requires applications with User Interfaces and a lot of graphics or if it must be network enabled, Linux is probably a good choice since a lot of this functionality is available.

If you have come to the conclusion that Linux is the operating system for your embedded project and that the microcontroller you are using is lacking a memory management unit, the µClinux distribution is the right choice for you and this book will help you start to investigate the characteristics and features of µClinux.

1.4 Organization of This Book

This book is organized in one theoretical part and a second more practically oriented part with mostly step-by-step guides. Chapters 1 to 7 are theoretically oriented and chapters 8 to 12 are more practically oriented.

- **Chapter 2** – *Linux vs. µClinux*
  Compares Linux with µClinux and highlights the differences, limitations and benefits.
- **Chapter 3** - *The µClinux Port*
  Gives an introduction to how the Linux kernel is organized and highlights those parts you need to know more about before porting Linux to a new platform. Platform specific parts of the µClinux distribution are also described in this chapter.
- **Chapter 4** - *Boot Loader*
  This chapter gives an introduction to what a boot loader is and then describes the Universal boot loader, u-boot, in more detail.
- **Chapter 5** - *Device Drivers*
  The first part of this chapter gives an introduction to what a device driver is and the second part describes the devices drivers that have been developed for the Embedded Artists µClinux distribution.
- **Chapter 6** - *Application Development*
  Explain what it means to develop applications that will execute in a Linux environment.
- **Chapter 7** - *Development Environment*
  Virtualization, the VMware Player and the Debian Linux distribution are covered in this chapter.
- **Chapter 8** - *Guides – VMware Player*
  This chapter helps you getting started with and using the VMware Player.
- **Chapter 9** - *Guides – Debian Linux*
  Describes how you can get started with the Debian Linux distribution, working with the shell, the desktop, file system, package management and much more.
- **Chapter 10** - *Guides – U-boot*
  This chapter guides you through building the u-boot, working with the u-boot environment and describes a number of different booting options available in the u-boot.
- **Chapter 11** - *Guides – µClinux*
  Guides you through building µClinux, the startup scripts, which users are available, a lot of network related functionality and how to use the device drivers that have been developed for the distribution.
- **Chapter 12** - *Guides – Create Your Own SDK*
  The final chapter guide you through setting up your own development environment in a Debian Etch distribution. The chapter describes which tools you need to install and how to build both the u-boot and µClinux.

1.5 Conventions in This Book

A number of conventions have been used throughout the book to help the reader better understand the content of the book.

`Constant width text` – is used for file system paths and command, utility and tool names.

```
$ This field illustrates user input in a terminal running on the
development workstation, i.e., on the workstation where you edit,
configure and build, for example, µClinux
```

```
# This field illustrates user input on the target hardware, i.e.,
input given to the terminal attached to the LPC2468 OEM Board or
LPC2478 OEM Board
```

```
This field is used to illustrate example code or excerpt from a
document.
```

2 Linux vs. µClinux

2.1 Introduction

This chapter will give an overview of Linux and µClinux and mention the modifications that have been necessary to make in order to get Linux running on MMU-less processors. There is also a short section about Linux and real-time capabilities.

2.2 Linux

Linux is multi-user and true multitasking operating system based on the Linux kernel (the kernel is the heart of the operating system). The first that usually comes to mind when talking about Linux is *open-source software,* which means that all the source code is freely available for everyone to look at, modify, use, and redistribute.

The Linux kernel was originally developed by Linus Torvalds in 1991 while he was attending the University of Helsinki in Finland. The inspiration was MINIX, a Unix-like operating system intended for academic use. It is said that Linus wasn't satisfied with MINIX and didn't get attention for his improvement ideas by the author of MINIX. This should be the reason for why he started to develop his own OS which should be non-commercial and open for everyone to use and modify/improve. Besides the kernel, Linux is also bundled with utilities and libraries from the GNU project see ref [2], and that is why Linux is sometimes known as GNU/Linux.

2.2.1 Different Aspects

If asked to learn more about Linux you should ask which aspect or segment of Linux to know more about. Different people could mean different things when they want to know more about Linux. In Figure 2 Linux has been divided into 4 different segments.

- **Using Linux** – The majority of people wants to know more about how to *use* Linux as an operating system for everyday work, such as a replacement for Microsoft's Windows or Apple's Mac OS X. It could be used for word processing, software development, web browsing, playing games, and so on. A different usage could be server (web server, e-mail server …) administration.
- **Application development** – There is also a lot to know about when it comes to developing applications for Linux; which APIs are available, how do you access a file, how to open a network connection, how to output graphics to the screen, and how to communicate between applications.
- **Porting and Driver Development** – For an embedded developer this area could be the most interesting to know more about. Linux may have to be ported to new hardware and drivers need to be developed for new devices. There is a lot to know about in this area.
- **Open Source Software** – It is important to know that Linux is freely available for everyone, but that doesn't mean that it is allowed to do whatever you want with the code since it comes with a software license, i.e., there are some rules to follow. The license for the Linux kernel is the GNU Public License (GPL) which, for example, has restrictions about how the code may be used in commercial applications. It is important to study the licenses and know which restrictions they set before developing a product based on open-source. You may end up in a situation where all your own source code is *affected* by the license used by the open source code.

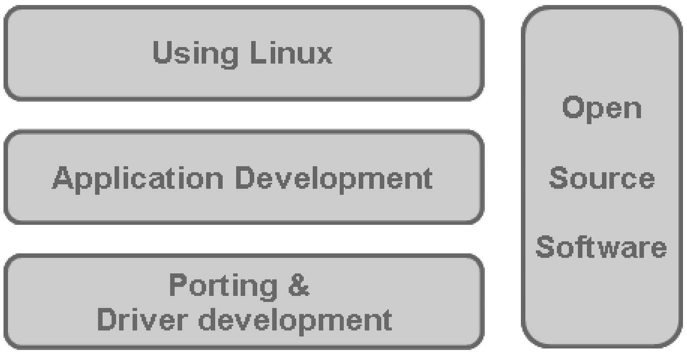


Figure 2 Different Aspects of Linux

This book will cover a small part or at least give an introduction to most of these aspects (the open source software and licenses part will not be covered).

2.2.2 Important Features

It is interesting to highlight different main features about an operating system. It will tell you a little bit about what the operating system is capable of without going into too many details.

- **Multi-user** – It is designed to support several users on the same machine at the same time.
- **Multitasking** – It allows several application/processes to run simultaneously and if the hardware has support for several processors the applications will truly execute in parallel.
- **Highly portable** – The Linux source code has a structure that simplifies porting and it has already been ported to numerous architectures and platforms. It is likely that the processor/microcontroller being used in your project already has a port available that only needs minor changes to comply with the entire hardware design.
- **Dynamic loadable modules** – It is possible to dynamically at runtime load or unload modules in the kernel. This is a powerful feature which allows the core part of the kernel to remain rather small, but possible to extend with new functionality through the use of modules.
- **Networking support** – The networking support in Linux is excellent. The first thing you usually want up-and-running when porting Linux to a new platform is the networking functionality. After this has been done continued porting and development will be much easier.
- **C code** – The kernel is almost entirely implemented in C code. Small parts have been implemented in assembly language for performance reasons.

Linux has also support for and is designed around virtual memory, paging and memory protection which require that the hardware has support for a Memory Management Unit (MMU), something not all processors have.

2.3 Memory Management Unit

A Memory Management Unit (MMU) is a hardware component responsible for handling access to the memory. It is usually used to implement support for virtual memory, i.e., a technique that gives an application or process the impression that it has a contiguous address space to work with while in reality the space could be physically fragmented. For an application it could also look as though it has exclusive access to the entire memory while in reality several applications share the physical memory.

The memory in a computer system could be thought of as a table with several entries (or pages), each representing storage space for data. Looking at Figure 3 there is one table representing the virtual memory, i.e., the memory the application will access, and another table representing the physical memory available in the hardware.

If an application is trying to access address 0 (in the example seen in Figure 3) it will actually access address 12288 in the physical memory, but does not need to know that this mapping is happening since it is handled by the MMU. The mapping shown in Figure 3 is also known as a *page table*. If virtual memory isn't supported the address the application is accessing is the same as the address of the physical memory, i.e., address 0 and not 12288 in the physical memory would be accessed in the example described earlier.

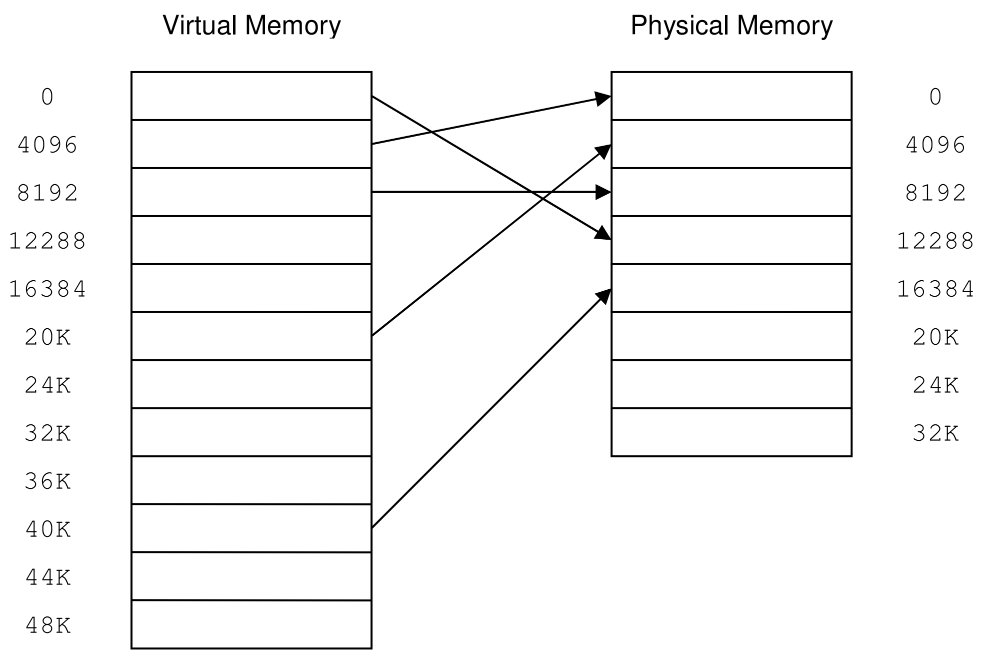


Figure 3 Virtual memory mapped onto physical memory

For each application to be able to believe that it has access to the entire memory each application will have its own page table with its own mappings. If the address space is only 48KB it would be no problem to keep this page table in memory for each process since it doesn't take that much space, but for a large address space (for example 32 bits which give 4GB) the page table would be huge (1 million entries if each page is 4KB). When the page table is large the actual table lookup of an address mapping could be relatively time consuming resulting in a slow system.

The described issues have of course been thought of and solutions exist. There is, for example, a technique of having multilevel page tables where only those page tables actually being used is kept in memory, thus saving space. There is also a technique called *translation lookaside buffer* (TLB) which is a small memory that will keep a mapping (virtual to physical) of the most frequently used pages for fast access. The usual behaviour of an application is that a small number of pages are heavily used and with the TLB the mapping between virtual address and physical address doesn't have to be expensive. The TLB is often located inside the MMU.

2.3.1 Swapping

Swapping is a technique of moving application code from storage, such as a hard disk, into memory or vice versa, i.e. from memory to storage. When an application starts the complete application image does not have to be loaded into memory, only those parts that are initially accessed. Other parts of the application will be loaded as they are needed which is also known as *demand paging*.

Looking at Figure 3 there are pages in the virtual memory which are not mapped to any physical memory. When the application tries to access such a page the MMU will detect this and the page being referenced will be loaded from storage into memory. If no physical memory is available the least frequently used page in the physical memory will be swapped out to storage before the new page is swapped into memory.

An MMU is required for swapping and demand paging, but even if the processor has support for an MMU it is not certain that swapping and demand paging is used in embedded systems because of limited and/or slow storage space.

2.3.2 Memory protection

Memory protection prevents applications and processes from accessing memory or address space that doesn't belong to them. With an MMU and support for virtual memory there is also memory protection since each process is assigned its own address space and thus not allowed to access anything outside its address space. There could also be a Memory Protection Unit (MPU) available on the hardware to support memory protection.

Memory protection is an important feature since it will enhance the stability of a system. When an application has been limited only to access memory assigned to the application it won't be possible to corrupt memory used by other applications. If one application crashes it won't affect the other applications or the operating system.

When Linux was originally designed and implemented it was dependent on the support of virtual memory and therefore support for an MMU. Since many processors used in the embedded world lack an MMU, µClinux was developed so that Linux could run on those processors.

2.4 µClinux

µClinux, see ref [3], stands for microcontroller Linux and is a version of Linux that can run on processors without a Memory Management Unit. The project was started by D. Jeff Dionne and Kenneth Albanowski in 1998 and was based on the Linux 2.0 kernel. The goal was to get it running on a Motorola DragonBall 68k processor.

Since version 2.5.46 of the Linux kernel major parts of µClinux is now integrated with the main kernel. Besides the updates in the kernel the µClinux distribution also contains a collection of user applications, libraries and tool chains.

The distribution has become very popular and has been ported to several architectures and is running in many products. Some of the supported architectures are.

- ARM
- Blackfin
- Freescale m68k
- Hitachi H8
- Intel i960
- Xilinx MicroBlaze
- Motorola Coldfire

2.4.1 Limitations

There are of course a few limitations with running Linux on an MMU-less processor. Some of the limitations have been mentioned in section 2.3 , but a few more are described in this section.

- **No separation of address space** – This means that all applications and processes share the same address space and thereby makes it possible for a process to corrupt the address space of another process, i.e., if one application crashes, the entire system may crash.

  This also leads to the fact that there will be no real separation between user space and kernel space; memory protection is used to solve this. An application running in user space can corrupt kernel space memory when using a MMU-less system.
- **No demand paging** – This means that a whole application must be read into RAM instead of just the pages being accessed, i.e., more memory consumption. This also means that there is no support for swapping, i.e., swapping code from RAM to and from, for example, a hard drive. This is, however, not a major issue for embedded devices.
- **Dynamic memory** – In Linux, dynamic memory allocation, for example, by using the library function `malloc`, is achieved by increasing the virtual address space belonging to the process doing the allocation. This means that physical memory won't be allocated until the virtual memory is actually used. Since there is no support for virtual memory in µClinux this technique cannot be used. Instead dynamic memory allocation is implemented by using a global memory pool which is available for all processes. This also means that one process could allocate all free memory making it impossible for other processes to allocate memory.
- **mmap** – The `mmap` system call is used to map a file in the file system into memory so that the file can be accessed as though you where accessing raw memory. The technique used to achieve this is demand paging, but since µClinux lack support for demand paging the `mmap` system call needs to be implemented differently.

  What it basically means is that memory may need to be allocated for the entire file that is memory mapped and the content of the file copied to the allocated piece of memory. The exception is if you have a read-only file system where the files are guaranteed to be stored sequentially and contiguously.
- **fork** – `fork` is a system call used by an application to create a new process. The way it works is that it creates a copy of the current process making what is called a child process. The child process will have a separate address space from the parent process, but with a copy of the parent's data. The copy procedure is however a bit special for fork since it won't automatically copy the data to new physical memory. Instead it will utilize demand paging and only copy the data when it is actually accessed for modification. In other words all read-only memory and memory not yet modified by child or parent process will use the same physical memory and thereby not waste any memory.

  The fork system call cannot be efficiently implemented without an MMU and therefore it doesn't exist in µClinux. An alternative system call, however with significant differences, that is available in µClinux is the `vfork` system call.

2.4.2 Benefits

Despite of the limitations mentioned in the previous section the great benefit with µClinux is to be able to run Linux at all on MMU-less processors. Linux is today a well-known operating system with lots of applications developed for it and lots of developers familiar with it. Moving Linux to the embedded world opens up for more application development and maybe even more powerful products since much source code is already available for everything from math libraries, network management to graphical libraries.

Other benefits with µClinux are:

- **Better performance** in some situations. An MMU can introduce significant time overheads and that is why the MMU sometimes is turned off in systems with hard real-time constraints. In other words running Linux without a MMU could lead to better performance.
- **Execute In Place (XIP) –** This is a technique where an application executes directly from storage such as flash and ROM. In standard Linux only a few parts of an application are loaded into memory before it is executed. Demand paging is then used to load more parts (pages) of the application when needed. With XIP the application doesn't have to be loaded into memory, but can instead be executed directly from storage and thereby save RAM.

2.4.3 Modifications

It has been necessary to do a few clever changes to Linux in order to get it running on MMU-less processors. First of all the memory management part of Linux has of obvious reasons been forced to be modified due to the lack of a MMU, but another important part of getting Linux to run on MMU-less processors is to get a toolchain, i.e., compiler, linker, loader, etc, modified to support the new way of utilizing the memory. This is actually where a large part of the effort has been spent when getting µClinux up-and-running on a new architecture.

The toolchain used when compiling applications for µClinux has been modified so that the porting effort of moving applications from standard Linux is minimized. The standard execution file format in Linux is the Executable and Linkable Format (ELF), but for µClinux the toolchain generates a new file format called binary flat format (bFLT). This format is a compact format, i.e., more memory efficient than ELF. The binary flat format also supports relocation which means that references in the code can be rearranged at load time to where the executable is located.

For more details about the binary flat file format, how relocation and the new loader in the Linux kernel work, read the book "Embedded Linux System Design and Development", see ref [4].

2.5 Linux and Real-Time

A system that can meet real-time requirements is said to be a system that can and must respond, to a request, within defined deadlines. Real-time systems are often divided into systems that meet either hard or soft real-time requirements. A system that must meet hard real-time requirements must have guaranteed and deterministic response times, while soft real-time requirements will tolerate smaller latencies. An example of a system that must meet hard real-time requirements is the control system of an airplane. It could lead to a disaster if the system didn't respond within defined deadlines.

A streaming video system is an example of a soft real-time application where it is no catastrophe if the deadline is sometimes not met. It just means that the quality of the video will be slightly worse during short periods of time.

Linux cannot guarantee to respond deterministically and within set deadlines and is therefore not an OS with hard real-time capabilities. There are, however, a couple of projects that aim to improve the real-time capabilities of Linux and this section will mention the most well-known. Only an overview will be given not in-depth analysis of each project.

2.5.1 Real-time Preemption patch

The Real-time Preemption patch, RT-preempt, is a patch for the Linux kernel that makes it fully preemptible. The work was started by Ingo Molnar and many of the features and capabilities from the patch have now made it into the mainline kernel. The patch introduces, for example, high resolution timers, preemptible locking primitives, priority inheritance for in-kernel spinlocks and semaphores, conversion of interrupt handlers into preemptible kernel threads and more.

More information about this project can be found on their website, see ref [5], and the patch is available at kernel.org.

2.5.2 RTLinux

RTLinux is a short for Real-Time Linux and is an operating system that will offer real-time capabilities and at the same time run ordinary Linux. The approach RTLinux has selected is to run the Linux kernel as one of its processes with a low priority. The RTLinux kernel will also be the one that primarily receives all interrupts and only if there is no real-time process to run, the interrupts will be passed to the Linux kernel.

RTLinux offers lock-free queues and shared memory as ways of communicating between user applications in Linux and processes running in the RTLinux kernel.

RTLinux started as a project at New Mexico Institute of Mining and Technology, but has since then been acquired by Wind River Systems and is now part of their product portfolio, see ref [6]. There is also a version of RTLinux called Open RTLinux which is available for academic, research and other open software projects, see ref [7].

2.5.3 RTAI

RTAI – the Real-Time Application Interface for Linux lets you write applications with strict timing constraints. RTAI is similar to RTLinux in the sense that it is also running the Linux kernel as a low priority process which is basically only allowed to run when there are no real-time processes running. One difference between RTLinux and RTAI is the way the Linux kernel is modified to support respective real-time kernel. RTLinux does a lot of changes in the kernel code whereas RTAI has minimized the needed modifications in the Linux kernel and instead introduced a hardware abstraction layer (HAL). RTAI basically consist of a patch that introduces the HAL and a number of real-time related services.

The HAL patch for RTAI is an Adeos (Adaptive Domain Environment for Operating Systems) based patch, see ref [8]. Adeos is a nanokernel HAL that operates between the hardware and the operating system that runs on it and handles, for example, sharing of hardware resources.

RTAI is an open source project and more information can be found on the project website, see ref [9].

2.5.4 Xenomai

The following description is from the Xenomai project website, see ref [10].

Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment.

Xenomai is, as RTAI, also using the Adeos hardware abstraction layer as the base for the kernel. There are other similarities, but also of course differences. The Xenomai project has, for example, as one of its main goals to ease the transition from working with a traditional RTOS to instead work with the Linux kernel. This is, for example, achieved by not having to completely rewrite the applications running on top of the traditional RTOS. Xenomai provides emulators for VxWorks, pSOS+, VRTX, and uITRON real-time APIs.

3 The µClinux Port

3.1 Introduction

This chapter will give an introduction to the Linux kernel and the µClinux port done for the LPC24xx family of processors. It is by no means a complete description of what is needed when porting the Linux kernel and µClinux distribution to a new architecture. Instead it should be seen as an introduction that will highlight some of the parts that are important to know about before taking up the challenge of porting Linux to a new architecture.

3.2 The Kernel Source Code

When starting to work with the Linux kernel it is important to know about how the source code for the kernel is organized. Downloading and uncompressing the kernel will give you a long list of directories and it might be difficult to know where to start looking when the kernel is to be ported to a new architecture or when developing a new device driver.

This section will go through all the directories found at the root level of the kernel source tree and explain what they contain. Other sections of this chapter will then go into more detail about those directories that are interesting from a porting and driver development perspective.

- **`arch`** – This is where most of the architecture specific code is located. There are, for example, sub-directories called `arm`, `i386`, `m68k`, `powerpc`, and `sparc`, which all represent their respective architecture. The source code in this directory takes up a large part of the kernel sources. Running a disk usage tool (`du`) on a vanilla (plain, unmodified and unpatched) 2.6.21 kernel shows that about **20%** of the code is located here. This directory will be explained a bit more later on in this chapter, especially the parts that are interesting from a porting perspective.
- **`block`** – Contains the code for the block layer in the kernel. The block layer is responsible for block devices (a hard disk is handled as a block device).
- **`crypto`** – This directory contains cryptographic algorithms such as AES, Blowfish, SHA1, and MD5 hash.
- **`Documentation`** – A lot of descriptions and documents for different parts of the kernel is located in this directory. There is, for example, a file called `CodingStyle` which describes the preferred coding style for the kernel. Start reading the file `00-INDEX` to get an overview of what the documentation directory contains.
- **`drivers`** – This is where the implementation of the device drivers is located. A lot of device drivers for many types of peripherals and devices are available in the kernel. Looking at the disk usage for this directory shows that more than **40%** of the code is located here. If you have a relatively standard device it is likely that a device driver is already available in the kernel. It should also be noted that when porting Linux to a new platform most of the time will be spent on writing/porting drivers.
- **`fs`** – This directory contains different file system implementations, such as FAT, ext2, ext3, cramfs, ntfs, and many more. The file system is an important part of the kernel and if you become a more advanced user of Linux you will soon realize that almost everything is represented as a file in Linux. The disk usage for this directory shows that about **9%** of the code in the kernel is located here.
- **`include`** – This is where “shared” header files are located, i.e., header files that may need to be included and used in different subsystems in the kernel. This directory also contains architecture specific files. Symbolic links are used during the build process to find the correct architecture specific files to include. More about this later

in this section. The header files take up more than **16%** of the total kernel disk usage.

- **`init`** – The kernel initialization code is located in this directory. There is, for example, a `main.c` file which contains the `start_kernel` function.
- **`ipc`** – This directory contains code for Inter-Process Communication (IPC), such as message queues and semaphores.
- **`kernel`** – This is the location of the core parts of the kernel such as the scheduler, timers, interrupt management, power management, threads, and more.
- **`lib`** – This directory contains library functions such as CRC, compression, text search support, priority search tree, random generator, sorting routine, kernel command line parsing, and more.
- **`mm`** – This is where the memory manager code is located. Here is code for memory buffer pool support, paging support, file mmap (memory map) functionality, and much more.
- **`net`** – Linux is a very network oriented operating system and hence there is a lot of networking related code for the kernel. This directory contains implementation of numerous networking protocols such as TCP/IP (IPv4 and IPv6), Ethernet-type device handling, 802.11 Networking stack, X.25 packet layer, and much more. The disk usage for this directory shows that about **5%** of the code in the kernel is located here.
- **`scripts`** – This directory contains a lot of scripts especially used during the build process of the kernel.
- **`security`** – This is the location of the security related code in the kernel. Code for the Security-Enhanced Linux (SELinux) is, for example, available here.
- **`sound`** – Sound related code is located in this directory. There is quite a lot of code available for such devices as sound cards, USB related audio devices, ALSA ARM devices and much more. The sound related files take up close to **6%** of the total kernel disk usage.
- **`usr`** – This is a really small directory which contains code for `initramfs` (Initial RAM file system).

Besides all of these directories the root level also contains two interesting files; the `Makefile` which is the main `Makefile` for the Linux kernel. This is basically where the build process starts. If you are interested in the build process and have some basic knowledge about GNU make files, have a look at this file to get an understanding of which build rules that are available in the Linux kernel. It is in this make file where the `ARCH` and `CROSS_COMPILE` variables are set. The `ARCH` variable is set to the architecture you intend to build the kernel for and the `CROSS_COMPILE` is set to the tool-chain intended to be used when building the kernel. By default these variables have empty values and can also be set as environment variables, but for the Embedded Artists patch the `ARCH` variable has been given the value `arm` and the `CROSS_COMPILE` variable has been given the value `arm-linux-`. Later in the same file you can see how the `CROSS_COMPILE` variable is used. The build tools, such as the compiler or the linker aren't invoked directly. Instead variables are setup to be used whenever a build tool must be called upon, see the example below.

```
LD          = $(CROSS_COMPILE)ld
CC          = $(CROSS_COMPILE)gcc
```

In our case this means that the `LD` variable (linker) will be expanded to `arm-linux-ld` and the CC variable (compiler) will be expanded to `arm-linux-gcc`.

The second file worth mentioning is the `README` file which contains some basic information about Linux; what it is, how to install the kernel, how to configure the kernel, and how to compile the kernel.

3.2.1 Configuring the Kernel

Since the Linux kernel can be used for many purposes and on many architectures it has become highly configurable. All architectures may, for example, not support all drivers available in the kernel source tree and therefore it must be possible to remove those that are not supported when building the kernel.

The Linux configuration system is called Kconfig and consists of a number of files written in the Kconfig language. With this language it is possible to define configuration options, dependencies, input prompt, help texts and default values.

When configuring the kernel a tool (several exists) is executed that will parse the Kconfig files and present a configuration tree where the functionality that must be included in the kernel is selected.

Below is an example of how a Kconfig file could look like. The example is an excerpt from the configuration file for the I^2C subsystem. A menu called I^2C support is presented with a number of configuration options. Two of the options are shown in the example, where the second actually depend on the first option. This means that the second option won't be visible in the configuration menu unless the first option has been selected.

```
#
# I2C subsystem configuration
#

menu "I2C support"

config I2C
    tristate "I2C support"
    ---help---
      I2C (pronounce: I-square-C) is a slow serial bus protocol used in
      many micro controller applications and developed by Philips. SMBus,
      or System Management Bus is a subset of the I2C protocol.  More
      information is contained in the directory
      <file:Documentation/i2c/>, especially in the file called "summary"
      there.

      Both I2C and SMBus are supported here. You will need this for
      hardware sensors support, and also for Video For Linux support.

      If you want I2C support, you should say Y here and also to the
      specific driver for your bus adapter(s) below.

      This I2C support can also be built as a module.  If so, the module
      will be called i2c-core.

config I2C_CHARDEV
    tristate "I2C device interface"
    depends on I2C
    help
      Say Y here to use i2c-* device files, usually found in the /dev
```

```
	  directory on your system.  They make it possible to have user-space
	  programs use the I2C bus.  Information on how to do this is
	  contained in the file <file:Documentation/i2c/dev-interface>.

	  This support is also available as a module.  If so, the module
	  will be called i2c-dev.

source drivers/i2c/algos/Kconfig
```

The last part of the Kconfig example shows that a second Kconfig file is included by using the *source* keyword. The included Kconfig file contains configuration options for I^2C algorithms.

The configuration options defined in the Kconfig files can then be used in Makefiles and source files to select the functionality to include in the kernel. Below is an excerpt from the Makefile containing the build rules for the I^2C subsystem. The files to include when building the kernel are selected using the configuration options shown in the Kconfig file above.

```
#
# Makefile for the i2c core.
#

obj-$(CONFIG_I2C)		+= i2c-core.o
obj-$(CONFIG_I2C_CHARDEV)	+= i2c-dev.o
```

A similar approach is used for source files. The pre-processor directives can be used to select if code will be included or not. In the example below there is some code that will only be included if the configuration option for ARM has been enabled.

```
static int __exit tps65010_detach_client(struct i2c_client
*client)
{
    struct tps65010         *tps;

    tps = container_of(client, struct tps65010, client);
    free_irq(tps->irq, tps);

#ifdef    CONFIG_ARM
    if (machine_is_omap_h2())
          omap_free_gpio(58);
    if (machine_is_omap_osk())
          omap_free_gpio(OMAP_MPUIO(1));
#endif

    cancel_delayed_work(&tps->work);
    flush_scheduled_work();
    debugfs_remove(tps->file);
    if (i2c_detach_client(client) == 0)
          kfree(tps);
    the_tps = NULL;

    return 0;
}
```

3.3 Architecture Specifics in the Kernel

Looking closer at the `arch` directory we find that it only contains sub-directories and no files. Each sub-directory represents an architecture and a few examples of architectures that are available in the Linux kernel are `arm`, `i386`, `m68k`, `powerpc`, and `sparc`. For this book we will focus on the ARM architecture since the target processor we will use in all exercises and examples is from the NXP LPC 24xx family, an ARM7TDMI based processor.

The `arch/arm` directory mostly consists of sub-directories and a few files related to the build process (`Makefile`) and kernel configuration. Most of the sub-directories begin with `mach-` and these directories contain target specific code. There are many variants of ARM processors, with different capabilities, and therefore there is a need to further divide the `arm` directory into more target specific directories. Target specific code for the LPC24xx family is, for the Embedded Artists distribution, located in the `arch/arm/mach-lpc22xx` directory. The reason for naming the directory `mach-lpc22xx` is because the LPC24xx-port has been based on a port that was done for the LPC22xx family. A better name for the directory would be `mach-lpc2xxx` since it is possible to have one port for this entire LPC2xxx family (note that not all LPC2xxx processors support the use of external memory and can therefore not run Linux, the available RAM would be too small).

The `arch/arm` directory also contains a number of sub-directories that are common for all ARM based processors. The sub-directory named `kernel`, for example, contains the kernel startup entry point. For processors without MMU it is the file `head-nommu.S` which contains the entry point (`stext`). This entry point is implemented in ARM assembler code and will end by calling the `start_kernel` C-function.

The `arch/arm/mm` directory contains arm-specific parts of the Linux memory manager, such as code for handling an MMU or the lack of an MMU, mapping of IO ports, and more. The `arch/arm/lib` directory contains optimized library functions such as `memcpy`, `memmove`, and `memset`, all implemented in ARM assembler code.

3.3.1 The mach-lpc22xx Directory

The code that is specific for the LPC2xxx family of processors is located in `the mach-lpc22xx` directory and specifically the code that is specific for the Embedded Artists development boards using the LPC2468 or LPC2478 processor is located here.

There are routines that retrieve the processor clock frequency (the frequency has been set by the boot loader) and sets up a timer interrupt that will generate the tick in the kernel, i.e., call the `timer_tick` function. This function will update different timers in the kernel, which are used, for example, by the scheduler to decide how long a process has been running and when it is time to preempt that process and give a new process the chance to run.

There is also code for initializing and setting up interrupts, but more importantly there are files that handle device specific initialization and files that handle board specific initialization. For the Embedded Artists distribution the files below handle device and board specific initialization.

- `lpc24xx_devices.c` – The peripherals of the LPC24xx MCU are defined and registered as devices in this file. This file is common for both the LPC2468 and the LPC2478 microcontrollers.
- `lpc2468_ea_board.c` – This is a file that contains board specific initialization code for the Embedded Artists LPC2468 OEM Board with Base Board Basic.
- `lpc2478_ea_board.c` – This is a file that contains board specific initialization code for the Embedded Artists LPC2478 OEM Board with QVGA Base Board.

- lpc2478_microblox_board.c – This is a file that contains board specific initialization code for the Future Electronics LPC2478 Micro-Blox Board with LongBow Board.

To better understand how these files are used we will go through the I^2C peripheral as an example. Below is an excerpt from the lpc24xx_device.c file showing how the I^2C peripheral is registered as a platform device in the kernel.

The first part show how two resources are setup; a memory resource specifying the register base address for the I^2C device, and an interrupt resource specifying the interrupt number used by the I^2C device. Setting up this information in this file will make the driver code a bit more generic and less platform dependent. If µClinux would be ported to a new LPC processor with a different base address for its registers or a different interrupt number the driver code wouldn't have to be modified, only the device specific file.

The second part show how a platform device is initialized. The resource array is stored in the device structure as well as some platform specific data (lpc2xx_i2c_data) which will be covered when the board specific file is analyzed later in this section.

The last part shows the actual registration call where the device structure is registered with the kernel. The lpc2xxx_add_i2c0_device function is called during startup of the Linux kernel.

```
/*
 * lpc2xxx_i2c_pdata must be created and initialized in the
 * board specific file (see e.g. ea_lpc2478_board.c).
 */
extern struct lpc2xxx_i2c_pdata lpc2xxx_i2c_pdata;


#if defined(CONFIG_LPC2XXX_I2C0)
static struct resource lpc2xxx_i2c0_resource[] = {
    {
          .name        = "lpc2xxx-i2c0",
          .start       = APB_I2C0_BASE,
          .end         = APB_I2C0_BASE + APB_I2C0_SIZE - 1,
          .flags       = IORESOURCE_MEM,
    },

    {
          .name        = "lpc2xxx-i2c0",
          .start       = LPC2xxx_INTERRUPT_I2C0,
          .flags       = IORESOURCE_IRQ,
    },
};

static struct platform_device lpc2xxx_i2c0_device = {
    .name              = "lpc2xxx-i2c",
    .num_resources     = ARRAY_SIZE(lpc2xxx_i2c0_resource),
    .resource          = lpc2xxx_i2c0_resource,
    .dev = {
          .platform_data = &lpc2xxx_i2c_pdata,
    },
};

static void __init lpc2xxx_add_i2c0_device(void)
{
    platform_device_register(&lpc2xxx_i2c0_device);
}
```

The excerpt below is from the `lpc2478_ea_board.c` and it shows how board specific initialization is done for the I^{2}C device. First a structure with board specific data is initialized. It contains four fields; peripheral clock frequency, the frequency of the I^{2}C interfaces, a timeout value, and retry count.

The second part shows how the pin select registers are initialized so that the I^{2}C peripheral can be used. Only two pins need to be initialized. The `lpc2478_ea_init_i2c` function is called during startup of the Linux kernel.

```
/* I2C board specific data */

struct lpc2xxx_i2c_pdata lpc2xxx_i2c_pdata = {
    0,    // fpclk is set in the init function below
    100000,     // freq in Hz
    100,  // timeout
    3,    // retries
};

static void __init lpc2478_ea_init_i2c_pins(void)
{
    lpc2xxx_i2c_pdata.fpclk = lpc_get_fpclk();


#if defined(CONFIG_LPC2XXX_I2C0)
    lpc22xx_set_periph(LPC22XX_PIN_P0_28, 1, 0);    // SCL0
    lpc22xx_set_periph(LPC22XX_PIN_P0_27, 1, 0);    // SDA0
#endif
}
```

Section 5.11 will explain how the driver for the I^{2}C device is setup so that it will use the resources that were registered in the `lpc24xx_devices.c` file.

3.3.2 The Include Directory

Shared header files are located in the `/include` directory. This directory contains both generic header files that are independent of the platform and architecture specific header files. The architecture specific header files have been put in a structure similar to the structure used for the architecture specific source files. The architecture specific files are found in directories with the prefix `asm-`, for example, `/include/asm-arm` for the ARM architecture. Within the architecture specific directory there is both generic files valid for all different targets within the architecture and target specific directories (with prefix `arch-`), for example, `/include/asm-arm/arch-lpc22xx` for the LPC2xxx family of processors.

To avoid having to specify the architecture and target specific path when including a header file in a source file, the build process will setup symbolic links to the directories. There will be one symbolic link called `asm` that will point to the architecture specific directory and one symbolic link called `arch` pointing to the target specific directory.

```
// same as <asm-arm/arch-lpc22xx/io.h>
#include <asm/arch/io.h>

…
```

3.4 Board Specifics in µClinux

The µClinux distribution (not talking about the kernel now) contains a directory structure that makes it easy to support several different kinds of development boards. The root directory of µClinux contains a directory called `vendors` which contain, as the name suggests, vendor specific directories (such as `EmbeddedArtists` or `FutureElectronics`). A vendor specific directory then contains the boards supported by a specific vendor, for example, the `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/` for the LPC2478 OEM Board with QVGA Base Board and the `uClinux-dist/vendors/EmbeddedArtists/LPC2468OEM_Board/` for the LPC2468 OEM Board with Base Board Basic.

The vendor and board directory that will be used is selected during the configuration of µClinux, i.e., prior to building anything.

3.4.1 Makefile

The make file in the board directory contains build rules for generating file system images as well as creating the kernel boot image used by the U-boot as described in section 4.5.3 Parts of the make file used with the Embedded Artists configuration will be discussed below.

There is one make target called `romfs` which creates vendor and board specific directories and files in the file system that will be used as the root file system in Linux. In the excerpt below directories listed in the `ROMFS_DIRS` variable are created and files such as drivers and applications are copied to the file system using the `ROMFSINST` command.

```
romfs: drivers applications
    [ -d $(ROMFSDIR)/$$i ] || mkdir -p $(ROMFSDIR)
    for i in $(ROMFS_DIRS); do \
          [ -d $(ROMFSDIR)/$$i ] || mkdir -p $(ROMFSDIR)/$$i; \
    done

    for i in $(DEVICES); do \
          touch $(ROMFSDIR)/dev/@$$i; \
    done

    $(ROMFSINST) -s /var/tmp /tmp
    $(ROMFSINST) -s /bin /sbin
    $(ROMFSINST) /etc/rc
    $(ROMFSINST) /etc/inittab
    $(ROMFSINST) ../../Generic/romfs/etc/services /etc/services
    case "$(LINUXDIR)" in \
    *2.6.*) \
    $(ROMFSINST) -S drivers/2.6.x/adc/adc.ko /drivers/adc.ko & \
    $(ROMFSINST) -S drivers/2.6.x/pwm/pwm.ko /drivers/pwm.ko & \
    $(ROMFSINST) -S drivers/2.6.x/sfr/sfr.ko /drivers/sfr.ko & \
    $(ROMFSINST) -S drivers/2.6.x/lpc2468mmc/lpc2468mmc.ko
/drivers/lpc2468mmc.ko ;; \
         *) echo "ttyS0:linux:/bin/sh" >> $(ROMFSDIR)/etc/inittab
;; \
    esac

    $(ROMFSINST) applications/led /bin/led
    $(ROMFSINST) applications/key /bin/key
    $(ROMFSINST) applications/eeprom /bin/eeprom
    $(ROMFSINST) applications/calibrate /bin/calibrate
    $(ROMFSINST) /etc/motd
    $(ROMFSINST) /etc/passwd
    echo "$(VERSIONSTR) -- " `date` > $(ROMFSDIR)/etc/version
```

The make target called `image` is a bit more interesting since this target will create the file system and kernel images. It is these images that will be downloaded to the development board. There are a number of different actions in this target:

- `genromfs` – This is a tool that creates a romfs file system given a directory.
- `mkfs.jffs2` – This tool creates a JFFS2 file system image given the romfs directory.
- `mkcramfs` – this tool creates a cramfs file system image given the romfs directory.
- `gzip` – In this step the Linux kernel image is compressed. Compressing the Linux kernel will save storage space if the kernel image is, for example, stored in flash. The disadvantage is that the boot time will be a bit longer since the kernel must first be uncompressed before it can be started.
- `mkimage` – This tool creates the boot image used by the U-boot. Please note how the load address `0xa0008000` is set. Also note the use of the option –C to tell the tool that the image is compressed. If compression shouldn't be used remove the –C option and the gzip action at the row above.

```
image:

    [ -d $(IMAGEDIR) ] || mkdir -p $(IMAGEDIR)
    genromfs -v -V "ROMdisk" -f $(ROMFSIMG) -d $(ROMFSDIR)
…

…

    mkfs.jffs2 -d $(ROMFSDIR) -D dev_table.txt -l -o $(JFFS2IMG) -
e 128 -n -m none -p -v || echo "Warning: could not build
$(JFFS2IMG)"

    mkcramfs -v -D dev_table.txt $(ROMFSDIR) $(CRAMFSIMG) || echo
"Warning: could not build $(CRAMFSIMG)"

    gzip -c $(ROOTDIR)/$(LINUXDIR)/arch/arm/boot/Image >
$(COMPKERN)

    mkimage -A arm -O linux -T kernel -C gzip -a 0xa0008000 -e
0xa0008000 -n "Linux 2.6.21" -d $(COMPKERN) $(IMAGEDIR)/uLinux.bin
|| echo "Warning: could not build $(IMAGEDIR)/uLinux.bin"
```

3.4.2 Configuration Files

The board specific directory also contains configuration files. These files contain default values for the configuration choices used when configuring the Linux kernel, µClinux applications and libraries, uClibc, and architecture specific settings.

- `config.arch` – Architecture specific settings such as setting the ARCH variable to `arm` and also setting some compiler flags.
- `config.linux-2.6.x` – Configuration settings for the Linux kernel, such as choosing which drivers to enable and which CPU to use.
- `config.uClibc` – configuration of the uClibc library.

- `config.vendor` – configuration of µClinux applications and libraries.

3.4.3 Applications and Drivers

The board directory can also contain applications that the vendor would like to distribute with its board. For the Embedded Artists board the following applications are provided.

- `calibrate` – this is an application that calibrates the touch screen. The application is using the frame buffer to draw coordinates (as squares) for the user to touch when configuring the screen as well as the touch screen device.
- `eeprom` – this application is using the I^2C driver to read and write to the EEPROM.
- `key` – this application checks the state of the four buttons on the QVGA Base board.
- `led` – this application can be used turn the four LEDs on the QVGA Base Board on or off.

There are also some device drivers that for different reasons haven't been integrated in the kernel.

- `adc` – this is a character device driver that illustrates how to access the ADC device on the board.
- `joystick` – this is a driver that illustrates how the joystick on the QVGA Base Board can be controlled.
- `lpc2468mmc` – this is the driver for the MMC/SD card.
- `pwm` – this is a driver that controls the PWM output on the LPC24xx board.
- `sfr` – this is a driver called Special Function Driver and it allows user space applications to check and modify internal registers in the LPC24xx MCU.

4 Boot Loader

4.1 Introduction

On many computer systems the first program that will run when the system powers up is the boot loader. The boot loader's responsibility is to load the operating system kernel into memory and start the kernel's execution. To be able to achieve this the boot loader must begin by initializing the hardware, such as the processor, memory controller and the devices that are needed to be able to load the kernel image.

The devices that need to be initialized are highly dependent on the system. On a PC it is typically a hard drive or a floppy disk, but on an embedded system the kernel image could reside in flash memory, on a memory card, a network device, a USB memory stick or any other device that have a storage area. If a user need to take action during the boot procedure, for example, by selecting which image to load if several exists, the boot loader might also need to initialize a display or other output device that can make a user aware of what is happening and present a list of choices to the user.

Besides loading the kernel image the boot loader might also need to load an initial root file system and make that available to the operating system. This is especially true when using Linux on an embedded system which doesn't have a storage area for the file system.

Another important feature of a boot loader is the support for passing boot arguments to the operating system kernel. This could be arguments that tell the kernel where to find the root file system, how to setup the console for output, and other arguments that configures the behaviour of the kernel. This allows for building generic kernel images that can be used without modification on slightly different systems, but where the behaviour is controlled from the boot loader.

4.1.1 Different Boot Loaders

Many different boot loaders exist and some of the more common are briefly described below.

- **APEX** – This boot loader was primarily written to support the Sharp LH series of processors (now belonging to NXP). It is using the Linux kernel Kbuild build system and has support for several booting options; TFTP, FAT, EXT2 and JFFS2 file systems, NOR and NAND flash. More information and access to the software can be found at the APEX website, see ref [11].
- **RedBoot** – The Red Hat Embedded Debug and Bootstrap firmware, RedBoot, is a boot loader that is based on the eCos Hardware Abstraction Layer. It can boot from serial or Ethernet interfaces and can also offer debugging capabilities through the GNU Debugger (GDB). RedBoot has been ported to several architectures including ARM, ColdFire, MIPS, PowerPC and more. Documentation and source code are available at the RedBoot website, see ref [12].
- **U-Boot** – More about this boot loader in section 4.2
- **MicroMonitor** – This boot loader is centered around an extensible embedded flash file system called TFS. It supports network boot and flash file storage, features such as decompression and scriptable configuration management. There are ports available for ARM, ColdFire, MIPS, PowerPC, Blackfin, and more. The MicroMonitor website contains more information, see ref [13].
- **LILO** – The LInux LOader , LILO, has for a very long time been the number one boot loader for Linux on PCs. Since it is being used on standard PCs it is using the BIOS and the Master Boot Record on a hard disk or floppy disk.

- **GRUB** – The Grand Unified Boot loader, GRUB, is a boot loader primarily used with AMD and x86 systems. It is a multiboot boot loader which several different operating systems on a computer at once. The user can choose which operating system to run when the computer starts. GRUB is now becoming more common than LILO when using Linux on PCs. More information can be found at the GRUB website, see ref [14].

More boot loaders exist, but for the remaining part of chapter 4 we will focus on Das U-Boot.

4.2 Das U-Boot

Das U-boot also known as the Universal Boot Loader or u-boot for short, see ref [15], is an open-source boot loader that supports a wide range of different architectures such as ARM, PowerPC, XScale, x86, MIPS, Coldfire, 68k, and MicroBlaze. Many board configurations have been made available, for each architecture, by an active community. The u-boot boot loader has actually become the most widely used boot loader on ARM based systems.

Besides supporting a wide range of architectures the u-boot also supports a wide range of booting options. Below is a list of some of these booting options:

- From Flash memory (for example NOR or NAND)
- From a USB mass storage device
- From an MMC/SD memory card
- From a harddisk or CDROM
- Using Ethernet: TFTP, BOOTP, DHCP or NFS
- Using a serial connection

A booting option means a location from where the u-boot searches for the kernel image to load. If a MMC/SD card has been selected the u-boot will initialize the memory card controller and try to read the image(s) from that device.

It is not sure that the u-boot you are working with has support for all the booting options mentioned above. It depends on how the u-boot has been configured during compile time, see section 4.3 and it also depends of which peripherals the embedded system has support for. Even though the embedded system might have support for a peripheral the u-boot might not have been configured to support that peripheral. Most often this is because the code size of the u-boot must be kept to a minimum in order for it to fit into, for example, the internal flash memory of the microcontroller.

Section 4.5 contains more information about how to select a booting option during runtime.

4.3 Configuration Options

This section will explore a number of configuration options that exist for the u-boot in general and for the Embedded Artists patch in particular.

4.3.1 Make Target

The make target for a board is defined in the base make file located in the u-boot root (`u-boot-1.1.6/Makefile`). Open this file and search for ARM7TDMI and the make targets for the Embedded Artists boards will be found. Below is an example of how a make target looks like.

```
LPC2468OEM_Board_16bit_config: unconfig
```

```
    @./mkconfig $(@:_config=) arm arm720t LPC24xxOEM_Board
```

The first line contains the name of the target. In this example it is LPC2468OEM_Board_16bit_config. The second line will invoke a script named `mkconfig` with the parameters `arm`, `arm720t` and `LPC24xxOEM_Board`. Below is an explanation of these parameters.

- `arm` – The first parameter defines the architecture to use for the board. In this example it is an ARM target. The architecture is, for example, used when selecting which compiler to use when building the u-boot.
- `arm720t` – The second parameter defines which kind of CPU to use and will tell the build environment which CPU specific directory to build. In this example the `u-boot-1.1.6/cpu/arm720t/` directory will be used.
- `LPC24xxOEM_Board` – the third parameter defines which board specific directory to use when building the u-boot. In this example the `u-boot-1.1.6/board/LPC24xxOEM_Board/` directory will be used. This directory contains files that are common for all the Embedded Artists LPC24xx based OEM boards.

Besides these three parameters the `mkconfig` script will also generate a header file called `config.h`. The content of this header file will be an include statement for the board specific configuration file. The name of the board specific configuration file is the same as the make target, excluding the `_config` part. Below is an example of how the `config.h` file looks like for the above mentioned make target.

```
/* Automatically generated - do not edit */

#include <configs/LPC2468OEM_Board_16bit.h>
```

For the Embedded Artists patch the following board configurations are available:

LPC2468 OEM Board

- `LPC2468OEM_Board_16bit_config` – LPC2468 OEM Board with a **16 bit** data bus and an operating CPU frequency of **72 MHz**
- `LPC2468OEM_Board_32bit_config` – LPC2468 OEM Board with a **32 bit** data bus and an operating CPU frequency of **72 MHz**
- `LPC2468OEM_Board_16bit_48MHz_config` – LPC2468 OEM Board with a **16 bit** data bus and an operating CPU frequency of **48 MHz**
- `LPC2468OEM_Board_32bit_48MHz_config` – LPC2468 OEM Board with a **32 bit** data bus and an operating CPU frequency of **48 MHz**

LPC2478 OEM Board

- `LPC2478OEM_Board_16bit_config` – LPC2478 OEM Board with a **16 bit** data bus and an operating CPU frequency of **72 MHz**
- `LPC2478OEM_Board_32bit_config` – LPC2478 OEM Board with a **32 bit** data bus and an operating CPU frequency of **72 MHz**
- `LPC2478OEM_Board_32bit_48MHz_config` – LPC2478 OEM Board with a **32 bit** data bus and an CPU frequency of **48 MHz**

4.3.2 Configuration Files

The configuration files are located in the `u-boot-1.1.6/include/configs/` directory and there is one header file for each make target. Since the majority of the configurations are identical between the Embedded Artists make targets, one header file containing all the common settings have been created with the name `LPC24xxOEM_Board_common.h`. This header file is then included in the board specific header file, such as the `LPC2468OEM_Board_16bit.h` file. The only configurations that are defined in the board specific header file are:

- CPU – if it is LPC2468 or LPC2478
- Data bus address width – 16 or 32 bit data bus.
- The CPU clock frequency – the PLL settings for the CPU
- The monitor command prompt – prompt displayed in the terminal

4.3.3 Highlighted Configurations

Configurations are done using C preprocessor defines in the header file. For the u-boot there are two different classes of configurations.

- Configuration options – These are selectable by the user and have names beginning with `CONFIG_`.
- Configuration settings – These depend on the hardware and should not be modified unless you know what you are doing. These defines have names beginning with `CFG_`.

Below is a list of some of the configurations set in `LPC24xxOEM_Board_common.h` that might be of interest for those who would like to do modifications to the configuration.

- `CONFIG_BAUDRATE` – defines the baud rate, for example 115200, used by the u-boot serial console.
- `CONFIG_COMMANDS` – defines which commands that will be enabled in the u-boot, i.e., basically which booting options that will be available. If this define is set to `CFG_CMD_NAND | CFG_CMD_MMC | CFG_CMD_USB` it means that commands for NAND flash, memory card and USB interface are enabled.
- `CONFIG_BOOTDELAY` – defines the delay, in seconds, before automatically booting. Can be set to -1 to disable autoboot.
- `CONFIG_BOOTARGS` – the value of this define goes into the environment variable named `bootargs` and is passed to the `bootm` command, i.e., it can be used to send boot arguments to the Linux kernel. For the Embedded Artists configuration the default value of this define is "`root=/dev/ram initrd=0xa1800000,4000k console=ttyS0,115200N8`". This means that the root file system is located in RAM at address `0xa1800000` and the console is located on the `ttyS0` device running at a baud rate of 115200.
- `CONFIG_BOOTCOMMAND` – defines a command string that is automatically executed after the boot delay.
- `CONFIG_EXTRA_ENV_SETTINGS` – contains default environment variables that will be compiled into the u-boot image.
- `CFG_LOAD_ADDR` – defines the default load address for the kernel image.
- `CFG_ENV_ADDR` – Specifies the start address of where the environment variables are stored.

- `CFG_ENV_SIZE` – defines the storage size for the environment variables

More information about many of the configuration options can be found in a readme file located in the u-boot root directory; `u-boot-1.1.6/README`

If you would like to minimize the size of the u-boot image the `LPC24xxOEM_Board_common.h` file is the file to start working with. Disable those commands and booting options that aren't needed for your specific project.

4.4 Console / Environment

The u-boot has support for a command line interface usually accessed over a serial console port. Connecting a terminal application to the serial port will give you access to the command line interface. If a boot delay is enabled (which it is for the Embedded Artists configuration) the u-boot will not automatically load an image directly after reset of the target board. Instead it will delay for a certain number of seconds allowing a user to stop the boot process. When having a terminal application connected to the serial port associated with the target board the boot process is stopped by hitting a key before the delay has expired. The u-boot will then enter into command mode. The command mode allows you to manually type in boot commands or to update the environment variables that can later be used as boot options.

4.4.1 Commands

It is possible to find out which commands are available by using the `help` command.

```
# help
```

When issuing the help command a list of all the available commands will be presented. These commands are the ones that have been selected to be supported when configuring the u-boot. If more information is needed about a specific command type `help` followed by the name of the command. In the example below more information is requested about the `setenv` command.

```
# help setenv
```

Below is a list of some of the commands used to modify, list and execute variables in the u-boot environment.

- `printenv` – This command will print the environment variables to the console. The command `print` can also be used since it is an alias for `printenv`.
- `setenv` – This command is used to set the value of an environment variable. If the variable doesn't exist when calling `setenv` it will be created. The command `set` is an alias for `setenv`.
- `saveenv` – This command will save any changes done to the environment and must be called after `setenv` has been used in order for the changes to be saved persistently. The command `save` is an alias for `saveenv`.
- `run` – execute the commands found in an environment variable.

```
# setenv serverip 192.168.5.10

# setenv nand_boot nboot a1500000 0\;bootm a1500000

# saveenv
# run nand_boot
```

The above example shows how two environment variables are set and then saved to persistent storage. Please note that when a variable contains many commands they must be separated with a semicolon (';'). When using the `setenv` command the semicolon must be escaped using a backslash ('\') as can be seen in the example above.

Memory related commands

The u-boot contains several commands that inspect and/or modify memory positions. Since several of the booting options somehow copy images to and from different memory locations these commands are important to know about.

Several of the memory commands can work in different modes; 8-bit, 16-bit or 32-bit. These modes are specified by a qualifier that is added to the command, for example, .b (byte) for 8-bit, .w (word) for 16-bit and .l (long) for 32-bit. When used together with the copy command it looks like `cp.b`, `cp.w` or `cp.l`.

- `md` – The Memory Display command lists the content of a specified memory region.
- `mm` – The Memory Modify command is an interactive command to modify the memory. Once started it will ask for a value to write to the memory and then auto increment the location and ask for a new value.
- `mw` – The Memory Write command will fill a memory region with a fixed value.
- `cmp` – This command compares two memory regions.
- `cp` – The copy command copies data from one memory region to another, for example, from RAM to flash.
- `protect` – This command is used to enable or disable write protection for a flash memory region. When memory protection is enabled the memory cannot be modified with, for example, the `cp` command or the `erase` command.
- `erase` – This command is used to erase a flash memory region.
- `flinfo` – prints flash memory information in the console. Displays information about the flash banks in the system and their protection status.

4.4.2 Variables

The variables listed below need special attention since they are used by some of the boot commands.

- `ethaddr` – this variable defines the MAC (Ethernet) address used by the u-boot. For the Embedded Artists configuration the value set in this variable will then also be used by the Linux kernel.
- `ipaddr` – this variable defines the IP address used by the u-boot. Note that the Linux kernel will *not* use the value set in this variable for its IP address.
- `netmask` – this variable defines the network mask used by the u-boot.
- `serverip` – this variables defines the TFTP server IP address used when downloading images using TFTP.
- `bootargs` – this variable defines the boot arguments sent to the Linux kernel. It contains, for example, information about where to find the root file system and console.

- `bootcmd` – this variable contains the boot command(s) that will be run during auto booting.
- `bootdelay` – this variable defines the delay in seconds until an autoboot will take place. Autoboot can be cancelled by hitting any key during boot.
- `fileaddr` – this is a special kind of variable where the value is set by the `tftpboot` command. The value will be the address where the last file has been downloaded. It can then be used by other boot commands.
- `filesize` – this is a special kind of variable where the value is set by the `tftpboot` command. The value will be the size of the latest downloaded file. It can then be used by other boot commands.

4.4.3 Erase the Environment

For the Embedded Artists configuration the environment variables are located in flash at address `0x7c000`, see the `CFG_ENV_ADDR`. Sometimes it might be needed to reset any changes made to the variables and get back the default values. One such situation is when updating the u-boot itself. If the update is significant new environment variables might have been added and the default values of previous variables might have been changed. The environment can be erased by following the steps below.

1. Begin by restarting the board and aborting the autoboot by hitting any key in the terminal connected to the board.
2. Enter the commands given below.
3. Restart the board again and the environment will be erased and all variables set to their default values.

```
# protect off 7c000 7cfff
# erase 7c000 7cfff
```

4.5 Booting Options

4.5.1 Important Remarks

Important to notice for the Embedded Artists configuration is that the u-boot occupies the RAM starting at address `0xA1F80000`. Therefore this memory cannot be used during the boot process to store kernels and root file systems. Regardless of which storage device is used the Linux kernel will be placed at `0xA0008000` before it is started. Make sure that enough room is reserved for the Linux kernel when using the RAM to store Linux kernel images and root file systems. The memory `0xA0000000` to `0xA0007FFF` is used for kernel parameters and tagged values and cannot be used for other purposes.

4.5.2 Boot Arguments

Boot arguments can be set by the u-boot and sent to the Linux kernel during start-up of the kernel. The arguments usually define where to find the root file system and the console. It was briefly mentioned in section 4.3.3 and this section will give examples of a couple of different boot arguments to use with the Embedded Artists OEM Boards.

RAM based root file system

For a RAM based root file system the boot argument below can be used. It specifies that the root file system can be found on the device `/dev/ram` at address `0xa1800000`. The second part of the boot argument defines that the console is accessible on the `ttyS0` device with a baud rate of 115200, no parity and 8 bits data.

```
bootargs=root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
```

MTD block on NOR flash

In the Linux kernel there is a subsystem to be used for memory devices, especially Flash devices, called the Memory Technology Device (MTD) subsystem, see section 5.14 for how it is setup for the Embedded Artists LPC24xx OEM Boards. More information about MTD in general is found in ref [16].

For the Embedded Artists Linux kernel configuration the MTD partition 3 is dedicated for the NOR flash memory. In the example below you can see how the boot arguments are setup when using MTD partition 3.

```
bootargs=root=/dev/mtdblock3 console=ttyS0,115200N8
```

MTD block on NAND flash

For the Embedded Artists Linux kernel configuration the MTD partition 1 is dedicated to the root file system when it is stored in NAND flash memory. In the example below you can see how the boot arguments are setup when using MTD partition 1.

```
bootargs=root=/dev/mtdblock1 console=ttyS0,115200N8
```

4.5.3 Boot Images

In the examples below the command `bootm` will be used to load and execute the Linux kernel. This command is expecting to find an application image (in this case the Linux kernel) in a special format to load. The image will contain information about the load and execute address, the name and type of the image and a CRC32 checksum.

The images are created by using a tool called `mkimage` which is distributed with the u-boot source code. Section 3.4 contains more information about how the `mkimage` tool is used when building µClinux.

4.5.4 TFTP

During development of the Linux kernel it is very convenient to use the Trivial File Transfer Protocol (TFTP) to download a newly created kernel and file system image. There is no need to transfer those images onto any other media before loading them into the development board. This means that you get short development cycles. As long as you have a working network interface on the board and a TFTP server running on the development computer you can download images over TFTP. For information about how to setup a TFTP server in a Debian Etch distribution, see section 9.9

It is also convenient to use TFTP when updating other storage locations on the board such as flash storage, for example, NAND or NOR flash.

The command

```
# tftpboot <load address> <boot filename>
```

- `load address` – this is the address where the file is downloaded to, for example, `0xa1500000` to store it in external SDRAM.
- `boot filename` – the name of the file to download.

Dependencies

Before using the TFTP boot command you must make sure that the variables `ethaddr`, `ipaddr`, `netmask` and `serverip` have been correctly set, see section 4.4.2 and that a TFTP server is running with access to the files about to be downloaded.

Example usage

The steps below explain how the µClinux image and the root file system can be downloaded and started using TFTP.

1. Download the µClinux image `uLinux.bin` to address `0xa1500000`.

```
# tftpboot a1500000 uLinux.bin
```

2. Download the root file system `romfs.img` to address `0xa1800000`. Please note that the boot arguments must be setup so that the Linux kernel is looking for the root file system at address `0xa1800000`, see section 4.5.2

```
# tftpboot a1800000 romfs.img
```

3. Load and execute the µClinux image.

```
# bootm a1500000
```

In the Embedded Artists configuration all these three steps have been put into one environment variable named `tftp_boot`.

```
tftp_boot=tftpboot a1500000 uLinux.bin;tftpboot a1800000
romfs.img;bootm a1500000
```

A second example where the TFTP command can be used is to update the u-boot itself. The steps below show how this can be done.

1. Download the u-boot image, `u-boot.bin`, to address `0xa1000000`.

```
# tftpboot a1000000 u-boot.bin
```

2. Disable write protection in the flash area where the u-boot will be stored.

```
# protect off 0 2ffff
```

3. Erase the flash area where the u-boot will be stored.

```
# erase 0 2ffff
```

4. Now copy the downloaded u-boot image from RAM address `0xa1000000` to flash address `0`. Note how the variable `filesize` is used to define how many bytes to copy using the `cp.b` command. The `filesize` variable has been set by the `tftpboot` command when downloading the `u-boot.bin` file.

```
# cp.b a1000000 0 $(filesize)
```

5. Reset the board and the new u-boot will be started.

4.5.5 FAT File System

The u-boot supports loading images from FAT file systems. These file systems can reside on many different kind of media such as a memory card or USB mass storage device.

The command

```
# fatload <interface> <dev[:part]> <addr> <filename> [bytes]
```

- `interface` – this is the interface to use when accessing the FAT file system, for example., `usb` for a USB device or `mmc` for a memory card.
- `dev` – this specifies the device to use. An interface can have several devices and you must choose which device to use. If you only have one device the `dev` parameter is set to `0` indicating device number 0.
- `part` – this is an optional parameter specifying which partition to use on a specific device.
- `addr` – this is the address where the file will be loaded to.
- `filename` - the name of the file to load.
- `bytes` – this is an optional parameter defining how many bytes to load.

Example usage

These two examples show how to load the µClinux image, `uLinux.bin`, from the MMC or USB interface to address `0xa1500000`.

```
# fatload mmc 0 a1500000 uLinux.bin
# fatload usb 0 a1500000 uLinux.bin
```

For more information about how to use the `fatload` command with MMC and USB commands see section 4.5.6 and section 4.5.7

4.5.6 USB Mass Storage

Almost all computers today have a USB connection and most operating systems support USB and have drivers for USB mass storage devices. This makes it quite simple to use a USB memory stick to transfer boot images from the development computer to the development board.

The command

```
# usb (start | stop)
```

The `usb` command can do more than just start and stop a USB device, but basically that is all that is needed. Call “`usb start`” before loading the files using `fatload` and end by calling “`usb stop`” to disconnect the USB interface.

Example usage

In this example the following takes place:

1. USB interface is initialized and started

```
# usb start
```

2. The `uLinux.bin` image is loaded to address `0xa1500000`

```
# fatload usb 0 a1500000 uLinux.bin
```

3. The root file system, `romfs.img`, is loaded to address `0xa1800000`. Please note that the boot arguments must be setup in a way where the Linux kernel is looking for the root file system at address `0xa1800000`, see section 4.5.2

```
# fatload usb 0 a1800000 romfs.img
```

4. The USB interface is then stopped

```
# usb stop
```

5. Finally the µClinux image is loaded and started by using the `bootm` command.

```
# bootm a1500000
```

4.5.7 MMC/SD Card

Using a memory (MMC/SD) card is another convenient way to transfer boot images to an embedded system. The way to handle a memory card is similar to a USB memory stick.

The command

```
mmc
```

The command is really simple. All you have to do is call `mmc` and the MMC interface will be initialized and ready to be used. There is no command for stopping the interface.

Example usage

In this example the following takes place:

1. The MMC interface is initialized

```
# mmc
```

2. The `uLinux.bin` image is loaded to address `0xa1500000`.

```
# fatload mmc 0 a1500000 uLinux.bin
```

3. The root file system, `romfs.img`, is loaded to address `0xa1800000`. Please note that the boot arguments must be setup in a way where the Linux kernel is looking for the root file system at address `0xa1800000`, see section 4.5.2

```
# fatload mmc 0 a1800000 romfs.img
```

4. Finally the µClinux image is loaded and started by using the `bootm` command.

```
# bootm a1500000
```

4.5.8 NOR Flash

If the development board has a NOR flash it is very convenient to have the boot images stored in that storage location. There is no need to attach any additional devices to the board such as a memory card or a USB device. Therefore this is a suitable storage location for production ready boards.

The command

There isn't really any NOR specific boot command. Instead the `cp`, i.e., copy command will be used to copy the boot images between the flash memory and the RAM memory, see the examples below.

Example usage

The first example show how the NOR flash can be updated with boot images.

1. First, flash bank number 2 is erased which is the bank used for NOR flash on the Embedded Artists Boards.

```
# erase bank 2
```

2. TFTP is used to transfer the µClinux image to address `0xa0000000`. Please note that loading from USB memory stick or MMC/SD card could also have been used.

```
# tftpboot a0000000 uLinux.bin
```

3. The µClinux image is copied to NOR flash, address `0x80000000`. Please note the use of variables `fileaddr` and `filesize` which are automatically set by the `tftpboot` command.

```
# cp.b $(fileaddr) 80000000 $(filesize)
```

4. The root file system, in this example a compressed version named `cramfs.img`, is downloaded using TFTP to address `0xa0000000`.

```
# tftpboot a0000000 cramfs.img
```

5. The file system is then copied to NOR flash address `0x80200000`.

```
# cp.b $(fileaddr) 80200000 $(filesize)
```

The reason for using a compressed version of the root file system is that NOR flash memory usually is quite small in size. However, if the uncompressed version of the file system can fit in the NOR flash memory there is no need to use a compressed version.

Please note that in a cramfs file system each file is compressed individually which means that the kernel doesn't need to uncompress the entire file system before it is used. Instead only those files that are accessed will be uncompressed.

The second example shows how to boot from NOR flash. In this case the image will be loaded directly from NOR flash, address `0x80000000`, using the `bootm` command. Please note that the file system hasn't been loaded to memory before calling the `bootm` command. This means that the boot argument must be setup in a way where the root file system is mounted in NOR flash instead. This can be achieved by using Memory Technology Device

(MTD) support in Linux, see 4.5.2 how to setup the boot argument variable in a way where MTD is used.

```
# bootm 80000000
```

It is also possible first to copy the boot images to RAM similar to the other boot options described above and then boot. A combination of these two examples could also be used where the root file system is first copied to RAM, address `0xa1800000`, and then the µClinux image is directly loaded from NOR flash using `bootm 80000000`. The example below show how all this is put into one U-boot environment variable.

```
nor_boot=cp.b 80000000 a1500000 200000;cp.b 80200000 a1800000
200000;bootm a1500000
```

4.5.9 NAND Flash

If the development board has a NAND flash it is very convenient to have the boot images stored in that storage location. There is no need to attach any additional devices to the board such as a memory card or a USB device. This is therefore a suitable storage location for production ready boards.

The commands

```
nand <argument>

nboot <toAddress> <fromAddress>
```

Two commands will be used with NAND boot. The first command named `nand` can be used to read and write from and to the NAND flash. The second command named `nboot` can be used to directly load the image from NAND flash. It is similar to a `nand read` request.

The arguments to the `nand` command are everything from read and write requests to information requests about the NAND device. Use the `help nand` command to get a complete list. Below is a description of the arguments later used in the example usage section.

- `nand erase [clean] [off size]` – The NAND memory will be erased. If no arguments are given to the erase request the entire memory will be erased.
- `nand write fromAddr toAddr size` – read `size` bytes of data stored in the `fromAddr` address and write it to the `toAddr` address.

The `nboot` command takes an address to read from and a device number, see the examples below to get a better understanding of this command.

Example usage

The first example show how the NAND memory is updated with µClinux and with root file system images.

1. The complete NAND memory is erased.

```
# nand erase
```

2. The µClinux image, `uLinux.bin`, is then downloaded to address `0xa0000000` using TFTP. Please note that loading from USB memory stick or MMC/SD card could also have been used.

```
# tftpboot a0000000 uLinux.bin
```

3. The newly downloaded image is written to address `0` with a size of `0x00300000` bytes.

```
# nand write $(fileaddr) 0 0x00300000
```

4. The root file system in a JFFS2 format (see more about JFFS2 below) is downloaded to address `0xa0000000` using TFTP.

```
# tftpboot a0000000 jffs2.img
```

5. The newly downloaded image is written to address `0x00300000`, i.e., right after the µClinux image.

```
# nand write $(fileaddr) 0x00300000 $(filesize)
```

The second example shows how to boot from NAND memory. When using the `nboot` command it is simple to load the image to RAM and then boot using the `bootm` command.

```
# nboot a1500000 0
# bootm a1500000
```

Please note that the file system hasn't been loaded to RAM memory before calling the `bootm` command in the example. This means that the boot argument must be setup in a way where the root file system is mounted in NAND flash instead. This can be achieved by using Memory Technology Device (MTD) support in Linux, see 4.5.2 for information of how to setup the boot argument variable in a way where MTD is used.

Journalling Flash File System version 2 (JFFS2)

In the above example a JFFS2 root file system (`jffs2.img`) is used. This file system format is suitable for flash memory devices in general and for NAND devices in particular. By using a JFFS2 file system it is possible to mount it as a read and writable file system in Linux and thereby persistently store any updates done to the file system. The opposite is a RAM based file system where any changes are lost when the power is turned off. For more information about how to use JFFS2 in Linux on an embedded device see section 5.14.4

5 Device Drivers

5.1 Introduction

Hardware, machines, or components attached to a computer system are often referred to as devices or peripherals. These components are used to extend the functionality of a computer system, i.e., to either input data to the system or to receive output from the system. Typical examples of such devices are displays, keyboards, mice, USB devices, network devices, storage devices, and so on.

In order for a device to function correctly with the computer system a piece of software is usually needed to initialize and control the device. This piece of software will become hardware dependent since it needs to interact directly with the hardware, such as with internal registers of the device. The software will therefore not be especially portable and can't, in the general case, be used with hardware from different manufactures, for example, Ethernet adapters from different manufactures would require different initialization and control code.

A good software design rule is to isolate those parts of the software that are dependent of the hardware to a separate layer and also into separate modules. By doing this only small parts of the software application must be replaced when changing parts of the hardware. The part of the software that isolates the dependency towards a device is known as the device driver.

5.2 Linux Devices and Drivers

The device driver plays an important role in the Linux kernel. As a matter of fact the largest part, measured in file size (bytes), of the kernel sources is the device drivers. In the 2.6.21 version of the kernel almost 50% of the source code consists of device driver code.

The device driver model in Linux offers a well-defined and uniform interface towards the devices making it easier to implement applications using the devices. In general many of the devices are exposed as a file in the file system for easy access by the application.

In Linux the devices and their associated drivers have been classified into three types.

- **Character Device** – This is a device that can be accessed as a stream of bytes, i.e., it is often a sequential access of the data on the device. The typical system calls implemented by a character driver are the `open`, `close`, `read` and `write` calls. Examples of character devices are serial ports (exposed as `/dev/ttyS0` in the file system), printers (`/dev/lp0`), and the real time clock (`/dev/rtc`).
- **Block Device** – This is a device where the data exchange is handled in one or more blocks of data as opposed to single characters. These devices are used for hosting file systems and although the device itself is exposed as a file an application does not access it directly. Instead the application access the device through a file system interface. Examples of block devices are a floppy disk (`/dev/fd0`), a hard drive (`/dev/hda`), and a memory card (`/dev/mmca1`).
- **Network Device** – This is a device that exchanges data with other hosts and the responsibility of the driver is to send and receive data packets to and from the network device. This device isn't accessed directly by applications. Instead the driver typically interacts with a network protocol stack which has an interface used by applications (for example the socket interface). Although the network device isn't mapped to the file system a unique name is given to the device, for example, `eth0` to the first Ethernet interface.

For more detailed information about device drivers there is a good book on the subject called "Linux Device Drivers", see ref [17].

The remaining part of this chapter will describe the drivers that have been developed for the Embedded Artists distribution. Each section contains a short description of the device which is followed by an explanation of how to enable support for the device in the Linux kernel. It is also explained where the driver code is located in the kernel source tree and most importantly how to use the device.

5.3 Porting

To design and implement drivers for Linux is out-of-scope for this book. This section will just shortly describe the steps you need to take in order to develop a driver.

First of all a more appropriate term than develop would be to port a driver. It is really rare that a driver implementer must implement a complete driver from scratch. A driver implementer may not even have to know if the driver he/she is implementing is a character driver, block driver or network driver. The reason for this is that the Linux kernel already contains a lot of the generic, hardware independent part, of the device driver. As for software in general a device driver should be divided into different layers ranging from the hardware dependent part up to the application interface. This is exactly what the kernel developers have done. Most often the driver writer only has to concentrate on the hardware dependent part of the driver and on the interface defined by the generic part of the driver.

Below are a number of steps a driver implementer usually must take.

1. Identify in which part of the kernel source tree the driver you are about to develop belongs. For example, if a display driver is developed it is located somewhere in the `/drivers/video/` directory or if a USB host driver is developed it is located in the `/drivers/usb/host/` directory.
2. Study the data sheet of the device to find out how the interface looks like. It might be that the device is using a more or less standardize interface, such as the OHCI (Open Host Controller Interface) interface for USB hosts in which case porting might be easy, that is, most of the code may already exist in the kernel.
3. Go through the drivers already present in the Linux kernel sources. It is very likely that a driver identical or at least similar to yours have already been developed.
4. Copy the driver that is similar to yours and modify those parts that differ between the devices. You will find that it is very common that one driver is based on another driver.
5. If there isn’t a similar driver at least have a look at the existing drivers for your category of device to find out how to use the interface defined by the generic part of the driver.

5.4 Frame Buffer

A frame buffer is a video output device that drives a video display from a memory buffer. The abstraction of accessing the video hardware is done in a way that it seems as though you are writing directly to the video hardware by writing to a piece of memory. Since the frame buffer device work identically between different Linux ports, i.e., different architectures, the implementation of applications that want to output graphics are easier and more portable.

5.4.1 Configuration

Frame buffer support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → Graphics support → Support for frame buffer devices

After enabling frame buffer support the hardware driver must also be selected. For the Embedded Artist LPC2478 board there is the following configuration choice:

Device Drivers → Graphics support → LPC2478 LCD controller support

There is also a choice to select the actual hardware, i.e., the LCD. The previous choice was only for the LCD controller that is to be used, but different LCDs could be connected to the board as well.

Device Drivers → Graphics support → Select LCD hardware

5.4.2 Driver Code

The source code for the hardware specific parts of the frame buffer code is located in `uClinux-dist/linux-2.6.x/drivers/video/lpc2478/`. The file `lcdctrl.c` is where the driver is registered with the platform bus and initialized when the probe function is called by the kernel. Important to notice is that the driver is using the SPI bus to communicate with the LCD hardware during initialization in order to correctly initialize the display. To achieve this the LCD specific part of the code is registering itself as a SPI driver, see the `tft_g240320.c` file and the `lcd_hw_init` function.

5.4.3 Usage

The frame buffer is exposed to user space applications as a device file called `/dev/fb0`. It can be accessed directly by an application to output graphics onto the display, for example, by memory mapping the file. An example of an application using the frame buffer can be found in `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/applications/calibrate.c`. This application is used to calibrate the touch screen and will use the frame buffer to output the coordinates where a user should touch using, for example, a stylus pen in order to do the calibration. The application will open the file, get some information about the video hardware using the `ioctl` function and then memory map (using `mmap`) the frame buffer file before outputting graphics to the display.

If several applications need to access the display it is better to use a Window System instead of direct access to the frame buffer. The µClinux distribution provided by Embedded Artists has enabled the Nano-X Window System (previously called MicroWindows), see ref [18]. This window system contains two APIs that can be used by applications to access a display. One of the APIs is a Win32 like API and the other is an Xlib-like API.

Section 6.8 shows an example of how to use the Nano-X interfaces.

5.5 Touch Screen

A touch screen is a display with a surface that can detect light pressure from, for example, a finger, stylus pen or other passive object. It is used as an input method to interact and control a device with a display and replaces a keyboard, mouse or joystick. It is commonly used on modern mobile phones, PDAs and computer terminals.

The touch screen used on the Embedded Artists Boards is controlled by the Texas Instruments TSC2046 touch screen controller. This controller is attached to the SPI interface on the OEM board.

5.5.1 Configuration

Touch screen support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → Input device support → Touchscreen interface

The specific touch screen driver to use must also be selected and enabled. In this case the TSC2046 touch screen controller is compatible with the ADS 7846 touch screen controller.

Device Drivers → Input device support → Touchscreens → ADS 7846/7843 based touchscreens

The horizontal and vertical screen resolution must also be set when enabling the touch screen interface.

Device Drivers → *Input device support* → *Horizontal screen resolution*

Device Drivers → *Input device support* → *Vertical screen resolution*

5.5.2 Driver Code

The TSC2046 touch screen controller is compatible with the ADS7846 touch screen controller. A driver for the ADS7846 is integrated in the kernel source tree and that driver has been able to be used without modification for the TSC2046 controller. The source code is available in the `uClinux-dist/linux-2.6.x/drivers/input/touchscreen/ads7846.c` file.

5.5.3 Usage

The touch screen is exposed to applications as two device files called `/dev/tsraw0` and `/dev/ts0`. The `tsraw0` device delivers unmodified raw values from the touch screen while the `ts0` device delivers values that have been converted by a calibration algorithm.

The calibrate application found in the `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/applications/calibrate.c` file illustrates how to use the touch screen device files and one way of how to calibrate the touch screen.

Below is a simple example of how to open the calibrated touch screen device, read values from it and output the values as points on a display.

```
/* Touch screen data */
struct ts_event {
    short pressure;
    short x;
    short y;
    short millisecs;
};

…

…

    ts_fd = open("/dev/ts0", O_RDONLY);

    /* failed to open device */
    if (ts_fd < 0) {
          goto err_close_fb;
    }

    while(1) {
          len = read(ts_fd, &event, sizeof(struct ts_event));


          /* failed to read from the device */
          if (len <= 0) {
                goto err_close_ts;
          }

          lcd_point(frame_map, event.x, event.y, COLOR_BLACK);

    }

…
```

5.6 Ethernet

Ethernet is the most common communication technique used to create local area networks (LANs) in, for example, offices today. It is a frame based protocol using 48-bit destination and source addresses, a 16 bit type field indicating the type of the data, an up to 1500 byte long data field, and finally a 32 bit checksum.

The LPC24xx processors come with an embedded Ethernet block containing a full featured 10 Mbps or 100 Mbps Ethernet Media Access Controller. The Ethernet block must then interface with an off-chip Ethernet PHY using either the MII (Media Independent Interface) or the RMII (reduced MII). In the Embedded Artists boards the RMII mode is used and the Ethernet PHY is either a National DP83848 or a Micrel KSZ8001L.

5.6.1 Configuration

Networking support is enabled in the kernel at the following place in the configuration tree:

Networking → Networking support

Several networking options have also been enabled in the Embedded Artists configuration. These options can be found in (and are not further described here):

Networking → Networking options

Besides networking support in the kernel there must also be support for networking drivers. This support is enabled at:

Device Drivers → Network device support → Network device support

The support for the link layer Ethernet must also be enabled:

Device Drivers → Network device support → Ethernet (10 or 100MBit)

Finally the Ethernet hardware support must be chosen.

Device Drivers → Network device support → NXP LPC2XXX Ethernet support

5.6.2 Driver Code

The source code for the Ethernet driver is located in `uClinux-dist/linux-2.6.x/drivers/net/arm/lpc22xx_eth.c` file. When the kernel is probing the driver, i.e., checking if it is available to be assigned to a device, the `lpc2xxx_eth_probe` function will be called. This function will start to initialize Ethernet registers and detect which PHY that is attached to the Ethernet block. After this is done the initialization will continue by setting up interrupt handler, MAC address, detecting link speed (if cable is connected), and so on.

5.6.3 Usage

The Ethernet driver isn't used directly from user space applications. The driver is used internally by the networking subsystem in the kernel and the applications use a high-level interface such as a socket API to get networking capabilities.

To enable the network interface when Linux is up and running the `ifconfig` command can be used. The following example shows how the Ethernet interface is assigned to the IP address 192.168.5.10 and then activated (the `up` flag activates).

```
# ifconfig eth0 192.168.5.10 up
```

Chapter 10.3 shows more examples of how to use the network interface in Linux. There is, for example, a section describing how to setup DHCP for dynamically allocated IP addresses.

5.7 MMC / SD

The LPC24xx family of processors comes with a Secure Digital and Multimedia Card Interface (MCI). This interface conforms to the Multimedia Card Specification v2.11 and the Secure Digital Memory Card Physical Layer Specification v0.96.

The Embedded Artists µClinux distribution contains an MMC/SD card driver, but it isn't integrated in the Linux kernel source tree and is not using the Linux MMC/SD card subsystem.

One important limitation to know about with the current implementation is that it doesn't support insertion and removal of the memory card at run time. The card must be inserted during boot. The card can be removed during runtime, but it is important to do this after the memory card has been unmounted in order not to lose any data. If the card is removed during runtime only the exact same card can be inserted again. The driver doesn't detect insertion/removal of the card and hence doesn't reinitialize the card after insertion.

5.7.1 Configuration

The supported file system format is FAT and therefore support for this file system must be enabled in the kernel:

File Systems → DOS/FAT/NT Filesystems → MSDOS fs support

File Systems → DOS/FAT/NT Filesystems → VFAT (Windows-95) fs support

If the driver were to use the MMC/SD card subsystem in the kernel this would have to be enabled:

Device Drivers → MMC/SD Card support →MMC support

5.7.2 Driver code

Since the driver is located outside of the kernel source tree it is placed at the following location: `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/drivers/2.6.x/lpc2468mmc/`. The module initialization function, i.e., the starting point of the driver, is located in the `ea-mmc_26.c` file.

If the driver were to be integrated in the Linux kernel source tree the correct location to put it would be in the `uClinux-dist/linux-2.6.x/drivers/mmc/`directory.

5.7.3 Usage

Before the driver can be used the driver module must be loaded into the Linux kernel by using the following command. Make sure the memory card is inserted before this command is executed.

```
# insmod /drivers/lpc2468mmc.ko
```

The `rc` script provided by Embedded Artists will by default load this module if it is available. The part of the script that loads the module is shown below. It will first check if the module is available and then run the `insmod` command to load the module into the kernel.

```
if [ -f /drivers/lpc2468mmc.ko ]; then
  insmod /drivers/lpc2468mmc.ko
fi
```

After the driver module has been successfully loaded, it is time to mount the memory and making it accessible. This is done by using the mount command.

```
# mount -t vfat /dev/mmca1 /mnt/mmc
```

The above command will mount a file system of type vfat onto the directory `/mnt/mmc/`. The block device (remember that almost all devices and hence drivers are exposed as device files) that will be mounted is the `/dev/mmca1` device.

5.8 USB Host

The Universal Serial Bus (USB) is an interface that has become widely used to connect a computer to different kind of devices (mouse, keyboard, digital cameras, and so on). USB is a host-centric bus which means that a USB host must initiate all data transfers, both inbound and outbound. This section will discuss the USB host part of the USB interface.

The LPC24xx comes with an embedded Open Host Controller Interface (OHCI) compliant host controller which makes it rather straight forward to implement a driver in Linux. The Linux kernel implements a USB subsystem with support for OHCI controllers. This means that all that is needed for the driver implementer is to setup access to the hardware, that is, get access to the registers and interrupt handler.

5.8.1 Configuration

USB host and OHCI support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → USB support → Support for Host-side USB

Device Drivers → USB support → OHCI HCD support

When a USB device is attached to the USB bus a USB device driver must be loaded in order to properly interact with the device. In the Embedded Artists configuration two standard classes of USB device drivers have been enabled: the USB Mass Storage and USB HID (Human Interface Device) classes. USB Mass Storage is needed to support USB memory sticks. The USB HID class is needed to support devices such as a mouse or a keyboard. These classes of drivers are enabled at the following place in the configuration tree:

Device Drivers → USB support → USB Mass Storage support

Device Drivers → USB support → USB Human Interface Device (full HID) support

The USB Mass Storage functionality needs SCSI support to be enabled.

Device Drivers → SCSI device support → SCSI device support

Device Drivers → SCSI device support → SCSI disk support

5.8.2 Driver Code

The source code for the USB Host driver is located in the `uClinux-dist/linux-2.6.x/drivers/usb/host/ohci-lpc24xx.c` file. This file is handled a bit differently than the other driver files. It doesn't contain any module initialization code run during start-up, but is instead directly included (by preprocessor) in the `ohci.hcd.c` file. At the same place as the inclusion, a constant with name `PLATFORM_DRIVER` has the value `ohci_hcd_lpc24xx_driver` which is the name of the platform driver structure which contains, for example, the probe callback function.

During the initialization phase of the ohci hcd module the platform driver structure defined in the `PLATFORM_DRIVER` constant will be registered.

5.8.3 Usage

The USB subsystem and particularly the host part aren't directly exposed to the user domain. Instead USB devices attached to the USB host will be exposed when their device drivers have been loaded.

If a USB memory stick is attached to the USB bus a block device with name `/dev/sda1` becomes available. In order to use this device it must be mounted.

```
# mount -t vfat /dev/sda1 /mnt/usbmem
```

In the above example the block device `/dev/sda1` will be mounted as a vfat file system onto the `/mnt/usbmem` directory.

Below is an example of the output in the console when a USB memory stick is attached to the Base Board. You can see that the SCSI emulation is enabled and that a SCSI device is associated with the memory stick.

```
usb 1-2: new full speed USB device using lpc24xx-ohci and address
2
usb 1-2: configuration #1 chosen from 1 choice
scsi0 : SCSI emulation for USB Mass Storage devices
scsi 0:0:0:0: Direct-Access     SanDisk  U3 Cruzer Micro  2.18 PQ:
0 ANSI: 2
SCSI device sda: 8015505 512-byte hdwr sectors (4104 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
SCSI device sda: 8015505 512-byte hdwr sectors (4104 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
 sda: sda1
sd 0:0:0:0: Attached scsi removable disk sda
```

5.9 USB Device

The Universal Serial Bus (USB) is an interface that has become widely used to connect a computer to different kind of devices (mouse, keyboard, digital cameras, and so on). USB is a host-centric bus. This means that a USB host must initiate all data transfers, both inbound and outbound. This section will discuss the USB device part of the USB interface.

As mentioned in the USB host section, see 5.8 , a USB device driver is loaded on the host when a USB device is attached to the host. In order to minimize confusion the driver used on the actual USB device is therefore called a USB gadget driver instead of USB device driver. The driver interfacing the peripheral on the processor is called a USB device controller driver.

The LPC24xx comes with an embedded USB device controller which is fully compliant with the USB 2.0 specification.

5.9.1 Configuration

USB device (gadget) support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → USB support → USB Gadget Support → Support for USB Gadgets

Support for the actual gadget drivers must also be enabled. In the Embedded Artists configuration a gadget driver for USB Mass Storage has been enabled.

Device Drivers → USB support → USB Gadget Support → USB Gadget Drivers

Device Drivers → USB support → USB Gadget Support → File-backed Storage Gadget

5.9.2 Driver Code

The source code for the USB device controller driver is located in the `uClinux-dist/linux-2.6.x/drivers/usb/gadget/lpc24xx_udc.c` file.

5.9.3 Usage

The USB Device driver is compiled as a separate driver module that isn't loaded automatically by the kernel. Instead the module is copied to a special directory in the root file system where it can be dynamically loaded at runtime into the kernel.

```
# cd /lib/modules/2.6.21-uc0/kernel/drivers/usb/gadget
# insmod lpc24xx_udc.ko
```

When the USB device driver has been loaded into the kernel it is time to load the gadget driver for the mass storage device. At the same time as loading this driver the file used as a backing storage device, i.e., the file system for the mass storage device is also specified. In this example the block device that represents the MMC/SD card is chosen.

```
# insmod g_file_storage.ko file=/dev/mmca1
```

When all this has been done it is just to connect a USB cable between the USB device connector on the Base Board and your computer and the board should show up as a mass storage device on your computer.

5.10 UART

The acronym UART stands for Universal Asynchronous Receiver/Transmitter and is usually used for serial communication over a serial port. It is today common for microcontrollers to include one or more embedded UARTs to be used for log/console output or communication between devices.

The LPC24xx have four UARTs where one of them (UART1) contains a modem interface. All the UARTs have register locations that confirm to the '550 industry standard, for example, the 16550 UART.

5.10.1 Configuration

The Linux kernel contains support for standard serial ports such as the 8250/16550 standard. This functionality is enabled at the following place in the configuration tree:

Device Drivers → Character devices → Serial drivers → 8250/16550 and compatible serial support

In the Embedded Artists configuration, support for the Console on the serial port has also been enabled. This functionality is available at:

Device Drivers → Character devices → Serial drivers → Console on 8250/16550 and compatible serial port

5.10.2 Driver Code

A driver for 8250/16550 compatible serial ports are integrated in the kernel source tree. That driver is possible to use without modification for the LPC24xx. The source code is available in the `uClinux-dist/linux-2.6.x/drivers/serial/8250.c` file.

5.10.3 Usage

In the boot arguments described in section 4.5.2 the console is setup for usage on UART0 which is represented by the device file ttyS0. The boot argument is setup as `console=ttyS0,115200N8`, which means that a baud rate of 115200 will be used with no parity and 8 bit data.

5.11 I^2C

Inter-Integrated Circuit (I^2C) is a two-wire serial bus protocol used by low-speed devices. The nodes on the bus have either a master or a slave role where the master initiates the data transfer. It is the master that selects which slave to communicate with. The I^2C bus is a multi-master bus which means that several master nodes can exist on the bus.

The LPC24xx has three I^2C interfaces that can be configured as Master, Slave or Master/slave. On the Embedded Artists boards the I2C0 interface is used and two devices are attached to this interface; a PCA9532 16 bit I/O Expander and a 24xx256 E2PROM.

In the Linux I^2C subsystem the drivers have been divided into these categories:

Category	Description
Bus driver	This part of the driver is controlling the I2C bus and is usually divided into two parts: an algorithm driver and an adapter driver. An algorithm driver contains code that can be used for a wide class of I2C adapters while an adapter driver depends on one algorithm driver.
Client driver	The client driver (in the kernel source tree it is located in a subdirectory named chips) contains the code that accesses a specific I2C device, such as the PCA9532 16 bit I/O Expander.

5.11.1 Configuration

I^2C support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → I2C support → I2C support

Device Drivers → I2C support → I2C device interface

The actual hardware interface for the LPC devices is enabled at:

Device Drivers → I2C support → I2C Hardware Bus support → NXP LPC2XXX I2C support

Two client drivers, for the I^2C devices attached to the I^2C interface on the Embedded Artists boards have been implemented. Support for these are enabled at the following location in the configuration tree:

Device Drivers → I2C support → Miscellaneous I2C Chip support → Microchip 24XX256 EEPROM driver

Device Drivers → I2C support → Miscellaneous I2C Chip support → Philips PCA9532 16-bit I/O expander

5.11.2 Driver Code

The source code for the bus driver is located in the `uClinux-dist/linux-2.6.x/drivers/i2c/busses/i2c-lpc2xxx.c` file. This file contains both the algorithm and adapter code. Besides the bus driver two client drivers were also developed. The source code for these drivers is available here:

```
uClinux-dist/linux-2.6.x/drivers/i2c/chips/24xx256.c
uClinux-dist/linux-2.6.x/drivers/i2c/chips/pca9532.c
```

Section 3.3.1 describes how to setup resources and do board specific initialization for devices. The example that was given was for the I^{2}C device and in this section we will continue to describe how the resources are used in the driver code.

Below is an excerpt from the I^{2}C driver code. The init function is shown and it register a driver structure with the platform bus. Compare this with how a device structure was registered with the platform bus in section 3.3.1

```
static struct platform_driver lpc2xxx_i2c_driver = {
    .probe              = lpc2xxx_i2c_probe,
    .remove             = lpc2xxx_i2c_remove,
    .driver      = {
          .owner       = THIS_MODULE,
          .name = DRIVER,
    },
};

static int __init
lpc2xxx_i2c_init(void)
{
    return platform_driver_register(&lpc2xxx_i2c_driver);
}
```

When a device is associated with its driver the probe function in the driver code is invoked and the device structure is provided as an argument. In the function you can see how the platform data is retrieved from the device structure. The platform data contained information about I^{2}C frequency, peripheral clock, timeout and retry value. These values are copied to a locally allocated structure to be used later in the driver.

The `lpc2xxx_i2c_map_regs` function is then called and this function (look further down in the excerpt) will retrieve the registered memory resource, i.e., the base address for the I^{2}C registers and copy it to a local structure. The base address will later be used when the I^{2}C registers must be accessed.

The second resource that was registered together with the device was an IRQ resource. This resource is retrieved using the `platform_get_irq` function and then used to setup and register an interrupt routine using the `request_irq` function.

```
static int __devinit
lpc2xxx_i2c_probe(struct platform_device *pd)
{
    struct lpc2xxx_i2c_data  *drv_data;
    struct lpc2xxx_i2c_pdata *pdata = pd->dev.platform_data;

    int   rc;



    if ((pd->id != 0) || !pdata) {
          return -ENODEV;
    }
    drv_data = kzalloc(sizeof(struct lpc2xxx_i2c_data),
GFP_KERNEL);
    if (!drv_data)
          return -ENOMEM;

    if (lpc2xxx_i2c_map_regs(pd, drv_data)) {
          rc = -ENODEV;
          goto exit_kfree;
    }

…

    drv_data->freq  = pdata->freq;
    drv_data->fpclk = pdata->fpclk;

    drv_data->irq = platform_get_irq(pd, 0);

    if (drv_data->irq < 0) {
          rc = -ENXIO;
          goto exit_unmap_regs;
    }

    drv_data->regs = (struct lpc2xxx_i2c_regs *)drv_data-
>reg_base;
    drv_data->adapter.dev.parent = &pd->dev;
    drv_data->adapter.id = I2C_HW_A_LPC2XXX;
    drv_data->adapter.algo = &lpc2xxx_i2c_algo;
    drv_data->adapter.owner = THIS_MODULE;
    drv_data->adapter.class = I2C_CLASS_HWMON;
    drv_data->adapter.timeout = pdata->timeout;
    drv_data->adapter.retries = pdata->retries;

    platform_set_drvdata(pd, drv_data);
    i2c_set_adapdata(&drv_data->adapter, drv_data);

    lpc2xxx_i2c_hw_init(drv_data);

    if (request_irq(drv_data->irq, lpc2xxx_i2c_intr, 0,
                DRIVER, drv_data)) {
          dev_err(&drv_data->adapter.dev,
                "lpc2xxx: Can't register intr handler irq: %d\n",
                drv_data->irq);
          rc = -EINVAL;
          goto exit_unmap_regs;
    } else if ((rc = i2c_add_adapter(&drv_data->adapter)) != 0) {
          dev_err(&drv_data->adapter.dev,
                "lpc2xxx: Can't add i2c adapter, rc: %d\n", -rc);
          goto exit_free_irq;
    }

    return 0;

    exit_free_irq:
          free_irq(drv_data->irq, drv_data);
    exit_unmap_regs:
          lpc2xxx_i2c_unmap_regs(drv_data);
    exit_kfree:
          kfree(drv_data);
    return rc;
}



static int __devinit
lpc2xxx_i2c_map_regs(struct platform_device *pd,
    struct lpc2xxx_i2c_data *drv_data)
{
    struct resource   *r;

    if ((r = platform_get_resource(pd, IORESOURCE_MEM, 0)) &&
          request_mem_region(r->start, (r->end - r->start + 1),
                drv_data->adapter.name)) {
          drv_data->reg_base_size = r->end - r->start + 1;
          drv_data->reg_base = ioremap(r->start,
                drv_data->reg_base_size);
          drv_data->reg_base_p = r->start;

    } else
          return -ENOMEM;

    return 0;
}
```

5.11.3 Usage

The I^{2}C bus is generally not directly exposed to the user domain, but instead the clients are exposed and accessible from user space applications. In the Embedded Artists configuration the client devices have been exposed as files in the sysfs file system.

PCA9532 16 bit I/O Expander

The PCA9532 has a number of files exposed at the following location in the file system: `/sys/bus/i2c/devices/0-0060/`. The files in the list below are accessible and each file represent a register in the PCA9532.

File	Description
`input0`	This file reflects the state of the device pins (inputs 0 to 7). Writing to this file will have no effect.
`input1`	This file reflects the state of the device pins (inputs 8 to 15). Writing to this file will have no effect.
`ls0`	LED select 0 controls LED (output pin) 0 – 3.
`ls1`	LED select 1 controls LED (output pin) 4 – 7.
`ls2`	LED select 2 controls LED (output pin) 8 – 11.
`ls3`	LED select 3 controls LED (output pin) 12 – 15.
`psc0`	The PSC0 register is used to program the period of the PWM0 output.
`psc1`	The PSC1 register is used to program the period of the PWM1 output.
`pwm0`	The PWM0 register determines the duty cycle of BLINK0. The outputs are LOW (LED on) when the count is less than the value in PWM0 and HIGH (LED off) when it is greater. If the value is set to 0 the output is always HIGH.

`pwm1`	The PWM1 register determines the duty cycle of BLINK1. The outputs are LOW (LED on) when the count is less than the value in PWM1 and HIGH (LED off) when it is greater. If the value is set to 0 the output is always HIGH.

For more details about the PCA9532 registers look at the data sheet, see ref [19]. Below are some examples of how to access the files (it is assumed that the current working directory is the `/sys/bus/i2c/devices/0-0060/` directory).

Turn on LED1 on the QVGA Base board:

```
# echo 1 > ls2
```

Turn off LED1 and turn on LED2 on the QVGA Base board:

```
# echo 4 > ls2
```

Check the state of the device pins (connected to the LEDs):

```
# cat input1

253
```

Please note the value 253 which is the same as the binary value 11111101, i.e., bit 1 has the value 0 all others have the value 1. A LED is turned on when an output is LOW so the value 253 means that device pin 9 (note that input1 is used) is low and since this pin is connected to LED2 this LED is lit. The schematics for the QVGA Base board illustrate how the PCA9532 is connected.

24xx256 E^{2}PROM

The 24xx256 has one important file exposed at the following location in the file system: `/sys/bus/i2c/devices/0-0050/`. The name of the file is `data0` and represents the content of the E^{2}PROM memory. Writing to this file means writing to the E^{2}PROM memory and reading from this file means reading from the E^{2}PROM memory.

5.12 SPI

SPI stands for the Serial Peripheral Interface Bus and is a low-level synchronous 4-wire serial bus. A master/slave communication mode is used where the master initiates the data frame and selects a slave using chip select. SPI offers full duplex communication with data transfers up to tens of Mbit/sec. SPI is often used by microcontrollers to interface with sensors, controllers, flash memory, MMC/SD cards and real-time clocks.

On the Embedded Artists QVGA Base Board the Touch controller and the LCD controller are attached to the SPI bus.

5.12.1 Configuration

SPI support is enabled in the kernel at the following place in the configuration tree:

Device Drivers → SPI support → SPI support

An SPI Master controller driver must also be selected and for the Embedded Artists Board a Bitbanging SPI master has been selected

Device Drivers → SPI support → Bitbanging SPI master

Finally the hardware interface is selected.

Device Drivers → SPI support → NXP LPC2XXX series SPI

5.12.2 Driver Code

The source code for the hardware specific parts of the SPI driver is located in `uClinux-dist/linux-2.6.x/drivers/spi/spi_lpc2xxx.c` and the Bit banging code is located in `uClinux-dist/linux-2.6.x/drivers/spi/spi_bitbang.c`.

5.12.3 Usage

The SPI driver isn't used directly from user space, but instead used by other drivers such as the touch screen driver; see section 5.5 and the frame buffer driver see section 5.4

The example code below is taken from the touch screen driver. Only certain parts from the driver are shown to illustrate how the SPI driver can be used.

The first part shows how a SPI driver structure is setup and registered with the SPI bus. The name and type of the driver is specified as well as the typical driver callback functions; probe, remove, suspend and resume. All of this is setup in the `ads7846_init` function which is invoked when the module is loaded.

```
static struct spi_driver ads7846_driver = {
    .driver = {
          .name   = "ads7846",
          .bus    = &spi_bus_type,
          .owner = THIS_MODULE,
    },
    .probe       = ads7846_probe,
    .remove      = __devexit_p(ads7846_remove),
    .suspend     = ads7846_suspend,
    .resume      = ads7846_resume,
};

static int __init ads7846_init(void)
{
...

    return spi_register_driver(&ads7846_driver);
}
module_init(ads7846_init);
```

The second part shows some of the code in the `ads7846_probe` function. Here we can see how an SPI device is first setup with a mode and bit size. After that an SPI message is initialized and then a write request (`tx_buf` is set) and a read request (`rx_buf` is set) is initialized and added to the SPI message. The function `ads7846_rx_val` is then registered as the completion callback that will be called upon when the SPI transfer is complete.

```
static int __devinit ads7846_probe(struct spi_device *spi)
{

...
    struct spi_message              *m;
    struct spi_transfer             *x;

    /* We'd set TX wordsize 8 bits and RX wordsize to 13 bits ... except
     * that even if the hardware can do that, the SPI controller driver
     * may not.  So we stick to very-portable 8 bit words, both RX and
TX.
```

```
     */
    spi->bits_per_word = 8;
    spi->mode = SPI_MODE_1;
    err = spi_setup(spi);

    if (err < 0) {
          return err;
    }

...

    spi_message_init(m);

    /* y- still on; turn on only y+ (and ADC) */
    ts->read_y = READ_Y(vref);
    x->tx_buf = &ts->read_y;
    x->len = 1;
    spi_message_add_tail(x, m);

    x++;
    x->rx_buf = &ts->tc.y;
    x->len = 2;
    spi_message_add_tail(x, m);

    m->complete = ads7846_rx_val;
    m->context = ts;

...

}
```

The actual SPI transfer will occur when either the `spi_async` function or the `spi_sync` function is called.

```
...
    status = spi_async(spi, m);
...
    status = spi_sync(spi, m);
...
```

The SPI interface also contains other functions for reading and writing SPI data, such as an `spi_write` and a `spi_read` function. The interface is documented and available in the header file `/linux-2.6.x/include/linux/spi/spi.h`.

5.13 RTC

A Real Time Clock (RTC) is a clock that keeps track of the current time on a device and it usually has a separate power supply, such as a battery, so that it can keep track of the time even though the device is powered down.

The LPC24xx provides a low-power RTC with time measured in seconds, minutes, hours, day of month, year, day of week, and day of year. There is also an alarm output pin that can be used to wake up the microcontroller from Power-down mode.

5.13.1 Configuration

Real Time Clock support is enabled at the following place in the configuration tree:

Device Drivers → Real Time Clock → RTC class

In the Embedded Artists configuration, support for setting the Linux system time from the RTC source has also been enabled.

Device Drivers → Real Time Clock → Set system time from RTC on startup

In the configuration tree it is possible to select which RTC interface to use for user-domain applications. All the three alternatives have been enabled in the Embedded Artists configuration.

Device Drivers → Real Time Clock → sysfs

Device Drivers → Real Time Clock → proc

Device Drivers → Real Time Clock → dev

Finally the hardware support must also be enabled.

Device Drivers → Real Time Clock → NXP LPC2XXX RTC support

5.13.2 Driver Code

The source code for the hardware specific parts of the RTC driver is located in `uClinux-dist/linux-2.6.x/drivers/rtc/rtc-lpc22xx.c` file. The directory that contains the `rtc-lpc22xx.c` file also contains the generic RTC code, i.e., the code that is independent of hardware.

5.13.3 Usage

The Real Time Clock is exposed to user mode applications as a device file called `/dev/rtc`. Two applications from the µClinux distribution have been enabled and can be used to access the real time clock; `hwlock` and `date`.

The `hwclock` application is used to query and set the real time clock while the `date` application is used to query and set the Linux system time.

The system time can be set by issuing the command:

```
# date -i
```

The date application will then ask for the user to input current year, month, day, hour, minute and seconds. Please note that because of a bug in the date application you need to input year with two digits and not four. The year 2009 should, for example, be specified as 09.

When the system time has been set the `hwclock` application can be used to update the real time clock by issuing the following command.

```
# hwclock --systohc
```

The `hwclock` application will complain when issuing this command since it cannot write to the file system (since it is usually read-only), but the real time clock will still be updated.

The `hwclock` application can also be used to update the system time from the real time clock by issuing the following command.

```
# hwclock --hctosys
```

5.14 MTD

When running Linux on a desktop computer the file systems often reside on hard disks and are therefore handled by block devices. On embedded devices it is more common to have flash memories, such as a NOR or NAND memory, and a block device is not suitable to use for these kind of memories because of some fundamental differences, see the table below. To better support flash memories the Memory Technology Device (MTD) subsystem was developed.

Block device	Memory Technology Device
Consists of clusters and sectors	Consists of erase blocks
A hard disks sector size is in general small (512 – 1024 bytes)	Erase blocks are large, 32kB or 128kB are not unusual
Bad sectors are handled by hardware	Bad erase blocks are not hidden and should be handled in software
2 operations: read and write sector	3 operations: erase, read and write and erase block
	Erase blocks wear out after a certain number (depends on memory) of erase cycles. It is therefore important to avoid erasing the same erase block over and over again.

Please note that devices such as MMC/SD card, USB memory stick and compact flash are not MTD devices although they are referred to as flash memories. These devices are handled as block devices.

The conventional file systems such as FAT and ext3 are designed for block devices and not suitable to use with flash memories. Instead the Journalling Flash File System version 2 is recommended. This file system has a design which, for example, avoids erasing the same erase block on a regular basis to avoid the wear out problem.

For more information about the MTD subsystem in Linux see ref [16]. For more information about the JFFS2 file system, see ref [20].

5.14.1 Configuration

Support for Memory Technology devices in the Linux kernel is enabled at the following place in the configuration tree:

> *Device Drivers → Memory Technology Devices (MTD) → Memory Technology Device (MTD) support*

Especially when working with large flash memories it is convenient to be able to divide the memory into several partitions. Support for MTD partitioning is enabled at:

> *Device Drivers → Memory Technology Devices (MTD) → MTD partitioning support*

In order for the user domain to use `ioctl` to obtain information about the MTD device the following functionality must be enabled.

> *Device Drivers → Memory Technology Devices (MTD) → Direct char device access to MTD devices*

The following functionality is required for the Journalling Flash File System to obtain a handle on the MTD device.

> *Device Drivers → Memory Technology Devices (MTD) → Caching block device access to MTD devices*

There are many configuration options related to RAM/ROM/Flash Chip drivers. The options that have been selected in the Embedded Artists configuration are shown below.

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Detect flash chips by Common Flash Interface (CFI) probe

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Detect non-CFI AMD/JEDEC-compatible flash chips

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Specific CFI Flash geometry selection

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Support 16-bit buswidth

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Support 1-chip flash interleave

Device Drivers → Memory Technology Devices (MTD) → RAM/ROM/Flash chip drivers → Support for AMD/Fujitsu flash chips

In the "Mapping drivers for chip access" section the following options have been selected.

Device Drivers → Memory Technology Devices (MTD) → Mapping drivers for chip access → Support non-linear mappings of flash chips

Device Drivers → Memory Technology Devices (MTD) → Mapping drivers for chip access → CFI Flash device mapped on Embedded Artists LPC24xx OEM Board

5.14.2 Initialization Code

Below is an excerpt from the board specific initialization code, i.e., the `linux-2.6.x/arch/mach-lpc22xx/lpc2478_ea_board.c` file. It is only the partitioning of the flash memories that are shown and not all the initialization code.

The first part illustrates the partitioning of the NAND memory, which is divided into two partitions. The first partition is used for the kernel and the second partition is used for the root file system.

```
static struct mtd_partition lpc2478_ea_nand_partition_info[] =
{
    {
          .name = "nand_kernel",
          .offset     = 0,
          .size = 0x00300000,
    },
    {
          .name   = "nand_filesystem",
          .offset = 0x00300000,
          .size   = MTDPART_SIZ_FULL,
    },
};
```

The second part illustrates the partitioning of the NOR memory, which is also divided into two partitions. The first partition is used for the kernel and the second partition is used for the root file system.

```
static struct mtd_partition lpc2478_ea_flash_partition_info[] =
{
    {
          .name = "kernel",
          .offset     = 0,
            .size = 0x200000,
    },
    {
            .name   = "rootfs",
            .offset = 0x200000,
            .size   = MTDPART_SIZ_FULL,
    },

};
```

All these partitions are registered with the MTD subsystem using the `add_mtd_partitions` function starting with the NAND partitions. Because of this the NAND partitions will be mapped to the `/dev/mtdblock0` and `/dev/mtdblock1` devices while the NOR partitions will be mapped to the `/dev/mtdblock2` and `/dev/mtdblock3` devices. As you may recall from section 4.5.2 the `/dev/mtdblock1` device was specified in the boot argument when the root file system was stored in NAND memory and the `/dev/mtdblock3` was specified in the boot argument when the root file system was stored in NOR memory.

5.14.3 Driver Code

The source code for the MTD subsystem is located in `uClinux-dist/linux-2.6.x/drivers/mtd/` directory where the `uClinux-dist/linux-2.6.x/drivers/mtd/maps/lpc2468-ea-flash.c` file contains the NOR flash driver.

5.14.4 Usage

MTD is used when the root file system is stored in either the NOR or the NAND flash memory. In order for the kernel to find the root file system the boot argument must be setup in a way where MTD devices are used, see section 4.5.2 for an explanation of how this is done. The flash memories must also be updated with the root file system. This is described in section 4.5.8 for the NOR flash and in section 4.5.9 for the NAND flash.

When using a JFFS2 file system on NAND flash it is possible to modify the file system, i.e., add, delete or change files. To make sure that all changes made to the file system are not lost you need to issue the command below before the device is restarted or powered down.

```
# mount -o remount,ro /dev/mtdblock1 /
```

5.15 SFR

The Special Function Register (SFR) driver is a driver provided by Embedded Artists which is not integrated into the kernel source tree. The purpose of the driver is to give access to the registers of the processor from the user domain, mainly as a debug option. It is possible to access the registers by using the register's memory address or by its name.

5.15.1 Driver Code

The source code of the driver is located in `uClinux-dist/vendors/EmbeddedArtists/drivers/2.6.x/sfr/ea_sfr.c` file. The driver is implemented as a character device and is typically accessed directly from the console.

5.15.2 Usage

Before the driver can be used the driver module must be loaded into the Linux kernel by using the `insmod` command. When the module has been loaded it will be possible to access through the `/dev/sfr` device file.

```
# insmod /drivers/sfr.ko
```

The driver takes ASCII text strings as input from the console. Using the echo command is the easiest way to send text strings to the SFR device. First let's examine the read operation. The example below will read the value of the PINSEL2 register and the result will be printed in the console.

```
# echo PINSEL2:? > /dev/sfr
```

In the example above the name of the register was used, but it is also possible to use the register's address, which is illustrated in the example below.

```
# echo 0xE002C008:? > /dev/sfr
```

Note that register names are written with upper case letters. Addresses and numbers can be written in normal decimal (base 10) notation or in hexadecimal (base 16) notation. For the latter case, the prefix 0x must be used.

Now let's examine the write operation. Instead of writing a question mark to the driver after the register name or memory address you can specify the value to write to this memory address. The example below illustrates how to set pin P2.10 to an output and set the pin low. P2.10 is connected to a LED (just above the P2.10 button on the QVGA Base Board). By pulling the pin low, the LED will be lit.

```
# echo FIO2DIR:0x400 > /dev/sfr
# echo FIO2CLR:0x400 > /dev/sfr
```

By pulling the pin high the LED will be turned off which is illustrated below.

```
# echo FIO2SET:0x400 > /dev/sfr
```

There are some limitations with the driver. All read and write operations are 32-bit operations. Also, there is no protection for non-existing memory addresses. Writing to non-existing addresses will result in a data abort exception.

Normally it's not a good idea to show a bad example, but in this case it might be good to show how easy it is to crash the execution with this driver. The example below writes to a memory address that is in internal flash, which is read only. If you type the command below it will result in a data abort exception that will hang all program execution. Hitting the reset button is the only way forward after that.

```
# echo 0x00001234:0x12345678 > /dev/sfr
```

5.16 ADC

The LPC24xx contains Analog to Digital Converter (ADC) inputs with 10 bit resolution. Embedded Artists provides a driver for this device, but the driver is not integrated into the Linux kernel source tree. The QVGA Base Board connects the accelerometer to analog input channel 0 and 1 and a trim potentiometer to channel 2.

5.16.1 Driver Code

The source code of the driver is located in `uClinux-dist/vendors/EmbeddedArtists/drivers/2.6.x/adc/ea_adc.c` file. The driver is implemented as a character device and can be accessed from the console using the `cat` command.

5.16.2 Usage

Before the driver can be used the driver module must be loaded into the Linux kernel. This is done by using the `insmod` command.

```
# insmod /drivers/adc.ko
```

When the module has been loaded four devices will be available, `/dev/ad0`, `/dev/ad1`, `/dev/ad2`, and `/dev/ad3`, corresponding to analog inputs P0.23-A0.0, P0.24-A0.1, P0.25-A0.2 and P0.26-A0.3.

The driver will output a newline terminated string, representing zero to full scale with 10-bit resolution (0-1023). Note that resolution and precision is not the same thing. Consult NXP's LPC24xx user's manual for details about the precision of the ADC peripheral.

Reading a value from an analog input can be done using the `cat` command. In the example below the ADC2, i.e., A0.2 will be read. The value will be written to the console.

```
# cat /dev/adc2
```

5.17 Joystick

There is a 5-key joystick mounted on the QVGA Base Board. This joystick has 5 positions; up, down, left, right and center, and is connected to P2.22, P2.23, P2.25, P2.26 and P2.27. Embedded Artists provides a driver for this device, but the driver is not integrated into the Linux kernel source tree and is not using the input subsystem in the kernel. Instead it is exposed as a simple character device to the user domain.

5.17.1 Driver Code

The source code of the driver is located in `uClinux-dist/vendors/EmbeddedArtists/drivers/2.6.x/joystick/joy.c` file. The driver is implemented as a character device.

5.17.2 Usage

Before the driver can be used the driver module must be loaded into the Linux kernel. This is done by using the `insmod` command.

```
# insmod /drivers/joy.ko
```

When the module has been loaded it will be possible to access it through the `/dev/joy` device file and when reading this file the current state of the joystick will be returned. The following position mapping has been chosen for the joystick. Note that the values are given in hexadecimal (base 16) notation in the table.

Position	Value
UP	0x01
DOWN	0x02
LEFT	0x04
RIGHT	0x08
CENTER	0x10

Several of the joystick positions can be active at the same time, for example, both the UP and RIGHT position could be active at the same time meaning that the returned value from the driver would be an OR operation between the UP value and the RIGHT value, i.e., 0x09 would be returned.

Below is an example of a simple application that once a second will check the status of the joystick and print the result in the console. The application will end when the joystick is in the UP and RIGHT position.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define JOY_UP     0x01
#define JOY_DOWN   0x02
#define JOY_LEFT   0x04
#define JOY_RIGHT  0x08
#define JOY_CENTER 0x10

int main(int argc, char **argv)
{
    int          fd;
    int          data = 0;
    int          len = 0;

    fd = open("/dev/joy", O_RDONLY);
    if (fd < 0) {
          printf("Couldn't open /dev/joy\n");
          goto error;
    }

    while(data != (JOY_UP|JOY_RIGHT)) {
          len = read(fd, &data, 1);

          if (len <= 0) {
                printf("Failed to read from /dev/joy\n");
                goto error;
          }

          printf("joystick: %d\n", data);
          sleep(1);
    }

    close(fd);
    return 0;

error:
    return 1;
}
```

5.18 Frame Buffer Console

The Frame Buffer Console is a low-level frame buffer based console driver that allows the console to be displayed on the frame buffer device. With the Embedded Artists boards this means having the console on the QVGA display. Support for this functionality is included in the Linux kernel and only needs to be enabled. As long as there is a frame buffer device no extra source code needs to be added.

5.18.1 Configuration

Support for the Frame Buffer Console is enabled at the following place in the Linux configuration tree:

> *Device Drivers → Graphics support → Console display driver support → Framebuffer Console support*

If the font used in the console is different from the default font used by the frame buffer console support for built-in fonts can be enabled. A suitable font for a small display is the Mac 6x11 font. Support for fonts is enabled at:

> *Device Drivers → Graphics support → Console display driver support → Select compiled-in fonts*
>
> *Device Drivers → Graphics support → Console display driver support → Mac console 6x11 font*

It is also possible to enable frame buffer bootup logos, i.e., a logo being displayed in the frame buffer console. By default the logo being used is the Linux penguin also known as Tux. Support for bootup logos are enabled at the following place in the configuration tree:

> *Device Drivers → Graphics support → Logo configuration → Bootup logo*
>
> *Device Drivers → Graphics support → Logo configuration → Standard 224-color Linux logo*

5.18.2 Usage

When support for Frame Buffer Console has been enabled in the kernel configuration the boot arguments must be setup in a way where the console isn't mapped to the serial port. Section 4.5.2 describes how to setup boot arguments and a few examples are given. In all of those examples the boot argument takes a console parameter specifying that the console should be mapped to the serial port, see example below.

```
bootargs=root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
```

For the console to be mapped to the frame buffer the console parameter must be removed from the boot arguments see the example below.

```
bootargs=root=/dev/ram initrd=0xa1800000,4000k
```

The console is now visible on the display, but in order to control the display a keyboard must be attached to the board. This can be achieved by making sure the kernel has support for USB host and HID drivers, see section 5.8 . When the keyboard has been attached to the board the HID driver will be loaded and whatever is typed on the keyboard will be sent to the console.

6 Application Development

6.1 Introduction

The previous chapters have mainly focused on the Linux kernel; how to boot it and develop drivers for it. This chapter will describe how to develop applications that will run in a Linux environment. This chapter will only be an introduction to application development and not a complete description since that would be an entire book by itself.

6.2 Programming Language

When developing an application you need to select the programming language in which you are to write the application. The programming language will describe what the application does. As with spoken languages there are several programming languages to choose from; some which are very similar in the way they are structured while others are completely different.

For embedded systems running Linux you will most likely use the C or C++ programming languages when developing an application. These programming languages have a lot of similarities, but also some differences. The C programming language is, for example, a function oriented programming language while C++ is an object-oriented language.

In a function oriented programming language the application is constructed by a set of functions where each function performs a certain task. The functions are usually organized in modules to easier understand what they do and to group functions that belong together.

An object oriented programming language organize everything into objects where each object has some capabilities, can be modified and perform certain tasks. Many think of application development using an object oriented programming language as a way of modelling the real world since the real world consists of objects (car, road, human, animal, house, elevator…).

Programming languages such as C or C++ are for humans, software developers, to read and understand not for computers. The language for the computers are known as machine code and the C and C++ languages can be translated into machine code by using specific development tools.

6.3 Development Tools

When developing an application the software developer will come across several different tools that will be used to ease the development process. The list below describes the tools that almost all developers in different degrees will be using.

- **Editor** – a tool used to write the software application. This tool is very similar (and may be identical) to an ordinary text editor or word processor. Many different editors exist from really simple to more advanced editors. These will recognize the programming language being used and help the developer by, for example, highlighting specific keywords for the language and color code the text to make it more readable.
- **Compiler** – this tool will be used to translate the programming language into computer language, for example, from C code to ARM specific code. The compiler typically takes a source file (*.c) as input and produces an object or binary file (*.o) as output. When working in an environment different from the target environment in which the application will execute, for example, working in Linux on a PC (x86 processor) and developing an application for an ARM target, the compiler you are using is known as a cross-compiler.

- **Linker** – An application typically consists of several source files and each source file will result in an object file when compiled. In order to create *one* application all the object files must be combined into one executable file and for this purpose a linker is used. It is also the linker's responsibility to resolve symbols, i.e., to make sure that all the symbols, such as functions and global variables, being referenced in the source code are actually available. If a referenced symbol isn't available at link time the linker will end with an error message.
- **Make** – When an application becomes complex and consists of many source files the build process must be optimized in a way where the developer doesn't have to compile each file separately. A tool that aids the developer with the build process is often known as an automatic build utility and one of the most used utilities, especially in the Linux world, is the GNU *make* utility. With this utility it is possible to setup rules in so called makefiles that specify how the application is being built. The make utility will then automatically build the software using the rules in the makefiles.
- **Debugger** – When starting to execute the application it will most likely need to be debugged, i.e., analyzed for incorrect behaviour. There are several ways to do this, but the most efficient way is to use a debugger tool. In the Linux world the most used debugger is the GNU Debugger, also known as GDB.

6.4 APIs and Libraries

If you know the programming language being used the next step is to learn how to use the Application Programming Interfaces (APIs) and libraries that are available for your project. It is very unusual that a software project develops all the software from scratch since many standard APIs are available.

- **POSIX** – Portable Operating System Interface for Unix is a collection of standardized interfaces originally developed for Unix, but is now available for many operating systems
- **Standard C library** – When working with the C programming language you certainly need to know about the Standard C library. This library contains common functions such as string handling, input/output functions such as file manipulation, memory allocation, time conversion and much more. The standard C library is also known as libc or for embedded, memory constraint devices, you may come across uClibc.
- **Pthreads** – or POSIX threads is an interface for creation and manipulation of threads. If you need to have several tasks running in parallel in your application, for example one part reading data from an analog input while another part is updating a user interface, you will most likely use the Pthreads API to achieve this.
- **Sockets** – If the application need to exchange data over a network the socket API (also known as Berkely sockets or BSD sockets) will most likely be used. When working with sockets you typically setup a server socket if you publish a service, for example a web server with web pages. The application accessing the service is known as a client and will create a client socket to connect to the server and retrieve the information being published.
- **User Interface** – If the application needs to present something to a user, for example, by drawing onto a display it will need a graphical user interface. Several exists from the simplest that will only provide basic drawing mechanism such as draw a pixel, a line, or a rectangle to more advanced that will support complete windowing systems. For standard Linux you will come across the X-Window

System, but on embedded resource constraint devices the interface might be the Nano-X Window System (previously known as Microwindows).

6.5 Hello World Example

The “Hello World” application is the classical example used to show how to implement a simple application in a (new) programming language. It is the most basic application that will output the message “Hello World” to a user, most often to a console.

The C code for a Hello World application is given below. The function named `main` will be the first function called when the application starts. In this function the message “Hello World” will be printed using the `printf` function. The `printf` function is declared in the header file named `stdio.h` which is included at the beginning of the file.

```
#include <stdio.h>

int main(void)
{
    printf("Hello world\n");
    return 0;
}
```

When the C file has been created it is time to build it, i.e., compile and link it into an executable application. This could be done by invoking the compiler and linker manually, but instead a makefile with build rules will be used since that is the conventional way to build applications.

An example of a makefile is given below. This file starts with defining variables that will later be used in the build rules. The first variable, `CFLAGS`, contain flags, i.e., options sent to the compiler. The second variable, `LDFLAGS`, contain flags sent to the linker. The third flag contains the path to the cross-compiler that will be used to compile the C file.

The name of the application is defined in the `PROG` variable and the source files to compile are listed in the `SRC` variable. In this example only one source file is used. At the end of the makefile the build rules are defined that will actually build the application.

```
CFLAGS= -Wall -W
LDFLAGS=-Wl,-elf2flt
CC= /usr/local/bin/arm-elf-gcc
RM=rm -f


PROG=hello
SRC= hello.c


OBJ=$(SRC:%.c=%.o)

$(PROG): $(OBJ)
    $(CC) $(CFLAGS) -o $(PROG) $(OBJ) $(LDFLAGS)

.PHONY: clean all dep

clean:

    $(RM) $(PROG) $(OBJ) *~ *.gdb .depend *.elf2flt *.elf
```

When the make tool is invoked it will read and parse the makefile, build a dependency tree of build rules and files that need to be compiled and then actually build the application. The output will be an executable file that needs to be transferred to the target where it can be run, see section 6.9 for more information.

6.6 Threads Example

In this example two simple threads are created using the Pthreads API. Each thread will only print a message in the console where the message is a combination of a static string and an argument sent to the thread at the time of creation.

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

static void* mythread(void* arg)
{
    printf("mythread called arg = %s\n", (char *)arg);
    return NULL;
}

int main(int argc, char* argv[])
{
    int ret;
    pthread_t thread1;
    pthread_t thread2;


    if (argc < 3) {
          printf("Usage: pthreads arg1 arg2\n");
          exit(1);
    }

    ret = pthread_create(&thread1, NULL, mythread, argv[1]);
    if (ret != 0) {
          printf("Failed to create thread 1 ret=%d\n", ret);
          exit(1);
    }

    ret = pthread_create(&thread2, NULL, mythread, argv[2]);

    if (ret != 0) {
          printf("Failed to create thread 2 ret=%d\n", ret);
          exit(1);
    }
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    return 0;
}
```

The main function prototype is a little bit different from what it looked like in the Hello World example. In the Hello World example the main function didn't take any parameters, the parameter list was void declared. In this example two parameters are defined; `argc` and `argv`. Both of these are used when sending information, i.e., arguments to the application at startup time of the application. The first parameter will contain the argument count, i.e., how many arguments that are being sent to the application. The second parameter will contain the actual argument data (always string data).

The application doesn't accept the argument count to be less than three. If it is less than three the application will output a message telling the user about how to use the application and then the application will exit. The reason for the three arguments is that the first argument sent to an application is always the name of the application. Then the application needs two more values to send to each thread as individual data.

The next step is to create the thread which is achieved by calling the `pthread_create` function. The first parameter is a thread handle that can later be used to manipulate the thread. The second parameter is attributes, but when a NULL value is given default attributes will be used. The third is the function that will be run by the thread. The last parameter is the argument sent to the application which is given as data to the thread.

The last part of the main function is two calls to `pthread_join`. The join function will halt execution until the thread calling join on has finished.

```
CFLAGS= -Wall -W
LDFLAGS=-Wl,-elf2flt -lpthread

CC= /usr/local/bin/arm-elf-gcc

RM=rm -f

PROG=pthreads

SRC= pthreads.c

OBJ=$(SRC:%.c=%.o)

$(PROG): $(OBJ)
    $(CC) $(CFLAGS) -o $(PROG) $(OBJ) $(LDFLAGS)

.PHONY: clean all dep


clean:
    $(RM) $(PROG) $(OBJ) *~ *.gdb .depend *.elf2flt *.elf
```

The makefile for this application is very similar to the makefile for the Hello World application. One difference is that we need to explicitly tell the compiler to include the Pthreads library when linking the application. This library is not included by default.

6.7 Networking Example

This is an example that illustrates how to implement a simple server that will listen to connections from clients and write to the console whatever the client sends to the server.

The first thing to do is to create an unbound socket and specify domain (`AF_INET`) and type (`SOCK_STREAM`) for that socket. `AF_INET` means that the used domain is Internet addresses and `SOCK_STREAM` means that a streaming, reliable, connection is desired, i.e., basically a TCP connection.

The next step is to bind the socket, i.e., associate it with an address. The address structure is setup with information about protocol family, port number and address. Then the `bind` function is called.

Since this is to be a server application the next step is to set the socket into a listening mode where it will accept incoming connections. This is achieved by calling the `listen` function.

After the `listen` function has been called the server is ready to handle incoming connections. The application must, however, call the `accept` function to retrieve the socket that will be associated with an established connection. A common design of an application is to configure the socket in a blocking mode which means that the `accept` function will block execution until a connection is present. When having the socket configured for blocking mode it is very convenient or almost necessary to use threads enabling other parts of the application to continue to run while waiting for clients to connect.

The last part of this application is to retrieve the data sent by the client. This is done by calling the `recv` function.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 5000   // The port to listen for connections on
#define BACKLOG 10

int main ()
{
    int sockfd;          // Socket file descriptor
    int nsockfd;         // New Socket file descriptor
    int sin_size;        // to store struct size
    struct sockaddr_in addr_local;
    struct sockaddr_in addr_remote;
    int num;
    char data = 0;

    /* create a socket */
    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    {
        printf ("ERROR: cannot create a socket\n");
        return (0);
    }
    else
    {
        printf ("OK: sucessfully created a socket\n");
    }

    /* setup the local socket address */
    addr_local.sin_family = AF_INET;            // Protocol Family
    addr_local.sin_port = htons(PORT);          // Port number
    addr_local.sin_addr.s_addr  = INADDR_ANY;  // AutoFill local address
    bzero(&(addr_local.sin_zero), 8);  // Flush the rest of struct


    /*  bind the socket */
    if( bind(sockfd, (struct sockaddr*)&addr_local, sizeof(struct
sockaddr)) == -1 )
    {
        printf ("ERROR: cannot bind Port %d\n",PORT);
        return (0);
    }
    else
    {
        printf("OK: bound to port %d sucessfully\n",PORT);
    }

    /*  listen for incoming connections */
    if (listen(sockfd,BACKLOG) == -1)
    {
        printf ("ERROR: cannot listen to port %d\n", PORT);
        return (0);
    }
    else
    {
        printf ("OK: listening on port %d\n", PORT);
    }

    while(1)
    {
        sin_size = sizeof(struct sockaddr_in);


      /*
        * Wait for a connection, and obtain a new socket
        * file despriptor for single connection
        */
        if ((nsockfd = accept(sockfd, (struct sockaddr
*)&addr_remote, &sin_size)) == -1)
        {
            printf ("ERROR: failed to accept the connection\n");
            continue;
        }
        else
        {
            printf ("OK: A client has connected from %s\n",
inet_ntoa(addr_remote.sin_addr));
        }

        data = 0;
        while (data != '.')
        {
            if ((num = recv(nsockfd, &data, 1, 0)) == -1)
            {
                    printf("ERROR: receive failed\n");
                    close(nsockfd);
                    exit(1);
            }

            if (num > 0)
                printf("%c", data);
            else if (num == 0)
                break;      // exit if nothing was received
        }

        printf ("closing connection\n");
        close (nsockfd);
     }
}
```

6.8 Nano-X Example

Nano-X, previously known as Microwindows is a windowing system designed for resource constraint devices. It runs on Linux systems with kernel framebuffer support and is using a client / server model, which means that the application acts as a client requesting graphical services from the server.

The example below illustrates an application that paints a yellow rectangle in a white window where the yellow rectangle will be able to receive touch events. Each touch event will be drawn as a red point in the yellow rectangle.

The first step is to establish a connection with the server. This is done by a call to the `GrOpen` function. If the Nano-x server isn't started or the connection fails the `GrOpen` function will return an error code. In this example the application will in such a case exit.

Information about the screen, such as width and height is retrieved by a call to `GrGetScreenInfo`. This information is then used when creating the main window with a call to `GrNewWindow`. The window is created with a white background and black border.

The yellow rectangle is then created. This rectangle is also a window, but created with the main window as its parent.

A subscription for mouse events (touch) is then started only on the rectangle, which means that events will only be sent to the application when touching inside the rectangle.

A graphical context is created with a call to `GrNewGC`. Drawing can take place in a graphical context. In this example the foreground color is set to red, i.e., when something is drawn, such as a point, it will be red.

The windows are made visible by calls to `GrMapWindow`. Then the application enters into an infinite loop receiving and handling events. The events are handled by the `handleevent` function and whenever an event of type `GR_EVENT_TYPE_MOUSE_MOTION` is received a point will be drawn inside the rectangle.

```
#include <stdio.h>
#include <nano-X.h>

#define RECT_X0 50
#define RECT_X1 190
#define RECT_Y0 50
#define RECT_Y1 150

static GR_WINDOW_ID   mainwid;
static GR_WINDOW_ID   rectwid;
static GR_GC_ID rectgc;

static void
handleevent(GR_EVENT *ep)
{
    GR_EVENT_MOUSE  mouse;
    switch (ep->type) {
          case GR_EVENT_TYPE_MOUSE_MOTION:
                mouse = ep->mouse;
                if (mouse.wid == rectwid) {
                      GrPoint(rectwid, rectgc, mouse.x, mouse.y);
                }

                break;
          case GR_EVENT_TYPE_CLOSE_REQ:
                GrClose();
                exit(0);
    }
}

int main(void)
{
    GR_SCREEN_INFO    si;
    if (GrOpen() < 0) {
          printf("Cannot open graphics\n");
          exit(1);
    }

    GrGetScreenInfo(&si);

    /* create main window */
    mainwid = GrNewWindow(GR_ROOT_WINDOW_ID, 0, 0, si.cols,
si.rows, 0, WHITE, BLACK);

    /* create rectangle */
    rectwid = GrNewWindow(mainwid, RECT_X0, RECT_Y0, RECT_X1-
RECT_X0, RECT_Y1-RECT_Y0, 1, YELLOW, BLACK);

    GrSelectEvents(rectwid, GR_EVENT_MASK_MOUSE_MOTION);

    rectgc = GrNewGC();
    GrSetGCForeground(rectgc, RED);

    GrMapWindow(mainwid);
    GrMapWindow(rectwid);

    while (GR_TRUE) {
          GR_EVENT event;

          GrGetNextEvent(&event);
          handleevent(&event);
    }

    return 0;
}
```

In order to be able to build the application the Nano-X library must first have been built and made available. In the Embedded Artists distribution, the library is available in the `uClinux-dist/user/microwin/src/lib` directory.

In the makefile for the application, paths to the library and header files are setup and provided to the compiler as well as the linker.

```
NANOX_LIB=/home/user/uClinux-dist/user/microwin/src/lib
NANOX_INC=/home/user/uClinux-dist/user/microwin/src/include

CFLAGS= -Wall -W -L$(NANOX_LIB) -I$(NANOX_INC) -DMWINCLUDECOLORS
LDFLAGS=-Wl,-elf2flt -lnano-X

CC= /usr/local/bin/arm-elf-gcc
RM=rm -f

PROG=graphics
SRC= graphics2.c

OBJ=$(SRC:%.c=%.o)

$(PROG): $(OBJ)
    $(CC) $(CFLAGS) -o $(PROG) $(OBJ) $(LDFLAGS)

.PHONY: clean all dep
```

```
clean:
    $(RM) $(PROG) $(OBJ) *~ *.gdb .depend *.elf2flt *.elf
```

Before running the application the Nano-X server must have been started so that it can accept connections and handle requests from the client. If the Nano-X library and server have been built and made available, the server is started by issuing the command below.

```
# nano-X &
```

6.9 Run Application on Target

Since the application is built on a host computer and not on the actual target it must somehow be transferred to the target before it is possible to run the application. Several alternatives exist and a few of them are described below.

6.9.1 NFS

During the development phase you normally want short development cycles, i.e., the time it takes from that the application has been built until it can be tested on the target. If the target and the host computer are connected to a network a convenient way to get a hold of the application is to mount the host computers file system onto the target's file system. By doing that, the application will be available for execution as soon as it has been built.

To be able to mount the host computer's file system onto the target's file system you need to have support for a Network File System (NFS) protocol. When the host computer's file system has been mounted, using NFS, it will look as though it is a local file system on the target. The example below shows how to mount an external file system.

```
# mount -t nfs -o nolock,rsize=4096,wsize=4096
192.168.0.10:/home/user /mnt/nfs
```

In this example the host computers IP address is 192.168.0.10 and the exported file system is located at `/home/user`. The file system is mounted onto the local directory `/mnt/nfs`.

Note that the host computer must export the directory that is to be accessed from the target. Section 9.4.3 describes how to export a directory in the Debian Linux distribution.

6.9.2 USB Memory Stick

If the target isn't connected to a network an alternative way to get the application to the target is to copy it to a USB memory stick. When the application has been copied to the USB memory stick it has to be attached to the target and then mounted.

Section 5.8.3 describes how to mount a USB memory stick.

6.9.3 ROMFS

If the application is to be distributed with the file system associated with the target it is convenient to setup rules in a way where the application is included in the ROMFS when the ROMFS is being built.

With the µClinux distribution there is a shell script called `romfs-inst.sh` that can be invoked to copy files to the ROMFS file system. This script can also be accessed from a makefile through the variable named ROMFSINST. This variable is, for example, used in the makefile copying the Embedded Artists applications to the file system. The makefile is available at: `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/Makefile`. An example of how two applications are copied is given below.

```
    $(ROMFSINST) applications/led /bin/led
    $(ROMFSINST) applications/calibrate /bin/calibrate
```

7 Development Environment

7.1 Introduction

In order to build µClinux and the u-boot bootloader there is a need for a development environment. This development environment is preferably running a Linux operating system. An alternative to run Linux would be to use Cygwin, the Linux-like environment for Windows, and compile the sources on a Windows machine. The Linux kernel build environment is however depending on a number of tools that are commonly found in Linux distributions, but not as common in Cygwin or Windows.

Not to enforce anyone to install Linux on their workstation a virtualization approach can be used where Linux will run in a virtual machine. The virtualization technology that Embedded Artists has chosen for their development environment is the VMware Player, see ref [21], which can be run on a Windows or Linux PC.

This chapter will also give you an introduction to the Debian Etch Linux distribution and a description of some key concepts common for most Linux distributions.

7.2 Virtualization

Virtualization is a technique where hardware resources, such as CPU, memory and peripherals are being abstracted to a virtualization layer. Each virtual machine running on top of the abstraction layer will see the hardware resources as if they would have direct access to the resources and thereby makes it possible to have, for example, several operating systems running on the same physical machine at the same time. The operating system running in the virtual machine won't know if it is running in a virtual machine or directly on a physical machine.

Even though several virtual machines are running on the same physical machine they will be isolated from each other as if they where separate physical machines. The advantages are that different operating systems or applications won't interfere with each other.

Another advantage of running operating systems and applications in a virtual machine is hardware independence. It is only the virtual machine that needs to be ported to a new hardware, not the operating system or applications. This is, for example, exactly the approach taken by the Java programming language where the output from the Java compiler is byte code which will be interpreted by a Java virtual machine.

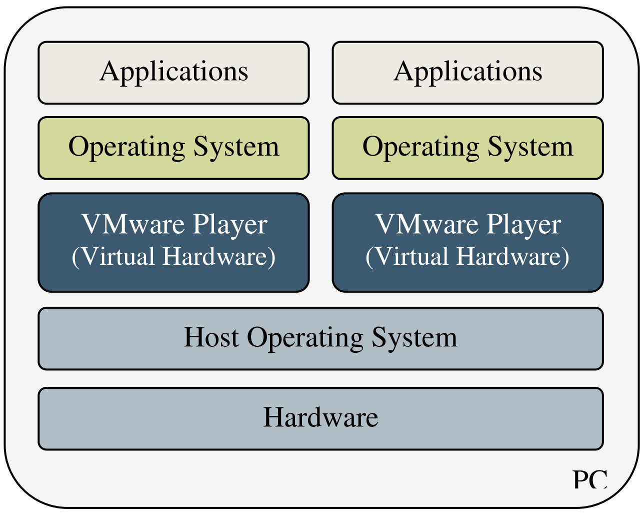


Figure 4 Virtualization overview

Figure 4 above illustrates a use case where a PC is running a host operating system such as Windows Vista and two virtual machines (VMware) where each virtual machine runs an operating system. The operating systems in the different virtual machine instances don't have to be the same kind of operating system, for example, one may run Linux and the other a Windows version such as Windows XP.

7.2.1 Virtualization Techniques

Several different virtualization techniques exist where each is trying to solve the same basic problem, but with some small differences.

- **Full Virtualization –** When using full virtualization the virtual machine can expose the hardware resources needed to run an unmodified operating system, i.e., exactly the same OS software that would be installed on a physical machine can be installed in the virtual machine.
- **Para Virtualization –** With Para-virtualization the guest operating system must be modified before running in the virtual machine. The reason is that the virtual machine only give a similar, but not an identical abstraction of the hardware. Para-virtualization is often used when you need to get almost the same performance from the virtual machine as you would have if you where running directly on the hardware.
- **Hardware assisted –** With hardware assisted virtualization the hardware, processor, is assisting the virtual machine with some hardware instructions so that it is possible to get full virtualization with better performance. Intel offers a technique they call VT-x and AMD is calling their technique AMD-V.

VMware has written a good whitepaper that explains the difference between different virtualization techniques in more detail, see ref [22] for more information.

7.2.2 VMware Player

As mentioned in the previous section the virtualization technology that will be used for the development environment is VMware Player from the VMware organization. VMware Player is a free to use virtualization software that enables you to run virtual machines created by, for example, VMware Workstation, WMware Fusion, WMware Server or WMware ESX.

With VMware Player it is possible to run several operating systems at the same time on a single PC where the PC is running either a Windows or Linux host operating system.

Virtual Appliances

For the VMware Player more than 900 virtual machines with pre-installed operating systems and applications, virtual appliances, exist on the Virtual Appliance Marketplace, see ref [23]. On this marketplace most of the popular Linux distributions, in different configurations, are available for download. Whenever you would like to evaluate, for example, a Linux distribution the Virtual Appliance Marketplace is a place to turn to.

7.3 Debian Distribution

A Linux distribution is a distribution of software applications on top of the Linux kernel, i.e., the Linux kernel and a set of libraries and utilities from the GNU project is the base of the distribution. Software applications such as editors, browsers, window managers, and more are built on top of this base. A popular web site with information, comparisons and news about Linux distributions is the DistroWatch.com web site, see ref [24]. That web site contains statistics about distributions making it possible to see which distribution is the most popular at any given moment.

There are more than 100 different Linux distributions today and Debian, see ref [25], is one of these distributions and is known to be one of the most popular. It comes with a very large number of software packages (more than 18000).

Debian is a free to use distribution available for about 10 architectures (i386, AMD, ARM, MIPS, ...) and many other distributions have been based on the Debian distribution, for example, Ubuntu, Damn Small Linux and Knoppix. It is known for its strict adherence to the Unix and free software philosophies and for being a very stable distribution.

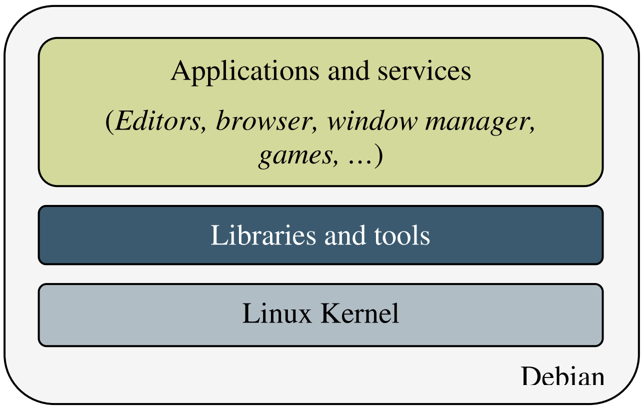

Applications and services
(Editors, browser, window manager, games, …)
Libraries and tools
Linux Kernel
Debian

Figure 5 Linux distribution overview

At the time of writing this book Debian 5.0 with code name Lenny had just been released. The distribution used by Embedded Artists is Debian 4.0 with code name Etch. The code names of Debian releases are names of characters from the computer animated film Toy Story.

7.3.1 Users and Login

Since Linux is a multi-user system, i.e., a system which allows several users to get access to the operating system, Linux requires you to login and authenticate who you are during start-up of the system. Different users may get different access rights, i.e., rights to access different parts of the operating system. With multi-user support it is possible to, for example, setup a system where user X won't to get access to user Y's files and folders whereas user Y will have access to user X's files and folders.

In a Linux system there is one user who has all rights and permissions. That user is known as the root or super-user of the system. Some refer to the root as the janitor of the system, i.e., the one that has the keys to everything, can access everything and makes sure everything is working as it should. However, given access to everything comes with a responsibility and you must know what you're doing or else it is very easy to get the system into an unstable state which may be hard to recover from. Don't take on the role as a janitor if you don't really need to.

For the Embedded Artists setup of Debian the user named "user" will automatically be logged in to the system and the Desktop environment (graphical user interface) named GNOME will be started.

Users and passwords

Below is a list of the users created in the Embedded Artists setup of Debian.

User name	Password

`user`	`user`
`root`	`root`

7.3.2 Basic Commands

Below is a list of common commands. Several of these are frequently used in everyday work when using Linux as an operating system.

Command	Description
`man`	Manual pages – This command will start the manual page for a specific command. The example below will get the manual for the ls command. `# man ls`

`ls`	List the contents of a directory, i.e., which files and folders that are available in a directory. The first example below will just list the content of the current directory and present it in the default way. The second example will give a long listing with more information, such as permissions; file size and modification date of the content. The third alternative will list the content of a specific directory.
	`# ls`
	`# ls -l`
	`# ls /home`

`pwd`	Print the name of the current directory.
`cd`	Change directory. The example below change the directory to /home/user. `# cd /home/user`
`mkdir`	Create a directory. The example below will create a directory named mydir in the current working directory. `# mkdir mydir`

`cp`	Copy files and directories. The first example will copy a specific file to a new location and at the same time change the name of the file. The second example will recursively copy the content of one directory to another directory.
	`# cp file1.txt /home/user/myfile1.txt`
	`# cp -r dir1 dir2`

`mv`	Move (rename) files. The first example renames the file file1.txt to file2.txt. The second alternative will move the file file1.txt from the current directory to the /home/user/dir1/ directory.
	`# mv file1.txt file2.txt`
	`# mv file1.txt /home/user/dir1/file.txt`

`rm`	Remove files or directories. The first example will remove the file file1.txt and the second example will recursively remove all the contents of directory dir1.
	`# rm file1.txt`
	`# rm -r dir1`

`echo`	Display a line of text. The first example will display the text "hello there" on standard output. The second example will display the text

	available in the variable named SHELL on standard output. The third alternative will echo the string mytext and direct the output to the file file.txt.
	`# echo hello there`
	`# echo $SHELL`
	`# echo mytext > file1.txt`

`cat`	Concatenate files and print in standard output. This command can be used to concatenate files, but most often it is just used to output the content of a file to standard output. `# cat file1.txt`

`less`	Less will display the content of a file one screen at a time, i.e., if the content of the file will take up more space than one screen the less command will stop and wait for input before continuing to display the content. You can navigate in the content by using the up and down arrows and exit the listing by pressing the 'q' key.
	The first example shows the content of the file file1.txt. The second example show the use of the pipe command ('|'). The output of the ls command will be "piped" to the less command.
	`# less file1.txt`
	`# ls -l /bin | less`

`locate`	List files in databases that match a pattern. The example below will list all files containing the string file1.txt in its absolute path. The system administrator can update the databases, see the updatedb command. `# locate file1.txt`

`find`	Search for files in a directory hierarchy. The first example will find all files containing the string "file" starting from the current directory and all sub-directories. The second example will start searching in the /home/user directory for the same pattern as in the first example.
	`# find -name *file*`
	`# find /home/user -name *file*`

`grep`	Find patterns in a file and print the line matching the pattern. The first example will search for the pattern "myword" in all files ending with .txt in the current directory. The second alternative will recursively search for the pattern "myword" in all files found starting from the current directory.
	`# grep myword *.txt`
	`# grep -r myword *`

`which`	Locate a command. This command will locate and return the pathname of the command given as input. The example below will locate the path to the find command `# which find`
`xargs`	Build and execute command lines from standard input. The example below will find all files with extension h, pipe them to xargs and then let grep find the string MY_CONSTANT in those files. `# find . -name *.h | xargs grep MY_CONSTANT`

7.3.3 The File System

The Linux file system deserves a section of its own since it is such a central part of the Linux operating system. A simple description of a Linux system is that:

"On a Linux system, everything is a file; if something is not a file, it is a process".

The statement is a simplification, but basically true since there are many different kinds of files. A directory is, for example, a special kind of file which contains a list of other files. Devices, such as a hard disk, mouse or a display are all represented as special kind of device files. Applications and services are also represented as files and in order to manage all of these files they are placed in a well-defined hierarchy.

It all begins with the root directory `'/'` and then expands into sub-directories. Even though a sub-directory may be a completely separate physical disk compared to the root directory it is still represented as an ordinary directory somewhere underneath the root directory. This way of organizing a file system is known as a unified file system. The opposite of a unified file system is the file system found in the Windows operating system. Here each physical disk drive or partition is given a separate drive letter (c:, d:, e:, and so on). It should be noted that on a Windows operating system it is also possible to mount other physical disk drives onto the "main" file system so that it looks as if it is an ordinary directory although it is not as common to do this as it is in a Linux operating system.

How come that different Linux distributions have a similar file system hierarchy. The reason for this is simply standardization. In the early years of Linux distribution development each distribution basically had its own hierarchy, but in order to minimize confusion and optimize understandability when moving from one distribution to another the *File system Hierarchy Standard* (FHS) was developed, see ref [26].

According to the FHS the root directory should contain the following sub-directories:

Directory	Description
`bin`	Essential command binaries, such as the shell and much used commands like cp, mv, rm, cat and ls.
`boot`	Static files of the boot loader
`dev`	Device files, such as hard drive, CD-ROM, floppy disk, printer, etc.
`etc`	Host-specific system configuration, such as the system startup scripts (/etc/rc.d)
`lib`	Essential shared libraries and kernel modules
`media`	Mount point for removable media
`mnt`	Mount point for mounting a file system temporarily
`opt`	Add-on application software packages
`sbin`	Essential system binaries. These binaries are essential to the working of the system and required by all users.
`srv`	Data for services provided by this system
`tmp`	Temporary files
`usr`	Secondary hierarchy. Contains user binaries such as telnet ftp, X window management.
`var`	Variable data, such as mail, output from the printer daemon and logs.

7.3.4 File Permissions

As mentioned in section 7.3.1 Linux allows several users to access to the operating system. In order to keep the system secure and stable, i.e., prevent a user from accessing and modifying critical system files, Linux has a file system where it is possible to set different kind of permissions on files and directories. Besides protecting system files it is also possible for a user to protect his/her files from access by other users, i.e., it is possible to have private files.

The permissions that have been assigned to a file can easily be discovered by running the `ls -l` command in a console. All the files and directories available in the directory where the command is executed will be listed. In the example given below two files and one directory is listed.

```
$ ls -l
-rw-rw-r--  1 joe students  17223 2008-12-02 10:21 report.txt
-rw-r--r--  1 joe joe       55322 2008-11-16 19:19 diary.txt
drwxr-xr-x  3 joe joe        4096 2009-02-01 12:17 tmp
```

To better explain what the information means the list has been divided into 7 columns as illustrated in the table below. The first column contains the file permissions. These symbols will be explained in more detail later in this section. The second column describe the number of available files which for a file will be 1, but for a directory can be a larger number, for example, 3 as in the example. The third column specifies who owns the file. The fourth column specifies which group that has access to the file. The fifth column specifies the size of the file in bytes. The sixth column specifies the last modification date and time of the file and the last column specifies the name of the file or directory.

Permissions	Number of files	User	Group	Size	Modification date and time	Name
`-rw-rw-r--`	1	joe	students	17233	2008-12-02 10:21	report.txt
`-rw-------`	1	joe	joe	55322	2008-11-16 19:19	diary.txt
`drwxr-xr-x`	3	joe	joe	4096	2009-02-01 12:17	tmp

For the permissions part there are 4 categories of information. The first part is a directory flag where a ‘d’ means a directory and ‘-‘ means a normal file. The second part is the permissions assigned to the owner of the file and the permissions are read (‘r’), write (‘w’) and execute (‘x’). A ‘-‘ means that the permission isn’t set. The read permission of course means that it is allowed to read the file, the write permission means that it is allowed to modify the file and the execute permission means that it is allowed to execute the file. The execute permission has only a meaning for programs and executable scripts when set for files. The execute permission on a directory means that it is allowed to list the content of the directory.

The third part are the permissions for the group associated with the file and the fourth part are permissions for all the other users, i.e., basically the whole system.

Type	Owner	Group	Other	…	Name
-	rw-	rw-	r--	...	report.txt

-	rw-	---	---	...	diary.txt
d	rwx	r-x	r-x	...	tmp

In the example above the file named `report.txt` can be read and modified by both the owner (`joe`) of the file and all users belonging to the group `students`, while all other users can only read the file.

The `diary.txt` file can only be read and modified by the owner; no one else may access the file.

The `tmp` directory is accessible, i.e., the content can be listed by all users, but can only be modified by the owner.

Sections 9.5.7 and 9.5.8 describe how permissions as well as group and owner settings can be changed on files and directories.

7.3.5 Desktop Environment

A Desktop Environment is the graphical user interface in which a user is working when using a modern computer. The alternative to a desktop environment is a command-line based interface, which still is very popular for Linux based servers.

In Linux the base of a desktop environment is the X Window system which provides the interface towards the display, such as basic drawing primitives and placement and appearance of windows. It also manages input device such as a mouse or a keyboard. The desktop environment will then define the look-and-feel of the user interface, i.e., the way, windows, toolbars, icons are drawn and how they behave.

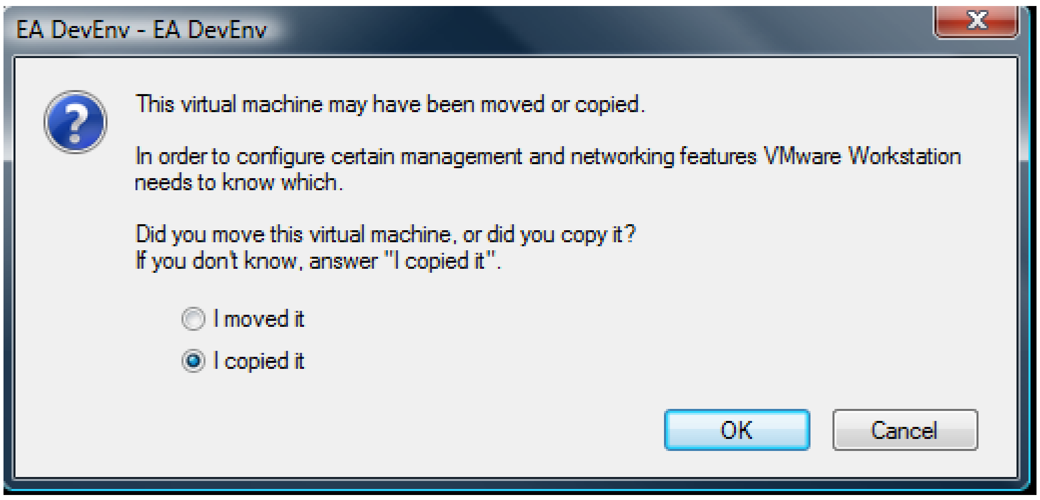
In Linux several desktop environments exist, but the two most popular are KDE, see ref [27], and GNOME, see ref [28]. The Debian Etch distribution provided by Embedded Artists is using the GNOME desktop environment.

8 Guides – VMware Player

8.1 Getting Started

This section contains a step by step guide of how to get up and running with VMware and the Debian Etch Linux distribution. The pre-requisite of this exercise is to have a DVD containing the development environment from Embedded Artists.

1. Begin by downloading and installing the VMware Player. See ref [21] for a link to where the player can be downloaded.
2. Unpack the development environment found on the DVD (the 3.1 version is called `EA_DevEnv_v3_1.7z` and is a 7-Zip packed file, see ref [29]).
3. A number of files will be unpacked and the most important one in this step is a file named `EA-dev.vmx` which is a configuration file for the virtual machine. Double-click on this file (if you are running a Windows OS) and the VMware Player will be started. If VMware Player has already been started the configuration file can be opened from within the player by selecting the Open command and then browsing to the configuration file.
4. When the VMware Player starts you will be prompted with a question asking you if the virtual machine was moved or copied, see Figure 6 below. Select “I copied it” and click OK. This screen will only appear the first time the virtual machine is started.
5. The Debian Etch Linux distribution will now start.


EA DevEnv - EA DevEnv

This virtual machine may have been moved or copied.

In order to configure certain management and networking features VMware Workstation needs to know which.

Did you move this virtual machine, or did you copy it?
If you don't know, answer "I copied it".

I moved it [✓ I copied it]

Figure 6 Moved or Copied VM

8.2 Changing Memory Allocation

Depending on the amount of RAM available in the workstation running the VMware Player the amount of RAM consumed by the player might need to be increased or decreased. If the Debian Etch distribution is perceived as running the reason could be too little allocated memory for the virtual machine. Increasing the memory allocation might give you better performance.

1. Go to the WMware Player Menu.
2. Select the Troubleshoot menu alternative, see Figure 7 below, and then select Change Memory Allocation.

3. A dialog window will appear where the amount of memory allocated to the virtual machine can be changed.
4. When the memory allocation has been changed the player must be restarted.

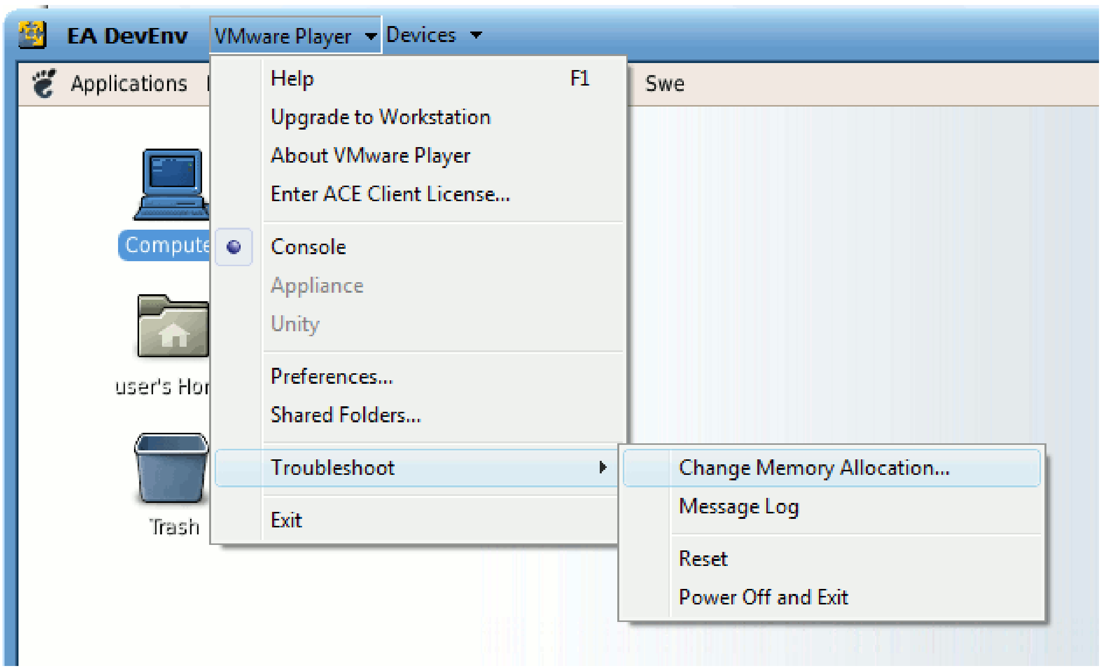


Figure 7 Change Memory Allocation in VMware

8.3 Enable / Disable Hardware Devices

In order for the virtual machine to get access to devices attached to the host computer, such as the CD-ROM drive, the device must be enabled in VMware Player. By default the removable devices are shown in the VMware Player Devices menu as can be seen in Figure 8 below where a CD drive, Ethernet interface and a Sound adapter are enabled and accessible from the virtual machine.

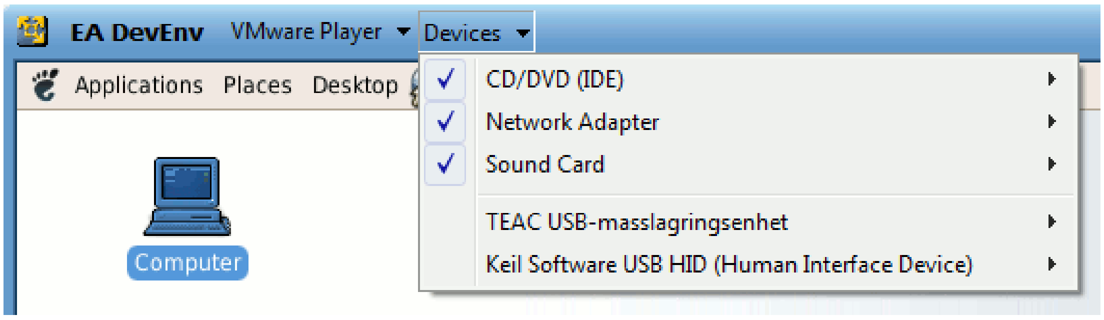

EA DevEnv
VMware Player
Devices
Applications Places Desktop
✓ CD/DVD (IDE)
✓ Network Adapter
✓ Sound Card
TEAC USB-masslagringsenhet
Keil Software USB HID (Human Interface Device)
Computer

Figure 8 VMware Toolbar Example 1

The accessibility of a device is toggled by going to the device’s menu and then selecting, for example, disconnect. In Figure 9 below the CD drive has been disabled, while the other devices are still enabled.

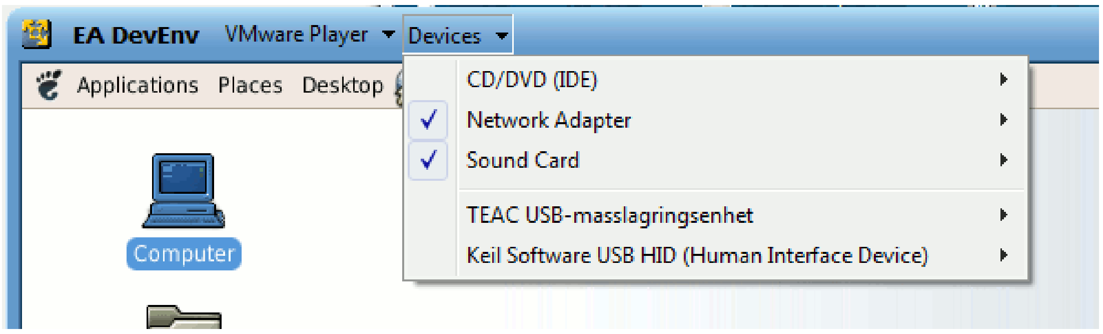


Figure 9 VMware Toolbar Example 2

Some of the devices may only be used by one machine at a time, i.e., either the virtual machine or the host computer has access to the device. This limitation is applicable for example for disk drives and USB devices. The Ethernet adapter is an example of a device which can be used by both the host computer and the virtual machine at the same time.

8.4 Share Data with Host OS

In many situations it is convenient to be able to share files and information between the virtual machine and the host computer. For VMware Player there are a number of ways to do this. The easiest way is to use a technique called *Shared Folders*.

8.4.1 Shared Folders

Follow the steps below to enable support for shared folders.

1. Go to the VMware Player menu
2. Select the Shared Folders menu alternative, see Figure 10.
3. A dialog window will appear, see Figure 11. Select one of the “Always Selected” or “Enabled until next power off or suspend” alternatives.
4. Click the OK button

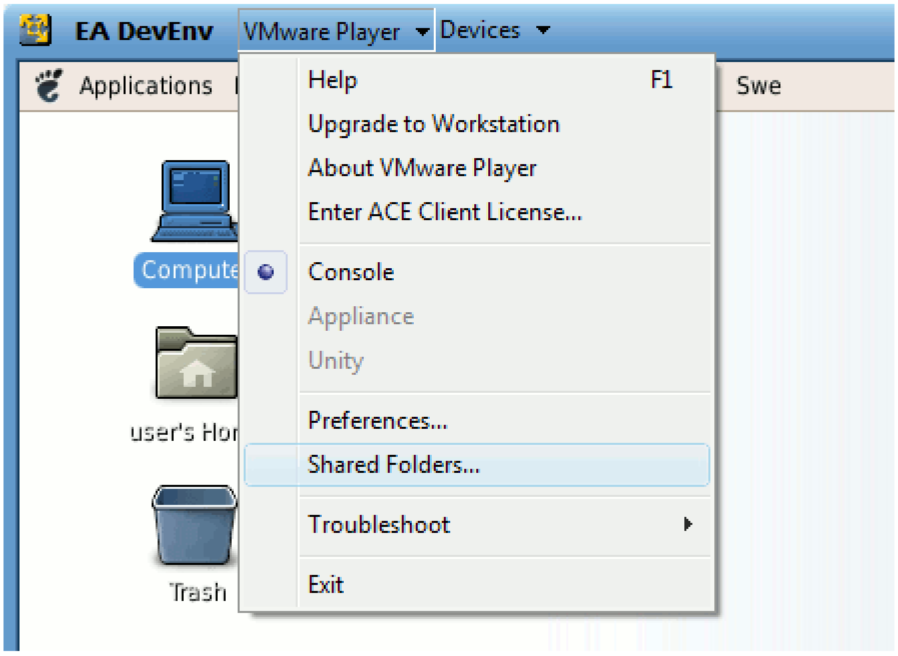


Figure 10 Shared Folders Menu

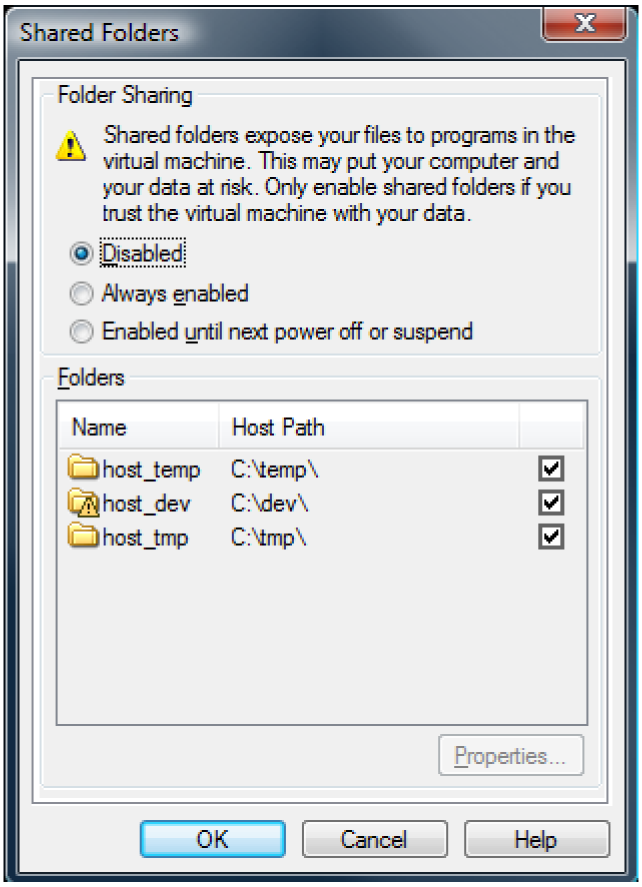
The folders that have been selected to be shared are listed in the settings dialog. If a folder isn't accessible, for example, not created, this is indicated with an icon with an exclamation point; see the `host_dev` folder in Figure 11.

If the virtual machine is running a Linux OS the shared folders will be available under `/mnt/hgfs`. If the virtual machine is running a Windows OS the shared folders must be mapped as a network drive.

In the Embedded Artists development environment the `host_temp` and `host_dev` folders, map to `c:\temp` and `c:\dev`, have been setup as shared folders. If you want to add, remove or change the folders that are shared you need to edit the configuration file (file extension .vmx). An example of how the configuration file may look like can be seen below.

```
sharedFolder0.present = "TRUE"
sharedFolder0.enabled = "TRUE"
sharedFolder0.readAccess = "TRUE"
sharedFolder0.writeAccess = "TRUE"
sharedFolder0.hostPath = "C:\temp\"
sharedFolder0.guestName = "host_temp"
sharedFolder0.expiration = "never"
sharedFolder1.present = "TRUE"
sharedFolder1.enabled = "TRUE"
sharedFolder1.readAccess = "TRUE"
sharedFolder1.writeAccess = "TRUE"
sharedFolder1.hostPath = "C:\dev\"
sharedFolder1.guestName = "host_dev"
sharedFolder1.expiration = "never"
sharedFolder.maxNum = "2"
```

Please note that usually the vmx file shouldn't be edited manually. Instead the VMware Workstation application from VMware could be used to create the configuration file.


Shared Folders

Folder Sharing

Shared folders expose your files to programs in the virtual machine. This may put your computer and your data at risk. Only enable shared folders if you trust the virtual machine with your data.

[✓ Disabled] Always enabled Enabled until next power off or suspend

Folders

Name	Host Path	
host_temp	C:\temp\	✓
host_dev	C:\dev\	✓
host_tmp	C:\tmp\	✓

Figure 11 Shared Folders Settings

8.4.2 Drag and Drop

If the virtual machine (the guest operating system) has VMware Tools installed (the Embedded Artists development environment has this) it is possible to drag and drop files as well as copy and paste information between the guest and host operating system. The VMware Tools service must be running in the guest operating system for this functionality to work. More information about VMware Tools can be found on the VMware website, see ref [21]. Instructions of how to install VMware tools are also available on the VMware website, see ref [39] for a link.

8.4.3 Additional Ways

The above described procedures are probably the easiest ways of sharing data between the host and guest OS, but there are more alternatives. It is, for example, possible to setup a network drive or a windows share that Linux can connect to. In section 9.4.2 there is more information about how to connect to a network drive.

8.5 Access to the Network

VMware offers several ways of letting the virtual machine get access to the network.

- **Bridged networking** – The virtual machine will get a unique identity on the network, separate from the host OS. Bridged networking will make the virtual machine visible to other computers on the network and they can communicate directly with the virtual machine. If you have an Ethernet adapter on your workstation, bridge networking is the easiest way of providing network access to your virtual machine.

- **Network Address Translation (NAT)** – With this alternative the virtual machine will share the IP and MAC addresses of the host. NAT is useful when you might be restricted to only use one IP address in your network. A limitation with this alternative is that it will not be possible for computers on the external network to initiate connections to the virtual machine.

There are more ways to configure network access, but the two options above are the most useful alternatives.

A setting in the configuration file called ethernet0.`connectionType` defines which network access mechanism that will be used by the virtual machine. The value of this setting can be “bridged” for bridged networking or “nat” for NAT enabled networking.

```
ethernet0.connectionType = "bridged"
```

Instead of changing the configuration file it is also possible to select the connection type in the Devices → Network Adapter menu.

8.5.1 Problems with Network Access

- **Shutdown** – Sometimes the network connection may be lost after a restart of the virtual machine without first shutting down Linux. This problem is most often solved by doing a proper restart of Linux. Section 9.10.6 describes how Linux can be properly restarted.
- **Firewall** – a software firewall might block access to the virtual machine. If you have problems with being unable to connect to your virtual machine try to disable your software firewall temporarily and do a new connection attempt.

9 Guides – Debian Linux

9.1 Getting Started

If you have followed the instructions in section 8.1 Getting Started with VMware Player you will have Debian Linux up-and-running, see Figure 12 below.

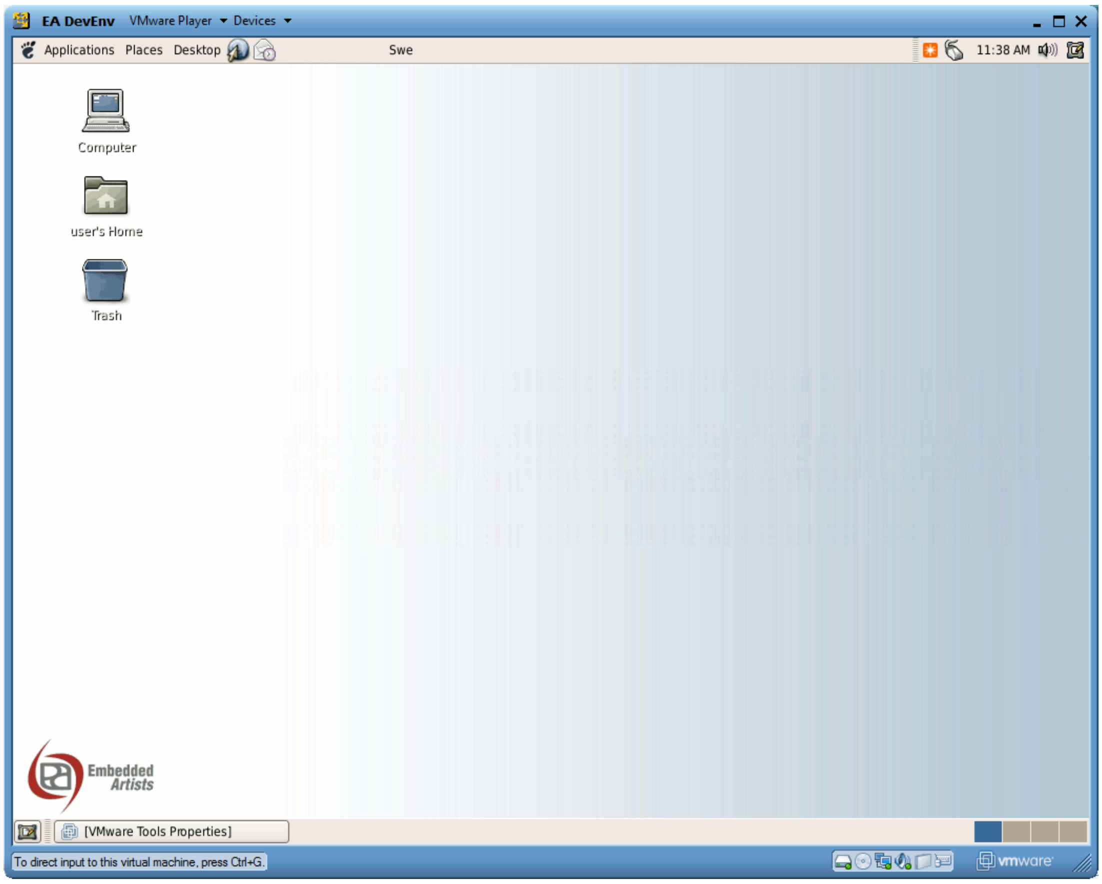


Figure 12 Debian after Login Running In a VMware Virtual Machine

Debian Linux has similarities with other operating systems, such as Microsoft Windows. They, for example, both have the concept of a desktop, icons and menus.

Let’s start by exploring the top left corner of the desktop. Here you will find the main menus in Debian, see Figure 13.

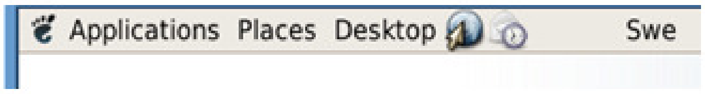


Figure 13 Main menus in Debian

- **Applications** menu – this menu contains links to most of the applications in Debian. You will find Accessories, Games, Editors, web browers and more in this menu.
- **Places** menu **–** this menu contains links to typical storage locations, such as your home folder, the CD Player, Network folders, and Recent Documents.
- **Desktop** menu – this menu contain links to Administration tools, Preferences settings, help document, and Shutdown button.

Directly to the right of the Desktop menu there are shortcuts to the web browser and to the e-mail client in Debian. A little bit further to the right is the keyboard layout indicator. In

Figure 13 it says "Swe" which means that a Swedish keyboard layout is being used. By clicking on this indicator you cycle between all activated keyboard layouts.

Now let's continue to the top right corner of the desktop which contains four different symbols that will be described.

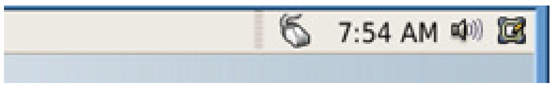


Figure 14 Top right panel of Debian

- **Network** – the first symbol indicates which kind of network you have. In Figure 14 the symbol for a wired connection is shown.
- **Clock** – the next part of the panel is the current time.
- **Volume** – the third symbol is used to control the volume of the speaker.
- **Window selector** – the last part is a window selector. By clicking on this icon you will bring up a list with all open windows (applications).

Just below the Applications menu you will find the icons, also known as launchers, on the desktop. By default there are three icons available. In Figure 15 an icon for the CD/DVD is also available since a DVD is inserted into the CD/DVD player. More icons can be added to the desktop and it is also possible to move them around and place them wherever you wish.

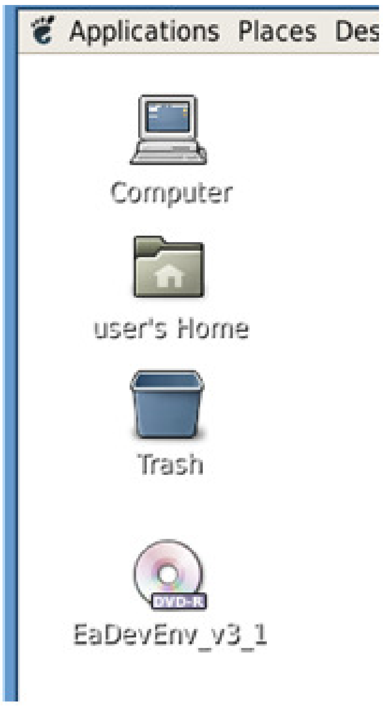


Figure 15 Icons in Debian

The last part that we will describe in this section is the workspace switcher which is located in the lower right corner of the desktop as shown in Figure 16. There are four workspaces, virtual desktops, available and you can switch between them by clicking on one of the squares.

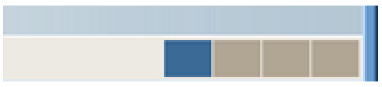


Figure 16 Workspace Switcher

9.2 Terminal / the Shell

9.2.1 Introduction

The most important application to users is the shell. This piece of software is used as the interface between the user and the operating system and can either be a graphical interface, such as the Desktop environment, or a text based interface also called command line interface (CLI). You are using a text based interface when running the Terminal application or when running Linux without a Desktop environment. The most important purpose of the shell is to allow a user to launch other applications.

Many people don't refer to the Desktop environment as a shell, but instead mean the command line interface when they are talking about the shell. For the remaining part of this section we will refer to the CLI when talking about the shell.

There are many types of shells available for Linux and the most popular is the GNU Bourne Again Shell (bash). Other well-known shells are C shell (csh), TENEX C shell (tcsh), Z shell (zsh) and Korn shell (ksh).

You can find out which shell you are running by issuing the following command in the Terminal application.

1. Start the Terminal application from the Applications menu.

   Application → Accessories → Terminal

2. Run the `ps` command to get information about a process.

```
$ ps -p $$
```

3. The result of running the command in a Debian distribution with bash as a shell is something like the example below.

```
  PID TTY          TIME CMD
 6242 pts/0    00:00:00 bash
```

The command you have run means that you want to get information about the current process. The current process when running in a shell is the shell itself.

Usually there is an environment variable available named SHELL which you can print to see which shell is being used. The variable will contain the absolute path to where the shell is located.

```
$ echo $SHELL
```

9.2.2 Browse the File System

Since the shell is usually a text based interface you need to be able to navigate through the file system by using simple commands. Section 7.3.2 describes the basic commands that you need to know about and a few of these are used to navigate through the file system.

1. Open a Terminal application as described in the previous section.
2. Enter the `pwd` command to see the absolute path to your current directory. The result in this example is `/home/user`

```
$ pwd
/home/user
```

3. Enter the `ls` command to see the content of the current directory. Non-hidden files and folders will be listed. In this example three directories were available.

```
$ ls
Desktop  u-boot-1.1.6  uClinux-dist
```

4. If you would like to get more details about the content as well as see “hidden” files (files where the name starts with a dot ‘.’ are considered to be hidden) you can use the options `–la`, see Figure 17 for the result of running this command.

```
$ ls -la
```

5. The previous command is usually stored in an alias `ll`.
6. Change the directory to `uClinux-dist`.

```
$ cd uClinux-dist
```

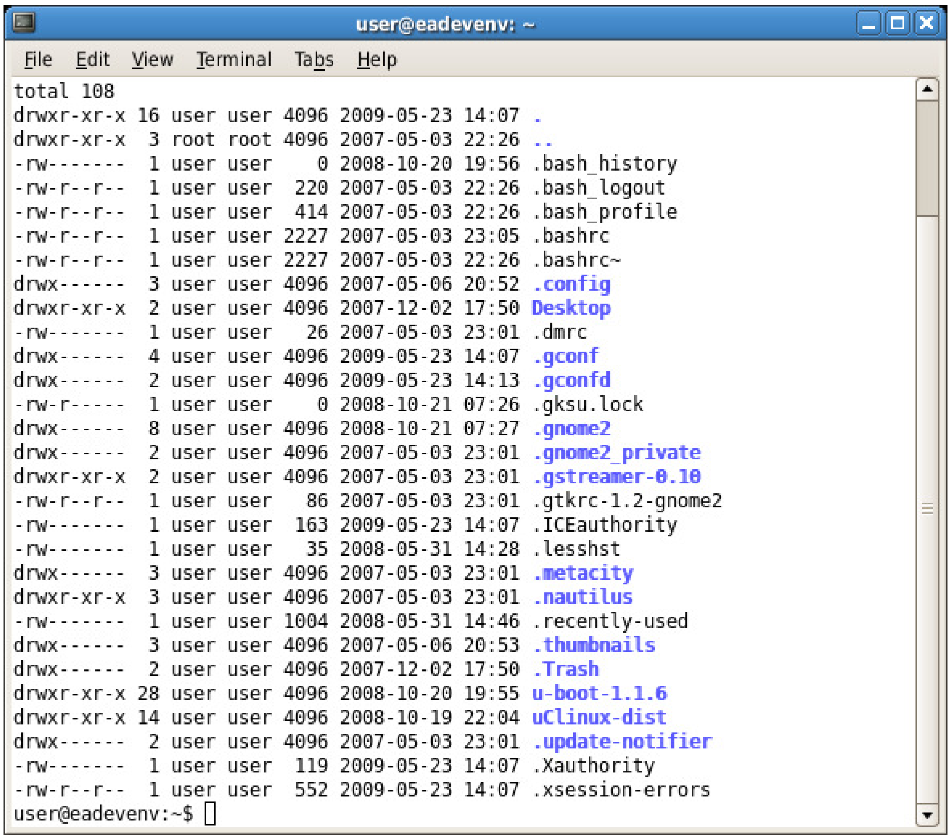
7. Change back to the previous directory (done by using the `cd` command with the `..` directory).

```
$ cd ..
```

8. Linux has also support for command line completion which means that you can start writing the name of, for example, a directory and then press the tab key to fill in the remaining part of the name. Change the directory to uClinux-dist again, but only write the beginning of the name (uC) and then press the tab key.

```
$ cd uC [Press tab key]
```

If there are several directories starting with the same letters then you would have to continue to fill in the name until you have a unique beginning. You could press the tab key twice to get a list of all the directories/files starting with the letters you have started to write.



```
total 108
drwxr-xr-x 16 user user 4096 2009-05-23 14:07 .
drwxr-xr-x  3 root root 4096 2007-05-03 22:26 ..
-rw-------  1 user user    0 2008-10-20 19:56 .bash_history
-rw-r--r--  1 user user  220 2007-05-03 22:26 .bash_logout
-rw-r--r--  1 user user  414 2007-05-03 22:26 .bash_profile
-rw-r--r--  1 user user 2227 2007-05-03 23:05 .bashrc
-rw-r--r--  1 user user 2227 2007-05-03 22:26 .bashrc~
drwx------  3 user user 4096 2007-05-06 20:52 .config
drwxr-xr-x  2 user user 4096 2007-12-02 17:50 Desktop
-rw-------  1 user user   26 2007-05-03 23:01 .dmrc
drwx------  4 user user 4096 2009-05-23 14:07 .gconf
drwx------  2 user user 4096 2009-05-23 14:13 .gconfd
-rw-r-----  1 user user    0 2008-10-21 07:26 .gksu.lock
drwx------  8 user user 4096 2008-10-21 07:27 .gnome2
drwx------  2 user user 4096 2007-05-03 23:01 .gnome2_private
drwxr-xr-x  2 user user 4096 2007-05-03 23:01 .gstreamer-0.10
-rw-r--r--  1 user user   86 2007-05-03 23:01 .gtkrc-1.2-gnome2
-rw-------  1 user user  163 2009-05-23 14:07 .ICEauthority
-rw-------  1 user user   35 2008-05-31 14:28 .lesshst
drwx------  3 user user 4096 2007-05-03 23:01 .metacity
drwxr-xr-x  3 user user 4096 2007-05-03 23:01 .nautilus
-rw-------  1 user user 1004 2008-05-31 14:46 .recently-used
drwx------  3 user user 4096 2007-05-06 20:53 .thumbnails
drwx------  2 user user 4096 2007-12-02 17:50 .Trash
drwxr-xr-x 28 user user 4096 2008-10-20 19:55 u-boot-1.1.6
drwxr-xr-x 14 user user 4096 2008-10-19 22:04 uClinux-dist
drwx------  2 user user 4096 2007-05-03 23:01 .update-notifier
-rw-------  1 user user  119 2009-05-23 14:07 .Xauthority
-rw-r--r--  1 user user  552 2009-05-23 14:07 .xsession-errors
user@eadevenv:~$ 
```

Figure 17 Long listing of /home/user

9.2.3 List Content of Files

There are several commands that can list the content of files. The most used are `cat`, `more` and `less`.

1. First try the `cat` command to list the content of a file.

```
$ cat uClinux-dist/README
```

2. As can be seen the complete content of the file is directly printed in the console. If the file is large it is better to output a part of the file at a time. This is something that the `more` and `less` command can do. These commands are similar, but `less` is more sophisticated than the `more` command since you can, for example, navigate back and forward in the file using the up and down arrow.

```
$ less uClinux-dist/README
```

3. It is also possible to direct the output of another command to, for example, the less command. The command below will list the content of the bin directory, but let the less command only display a small part of the list at a time.

```
$ ls -la /bin | less
```

9.2.4 Search for Files / Content

There are often situations where you would like to locate files, commands and/or content of a file. Several commands exist for this purpose.

1. Locate files using the `locate` command. The example below locates all files with the name bashrc in its path. Note that this command will not actually search the file system when it is run, but instead search in a database that contains an index of the files on the file system. Since the database isn't updated often recent files won't be a part of the database. The `updatedb` command can be used to update the database.

```
$ locate bashrc
```

2. A command that will search the file system is the `find` command. The example below will search for the files ending with the name "bashrc" in the current directory and all sub-directories.

```
$ find -name *bashrc
./.bashrc
./uClinux-dist/user/bash/examples/startup-files/bashrc
```

3. It is also possible to explicitly specify in which directory to start the search. The example is using the root directory ('/') as starting directory

```
$ find / -name *bashrc
```

4. If you are looking for which command that is used and where it is located (several with the same name could be installed) you can use the `which` command.

```
$ which ls
/bin/ls
$ which arm-linux-gcc
/usr/local/bin/arm-linux-gcc
```

5. If you are instead looking for patterns in a file you can use the `grep` command. The example below will look for the string "alias ll" in all files in the current directory and specify the line number where the string is found.

```
$ grep -n “alias ll” .*
.bashrc:63:alias ll='ls -lA'
```

6. You can also search recursively, i.e., the current directory and all sub-directories.

```
$ grep -r -n “alias ll” .
./.bashrc:63:alias ll='ls -lA'
./uClinux-dist/user/bash/examples/startup-
files/Bash_aliases:14:alias ll="ls -l"
./uClinux-dist/user/bash/examples/startup-files/bashrc:11:alias
ll='ls -l'
```

7. A little more advanced example is to combine several commands. The following example will find all files with extension .h and pipe that list to the `xargs` command and then let `grep` search that list for the string EMBEDDED.

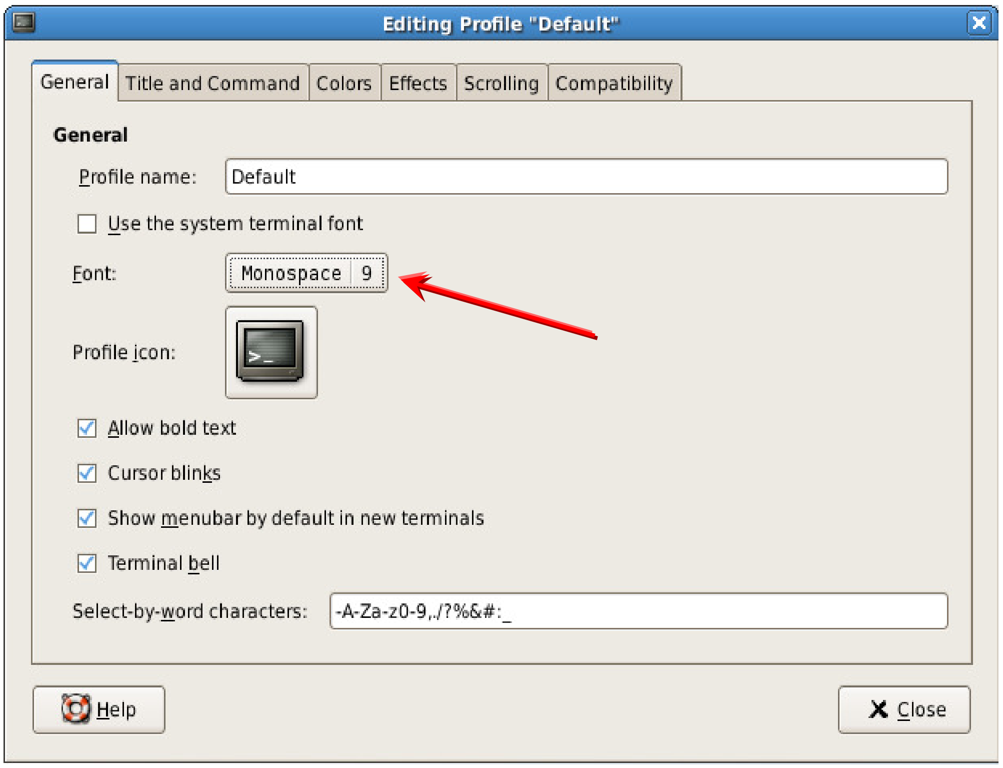
```
$ find . -name *.h | xargs grep EMBEDDED
```

9.2.5 Change Settings of Terminal

You change the font size of the terminal application by selecting the menu alternative

Edit → Current Profile.

You will then be presented with the Editing Profile dialog, see Figure 18, and there it is possible to change the font size.



Editing Profile "Default"

General

Profile name: Default

☐ Use the system terminal font

Font: Monospace 9

Profile icon:

☑ Allow bold text

☑ Cursor blinks

☑ Show menubar by default in new terminals

☑ Terminal bell

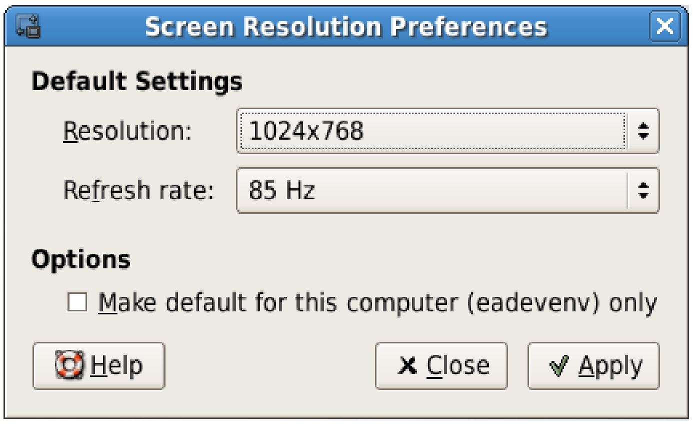
Select-by-word characters: -A-Za-z0-9,./?%&#:_

Figure 18 Editing Default Profile for the Terminal Application

9.3 Customize the Desktop

9.3.1 Changing Screen Resolution

The default screen resolution is set rather low to make sure it starts up correctly on most computer configurations. For most users the resolution is, however, too low since screens today are rather large and therefore the resolution needs to be changed.



Default Settings

Resolution: 1024x768

Refresh rate: 85 Hz

Options

☐ Make default for this computer (eadevenv) only

Figure 19 Screen Resolution Preferences Dialog

1. Go to the Desktop menu and select Preferences and then Screen resolution.

   Desktop → Preferences → Screen resolution

2. Select the resolution you wish and then click the Apply button.
3. A dialog window will appear asking you to either keep the resolution you have selected or go back to the previous resolution. Click on the "Keep the resolution" button to change the resolution.

If the resolution you want isn't available in the resolution list it is possible to use the VMware config tools to change the display size.

1. Open a terminal application.

   Applications → Accessories → Terminal

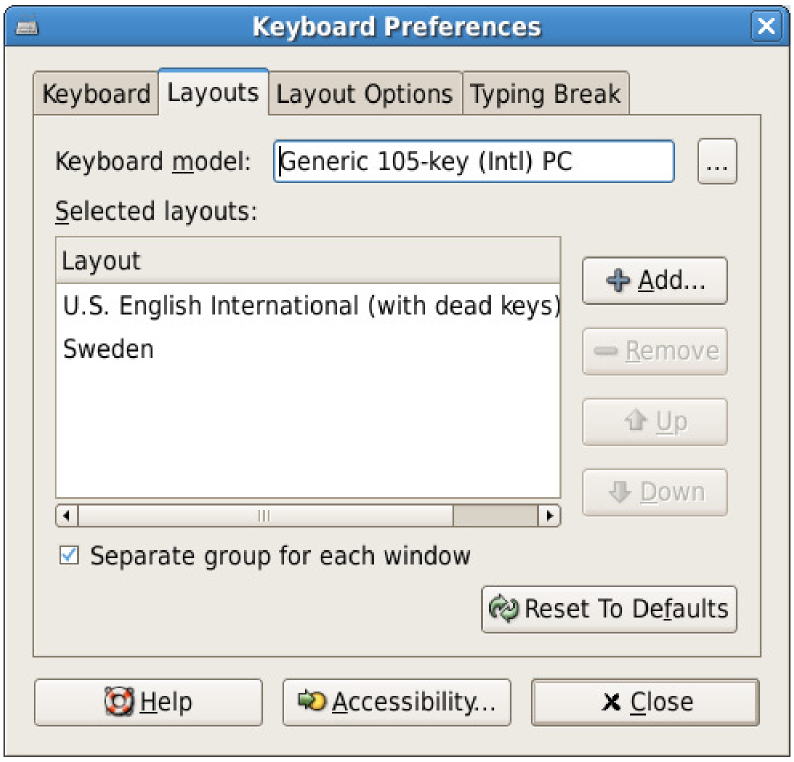
2. Run the VMware config tools script with administrative privileges.

```
$ sudo vmware-config-tools.pl
```

3. When asked for a password enter "user" (without the quotation marks).
4. You will be presented with a number of questions. Provide the default answer, i.e., press the Enter key until you get a question about the display size ("Do you want to change the display size that X start with?). Select the screen resolution you want by entering the number displayed in front of the resolution and then press Enter.

9.3.2 Changing Default Keyboard

The default keyboard layout is U.S. English. If you don't have such a keyboard the layout must be changed.



Keyboard model: Generic 105-key (Intl) PC

Selected layouts:

Layout
U.S. English International (with dead keys)
Sweden

☑ Separate group for each window

Figure 20 Keyboard Preference Dialog

1. Go to the Desktop menu and select Preferences and then Keyboard.

   Desktop → *Preferences* → *Keyboard*

2. In the dialog window that appears select the Layouts tab.
3. By default U.S. English and Swedish layouts are added, but more can be added by clicking on the Add button.
4. Change the preference by selecting the layout and then clicking the Up or Down button.

It is also possible to change layout by clicking on the Keyboard indicator on the menu bar, see Figure 21 below.

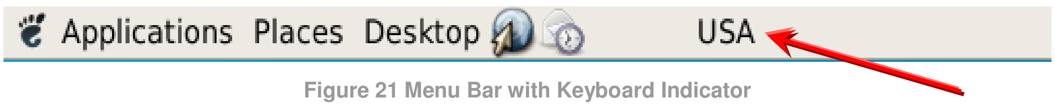
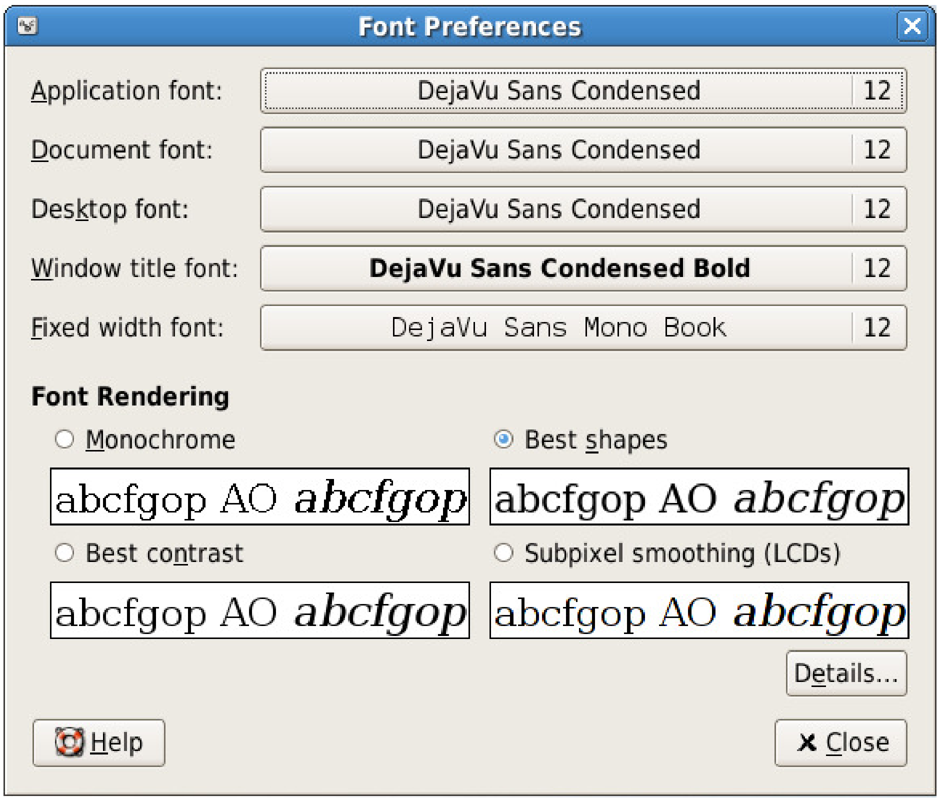


Figure 21 Menu Bar with Keyboard Indicator

By clicking on the Keyboard indicator you will toggle between the different layouts that have been added to the Keyboard Preferences.

9.3.3 Changing Font Sizes

The font sizes used throughout the graphical interface are by default rather large. If you have good eye vision you might want to change the size. By lowering the font size, dialogs and other application windows will also take up less space from the desktop.



Application font: DejaVu Sans Condensed 12
Document font: DejaVu Sans Condensed 12
Desktop font: DejaVu Sans Condensed 12
Window title font: DejaVu Sans Condensed Bold 12
Fixed width font: DejaVu Sans Mono Book 12

Font Rendering

( ) Monochrome
abcfgop AO *abcfgop*

(●) Best shapes
abcfgop AO *abcfgop*

( ) Best contrast
abcfgop AO *abcfgop*

( ) Subpixel smoothing (LCDs)
abcfgop AO *abcfgop*

Figure 22 Font Preferences Dialog

1. Go to the Desktop menu and select Preferences and then Font.

   Desktop → Preferences → Font

2. Change the font sizes for different parts of the desktop by clicking on the drop down list. By default the size is set to 12.
3. The size will be changed immediately for all windows.
4. Besides changing the size of the font it is also possible to change the type of the font.

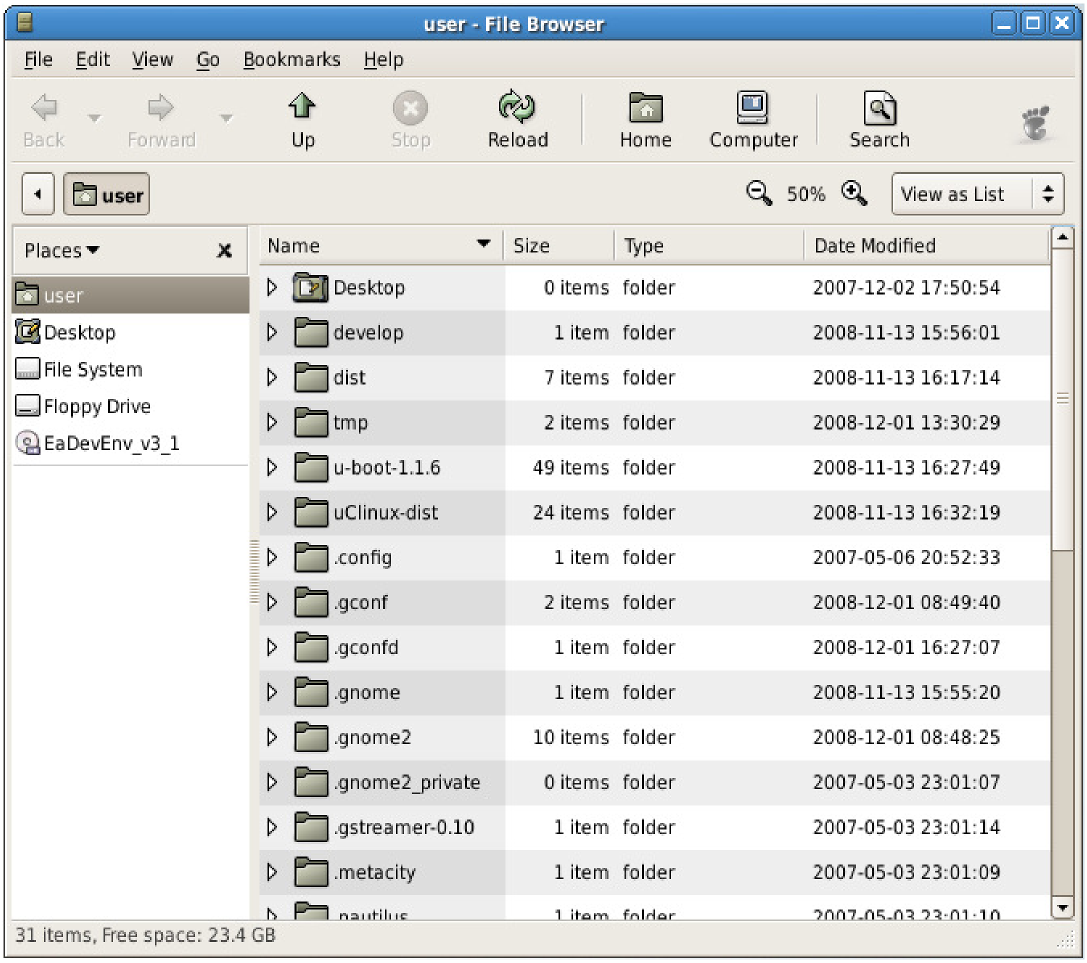
Note that not all applications will be affected by these settings. You might need to do local changes to alter the font size of specific applications, such as for the Terminal application or the Editor you are using.

9.4 File Systems

9.4.1 Browsing the File System

It is possible to browse the file system in several ways, for example, by using any of the file system related commands listed in section 7.3.2 from a Terminal. A more user-friendly way might be to use a graphical tool. Debian comes with a File Browser application which is found in the applications menu.

Applications → System Tools → File Browser



Name	Size	Type	Date Modified
Desktop	0 items	folder	2007-12-02 17:50:54
develop	1 item	folder	2008-11-13 15:56:01
dist	7 items	folder	2008-11-13 16:17:14
tmp	2 items	folder	2008-12-01 13:30:29
u-boot-1.1.6	49 items	folder	2008-11-13 16:27:49
uClinux-dist	24 items	folder	2008-11-13 16:32:19
.config	1 item	folder	2007-05-06 20:52:33
.gconf	2 items	folder	2008-12-01 08:49:40
.gconfd	1 item	folder	2008-12-01 16:27:07
.gnome	1 item	folder	2008-11-13 15:55:20
.gnome2	10 items	folder	2008-12-01 08:48:25
.gnome2_private	0 items	folder	2007-05-03 23:01:07
.gstreamer-0.10	1 item	folder	2007-05-03 23:01:14
.metacity	1 item	folder	2007-05-03 23:01:09
.nautilus	1 item	folder	2007-05-03 23:01:10

31 items, Free space: 23.4 GB

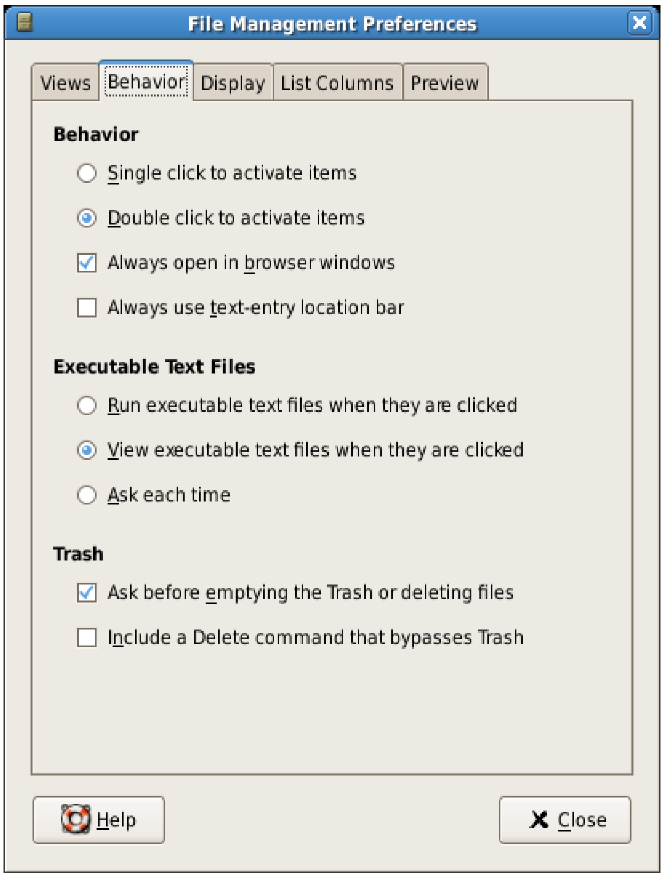
Figure 23 File Browser Application

There are several ways of starting the File Browser, for example, by using the Computer or Home shortcuts on the desktop, see Figure 15.

The look and behaviour of the File Browser application can be configured by opening the File Management Preference dialog which is found in the Edit menu of the File Browser application.

Edit → Preferences

One thing that a lot of people want to enable is the behaviour "Always open in browser windows". If this choice isn't selected each folder you navigate to will be opened in a new window.



Behavior

- ( ) Single click to activate items
- (●) Double click to activate items
- [✓] Always open in browser windows
- [ ] Always use text-entry location bar

Executable Text Files

- ( ) Run executable text files when they are clicked
- (●) View executable text files when they are clicked
- ( ) Ask each time

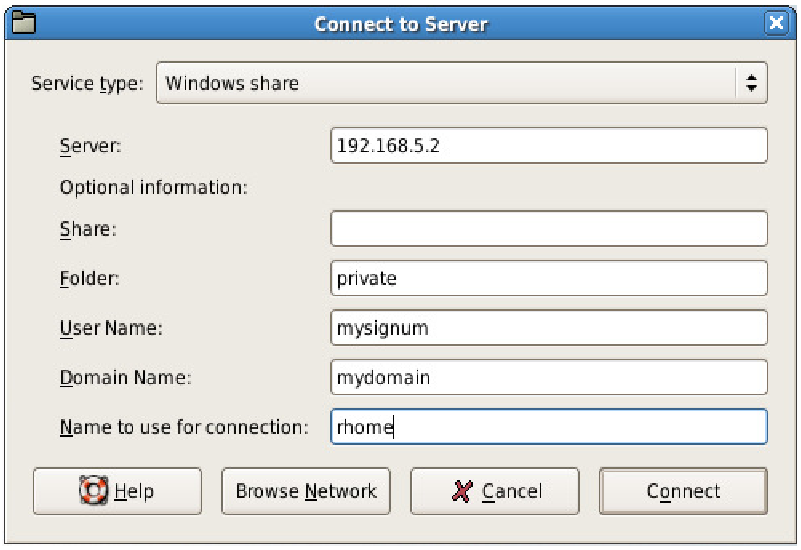
Trash

- [✓] Ask before emptying the Trash or deleting files
- [ ] Include a Delete command that bypasses Trash

Figure 24 File Management Preferences

9.4.2 Connecting to a Network Drive

Whether you want to connect to a network drive or a Windows share the steps to take will be the same.



Service type:	Windows share
Server:	192.168.5.2
Optional information:	
Share:	
Folder:	private
User Name:	mysignum
Domain Name:	mydomain
Name to use for connection:	rhome

Figure 25 Connect to Server Dialog

1. Start the File Browser application
2. Go to the File menu and select Connect to Server

   File → Connect to Server

3. In the Connect to Server window that opened, select Windows share in the Service type field.
4. Enter the name or the IP address of the computer you are connecting to in the Server field.
5. If you are connecting to a Windows share you need to specify the name of that share in the Share field, otherwise you can leave this field empty.
6. A folder on the server as well as login information and an optional name to use for the connection can also be specified.
7. Click the Connect button to connect to the network drive.
8. If successful a shortcut to the drive will be available on the desktop, see Figure 26.

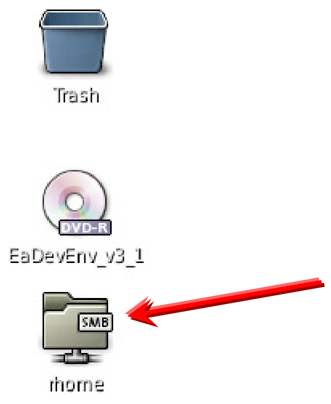


Figure 26 Shortcut to a Network Share

9.4.3 Setup a Network File System (NFS)

In Linux it is possible to mount remote file systems as if they where local file systems by using the Network File System protocol (NFS). The file access will be transparent to the user since files are accessed as if they were on the local computer.

This section describes how you export a part of your local file system to be accessible by other users and computers on the network, for example, from an embedded system running uClinux as the LPC2478 OEM Board.

1. Open the `/etc/exports` file in a text editor

```
$ sudo gedit /etc/exports
```

2. Enter “user” as the password if you are asked for it.
3. The opened file contains an access control list for the file systems that are exported. Make sure the following line is present in the file. The meaning of the values is described below.

```
/home/user/     192.168.0.0/255.255.0.0(ro,sync,all_squash,
no_subtree_check,anonuid=1000,anongid=100)
```

4. Save the file and make sure the changes are exported by running the following command.

```
$ sudo exportfs -ra
```

5. If asked for a password enter “user”.
6. It is now possible to connect to the exported file system from a different computer and mount the directory as a local file system. See section 6.9.1 for how this can be done in µClinux.

The different values in the exports file are explained below.

Part	Description
`/home/user`	The directory that is exported.
`192.168.0.0/255.255.0.0`	The subnet that will have access to the file system, i.e., any computer with IP address 192.168.x.x.
`ro`	Read only. We don’t allow the file system to be modified. If modification should be allowed exchange the `ro` for `rw`.
`sync`	The file system reply to requests when the changes have been committed to the file system.
`all_squash`	All user ids and group ids are mapped to the anonymous user.
`anonuid=1000`	Anonymous user ID is 1000
`anongid=1000`	Anonymous group ID is 1000

9.4.4 MMC/SD Card

This section describes how you can transfer files between the Debian distribution and an MMC/SD card and particularly when using a USB based memory card reader.

If the card you are using hasn’t been formatted before, this is the first thing to do before copying files to the card.

1. Install the dosfs tools.

```
$ sudo apt-get install dosfstools
```

2. If asked for a password enter “user”.
3. Attach the memory card reader to your computer and insert the memory card.
4. Make sure that the virtual machine has access to the memory card by looking at the VMware Player toolbar, see section 8.3 for an explanation and Figure 27 of what it could look like.
5. Run the dmesg command to see which device name is given to the memory card.

```
$ sudo dmesg



usb 1-1: new full speed USB device using uhci_hcd and address 2
usb 1-1: configuration #1 chosen from 1 choice
usb 1-2: new full speed USB device using uhci_hcd and address 3
usb 1-2: configuration #1 chosen from 1 choice
hub 1-2:1.0: USB hub found
hub 1-2:1.0: 7 ports detected
```

```
Initializing USB Mass Storage driver...
scsi1 : SCSI emulation for USB Mass Storage devices
usbcore: registered new driver usb-storage
USB Mass Storage support registered.
usb-storage: device found at 2
usb-storage: waiting for device to settle before scanning
  Vendor: Generic   Model: STORAGE DEVICE    Rev: 9335
  Type:   Direct-Access                      ANSI SCSI revision:
00
SCSI device sdb: 1984000 512-byte hdwr sectors (1016 MB)
sdb: Write Protect is off
sdb: Mode Sense: 03 00 00 00
sdb: assuming drive cache: write through
SCSI device sdb: 1984000 512-byte hdwr sectors (1016 MB)
sdb: Write Protect is off
sdb: Mode Sense: 03 00 00 00
sdb: assuming drive cache: write through
 sdb: sdb1
sd 1:0:0:0: Attached scsi removable disk sdb
usb-storage: device scan complete
```

6. As can be seen in this log the usb-storage device is given the name sdb (SCSI device **sdb**…) and will be mapped to the device file **`/dev/sdb1`** (sdb: sdb1).
7. If the card was already formatted it will automatically be mounted by Debian. If you still want to format the card you must first unmount it by issuing the following command (replace the device file name with the one shown in your log).

```
$ sudo umount /dev/sdb1
```

8. To format the card with a FAT32 file system issue the following command.

```
$ sudo mkfs.vfat /dev/sdb1
```

9. Now remove the card and insert it again and it will automatically be detected and mounted. You will see a shortcut on the desktop.
10. It can also be found mounted as `/media/usbdisk` in the file system.
11. It is now possible to transfer files between the card and Debian Linux.

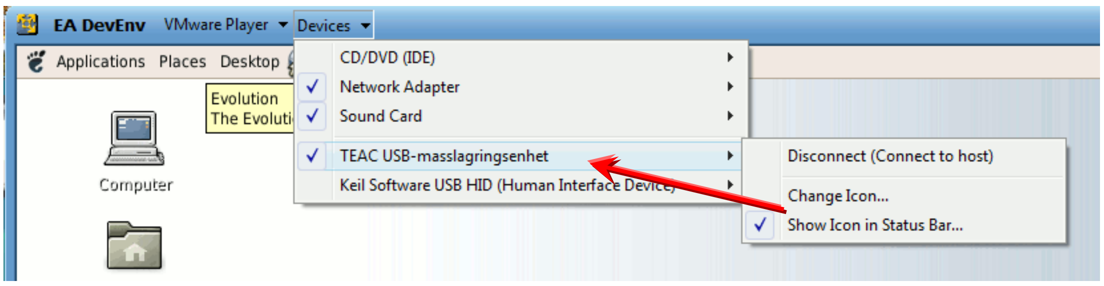


Figure 27 SanDisk Removable Disk Available in VMware Player

Before removing the card make sure to properly unmount the card first. Above an example was given where you run the umount command but it is also possible to right-click on the shortcut and selecting Unmount Volume as can be seen in Figure 28 below.

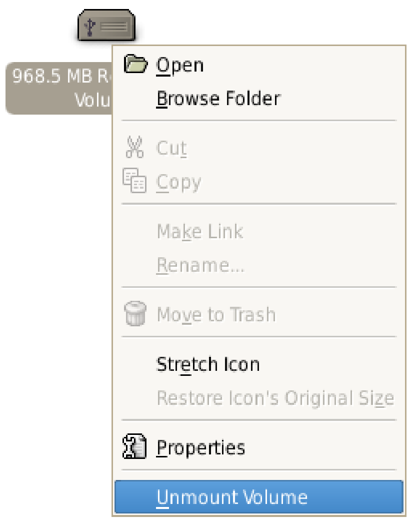


Figure 28 Unmount a Volume

9.4.5 USB Memory Stick

Working with a USB memory stick is similar to working with MMC/SD cards. The memory card will automatically be detected and mounted if it is attached when the virtual machine is active and in focus. As for the MMC/SD card you will get a shortcut on the Desktop and you can transfer files to and from the memory stick as if you were working with a local file system.

Before removing the memory stick make sure to unmount it first. You can do it in the same way as for the MMC/SD card, i.e., by right-clicking on the shortcut and then selecting Unmount Volume, see Figure 28.

9.5 Users

Section 7.3.1 gives a short introduction to users in a Linux system and also specifies which users and passwords that have been setup for the Embedded Artists Debian distribution. This section contains some guides of how to work with users.

9.5.1 Find out who is Logged On

By issuing the `whoami` command you will see which user you are currently logged in as. It is also possible to see all users currently logged on by issuing the `who` command.

```
$ whoami
user
$ who
user     :0           2009-05-23 14:07
user     pts/0        2009-05-23 14:12 (:0.0)
```

9.5.2 Add a User

This section describes different ways of adding a new user to the system.

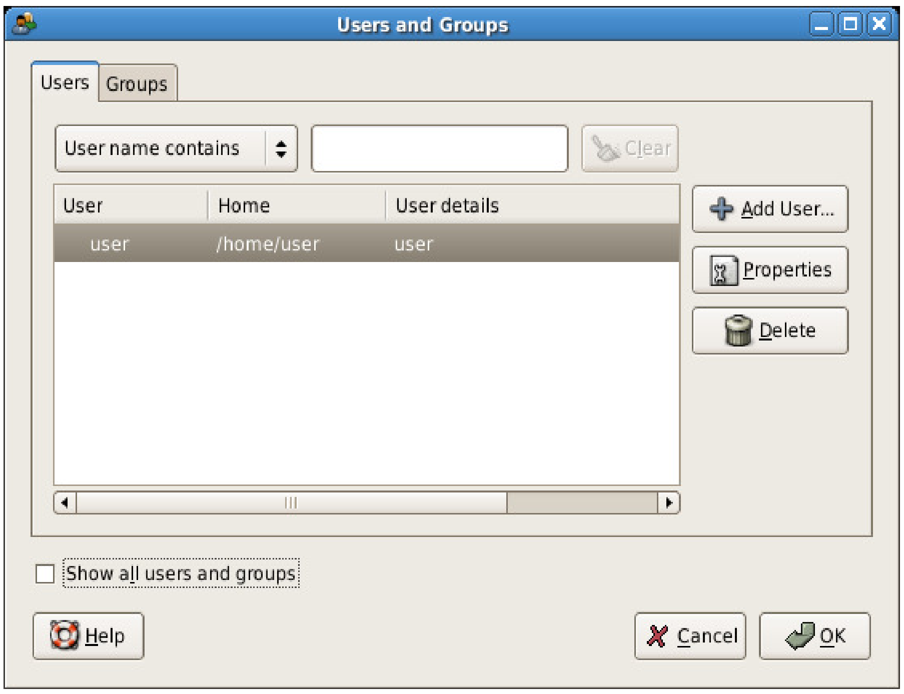
1. Start the Users administration tool from the Administration menu.

   Desktop → Administration → Users and Groups

2. When the application starts it will ask for the root password since managing users requires administrator privileges. Enter "root" as the password.
3. Click the "Add User" button, see Figure 25, to add a new user.
4. A window will appear where some details about the user must be entered. The most important details are the username and password. It is also possible to specify the real name of the user and some contact information if that is required, see Figure 30.
5. There are two more tabs in the window that allow more advanced editing of the user. For a normal user the default settings don't have to be changed.
6. Press the OK button when all necessary information has been provided and then also press the OK button on the tool window for the user to be added to the system.

The administration tool can also be started from the command-line by issuing the command below.

```
# users-admin
```



User	Home	User details
user	/home/user	user

Figure 29 Users administration tool

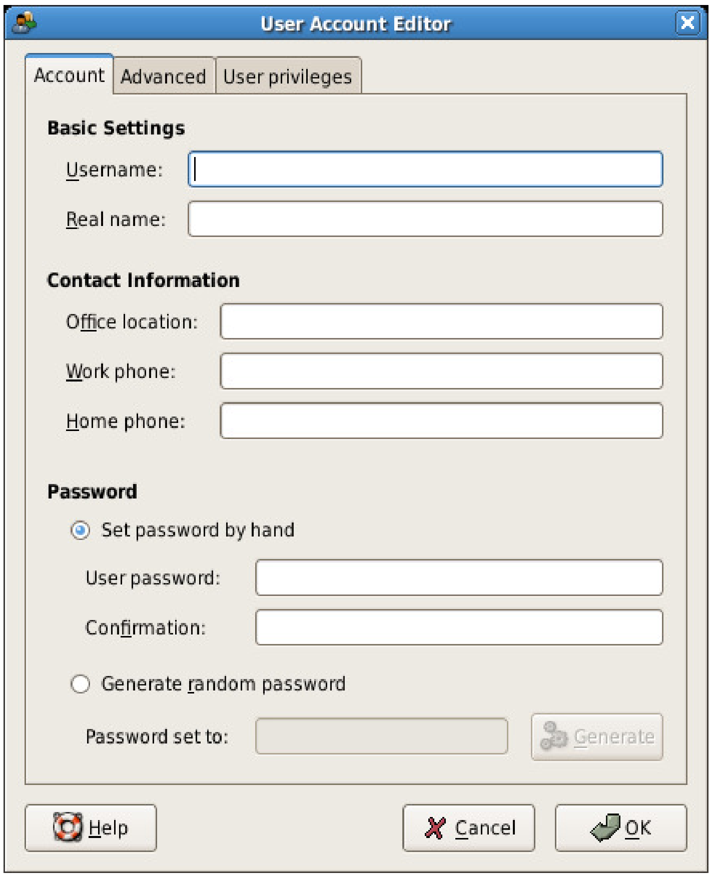


Figure 30 User Account Editor

It is also possible to add users from the command line by using the `adduser` command. In the example below the user "jdoe" will be added to the system. The `adduser` command will ask for password as well as some other information about the user. The only required information is the password.

1. Add the user.

```
$ sudo adduser jdoe
```

2. Enter "user" as password if asked for it (note this is the password for using the `adduser` command)
3. You will now be asked twice for the password to be used by user jdoe. Enter 123456 as password in this example.

```
Enter new UNIX password:
Retype new UNIX password:
```

4. You will be asked to enter some more information about jdoe, but can provide the default value by just pressing the ENTER key.
5. When all information has been provided you will be asked if the information is correct. Enter 'y' for the user to be added to the system.

```
Is the information correct? [y/N] y
```

9.5.3 Change Password

A user can change his/her own password by using the `passwd` command. This command will first ask for the current password and then twice for the new password.

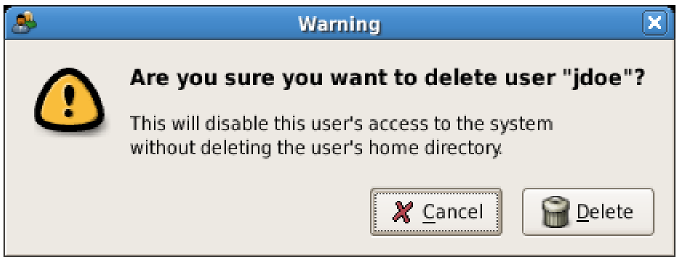
```
# passwd
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

It is also possible for the administrator to change a user's password by using the `passwd` command. In this case the user's username must be specified.

```
# sudo passwd jdoe
```

9.5.4 Deleting a User

From within the Users administration tool select the user to delete in the list of users and then press the "Delete" button. A window with a warning will appear, see Figure 31, asking if the user really should be deleted. Press the "Delete" button to remove the user and then press the "OK" button in the Users administration tool window to make the change take effect.



Are you sure you want to delete user "jdoe"?

This will disable this user's access to the system without deleting the user's home directory.

Figure 31 Warning when deleting a user

It is also possible to delete a user from the command line by using the `deluser` command. In the example below the user "jdoe" will be removed from the system. Please note that the user will be removed without a warning when issuing this command.

```
# sudo deluser jdoe
```

The user's home folder located in the `/home/` directory will not be removed when deleting a user irrespective of using the command line or the graphical tool. If the folder shall be removed this must be done manually, for example, by using the `rm` command.

```
# sudo rm -r /home/jdoe
```

9.5.5 Groups

During certain circumstances it is convenient to give several users the same access rights and privileges to the system, for example, the same access to a certain set of files. Linux has a built in mechanism for this called *groups*.

Groups can be added using the Users administration tool.

1. Start the Administration tool

Desktop → Administration → Users and Groups

2. Select the Groups tab and then press the “Add Group” button.
3. Enter a group name, for example, “students”.
4. Select the users to add to this group and click the “Add” button, see Figure 32.
5. Finish by clicking the “OK” button.

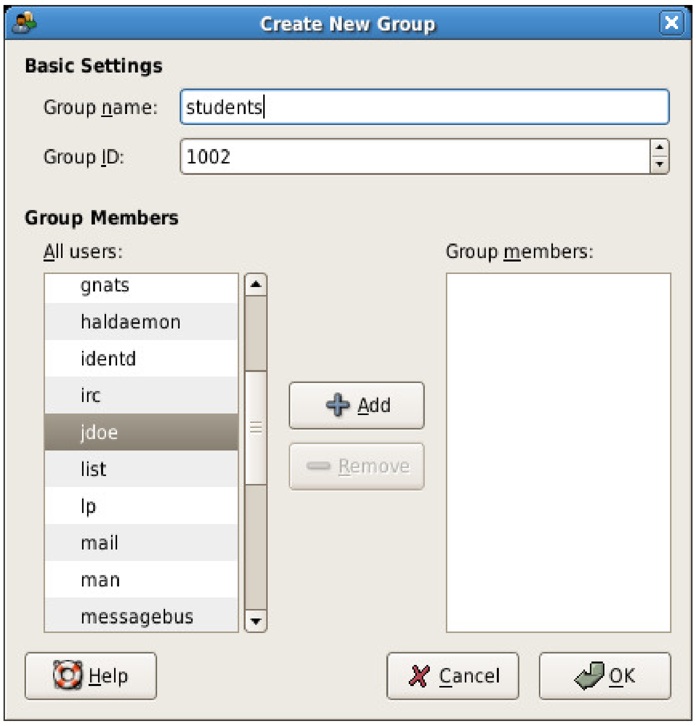


Figure 32 Creating a group

From the command line the `addgroup` command can be used. In the example below the group students are added to the system.

```
$ sudo addgroup students
```

Adding users to a group can also be handled from the command line. In this case the `adduser` command is used. In the example below the user jdoe is added to the students group.

```
$ sudo adduser jdoe students
```

Deleting groups are as simple as adding them. From the Users administration tool select the group to delete in the Groups tab and then press the “Delete” button. A warning will appear where the “Delete” button must be pressed to actually remove the group.

From the command line the command `delgroup` is used. In the example below the group students is removed.

```
$ sudo delgroup students
```

9.5.6 Sudo

The `sudo` command is used to run an application with administrator privileges. It is the configuration file `/etc/sudoers` that specifies which users and groups are allowed to use the `sudo` command. The tool `visudo` must be used when editing the `sudoers` file.

```
$ sudo visodo
```

Adding the following line to the `sudoers` file will allow the user jdoe to use `sudo`.

```
%jdoe   ALL=(ALL) ALL
```

Please consult the man page for sudoers for more information about how to write a sudoers file.

```
$ man sudoers
```

9.5.7 Changing Permissions of Files and Directories

If the permissions of a file need to be changed the `chmod` utility is the tool to turn to. The syntax of the tool is described below.

```
        chmod class operator permissions file
```

Some of the most commons options to the `chmod` tool are described in these tables.

Class	Description
a	The change will affect all users, i.e., owner, group and other.
u	The change will only affect the Owner (user) permissions
g	The change will affect the Group permissions
o	The change will affect the Other permissions

Operator	Description
+	Permissions will be added
-	Permissions will be removed

Permissions	Description
r	Read permission
w	Write permission
x	Execute permission

In the example below the group `students` will no longer have write permissions to the `report.txt` file.

```
$ chmod g-w report.txt
```

In this example both the group and others will be given read permissions to the `diary.txt` file.

```
$ chmod go+r diary.txt
```

The `diary.txt` file could also be made readable by everyone by using the “all” class as in the example below.

```
$ chmod a+r diary.txt
```

9.5.8 Changing Group and Owner Settings

One part remains about file permission and that is the owner and group settings of a file. To change these settings the `chown` tool is used. The syntax for the tool is given below.

`chown` *`[owner][:group] file`*

Option	Description
owner	This is an optional parameter specifying the owner of the file.
group	This is an optional parameter specifying the group the file should be associated with. Note that the group must be prefixed with a colon.
file	The file to change the owner and/or group for.

The owner of the file `report.txt` is changed in the example below.

```
$ chown mike report.txt
```

In the example below the group of the file `diary.txt` is changed from “joe” to “family”.

```
$ chown :family diary.txt
```

Both the owner and the group are changed for the file `report.txt` in the example below.

```
$ chown andy:teachers report.txt
```

9.6 Package Management

Most Linux distributions will offer some kind of package management system that will allow you to easily install, upgrade, configure and remove software packages, i.e., applications, services or libraries from your system.

The package manager used in Debian is `dpkg` and manages .deb formatted archives. The deb format keeps track of dependencies and files to make sure the system on which it is installed will be kept consistent. The dpkg tool is a low-level interface to the Debian package manager and high-level tools more easy to use exist.

9.6.1 Advanced Package Tool (APT)

One of the more common frontends to `dpkg` is the Advanced Package Tool (APT) suite of programs and more specifically the `apt-get` command. Listed below are the commands you will most likely run when using apt-get. The first command will update the apt cache with available packages (note that you need a network connection for this to function).

The second command is run when you want to install a software package. In section 9.4.4 the `dosfstools` are installed using this command.

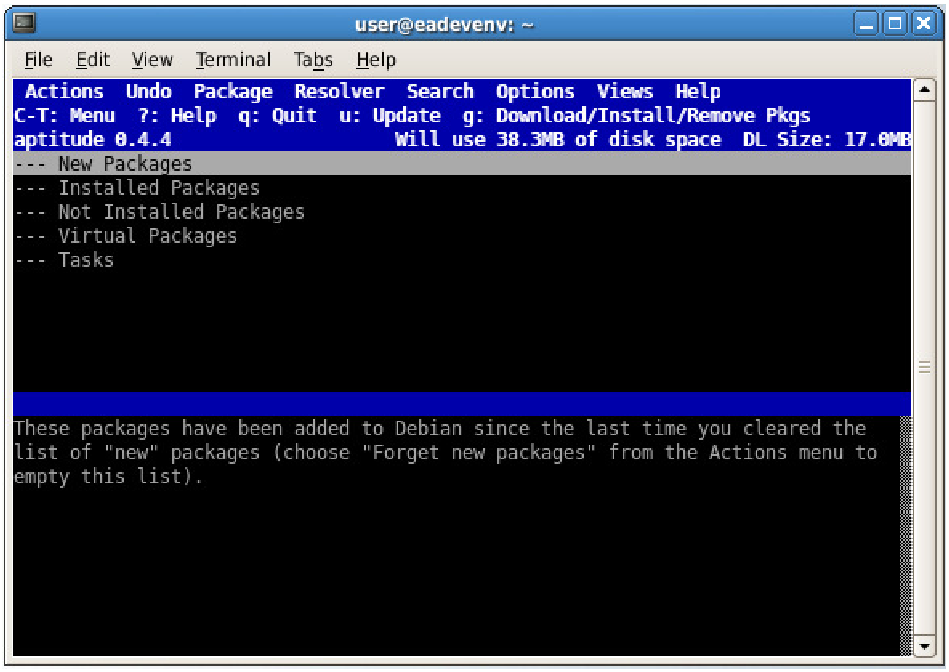
The third command could be issued if you want to upgrade the software packages already installed on your system.

```
# sudo apt-get update

# sudo apt-get install <program to install>

# sudo apt-get upgrade
```

If installation of a package fails run the `apt-get update` command and then try to install it again.

9.6.2 Aptitude

Aptitude is a frontend to APT that will allow the user to interactively choose packages to install or remove from the system. Aptitude is using the ncurses computer terminal library to give the user a simple graphical frontend to APT, see Figure 33. A command line interface similar to the one of apt-get also exists for those who prefer that kind of interface.

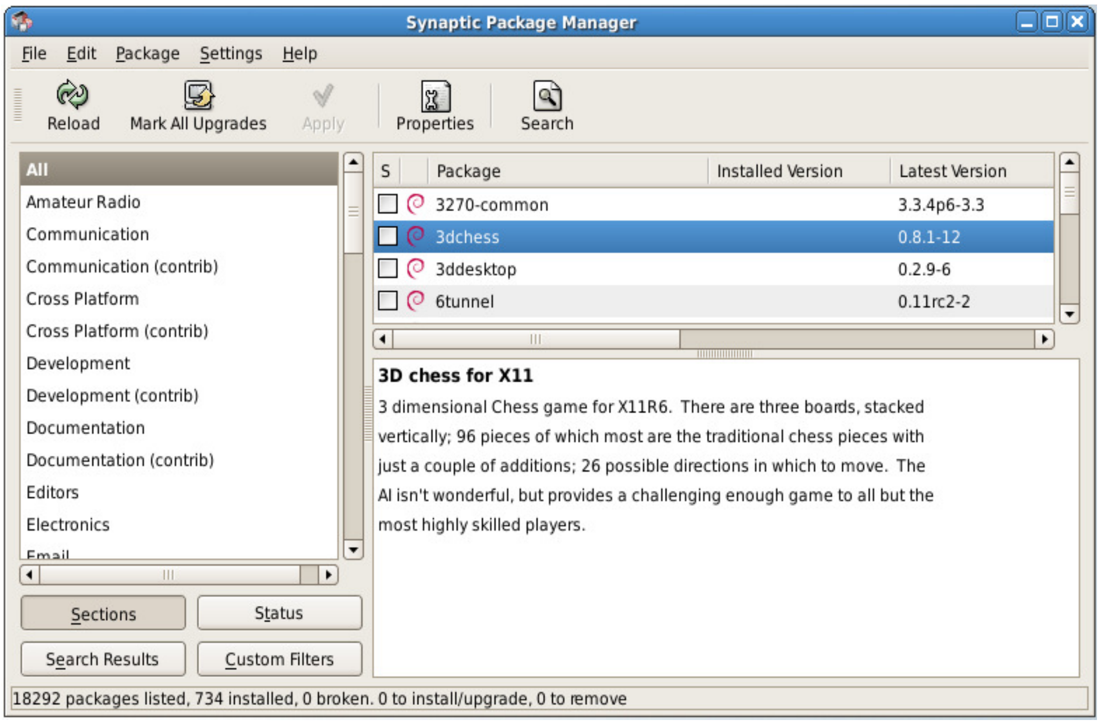


```
 Actions  Undo  Package  Resolver  Search  Options  Views  Help
C-T: Menu  ?: Help  q: Quit  u: Update  g: Download/Install/Remove Pkgs
aptitude 0.4.4                 Will use 38.3MB of disk space  DL Size: 17.0MB
--- New Packages
--- Installed Packages
--- Not Installed Packages
--- Virtual Packages
--- Tasks

These packages have been added to Debian since the last time you cleared the
list of "new" packages (choose "Forget new packages" from the Actions menu to
empty this list).
```

Figure 33 Aptitude user interface

9.6.3 Synaptic

Synaptic is a graphical frontend to APT using the GTK+ library to create the user interface. Because of its graphical user interface it is easy to use by non-experienced users, it’s a matter of point and click. It is easy to search, sort, change repositories and get information about the available software packages by using Synaptic. Figure 34 below is a snapshot of what Synaptic looks like.



S	Package	Installed Version	Latest Version
☐	3270-common		3.3.4p6-3.3
☐	3dchess		0.8.1-12
☐	3ddesktop		0.2.9-6
☐	6tunnel		0.11rc2-2

3D chess for X11

3 dimensional Chess game for X11R6. There are three boards, stacked vertically; 96 pieces of which most are the traditional chess pieces with just a couple of additions; 26 possible directions in which to move. The AI isn't wonderful, but provides a challenging enough game to all but the most highly skilled players.

18292 packages listed, 734 installed, 0 broken. 0 to install/upgrade, 0 to remove

Figure 34 Synaptic Package Manager

Remember that the package management tools must be run with root privileges, i.e., either as the root user or by using sudo.

9.7 Working with Archives

When a selection of files should be distributed between, for example, computers on a local or global network it is more efficient to distribute these files as one package instead of distributing them as separate files. Putting a number of files into a package is known as creating an archive. For Windows users the most common archive format is the zip format, which also compresses the archive, but for Linux users a more common format is the tar format.

Basically all Linux distributions have support for the `tar` command and it is also fairly easy to use. The example below show how all the files in the sources directory are put into an archive named sources.tar.

```
$ tar -cvf sources.tar sources
```

In the example below it is shown how to create an archive of all the files in the current directory with a specific file extension.

```
$ tar -cvf headers.tar *.h
```

If an archive has been received and the content needs to be extracted the following options to the `tar` command can be used.

```
$ tar -xvf headers.tar
```

To be even more efficient and save some bandwidth and storage space when distributing the archive it should be compressed. As mentioned previously in this section the zip format used on Windows PCs both archives and compresses files. On Linux the gzip or bzip2 commands would typically be used to compress files.

In the example below it is shown how to compress a file using gzip. The result of the command would be a file named `headers.tar.gz`.

```
$ gzip headers.tar
```

To unpack the compressed file the gunzip command would be used. This is illustrated in the example below. The result of running the command would be a file named `headers.tar`.

```
$ gunzip headers.tar.gz
```

In the example below it is shown how to compress a file using the bzip2 command. The result of the command would be a file named `headers.tar.bz2`

```
$ bzip2 headers.tar
```

To unpack the compressed file the bunzip2 command can be used. This is illustrated in the example below. The result of running the command would be a file named `headers.tar`.

```
$ bunzip2 headers.tar.bz2
```

To make it even simpler to work with archives and compression the `tar` command supports options that filter the archive through either `gzip` or `bzip2`. In the example below a compressed archive in gzip format is first created and then uncompressed and extracted. The “z” option is used to filter the archive through gzip.

```
$ tar -czvf sources.tar.gz sources
$ tar -xzvf source.tar.gz
```

In the example below a compressed archive in bzip2 format is first created and then uncompressed and extracted. The “j” option is used to filter the archive through bzip2.

```
$ tar -cjvf sources.tar.bz2 sources
$ tar -xjvf sources.tar.bz2
```

For more information about the above described commands and their options use the `man` command, for example, `man tar` to see the manual for the `tar` command.

9.8 Working with Patches

Software developers often end up in a situation where they need to find out the difference between two versions of the software being developed. The differences could have been made by different developers or by the same developer but during long periods of time. A tool that can easily and automatically show the differences is therefore of good use.

The reasons for having differences in the software could be that a bug has been corrected or new functionality has been added. A tool that can easily merge these updates to old versions of the software is also convenient to have. Both of the tools mentioned are available in most Linux distributions and are known as the `diff` and `patch` tools.

The `diff` tool is used to show the differences between two files or directories and the output from the tool can be in a number of different formats. The easiest way to show how it works is by an example. Below are excerpts from two versions of the same file. The first version contains a number of spelling errors that are corrected in the second version of the file.

Unmodified file

```
This is a sample text file with a
number of lines of text that will

be used as an axsample when showing
how the diff and patch tools
work In Linux.

On version of the file will contain
some errors that will be corrected
in the modified version of the file.
A diff between the files will then be
created using the diff tool.
```

Modified file

```
This is a sample text file with a
number of lines of text that will

be used as an example when showing
how the diff and patch tools
work in Linux.

One version of the file will contain
some errors that will be corrected
in the modified version of the file.
A diff between the files will then be
created using the diff tool.
```

The example below illustrates how the diff tool generates an output that shows the differences between the files.

```
$ diff file_unmodified.txt file_modified.txt > diff.txt
```

The result will be stored in the `diff.txt` file and have the content shown below. As can be seen the tool show the differences line by line and not word by word. The line being modified or replaced is prefixed with ‘<’ and the line replacing is prefixed with ‘>’.

```
4c4
< be used as an axsample when showing
---
> be used as an example when showing
6c6
< work In Linux.
---
> work in Linux.
8c8
< On version of the file will contain
---
> One version of the file will contain
```

A more common output format is the unified format which the diff tool can output given the –u option as can seen below.

```
$ diff -u file_unmodified.txt file_modified.txt > diff_unified.txt
```

The result of the above command can be seen below.

```
--- file.txt    2009-01-12 23:12:58.000000000 +0100
+++ file_updated.txt  2009-01-12 23:20:41.000000000 +0100
@@ -1,11 +1,11 @@
 This is a sample text file with a
 number of lines of text that will

-be used as an axsample when showing
+be used as an example when showing
 how the diff and patch tools
-work In Linux.
+work in Linux.

-On version of the file will contain
+One version of the file will contain
 some errors that will be corrected
 in the modified version of the file.

 A diff between the files will then be
```

As can be seen in the result above the name of the files are given as well as the last modified date of those files. The unmodified lines closest to the modified lines are also shown which makes it easier to understand the context of the changes.

The `diff` tool can also be used to show the difference between two directories. This is, for example, used when creating a patch for the changes made in µClinux. Below is an example of how a patch is created between two versions of µClinux.

```
$ diff -Naur uClinux-dist uClinux-dist_new > ea_uClinux-
081020.diff
```

The options given to the diff tool are explained below.

Option	Description
N	Treat new absent files as empty
a	Treat all files as text
u	Create unified output
r	Recursively compare any subdirectories found

Information about more options supported by the `diff` tool can be found by using the `man` tool.

When a diff file (also called a patch) has been created the changes described in that file can be merged into an old version of the file (or directory), i.e., the patch can be applied. Applying a patch is simple since it is just a matter of sending the diff file to the patch tool.

If the diff file is generated in the "normal" format the file to patch must be specified, see the example below.

```
$ patch file.txt < diff.txt
```

When the diff file is in unified format it is enough to change the directory to where the file to be patched is located. Remember that the unified format specifies the file name.

```
$ patch < diff_unified.txt
```

When applying a patch to entire directories it is important to specify the correct directory level using the `p` option. Setting a directory level means that the patch tool will take the path names given in the diff file into consideration when looking for files to patch. The patch created for the µClinux distribution should, for example, use the option –p1, i.e., removing the first part of the path (`uClinux-dist/`) when applying the patch while being located in the `uClinux-dist` directory, see the example below.

```
$ cd uClinux-dist
$ patch -p1 < ea_uClinux-081020.diff
```

9.9 Setup a TFTP Server

Using the Trivial File Transfer Protocol (TFTP) to transfer files to your embedded system is very convenient during the development cycle of, for example, µClinux. It is also convenient to use when updating kernel images to be used by the boot loader when starting µClinux. This section describes how you setup a TFTP server in your Debian distribution.

1. Begin by installing the TFTP server

```
$ sudo apt-get install tftpd
```

2. Enter “user” as the password if you are asked for it.
3. Open the `/etc/inetd.conf` configuration file.

```
$ sudo gedit /etc/inetd.conf
```

4. Add the following line (if it isn’t already present). Note that it should all be on one line. The last part specifies the TFTP server’s root directory, i.e., the directory where it will look for the files requested by TFTP clients.

```
tftp            dgram udp   wait   nobody      /usr/sbin/tcpd
    /usr/sbin/in.tftpd /home/user/
```

5. Restart the inetd service.

```
$ sudo killall -HUP inetd
```

9.10 The gedit Editor

When inspecting, creating or modifying text files a text editor is needed. Several different kind of editors exits for Linux and the choice of editor is often based on personal preference. The same person might use different editors for different purposes, such as one editor for modifying source code and another editor for writing a technical report. This section will describe the editor named gedit which is an editor simple to use, but still quite powerful.

Gedit is the official text editor for the GNOME desktop environment and is preinstalled in the Debian Etch distribution provided by Embedded Artists. This tool can be started from the Applications menu.

Applications → Accessories → Text Editor

It can also be started from the console. In the example below the file named `myfile.txt` will be opened in gedit.

```
$ gedit myfile.txt
```

From the file browser it is possible to right click on a file and select “Open with Text Editor” in the menu. Gedit will then be opened with the selected file since gedit is the default text editor in the Debian Etch distribution.

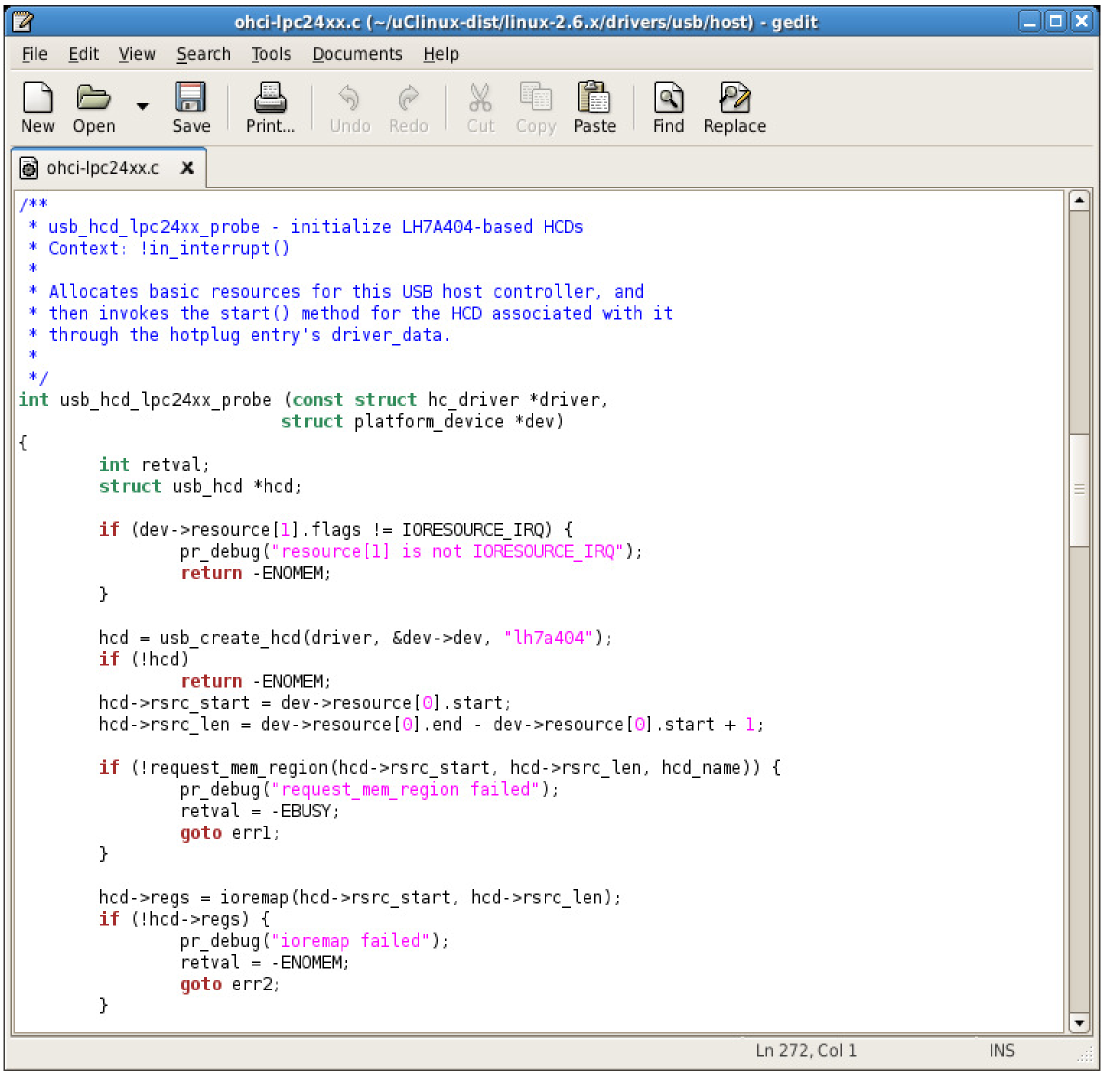
9.10.1 Syntax Highlighting

Gedit recognizes many programming languages and will highlight the keywords in those languages when a file is opened in gedit. This makes it easier to read the code in the file. In Figure 35 below a C file is opened in gedit and comments, keywords and strings can be easily distinguishable from, for example, function and variable names.

The colors used to highlight keywords in a programming language can be changed by opening the Preferences dialog and selecting the Syntax Highlighting tab.

Edit → Preferences → Syntax Highlighting

The language to change highlighting for is selected in the drop-down menu called Highlight mode and then the specific element, e.g. comment, string, or type, to change is selected. The foreground color, background color and font format (bold, italic …) can be changed.


ohci-lpc24xx.c (~/uClinux-dist/linux-2.6.x/drivers/usb/host) - gedit
File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut Copy Paste Find Replace
ohci-lpc24xx.c

```
/**
 * usb_hcd_lpc24xx_probe - initialize LH7A404-based HCDs
 * Context: !in_interrupt()
 *
 * Allocates basic resources for this USB host controller, and
 * then invokes the start() method for the HCD associated with it
 * through the hotplug entry's driver_data.
 *
 */
int usb_hcd_lpc24xx_probe (const struct hc_driver *driver,
                          struct platform_device *dev)
{
        int retval;
        struct usb_hcd *hcd;

        if (dev->resource[1].flags != IORESOURCE_IRQ) {
                pr_debug("resource[1] is not IORESOURCE_IRQ");
                return -ENOMEM;
        }

        hcd = usb_create_hcd(driver, &dev->dev, "lh7a404");
        if (!hcd)
                return -ENOMEM;
        hcd->rsrc_start = dev->resource[0].start;
        hcd->rsrc_len = dev->resource[0].end - dev->resource[0].start + 1;

        if (!request_mem_region(hcd->rsrc_start, hcd->rsrc_len, hcd_name)) {
                pr_debug("request_mem_region failed");
                retval = -EBUSY;
                goto err1;
        }

        hcd->regs = ioremap(hcd->rsrc_start, hcd->rsrc_len);
        if (!hcd->regs) {
                pr_debug("ioremap failed");
                retval = -ENOMEM;
                goto err2;
        }
```

Figure 35 C file opened in gedit

9.10.2 Indentation

When writing code some coding guidelines must often be followed. One of these guidelines is usually how the code should be indented, for example, if tabs or spaces should be used and how many of those to use for each indentation level. Gedit can be configured to follow these rules, i.e., it is possible to configure if tabs or spaces should be used and how many to use for each indentation level. It is also possible to enable automatic indentation which means that the next line will start at the same indentation level as the current line.

Indentation settings are configured on the Editor tab in the Preferences dialog.

Edit → Preferences → Editor

9.10.3 Spell Checking

It is possible to check the spelling of the text and code that is being written in gedit. This is done by using the key F7 or from the Tools menu.

Tools → Check Spelling

It is also possible to enable an automatic check of the spelling which means that all words considered to be incorrectly spelled will be underlined with a red wave formed line.

Before spell checking can be activated a dictionary for the language being used must be installed. A spell checker that gedit supports is the Aspell spell checker. Dictionaries to this spell checker can be installed using the apt-get tool. In the example below English and Swedish dictionaries are installed.

```
# sudo apt-get install aspell-en
# sudo apt-get install aspell-sv
```

When the dictionaries have been installed gedit must be restarted in order to use the dictionaries. The language to use during spell checking is selected from the Tools menu.

Tools → Set Language

9.10.4 Plugins

Gedit has a plugin system which can be used to dynamically add new extensions and features to gedit. The spell checker described in the previous section is, for example, a plugin. Other available plugins are.

- **Sort** – Sort selected text lines in ascending or descending order.
- **Indent lines** – indent block of text
- **Change Case** – Change the case of selected text
- **Snippets** – insert often used pieces of text in a fast way. The text is inserted by only writing a trigger word and then pressing tab or CTRL+Space. If a do-while statement should be written all that is needed is to write “do” and then press tab and the rest will be automatically inserted.

9.10.5 Alternative Editors

Other examples of popular editors in Linux are given below. An extensive description of these editors is, however, out of scope for this book.

- **Emacs** – powerful and extensible with probably the largest command set among all editors. Many extensions are available allowing the editor to be used as a complete IDE (Integrated Development Environment), web browser, e-mail client, diff tool and much more.
- **Vim** – acronym for Vi IMproved, i.e., an improved version of the vi editor which more or less has been a de-facto editor in Unix systems. As for Emacs the Vim editor is considered to be a powerful tool for software development and many times used as an entire IDE. It is also extensible with a lot of available plugins.

9.10.6 Shutting Down

To properly shut down or restart Linux go to the Desktop menu.

Desktop → Shut Down

You will then get the alternative to Restart or Shut Down the system. When Linux has been shutdown the virtual machine will automatically be closed as well.

You can also shut down Linux from the command line, i.e., from the shell by issuing the shutdown command.

```
$ shutdown -h now
```

10 Guides – U-boot

10.1 Build the U-boot

This section describes how to unpack a clean u-boot installation, apply patches, build the u-boot and finally how to download the compiled u-boot to the target. The u-boot source is available on the Embedded Artists resource DVD or can be downloaded from ref [30]. The patches can be downloaded from Embedded Artists support site.

1. Rename existing installation of the u-boot

```
$ cd /home/user
$ mv u-boot-1.1.6 old_u-boot-1.1.6
```

2. Unpack the source code. Make sure the Embedded Artists resource DVD is inserted in your DVD player. An alternative is to download the source code from the official u-boot website, see ref [30] .

```
$ tar -xjvf /cdrom/extra/u-boot-1.1.6.tar.bz2
```

3. Download the latest u-boot patch from the Embedded Artists support site. Start the web browser.

   Applications → Internet → Epiphany Web Browser

4. Login to the Embedded Artists support site and go to the patches section for µClinux.
5. Download the u-boot patch by right-clicking on the link and select “Save Link As”. Make sure the folder to use is `/home/user` and then click Save.
6. There might also be an incremental patch available (the name contains incrX in it where X is a number). Download that file as well.
7. Change directory to the u-boot directory.

```
$ cd u-boot-1.1.6
```

8. Apply the u-boot patch

```
$ gunzip -c ../u-boot-1.1.6-ea_v1_9_1.diff.gz | patch -p1
```

9. Apply incremental patches that are available in the correct order (in this example there is only one incremental patch).

```
$ gunzip -c ../u-boot-1.1.6-ea_v1_9_1_incr1.diff.gz | patch -p1
```

10. Select board configuration. Several exist and section 4.3.1 describes how you can find out which are available. In this example we will select the configuration for the LPC2478 OEM Board with 32-bit data bus.

```
$ make LPC2478OEM_Board_32bit_config
```

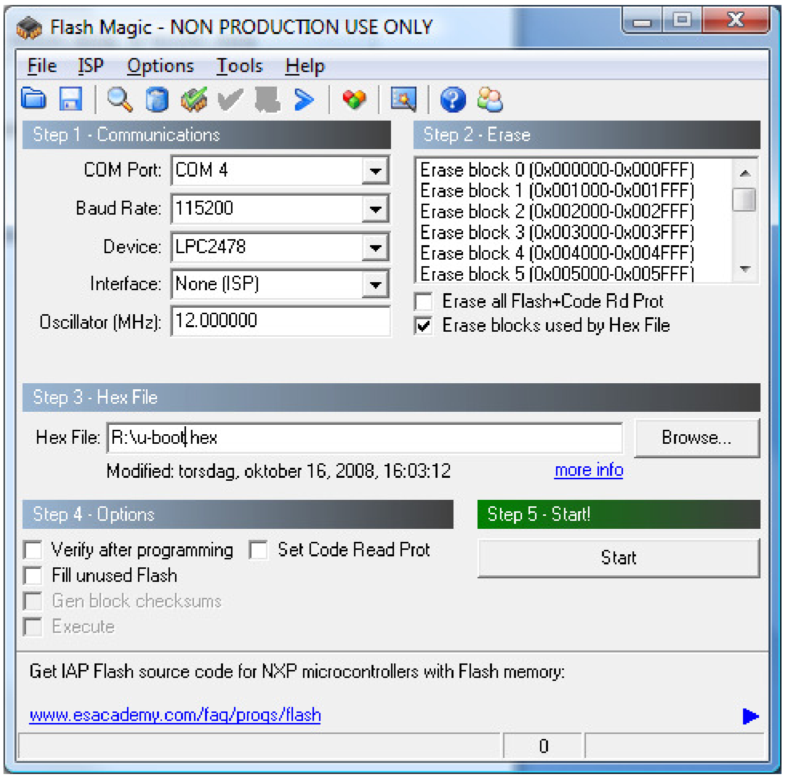
11. Now it is time to compile the u-boot.

```
$ make
```

12. When the build has finished successfully a binary file named `u-boot.bin` will be available in the `u-boot-1.1.6` directory.
13. Convert the bin file to a hex file.

```
$ arm-linux-objcopy -I binary -O ihex u-boot.bin u-boot.hex
```

14. Copy the hex file to a location where the tool FlashMagic can reach it (typically your PC), see section 8.4 for how to share data between the host OS and the virtual machine.
15. Start the FlashMagic tool (can be downloaded from ref [31]). See Figure 36 for what FlashMagic looks like when it is started.
16. Click the “Browse” button and browse to the location of the u-boot.hex file.
17. Change the COM port to the COM port used by your target. Select 115200 as baud rate.
18. Click the “Start” button to download the u-boot.bin file to the target.



COM Port: COM 4
Baud Rate: 115200
Device: LPC2478
Interface: None (ISP)
Oscillator (MHz): 12.000000

Erase block 0 (0x000000-0x000FFF)
Erase block 1 (0x001000-0x001FFF)
Erase block 2 (0x002000-0x002FFF)
Erase block 3 (0x003000-0x003FFF)
Erase block 4 (0x004000-0x004FFF)
Erase block 5 (0x005000-0x005FFF)
☐ Erase all Flash+Code Rd Prot
☑ Erase blocks used by Hex File

Hex File: R:\u-boot.hex
Modified: torsdag, oktober 16, 2008, 16:03:12

☐ Verify after programming ☐ Set Code Read Prot
☐ Fill unused Flash
☐ Gen block checksums
☐ Execute

Get IAP Flash source code for NXP microcontrollers with Flash memory:
www.esacademy.com/faq/progs/flash

Figure 36 FlashMagic

More information about how to program your board can be found in the User’s manual for your board.

10.2 Explore the U-boot Environment

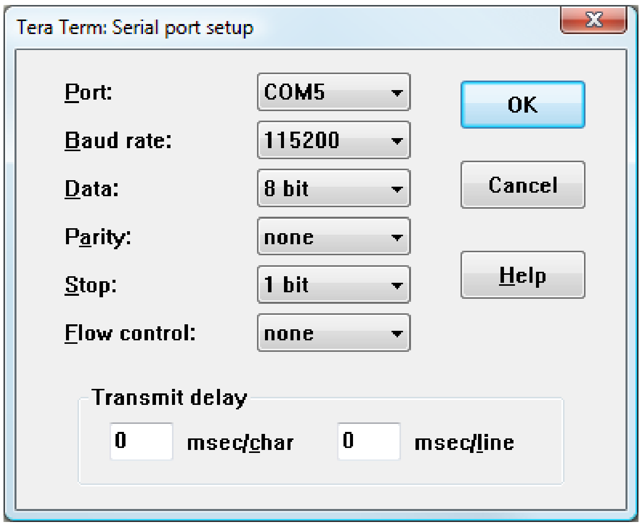
This section explains how you connect to the board with a terminal application and then explore the u-boot console and environment. You can use a terminal application (note that this is not the same application as in Debian Linux) of your choice. In this example we will use an application known as Tera Term, see ref [32].

10.2.1 Connect a Terminal to the Board

1. Start the Tera Term application and configure the serial port.

   Setup → Serial port

2. Configure the port to use a baud rate of **115200**, 8-bit data, no parity, 1 stop bit and no flow control, see Figure 37. Also choose the COM port that is associated with your development board.



Tera Term: Serial port setup

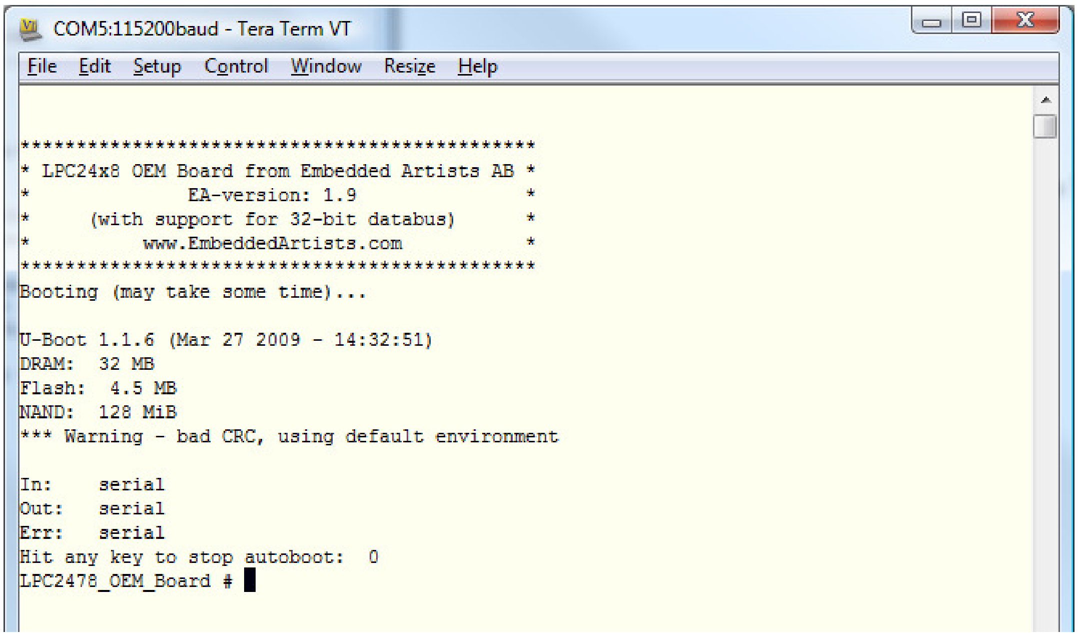
Port:	COM5
Baud rate:	115200
Data:	8 bit
Parity:	none
Stop:	1 bit
Flow control:	none

Transmit delay: 0 msec/char 0 msec/line

Figure 37 Serial port setup in TeraTerm

3. Click the “OK” button and Tera Term will connect to the board. If you have selected the wrong COM port or if it is already in use you will get an error message telling you that Tera Term cannot open the COM port.

4. Press the Reset button on the base board to make sure the board restarts. You should now see output from the u-boot in the terminal, see Figure 38. Make sure to hit any key to stop the auto boot procedure.


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Resize Help

```
*********************************************
* LPC24x8 OEM Board from Embedded Artists AB *
*             EA-version: 1.9                *
*     (with support for 32-bit databus)      *
*          www.EmbeddedArtists.com           *
*********************************************
Booting (may take some time)...

U-Boot 1.1.6 (Mar 27 2009 - 14:32:51)
DRAM:  32 MB
Flash:  4.5 MB
NAND:  128 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
LPC2478_OEM_Board #
```

Figure 38 U-boot output in Tera Term

5. If you see a warning message about bad CRC this means that there isn't any environment saved in flash memory. Default values will be used instead.

10.2.2 Basic Commands

Make sure to follow the instructions in the previous section so that the terminal is connected to the board and u-boot is running. This section will describe some basic commands you need to know about.

1. List all available commands by using the `help` command.

```
# help
?        - alias for 'help'
autoscr  - run script from memory
base     - print or set address offset
bdinfo   - print Board Info structure
boot     - boot default, i.e., run 'bootcmd'
bootd    - boot default, i.e., run 'bootcmd'
...
```

2. Get specific instructions about a command, the example shows help text for the `setenv` command.

```
# help setenv
setenv name value ...
    - set environment variable 'name' to 'value ...'
setenv name
    - delete environment variable 'name'
```

3. Print the current environment by using the `printenv` command.

```
# printenv
bootargs=root=/dev/ram initrd=0xa1800000,4000k console=ttyS0,115200N8
bootcmd=echo ;echo Booting from NAND FLASH (may take some seconds);echo
First loads 'uLinux.bin' and then 'jffs2.img';run nand_boot
```

```
bootdelay=3
baudrate=115200
tftp_boot=tftpboot a1500000 uLinux.bin;tftpboot a1800000 romfs.img;bootm
a1500000
nand_boot=nboot a1500000 0;bootm a1500000
nor_boot=bootm 80000000
...
```

4. The `printenv` command will list the u-boot environment, i.e., the variables that have been setup for the u-boot. Variables are typically used to store different booting options or update commands. The variables can also contain configuration information such as local IP address assigned to the board, IP address of a server where images can be downloaded from, and so on.
5. Add a variable to the environment by using the `setenv` command.

```
# setenv testvar myvalue
```

6. The `testvar` variable with value `myvalue` has now been added to the environment, but is not stored persistently, i.e., it will be lost in case of a power cycle. In order to save it persistently you need to use the `saveenv` command.

```
# saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...Erasing 1 sectors starting at sector 26.
This make take some time ... Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
```

7. Your changes have now been saved. If you would like to remove your variable use the `setenv` command by only specifying the variable name, i.e., without value.

```
# setenv testvar
```

8. Remember to use the `saveenv` command to save the change persistently.
9. One more important command is the `run` command. This command is used to execute commands stored in a variable. For now we will just get the help text for this command. Following sections will contain examples of its usage.

```
# help run
run var [...]
    - run the commands in the environment variable(s) 'var'
```

10.3 Network Related

This section describes network related functionality in the u-boot, for example, how to setup network addresses and use the TFTP protocol to transfer u-boot and kernel images to the development board.

10.3.1 Configuration of Addresses

The u-boot environment contains a number of variables that must be set before using the network functionality.

1. Set the Ethernet (MAC) address used by the u-boot. The Embedded Artists OEM Boards is delivered with a unique Ethernet address printed on a sticker. The address starts with `00:1A:F1`, which is Embedded Artists Organizationally Unique Identifier (OUI). Replace the last three 00 in the example with your values.

```
# setenv ethaddr 00:1a:f1:00:00:00
```

2. Set the IP address used by the u-boot. Please note that this must be an IP address that can be used on your network and it must be unique. Please ask you network administrator for an address to use.

```
# setenv ipaddr 192.168.5.233
```

3. Set the IP address of the computer or virtual machine where your TFTP server is running.

```
# setenv serverip 192.168.5.10
```

4. Now save these changes persistently.

```
# saveenv
```

10.3.2 Using `tftpboot` to update the u-boot

Section 4.5.4 describes the `tftpboot` command which allow downloading of files from a TFTP server. If you are modifying the u-boot it is very convenient to download those changes to the board using TFTP instead of having to use, for example, FlashMagic as described in section 10.1

1. Follow the instructions in section 9.9 in order to setup a TFTP server in Debian Linux.
2. Copy the u-boot image to the TFTP root. These instructions assume that you have the TFTP server's root as described in section 9.9 and that you have started a Terminal application in Debian. It also assumes that you have already built the u-boot as described in section 10.1

```
$ cd /home/user
$ cp u-boot-1.1.6/u-boot.bin .
```

3. Download the `u-boot.bin` file to external RAM at address `0xa1000000`.

```
# tftpboot a1000000 u-boot.bin
```

4. Disable write protection in the flash area where the u-boot will be stored.

```
# protect off 0 2ffff
```

5. Erase the flash area where the u-boot will be stored.

```
# erase 0 2ffff
```

6. Copy the u-boot from RAM at address `0xa1000000` to flash at address `0x00000000`. The variable `filesize` is automatically set by the `tftpboot` command to the size of the `u-boot.bin` file.

```
# cp.b a1000000 0 $(filesize)
```

7. Now restart the board and the u-boot will be loaded and started.

10.3.3 Using `tftpboot` to Boot µClinux with Root File System

Section 4.5.4 describes the `tftpboot` command which allow downloading of files from a TFTP server. During the development of the Linux kernel it is very convenient to be able to quickly download and test a change in the kernel. This can be achieved by using the TFTP boot command if your board is connected to a network.

1. Follow the instructions in section 9.9 in order to setup a TFTP server in Debian Linux.
2. Copy the kernel and file system images to the TFTP server's root directory.

```
$ cd /home/user
$ cp uClinux-dist/images/uLinux.bin .
$ cp uClinux-dist/images/romfs.img .
```

3. Setup the boot argument variable to use a root file system located in RAM at address `0xa1800000` and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits.

```
# setenv bootargs root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
# saveenv
```

4. Download the kernel image to RAM at address `0xa1500000`.

```
# tftpboot a1500000 uLinux.bin
```

5. Download the file system image to RAM at address `0xa1800000`.

```
# tftpboot a1800000 romfs.img
```

6. Use the `bootm` command to load the bootable image found at address `0xa1500000`.

```
# bootm a1500000
```

In order to download the Linux and file system images we had to call upon the `tftpboot` command twice and then call the `bootm` command to boot the kernel. A better way is to store these three commands in a variable and then use the `run` command to execute them sequentially.

1. Create an environment variable named `tftp_boot`. A semicolon (';') is used to separate the commands, but the semicolon must be escaped with a backslash ('\').

```
# setenv tftp_boot tftpboot a1500000 uLinux.bin\;tftpboot
a1800000 romfs.img\;bootm a1500000
# saveenv
```

2. You can now run all the three command by using the `run` command.

```
# run tftp_boot
```

10.3.4 Troubleshooting the `tftpboot` Command

If you don't get contact with the server when using the `tftpboot` command you will get the following output in the console.

```
TFTP from server 192.168.5.99; our IP address is 192.168.5.233
Filename 'u-boot.bin'.
Load address: 0xa1000000
Loading: T T T T T T T T T T T T T T T T T T T T
```

There could be several reasons for this behavior.

- You have entered the wrong IP address for the TFTP server.
    - Please check the `serverip` variable.
- The TFTP server isn't started.
- You have a (software) firewall that blocks the TFTP traffic from the development board.
    - Disable the firewall temporarily and try to run the `tftpboot` command again.
- The network functionality in VMware or Debian has stopped working.
    - See section 8.5.1 for information about how to solve this problem.

If the file you are trying to download isn't available in the server's root directory you will get the following output in the console.

```
TFTP from server 192.168.5.10; our IP address is 192.168.5.233
Filename 'u-boot.bin'.
Load address: 0xa1000000
Loading: *
TFTP error: 'No such file or directory' (0)
Starting again
```

Make sure to copy the file to the TFTP server's root directory. See section 9.9 for more information about how the TFTP server and its root directory is setup in Debian Linux.

10.4 FAT File Systems

Images can be loaded from the FAT file systems on different devices using the `fatload` command, see section 4.5.5 for more information about the `fatload` command.

10.4.1 USB Memory Stick

Almost all computers today have a USB connection and most operating systems support USB and have drivers for USB mass storage devices. This makes it quite simple to use a USB memory stick to transfer boot images from the development computer to the development board.

1. Attach a USB memory stick to your computer, for example, to the Debian Etch distribution running in VMware, see section 9.4.5 for more information.
2. Copy uLinux.bin and romfs.img from `uClinux-dist/images` to the root directory on the USB memory stick. In the example below it is assumed that the USB memory stick has been mounted onto `/media/usbdisk`.

```
$ cd /home/user
$ cp uClinux-dist/images/uLinux.bin /media/usbdisk/
$ cp uClinux-dist/images/romfs.img /media/usbdisk/
```

3. Unmount the USB memory stick, see section 9.4.5 for how to do this correctly, and then remove it from your computer.
4. Attach the USB memory stick to the development board.
5. Start the development board and make sure you enter into the u-boot console.
6. Setup the boot argument variable to use a root file system located in RAM at address `0xa1800000` and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits. Note that you don't have to do this if the `bootargs` has previously been setup.

```
# setenv bootargs root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
# saveenv
```

7. Initiate the USB interface. You should see output similar to what is illustrated below.

```
# usb start
(Re)start USB...
USB:   scanning bus for devices... 2 USB Device(s) found
       scanning bus for storage devices... 1 Storage Device(s) found
```

8. Load the `uLinux.bin` file using the `fatload` command from the USB memory stick to RAM at address `0xa1500000`.

```
# fatload usb 0 a1500000 uLinux.bin
```

9. Load the `romfs.img` file using the `fatload` command from the USB memory stick to RAM at address `0xa1800000`.

```
# fatload usb 0 a1800000 romfs.img
```

10. Stop the USB interface.

```
# usb stop
```

11. Use the `bootm` command to load the bootable image found at address `0xa1500000`.

```
# bootm a1500000
```

All of these commands can be put in one u-boot environment variable in order to execute them all at once using the `run` command.

1. Create an environment variable named `usb_boot`. A semicolon (';') is used to separate the commands, but the semicolon must be escaped with a backslash ('\').

```
# setenv usb_boot usb start\;fatload usb 0 a1500000
uLinux.bin\;fatload usb 0 a1800000 romfs.img\;usb stop\;bootm
a1500000
# saveenv
```

2. You can now run all commands by using the `run` command.

```
# run usb_boot
```

10.4.2 MMC/SD Card

As with USB interfaces it is really common to have a memory card reader attached to a computer today. A memory card can be used to transfer boot images from the development computer to the development board.

1. Insert an MMC/SD card in your memory card reader attached to your computer. Section 9.4.4 describes how to work with MMC/SD cards in the Debian Etch distribution.
2. Copy `uLinux.bin` and `romfs.img` from `uClinux-dist/images` to the root directory on the MMC/SD card. In the example below it is assumed that the card has been mounted onto `/media/usbdisk`.

```
$ cd /home/user
$ cp uClinux-dist/images/uLinux.bin /media/usbdisk/
$ cp uClinux-dist/images/romfs.img /media/usbdisk/
```

3. Unmount the MMC/SD card, see section 9.4.4 for how to do this correctly, and then remove it from your computer.
4. Insert the MMC/SD card in the MMC connector on the development board.
5. Start the development board and make sure you enter into the u-boot console.
6. Setup the boot argument variable to use a root file system located in RAM at address `0xa1800000` and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits. Note that you don't have to do this if the `bootargs` has previously been setup.

```
# setenv bootargs root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
# saveenv
```

7. Initiate the MMC interface. You should see an output similar to what is illustrated below.

```
# mmc
mmc_init

mci-cid (SD memory card):
  Manufacturer ID: 0x03 = SanDisk
  OEM/Application ID: SD
  Product Name: SU256
  Product Revision: 8.0
  Serial Number: 0x6013ae0f
  Date Code: 2006.5
  sector size = 512 (Bytes), card size = 253 (MBytes)
  dump csd data: 002600325f5983c4
                 eddacfff92404060
..
Return 0 after fat_register_device
```

8. Load the `uLinux.bin` file using the `fatload` command from the memory card to RAM at address `0xa1500000`.

```
# fatload mmc 0 a1500000 uLinux.bin
```

9. Load the romfs.img file using the `fatload` command from the memory card to RAM at address `0xa1800000`.

```
# fatload mmc 0 a1800000 romfs.img
```

10. Use the `bootm` command to load the bootable image found at address `0xa1500000`.

```
# bootm a1500000
```

All of these commands can be put in one u-boot environment variable in order to execute them all at once using the `run` command.

1. Create an environment variable named `mmc_boot`. A semicolon (';') is used to separate the commands, but the semicolon must be escaped with a backslash ('\').

```
# setenv mmc_boot mmc\;fatload mmc 0 a1500000 uLinux.bin\;fatload
mmc 0 a1800000 romfs.img\;bootm a1500000
# saveenv
```

2. You can now run all commands by using the `run` command.

```
# run mmc_boot
```

10.5 NOR Flash

10.5.1 Update NOR Flash via TFTP

This section describes how to update the NOR flash with the Linux kernel image and a compressed file system image by downloading them using the TFTP protocol.

1. Erase the NOR flash

```
# erase bank 2
```

2. Download the `uLinux.bin` file to RAM at address `0xa1500000`.

```
# tftpboot a1500000 uLinux.bin
```

3. Copy the downloaded image to NOR flash starting at address `0x80000000`. The `fileaddr` and `filesize` variables are set by the `tftpboot` command.

```
# cp.b $(fileaddr) 80000000 $(filesize)
```

4. Download the file system image to RAM at the address `0xa1500000`. A compressed file system image is downloaded in this example.

```
# tftpboot a1500000 cramfs.img
```

5. Copy the file system image to NOR flash starting at the address `0x80200000`.

```
# cp.b $(fileaddr) 80200000 $(filesize)
```

10.5.2 Update NOR Flash via USB

This section describes how to update the NOR flash with the Linux kernel image and a compressed file system image by loading them from a USB memory stick.

1. Copy the `uLinux.bin` and the `cramfs.img` file to a USB memory stick in a similar way as described in section 10.4.1
2. Attach the USB memory stick to the development board. Power up the board and enter into the u-boot console.
3. Erase the NOR flash

```
# erase bank 2
```

4. Initiate the USB interface.

```
# usb start
```

5. Load the `uLinux.bin` file from the USB memory stick.

```
# fatload usb 0 a1500000 uLinux.bin
```

6. Copy the image to NOR flash starting at the address 0x80000000.

```
# cp.b a1500000 80000000 200000
```

7. Load the compressed file system image to RAM at the address `0xa1500000`.

```
# fatload usb 0 a1500000 cramfs.img
```

8. Copy the file system image to NOR flash starting at the address 0x80200000.

```
# cp.b a1500000 80200000 200000
```

9. Stop the USB interface.

```
# usb stop
```

10.5.3 Update NOR Flash via MMC

This section describes how to update the NOR flash with the Linux kernel image and a compressed file system image by loading them from a memory card.

1. Copy the `uLinux.bin` and the `cramfs.img` file to a memory card in a similar way as described in section 10.4.2
2. Attach the memory card to the development board. Power up the board and enter into the u-boot console.
3. Erase the NOR flash

```
# erase bank 2
```

4. Initiate the MMC interface.

```
# mmc
```

5. Load the `uLinux.bin` file from the memory card.

```
# fatload mmc 0 a1500000 uLinux.bin
```

6. Copy the image to NOR flash starting at the address `0x80000000`.

```
# cp.b a1500000 80000000 200000
```

7. Load the compressed file system image to RAM at the address `0xa1500000`.

```
# fatload mmc 0 a1500000 cramfs.img
```

8. Copy the file system image to NOR flash starting at the address 0x80200000.

```
# cp.b a1500000 80200000 200000
```

10.5.4 Boot from NOR Flash with Images in RAM

In this example the kernel and file system images will first be copied to RAM before the kernel is started.

1. Setup the boot argument variable to use a root file system located in RAM at the address `0xa1800000` and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits.

```
# setenv bootargs root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
# saveenv
```

2. Copy the kernel image to RAM at the address `0xa1500000.`

```
# cp.b 80000000 a1500000 200000
```

3. Copy the file system image to RAM at the address `0xa1800000`.

```
# cp.b 80200000 a1800000 200000
```

4. Use the `bootm` command to load the bootable image found at the address `0xa1500000`.

```
# bootm a1500000
```

10.5.5 Boot from NOR Flash with Images in Flash

This section describes how to directly boot the kernel from NOR flash without first copying the kernel or file system images to RAM.

1. Setup the boot argument variable to use a root file system located in NOR flash and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits.

```
# setenv bootargs root=/dev/mtdblock3 console=ttyS0,115200N8
# saveenv
```

2. Use the `bootm` command to directly load the bootable image from NOR flash at address `0x80000000`.

```
# bootm 80000000
```

10.6 NAND Flash

In all of these guides we will be using a JFFS2 file system as the root file system. Section 4.5.9 has more information about JFFS2 file systems.

10.6.1 Update NAND Flash via TFTP

This section describes how to update the NAND flash with the Linux kernel image and a JFFS2 file system image by downloading them using the TFTP protocol.

1. Erase the NAND flash

```
# nand erase
```

2. Download the `uLinux.bin` file to RAM at the address `0xa1500000`.

```
# tftpboot a1500000 uLinux.bin
```

3. Copy the downloaded image to NAND flash at offset `0x0`. The `fileaddr` variable is set by the `tftpboot` command.

```
# nand write $(fileaddr) 0 0x00300000
```

4. Download the JFFS2 file system image to RAM at the address `0xa1500000`.

```
# tftpboot a1500000 jffs2.img
```

5. Copy the file system image to NAND flash starting at offset `0x00300000`.

```
# nand write $(fileaddr) 0x00300000 $(filesize)
```

10.6.2 Update NAND Flash via USB

This section describes how to update the NAND flash with the Linux kernel image and a JFFS2 file system image by loading them from a USB memory stick.

1. Copy the `uLinux.bin` and the `jffs2.img` file to a USB memory stick in a similar way as described in section 10.4.1
2. Attach the USB memory stick to the development board. Power up the board and enter into the u-boot console.
3. Erase the NAND flash

```
# nand erase
```

4. Initiate the USB interface.

```
# usb start
```

5. Load the `uLinux.bin` file from the USB memory stick.

```
# fatload usb 0 a1500000 uLinux.bin
```

6. Copy the image to NAND flash at offset `0x0`.

```
# nand write 0xa1500000 0 0x00300000
```

7. Load the JFFS2 file system image to RAM at the address `0xa1500000`.

```
# fatload usb 0 a1500000 jffs2.img
```

8. Copy the file system image to NAND flash at offset `0x00300000`.

```
# nand write 0xa1500000 0x00300000 0x00300000
```

9. Stop the USB interface.

```
# usb stop
```

10.6.3 Update NAND Flash via MMC

This section describes how to update the NAND flash with the Linux kernel image and a JFFS2 file system image by loading them from a memory card.

1. Copy the `uLinux.bin` and the `jffs2.img` file to a memory card in a similar way as described in section 10.4.2 10.4.1
2. Attach the memory card to the development board. Power up the board and enter into the u-boot console.
3. Erase the NAND flash

```
# nand erase
```

4. Initiate the MMC interface.

```
# mmc
```

5. Load the `uLinux.bin` file from the memory card.

```
# fatload mmc 0 a1500000 uLinux.bin
```

6. Copy the image to NAND flash at offset `0x0`.

```
# nand write 0xa1500000 0 0x00300000
```

7. Load the JFFS2 file system image to RAM at the address `0xa1500000`.

```
# fatload mmc 0 a1500000 jffs2.img
```

8. Copy the file system image to NAND flash at offset `0x00300000`.

```
# nand write 0xa1500000 0x00300000 0x00300000
```

10.6.4 Boot from NAND Flash Using a JFFS2 File System

This section describes how to directly boot the kernel from NAND flash with a JFFS2 file system also stored in NAND flash.

1. Setup the boot argument variable to use a root file system located in the NAND flash and the console located at device `ttyS0` with a baud rate of 115200, no parity and 8 data bits.

```
# setenv bootargs root=/dev/mtdblock1 console=ttyS0,115200N8
# saveenv
```

2. Load the kernel image from NAND flash at offset `0x0` to RAM at the address `0xa1500000`.

```
# nboot a1500000 0
```

3. Use the `bootm` command to load the bootable image found at the address `0xa1500000`.

```
# bootm a1500000
```

10.7 Boot Automatically

In all previous examples the auto boot procedure has been stopped in order to enter into the u-boot console. If the auto boot isn't stopped the command(s) found in the `bootcmd` variable will be run.

1. Set the bootcmd variable to boot from a USB memory stick. The variable (`usb_boot`) created in section 10.4.1 will be used.

```
# setenv bootcmd run usb_boot
# saveenv
```

The delay before the commands in the `bootcmd` variable are executed is by default set to 3 seconds. This can be changed by modifying the `bootdelay` variable.

1. Change the boot delay to 10 seconds.

```
# setenv bootdelay 10
# saveenv
```

11 Guides – µClinux

11.1 Build µClinux

This section describes how to unpack a clean µClinux distribution, apply patches, and build the kernel and file system images. The µClinux and kernel source is available on the Embedded Artists resource DVD or can be downloaded from ref [33] and ref [34]. The patches can be downloaded from Embedded Artists support site.

1. Rename existing installation of the µClinux if it exists

```
$ cd /home/user
$ mv uClinux-dist old_uClinux-dist
```

2. Unpack the µClinux distribution. Make sure the Embedded Artists resource DVD is inserted in your DVD player. An alternative is to download the source code from ref [33].

```
$ tar -xzvf /cdrom/extra/uClinux-dist-20070130.tar.gz
```

3. Remove the existing Linux kernel directories since a newer version of the kernel will be used.

```
$ cd uClinux-dist
$ rm -r linux-2.*
```

4. Unpack the kernel sources

```
$ tar -xzvf /cdrom/extra/linux-2.6.21.tar.gz
```

5. Rename the kernel directory

```
$ mv linux-2.6.21 linux-2.6.x
```

6. Download the latest µClinux patch from the Embedded Artists support site. Start the web browser.

   Applications → Internet → Epiphany Web Browser

7. Login to the Embedded Artists support site and go to the patches section for µClinux.
8. Download the patch by right-clicking on the link and select “Save Link As”. Make sure the folder to use is `/home/user` and then click Save. Also download any available incremental patches.
9. Apply the patch

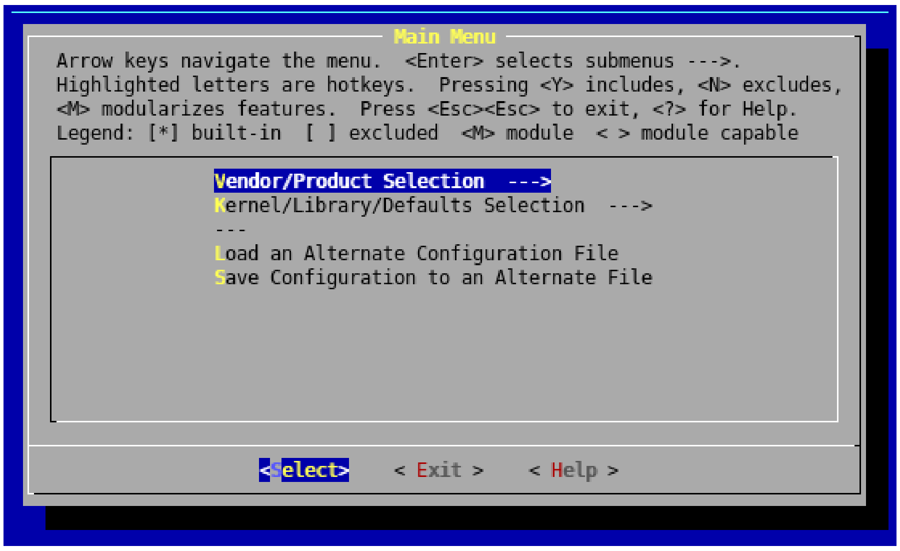
```
$ gunzip -c ../ea-uClinux-081029.diff.gz | patch -p1
```

10. Apply incremental patches. In this example only one is available.

```
$ gunzip -c ../ea-v3_1_incr1.diff.gz | patch -p1
```

11. Start the configuration tool.

```
$ make menuconfig
```



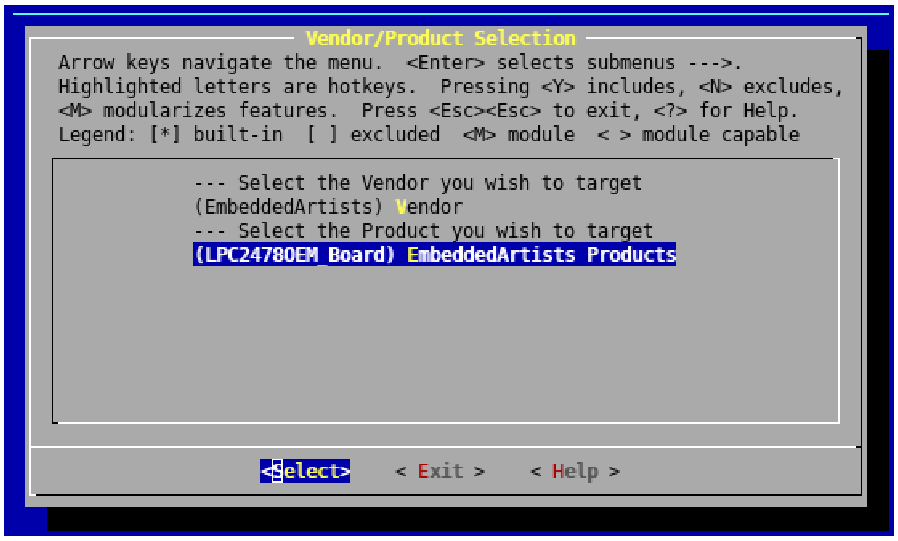
```
Main Menu
Arrow keys navigate the menu.  <Enter> selects submenus --->.
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
<M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                Vendor/Product Selection  --->
                Kernel/Library/Defaults Selection  --->
                ---
                Load an Alternate Configuration File
                Save Configuration to an Alternate File

                <Select>    < Exit >    < Help >
```

Figure 39 - Main menu in the configuration tool

12. Change vendor to Embedded Artists ("Select the Vendor you wish to target").

Vendor/Product Selection → *Vendor* → *EmbeddedArtists*

13. Select your board configuration ("Select the Product you wish to target").



```
Vendor/Product Selection
Arrow keys navigate the menu.  <Enter> selects submenus --->.
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
<M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

             --- Select the Vendor you wish to target
             (EmbeddedArtists) Vendor
             --- Select the Product you wish to target
             (LPC24780EM Board) EmbeddedArtists Products

                <Select>    < Exit >    < Help >
```

Figure 40 - Vendor/Product selection

14. Click Exit followed by Exit once more and then click the Yes button when asked to save the kernel configuration.

15. Now it is time to build µClinux

```
$ make
```

16. When the build has finished successfully the kernel and file system images will be available in the `uClinux-dist/images` directory.

17. Now follow any of the guides in chapter 10 to download the images to the target and boot µClinux.

11.2 Startup of Linux

When Linux is starting you will get an output similar to the example below in the console (note that parts of the output have been removed). The kernel initializes the system and then starts to load the built-in and enabled device drivers.

```
# Booting image at a1500000 ...
   Image Name:   Linux 2.6.21
   Image Type:   ARM Linux Kernel Image (gzip compressed)
   Data Size:    1125626 Bytes =  1.1 MB
   Load Address: a0008000
   Entry Point:  a0008000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK

Starting kernel ...

òLinux version 2.6.21-uc0 (user@eadevenv) (gcc version 3.4.4) #1 Fri Jun
5 07:59:47 CEST 2009
CPU: NXP-LPC2478 [1701ff35] revision 5 (ARMv4), cr=a023ae00
Machine: Embedded Artists LPC2478 OEM Board
Ignoring unrecognised tag 0x00000000
Built 1 zonelists.  Total pages: 8128
Kernel command line: root=/dev/ram initrd=0xa1800000,4000k
console=ttyS0,115200N8
PID hash table entries: 128 (order: 7, 512 bytes)
LPC2XXX Clocking Fin=12000000Hz Fcco=288000000Hz M=11 N=0
Fcclk=48000000 PCLKSEL=55515555 11555455
Console: colour dummy device 80x30
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory: 32MB = 32MB total
Memory: 26100KB available (2040K code, 227K data, 108K init)
Mount-cache hash table entries: 512
NET: Registered protocol family 16
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP reno registered
checking if image is initramfs...it isn't (bad gzip magic numbers); looks
like an initrd
Freeing initrd memory: 4000K
...
Welcome to
          ____ _  _
         /  __| ||_|
    _   _| |  | | _ ____  _   _  _  _
   | | | | |  | || |  _ \| | | |\ \/ /
   | |_| | |__| || | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/
```

```
Board specific drivers by Embedded Artists AB
http://www.EmbeddedArtists.com

(Release 2008-10-20: Check for updates)



mci-cid (SD memory card):
  Manufacturer ID: 0x03 = SanDisk
  OEM/Application ID: SD
  Product Name: SU256
  Product Revision: 8.0
  Serial Number: 0x6013ae0f
  Date Code: 2006.5
  sector size = 512 (Bytes), card size = 253 (MBytes)

  dump csd data: 002600325f5983c4
                 eddacfff92404060
 mmc: mmc1
eth0: Link now 100-FullDuplex
 mmc: mmc1
init: Booting to single user mode
#
```

11.2.1 The `rc` script

When the built-in drivers have been loaded a script called `rc` located in the `/etc` directory will be called upon. The Embedded Artists `rc` script, which in the µClinux source tree is located in `uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board/`, will begin by creating device files in the `/dev` directory, mount a number of file systems, load board specific drivers, setup the network interface and finally look for a user specific script (`/mnt/mmc/userrc`) on the memory card and execute that script if it is available. Below is an excerpt of the `rc` script.

```
/bin/mount -t tmpfs tmpfs /dev
/bin/mknod /dev/tty c 5 0
/bin/mknod /dev/console c 5 1
...
/bin/mknod /dev/fb0 c 29 0
/bin/mknod /dev/ts0 c 13 128
/bin/mknod /dev/tsraw0 c 13 144

hostname EA-LPC2478

/bin/expand /etc/ramfs.img /dev/ram1

mount -t sysfs sysfs /sys
mount -t proc proc /proc
mount -t ext2 /dev/ram1 /var
mkdir /var/tmp
mkdir /var/log
mkdir /var/run
mkdir /var/lock
mkdir /var/empty

cat /etc/motd

# load board specific drivers
```

```
if [ -f /drivers/lpc2468mmc.ko ]; then
  insmod /drivers/lpc2468mmc.ko
fi

if [ -f /drivers/sfr.ko ]; then
  insmod /drivers/sfr.ko
fi

# setup network interface
ifconfig eth0 192.168.5.233 up

# start internet services
inetd &

#mount -t nfs -o nolock,rsize=4096,wsize=4096 192.168.5.10:/home/user
/mnt/nfs

mount -t vfat /dev/mmca1 /mnt/mmc

# if user defined rc exists then execute it
if [ -f /mnt/mmc/userrc ]; then
  /bin/sh /mnt/mmc/userrc
fi
```

You can look at the `rc` script by issuing the `cat` command.

```
# cat /etc/rc
```

11.2.2 The `userrc` script

The user specific rc script (`userrc`) can typically be used to setup your own IP address, load your own drivers and modules or start applications without having to modify the `rc` script.

Setup your own IP address by adding the following row to the `userrc` file and then copy the `userrc` file onto a memory card.

```
ifconfig eth0 192.168.0.100
```

11.3 File Systems

The root file system is most often mounted as a read-only file system stored in RAM. This prohibits the creation of new files in runtime. There are, however, a number of exceptions.

- The `/dev` directory represents a file system of the type `tmpfs`, which is a temporary storage area which uses virtual memory (RAM) instead of a persistent storage device. This means that it is possible to create files in this directory in runtime, but any changes will be lost during a power cycle.
- The `/var` directory is a RAM based file system of the type ext2 which means that this directory also supports creation of files in runtime, but changes are lost during a reset or power cycle. Interesting to note is that the `/tmp` directory is a symbolic link to `/var/tmp`.

You can try to create a file in one of the read-only directories by using the echo command. The result will be an error message.

```
# cd /etc
# echo test > myfile.txt
myfile.txt: cannot create
```

If you instead try to create the file in one of the writable directories, such as the /tmp directory, the file will be created.

```
# cd /tmp
# echo test > myfile.txt
# ls
myfile.txt
```

If you need to be able to store or modify files persistently you can do this either by mounting a USB memory stick or a memory (MMC/SD) card onto the file system and make your changes there. It is also possible to store the root file system in, for example, NAND flash and the most suitable type of file system is then a JFFS2 file system, see the next section for more details.

11.3.1 JFFS2 – Journalling Flash File System version 2

A root file system of the type JFFS2 can be stored in NAND flash and thereby making it possible to create or modify files persistently. Section 10.6 contains guides of how to update the NAND flash with kernel and jffs2 images as well as information about how to set the boot arguments.

There are a couple of things you need to know about when using a JFFS2 file system in NAND flash.

- It takes some time to mount it at startup so the boot time for Linux might increase.
- The access to the file system is also slower than compared to a RAM based file system.
- You can create new files as well as modify existing and are not limited to the `/dev` or `/var` directories.
- When a file is created it won't be stored persistently immediately since Linux file systems have a RAM based cache. This means that in order to make sure that changes to the file system are stored persistently you should force this to happen. One example of how to force changes to be stored is given below.

```
# mount -o remount,ro /dev/mtdblock1 /
```

Please note that issuing the mount command with the remount option could take several minutes to complete.

11.4 Users

Two users have been added to Embedded Artists µClinux distribution. By default the user "root" is logged in when the system is started.

User Name	Password
`root`	`uclinux`
`guest`	`uclinux`

11.4.1 The `passwd` file

All user accounts added to the system can be found in the `/etc/passwd` file. Use the `cat` command to list the content of this file.

```
# cat /etc/passwd
guest:ajBsnELqCA.2Y:100:100:guest:/:/bin/sh
root:ajBsnELqCA.2Y:0:0:root:/:/bin/sh
```

Each line in this file represents a user/account with some additional information about each account. Each line has the following format.

`account:password:UID:GID:GECOS:directory:shell`

- `account` – the name of the user
- `password` – the encrypted user password
- `UID` – the numerical user ID
- `GID` – the numerical primary group ID for the user
- `GECOS` – optional field for informational purposes. Usually, it contains the full user name.
- `directory` – the user's home directory
- `shell` – the program to run at login

11.4.2 Adding the `addgroup`, `adduser` and `passwd` Commands

In order to add users dynamically in runtime the `adduser` and `passwd` commands must be available. By default these commands are not available in the Embedded Artists distribution. This section describes how to enable them.

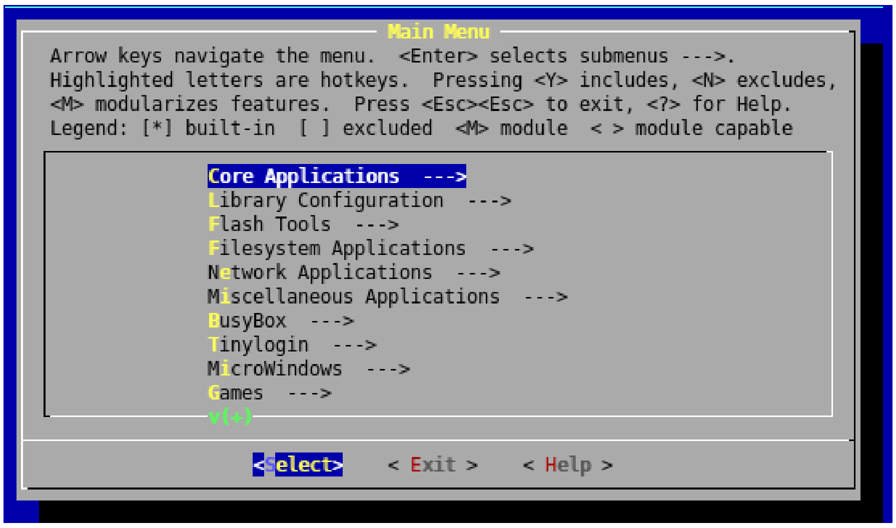
1. In the Debian Etch distribution, change directory to the µClinux distribution

```
$ cd /home/user/uClinux-dist
```

2. Start the configuration tool

```
$ make menuconfig
```

3. Go to the "Kernel/Library/Defaults Selection" menu alternative and click the Select button.
4. Go to the "Customize Vendor/User Settings and click the Space bar on your keyboard to select that option.
5. Click on the Exit button in order to go back to the Main Menu. Click once more on the Exit button and select Yes when asked to save the configuration.
6. You will now start the Main Menu for the Vendor/User Settings configuration, see Figure 41.



```
Main Menu
Arrow keys navigate the menu.  <Enter> selects submenus --->.
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
<M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                Core Applications  --->
                Library Configuration  --->
                Flash Tools  --->
                Filesystem Applications  --->
                Network Applications  --->
                Miscellaneous Applications  --->
                BusyBox  --->
                Tinylogin  --->
                MicroWindows  --->
                Games  --->
                v(+)

                <Select>    < Exit >    < Help >
```

Figure 41 Main Menu for Vendor/User Settings

7. Go to the BusyBox menu and you will be presented with a long list of commands.
8. Go to the addgroup command and press the space bar to select this command.
9. Go to the adduser command and press the space bar to select this command.
10. Continue down the list until you come to the passwd command and select it by pressing the space bar.
11. Now click on the Exit button to return to the Main menu. Click on the Exit button once more and select Yes when asked to save the configuration.
12. Clean the previous build

```
$ make clean
```

13. Build the system

```
$ make
```

14. When µClinux has been successfully built the images will be available in the `uClinux-dist/images` directory.
15. Now use any of the boot options described in chapter 10 to boot the system with these new images.

Please note that these commands can only be used with a writable root file system which basically means that you need to use a JFFS2 file system, see section 11.3.1 for more information.

When the commands have been added to the system it is time to add a user.

1. First you have to create the /etc/group file since by default it isn't available, but needed by the addgroup command.

```
# echo "# group file" > /etc/group
```

2. Now add the user joe to the system. You will be asked to enter a password for the user joe.

```
# adduser joe
Changing password for joe
Enter the new password (minimum of 5, maximum of 8 characters)
Please use a combination of upper and lower case letters and
numbers.
Enter new password:
Re-enter new password:
Jan  1 00:02:54 passwd[147]: password for `joe' changed by user
`root'
Password changed.
```

If you want to use a multi-user system other useful commands are `chmod` and `chown` which can be used to change the permissions on files, se sections 9.5.7 and 9.5.8 for more information about these commands.

11.5 Network Related

11.5.1 Static IP Address

The IP address of the system can be setup statically by using the `ifconfig` command. In sections 11.2.1 and 11.2.2 the `rc` and `userrc` scripts were described and it was also illustrated how `ifconfig` was used to setup the IP address of the system. The command can also be used in runtime to change the IP address.

```
# ifconfig eth0 192.168.5.240
```

The `ifconfig` command can also be used to see which IP address that has been assigned to your system.

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1A:F1:00:00:00
          inet addr:192.168.5.233  Bcast:192.168.5.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5455 errors:0 dropped:0 overruns:0 frame:0
          TX packets:308 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:285059 (278.3 KiB)  TX bytes:30143 (29.4 KiB)
          Interrupt:21
```

11.5.2 Dynamic IP Address – DHCP

For easier network administration the Dynamic Host Configuration Protocol (DHCP) is most often used to assign IP addresses to computers and devices connected to the network. A DHCP server is connected to the network and is responsible for assigning IP addresses to DHCP clients. The server is responsible for making sure that the assigned IP addresses are unique to avoid conflicts on the network, which can easily arise when assigning IP addresses statically.

Follow the steps below to enable DHCP (client) support in the µClinux distribution.

1. In the Debian Etch distribution change directory to the µClinux distribution

```
$ cd /home/user/uClinux-dist
```

2. Start the configuration tool

```
$ make menuconfig
```

3. Go to the “Kernel/Library/Defaults Selection” menu alternative and click the Select button.
4. Go to the “Customize Kernel Settings” and click the Space bar on your keyboard to select that option.
5. Also go to the “Customize Vendor/User Settings and click the Space bar on your keyboard to select that option.
6. Click on the Exit button in order to go back to the Main Menu. Click once more on the Exit button and select Yes when asked to save the configuration.
7. The Main menu for the kernel configuration will first be opened.
8. Go to the Networking menu and then to Networking options.
9. Go to the Packet Socket option and press the space bar on your keyboard twice (you should see a star ‘*’) to enable the option.
10. Click the Exit button three times and then select Yes when asked to save the configuration.
11. You will now start the Main Menu for the Vendor/User Settings configuration
12. Go to the Network Applications menu and you will be presented with a long list of applications.
13. Scroll down to the dhcpcd-new application and select it by pressing the space bar on your keyboard.
14. Click the Exit button to go back to the Main menu and then click the Exit button once more. Select Yes when asked to save the configuration.
15. Modify the `rc` script to run the DHCP client instead of statically assigning the IP address. Disable the `ifconfig` command and add the `dhcpcd` command instead (see example below).

```
$ gedit vendors/EmbeddedArtists/LPC2478OEM_Board/rc
```

```
# the line below is
# ifconfig eth0 192.168.5.233 up

# run the dhcp client
dhcpcd &
```

16. Save the file when you have done the changes and exit the gedit application.
17. Clean the previous build

```
$ make clean
```

18. Build the system

```
$ make
```

19. When µClinux has been successfully built the images will be available in the `uClinux-dist/images` directory.

20. Now use any of the boot options described in chapter 11 to boot the system with these new images.

When the system is started you will most likely see error messages about failing to write to some missing files. You can ignore these messages since the DHCP request will be carried out anyways.

```
Jan  1 00:00:44 dhcpcd[141]: dhcpConfig: failed to write cache
file /etc/dhcpc/dhcpcd-eth0.cache: No such file or directory

Jan  1 00:00:44 dhcpcd[141]: dhcpConfig: failed to write info file
/etc/dhcpc/dhcpcd-eth0.info: No such file or directory
```

If you don't want these error messages to appear you need to use a writable root file system and create the missing files.

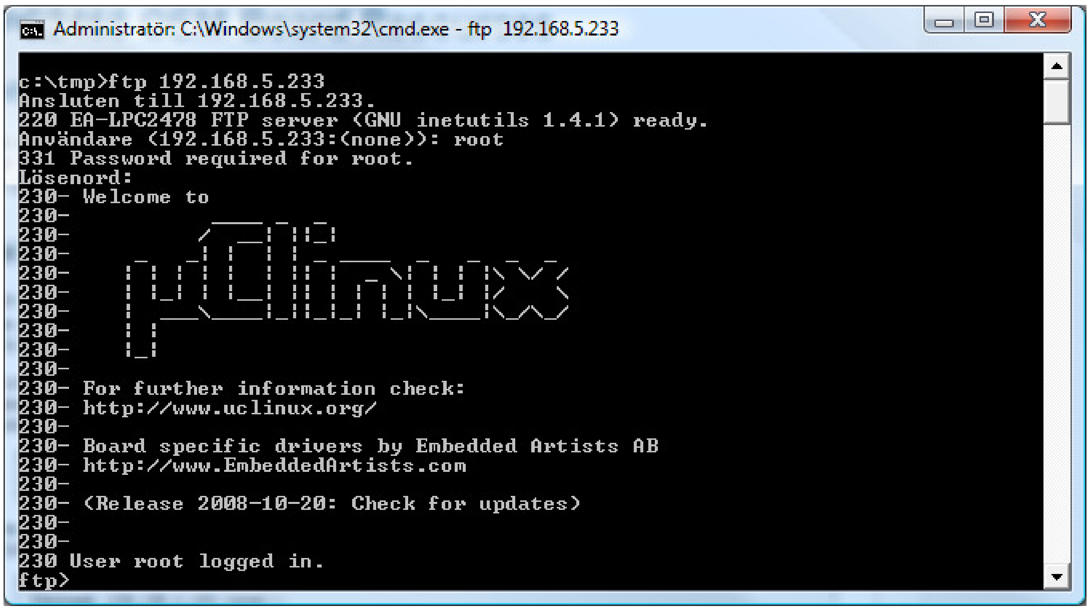
You can check if your system has got an IP address by using the `ifconfig` command.

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1A:F1:00:00:00
          inet addr:192.168.5.86  Bcast:192.168.5.255
Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1764 (1.7 KiB)  TX bytes:2926 (2.8 KiB)
          Interrupt:21
```

11.5.3 FTP Access

An FTP server is in the default configuration of Embedded Artists µClinux distribution running at startup of the system. FTP is conventient to use to update the contents of the file system. From anywhere on the network you can start an FTP client and download or upload files to your system.

1. Make sure you have µClinux up-and-running.
2. Start a command prompt in Windows and enter ftp followed by the IP address of your system, see Figure 42.



Administratör: C:\Windows\system32\cmd.exe - ftp 192.168.5.233

```
c:\tmp>ftp 192.168.5.233
Ansluten till 192.168.5.233.
220 EA-LPC2478 FTP server (GNU inetutils 1.4.1) ready.
Användare (192.168.5.233:(none)): root
331 Password required for root.
Lösenord:
230- Welcome to
230-           ____ _  _
230-          /  __| ||_|
230-     _   _| |  | | _ ____  _   _ _   _
230-    | | | | |  | || |  _ \| | | |\ \/ /
230-    | |_| | |__| || | | | | |_| |/    \
230-    |  ___\____|_||_|_| |_|\____|\_/\_/
230-    | |
230-    |_|
230-
230- For further information check:
230- http://www.uclinux.org/
230-
230- Board specific drivers by Embedded Artists AB
230- http://www.EmbeddedArtists.com
230-
230- (Release 2008-10-20: Check for updates)
230-
230-
230 User root logged in.
ftp>
```

Figure 42 FTP client in a DOS prompt

3. When asked for user you can answer with any of the users present in the system, for example root or guest, see section 11.4 for more information about users.
4. You will also have to enter the password for the user.
5. Now you will be logged in and can download or upload files.
6. Go to the tmp directory and upload a file.

```
ftp> cd tmp
ftp> put myfile.txt
```

7. You can now check in the terminal attached to your board that the file has been uploaded

```
# cd /tmp
# ls -la
```

8. If you would like to download files use the get command in the FTP client.

```
ftp> get myfile.txt
```

11.5.4 Telnet Access

By connecting to the system with a telnet client you will be able to execute commands and programs – both your own and the built-in ones. This can be done from anywhere on the network where you have access to a telnet client.

1. Make sure you have µClinux up-and-running
2. From Debian Linux start a telnet client and connect it to your system.

```
$ telnet 192.168.5.233
Trying 192.168.5.233...
Connected to 192.168.5.233.
```

```
Escape character is '^]'.
login:
```

3. You will be asked to login. Use any of the users and passwords setup in your system, for example root or guest, see section 11.4 for more information about users.
4. You have now access to the standard linux commands available in your µClinux configuration. The same commands you can access when having a terminal directly connected to the system.
5. Enter exit when you would like the telnet client to disconnect from the system.

```
$ exit
```

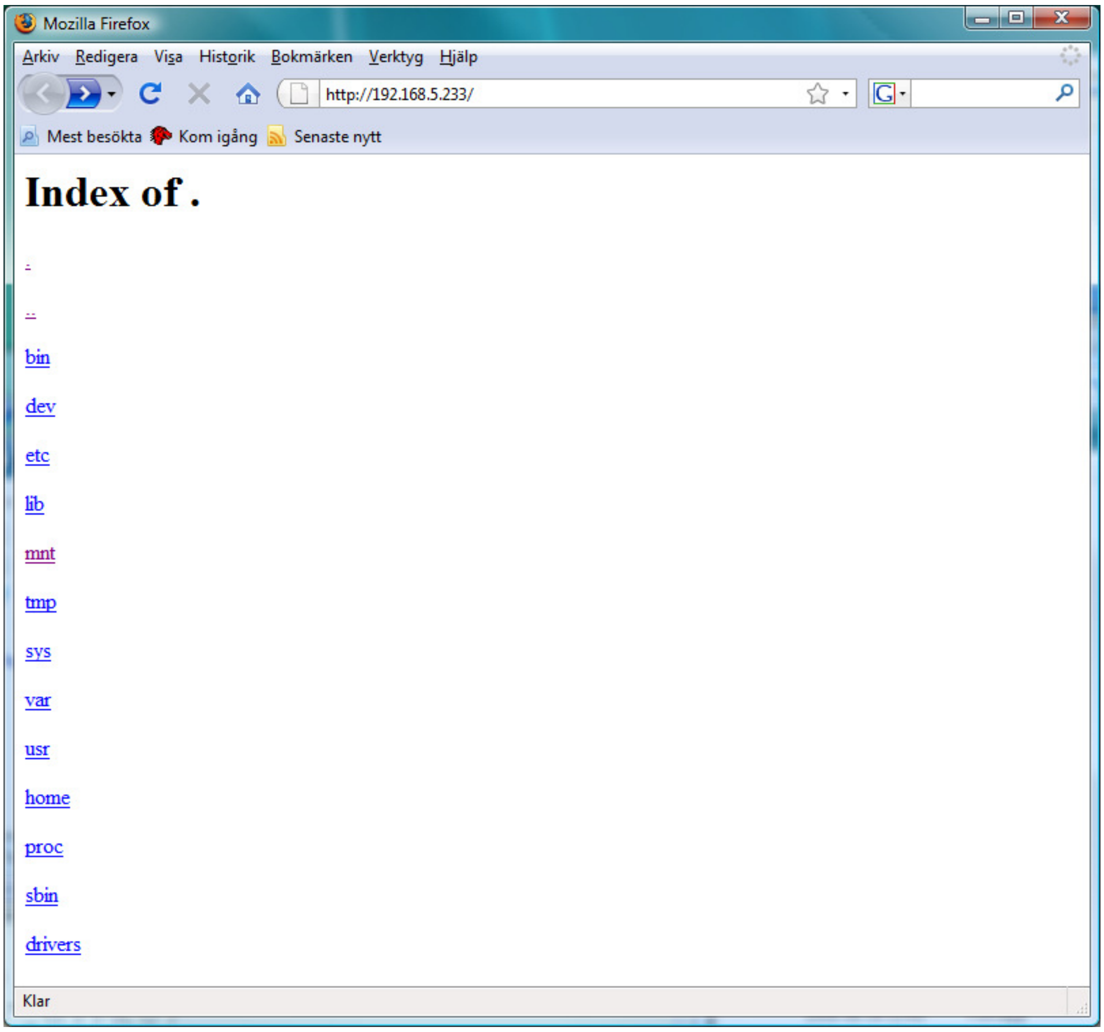
11.5.5 Web/HTTP Access

The µClinux distribution comes with a pre-installed webserver.

1. Since the webserver isn't started by default you have to do this. It is also possible to start the server from the `rc` or `userrc` script.

```
# httpd &
```

2. Now start a web browser and enter the IP address in the web browser's address field. The web server is configured to have its directory, i.e., look for content in `/home/httpd`, but if that directory doesn't exist it will show the content of the root of the file system, see Figure 43.
3. If an `index.html` file exists in a directory the web server will automatically load that file.


Mozilla Firefox
Arkiv Redigera Visa Historik Bokmärken Verktyg Hjälp
http://192.168.5.233/
Mest besökta Kom igång Senaste nytt
Klar

Index of .

.

..

bin

dev

etc

lib

mnt

tmp

sys

var

usr

home

proc

sbin

drivers

Figure 43 The root file system listed in web browser

11.5.6 NFS Mounting

Section 6.9.1 mentions that it is really convenient to mount a directory on you development computer so that it is reachable from the target board during, for example, application development. You can them compile the application on your computer and directly test and run it on your target board. This section describes how to perform an NFS mount.

1. Follow the guidelines in section 9.4.3 to export a directory on your development computer running Debian Etch.
2. Make sure µClinux is up-and-running and that you have a terminal application connected to the board.
3. Mount the remote directory onto `/mnt/nfs` (note that the command below should be on one line).

```
# mount -t nfs -o nolock,rsize=4096,wsize=4096
192.168.5.10:/home/user /mnt/nfs
```

4. Change the directory to `/mnt/nfs` and list the content.

```
# cd /mnt/nfs
# ls -la
```

5. You should now be able to access the content on the remote directory as if it was a local file system.

11.6 Graphics Related

11.6.1 Nano-X

Nano-X is a windowing system designed for resource constraint devices and runs on Linux systems with framebuffer support. Section 6.8 briefly describes Nano-X and shows how to use the interfaces. This section describes how to start nano-X and run the demo applications that are available in Embedded Artists µClinux distribution.

1. Make sure µClinux is up-and-running.
2. Start the nano-X server.

```
# nano-X &
```

3. You will get a warning message about tsdev being scheduled for removal, but ignore this warning.
4. Start the nano-X demo.

```
# demo &
```

5. Touch the screen in the upper left corner to end the application.
6. Start the nano-X server in landscape mode

```
# nano-X -L &
```

7. Start the demo again and see how the orientation has changed.

```
# demo &
```

8. Touch the screen in the lower left corner to end the application.
9. Start the nano-X server again.

```
# nano-X &
```

10. Start the nano-X window manager application.

```
# nanowm &
```

11. Start the demo and see how all the squares now looks like windows with borders and title bars.

11.7 USB Related

11.7.1 USB Host – Connect USB Memory Stick

USB Host functionality, which is described in more detail in section 5.8 is by default enabled in the µClinux configuration. The host functionality can, for example, be used to attach a USB memory stick to the board and thereby copy files to and from the memory stick.

1. Make sure you have µClinux up-and-running.

2. Attach a USB memory stick to the connector on the board and you will see output similar to the example below.

```
usb 1-2: new full speed USB device using lpc24xx-ohci and address
3
usb 1-2: configuration #1 chosen from 1 choice
scsi1 : SCSI emulation for USB Mass Storage devices
scsi 1:0:0:0: Direct-Access     SanDisk  Cruzer           8.02 PQ:
0 ANSI: 0 CCS
SCSI device sda: 15704063 512-byte hdwr sectors (8040 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
SCSI device sda: 15704063 512-byte hdwr sectors (8040 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
 sda: sda1
sd 1:0:0:0: Attached scsi removable disk sda
```

3. The output means that the kernel has discovered the USB memory stick and associated it with the device file `/dev/sda1`.
4. Mount the USB memory stick as a file system of type vfat onto the directory `/mnt/usbmem`.

```
# mount -t vfat /dev/sda1 /mnt/usbmem
```

5. You can now access the memory stick by changing directory to `/mnt/usbmem`.

```
# cd /mnt/usbmem
# ls -la
```

11.7.2 USB Device – Target is a USB Memory Stick

If you would like the target board to become a USB memory stick when connected to, for example, a computer you will have to use the USB device functionality, see section 5.9 for more information.

1. Make sure µClinux is up-and-running.
2. If you have the µClinux configuration for the LPC2468 OEM Board the USB device driver is compiled as a separate driver module and must therefore be loaded before the USB device functionality can be used. You can skip this step if you have the µClinux configuration for the LPC2478 OEM Board.

```
# cd /lib/modules/2.6.21-uc0/kernel/drivers/usb/gadget
# insmod lpc24xx_udc.ko
```

3. Now it is time to load the gadget driver for the mass storage device. The backing storage device, i.e., the file system for the mass storage device, is given as a module parameter when loading the driver. In this example we are using the MMC/SD card as a file system.

```
# cd /lib/modules/2.6.21-uc0/kernel/drivers/usb/gadget
# insmod g_file_storage.ko file=/dev/mmca1
```

4. Now connect a USB cable (A to mini-B) between the target board and your computer and it should appear as a mass storage device on your computer. You will get access to the files available on the MMC/SD card.

11.8 I^{2}C

Two devices are attached to the I^{2}C bus on the Embedded Artists boards. One of the devices is an I/O expander that controls the LEDs and some buttons on the board. The other device is an EEPROM device, i.e., a small storage area.

11.8.1 PCA9532 Device

The PCA9532 has a number of files exposed at this location in the file system: `/sys/bus/i2c/devices/0-0060/`. Read section 5.11.3 for more information about the files that are present in this directory.

1. Make sure µClinux is up-and-running.
2. Change directory to the pca9532 directory.

```
# cd /sys/bus/i2c/devices/0-0060
```

3. If you study the schematics for the base board you will see that LED1 to LED 4 are connected to LED selector 2 (ls2) on the PCA9532. Turn on LED1 by writing the value 1 to the ls2 file.

```
# echo 1 > ls2
```

4. Turn on LED2 by writing the value 4 to the ls2 file.

```
# echo 4 > ls2
```

5. Turn on LED 1 and LED2.

```
# echo 5 > ls2
```

6. You can check the state of the device pins (connected to the LEDs) by reading the input1 file.

```
# cat input1
252
```

7. The value 252 is the same as the binary value 11111100, i.e., bit 0 and bit 1 has the value 0 all others have the value 1. A LED is turned on when the output is LOW so the value 252 means that LED 1 and LED 2 are lit.
8. You need to study the schematics for the board as well as the datasheet for the PCA9532 device to see how to use the files.

11.8.2 EEPROM Device

The EEPROM device is exposed as a file in the sys file system. The location is `/sys/bus/i2c/devices/0-0050/data0`. The size of this file is always 32768 bytes which is the same as the size of the EEPROM memory.

An application which accesses the EEPROM file is provided in the Embedded Artists µClinux distribution.

1. Make sure you have µClinux up-and-running.
2. List the first 128 bytes in the EEPROM in hex format.

```
# eeprom r8 0 128 hex
```

3. Write 4 hex values to the start of the EEPROM.

```
# eeprom w8 0 h:10:30:50:70
```

4. Now list the first bytes in the EEPROM to see that it has changed.

```
# eeprom r8 0 32 hex
```

5. If you would like to see all options you can use with the eeprom application run it without any arguments.

```
# eeprom
```

The source code for the eeprom application is available in the µClinux distribution. By studying the source code you can see how to access a file in the file system.

```
$ cd /home/user/uClinux-dist
$ gedit vendor/EmbeddedArtists/LPC2478OEM_Board/applications/eeprom.c
```

11.9 Real-Time Clock (RTC)

The LPC24xx microcontrollers come with an embedded real-time clock (RTC). In Linux you can control this clock by using the `date` and `hwclock` commands.

1. Make sure you have µClinux up-and-running
2. Set the system date and time using the date command. Please note that the year should only be entered with two digits.

```
# date -i
Enter year [2000]:
09
Enter month [1-12]:
6
Enter day [1-31]:
17
Enter hour [0-23]:
18
Enter minute [0-59]:
16
Enter seconds [0-59]:
11
Clock: old time 1970/01/01 - 00:22:12 GMT
Clock: new time 2009/06/17 - 18:16:11 GMT
Wed Jun 17 18:16:11 2009
```

3. You can now use the `hwclock` command to update the real-time clock. If you are using a read-only file system you will get a warning, but can safely ignore it.

```
# hwclock --systohc
hwclock: Could not open file with the clock adjustment parameters
in it (/etc/adjtime) for writing, errno=30: Read-only file system.
Drift adjustment parameters not updated.
```

4. If you have a correctly set real-time clock you can update the system time by using the `hctosys` option.

```
# hwclock --hctosys
```

5. If you would like to check the system time you can issue the date command without any arguments.

```
# date
Wed Jun 17 18:19:50 2009
```

11.10 ADC

A proprietary driver has been developed for the ADC (analog to digital) device. This section describes how you can use the driver to monitor the ADC inputs.

1. Make sure µClinux is up-and-running.
2. Load the driver module.

```
# insmod /drivers/adc.ko
```

3. The ADC inputs are now available as four devices files, `/dev/ad0`, `/dev/ad1`, `/dev/ad2`, and `/dev/ad3`. Read the value from the ADC input connected to the trim potentiometer on the base board.

```
# cat /dev/adc2
```

4. Turn the trim potentiometer on the base board and then read the value again.

```
# cat /dev/adc2
```

11.11 SFR

Sometimes, especially for debugging purposes, it is convenient to be able to access the processor's registers from user space. A proprietary driver has been developed for this purpose and is included in the µClinux distribution, see section 5.15 for more information.

1. Make sure µClinux is up-and-running.
2. Load the driver module.

```
# insmod /drivers/sfr.ko
```

3. If you get the following error message the module was already loaded.

```
insmod: cannot insert `sfr.ko': File exists (-1): File exists
```

4. Read the value of the PINSEL2 register.

```
# echo PINSEL2:? > /dev/sfr
```

5. You can also access the registers with their address.

```
# echo 0xE002C008:? > /dev/sfr
```

6. Set pin P2.10 to an output.

```
# echo FIO2DIR:0x400 > /dev/sfr
```

7. Produce a low level output on pin P2.10 and the LED by the P2.10 button on the base board will be lit.

```
# echo FIO2CLR:0x400 > /dev/sfr
```

8. Produce a high level output on pin P2.10 and the LED by the P2.10 button on the base board will be turned off.

```
# echo FIO2SET:0x400 > /dev/sfr
```

11.12 Framebuffer Console

The Frame Buffer Console is a low-level frame buffer based console driver that allows the console to be displayed on the frame buffer device. With the Embedded Artists boards this means having the console on the QVGA display. Support for this functionality is included in the Linux kernel and only needs to be enabled. As long as there is a frame buffer device no extra source code needs to be added.

1. In the Debian Etch distribution, change directory to the µClinux distribution.

```
$ cd /home/user/uClinux-dist
```

2. Start the configuration tool

```
$ make menuconfig
```

3. Go to the "Kernel/Library/Defaults Selection" menu alternative and click the Select button.
4. Go to the "Customize Kernel Settings" and click the space bar on your keyboard to select that option.
5. Click on the Exit button to go back to the main menu and then click the Exit button again. Select Yes when asked to save the configuration.
6. Go to the following configuration option and click the space bar twice to select the option.

   Device Drivers → Graphics support → Console display driver support → Framebuffer Console support

7. Also select the "Select compiled-in fonts option" and choose the "Mac console 6x11 font"

   Device Drivers → Graphics support → Console display driver support → Select compiled-in fonts

Device Drivers → Graphics support → Console display driver support → Mac console 6x11 font

8. Click the exit button to go back to the “Graphics support” section.
9. Go the “Logo configuration” section and select the “Bootup logo” option.

   Device Drivers → Graphics support → Logo configuration → Bootup logo

10. Click the Exit button until you are asked to save the configuration. Select the Yes option.
11. Build the system

```
$ make
```

12. When µClinux has been successfully built the images will be available in the `uClinux-dist/images` directory.
13. Before you use these images to boot µClinux start the board and enter into the u-boot console.
14. Change the boot arguments so that the console is no longer mapped onto the ttyS0 device. In the example below the root file system is loaded to RAM.

```
# set bootargs root=/dev/ram initrd=0xa1800000,4000k
```

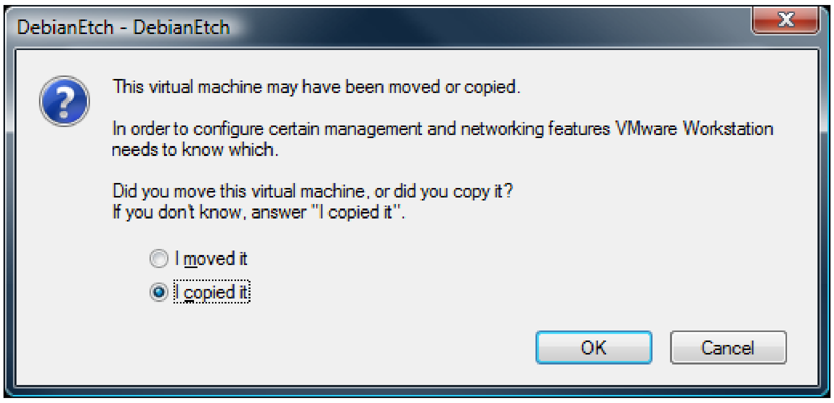
15. Now boot µClinux using the new images and a booting option described in chapter 10 and you will see the console output on the display (if you have a display attached to your board).
16. Attach a USB keyboard to the USB host connector on the base board and start typing on the keyboard. Whatever you type will be directed to the console and you should see it on the display.

12 Guides – Create Your Own SDK

12.1 Debian Etch as VMware Appliance

This section describes how you can get up-and-running with a Debian Etch Linux distribution as a VMware appliance (virtual machine).

1. Download and install the VMware Player, see ref [21].
2. Download Debian Etch as a WMware appliance. Ref [35] contains a direct link to a zip-file containing an appliance. It is also possible to search for appliances at the VMware site, see ref [23].
3. Unzip the VMware appliance file and double-click on the `DebianEtch.vmx` file to load Debian Etch in the VMware Player.
4. Since this is the first time this virtual machine is loaded you will be asked if it was moved or copied. Select the "I copied it" option, see Figure 44.
5. When Debian Etch has started you can login using the username "user" and password "user".
6. The default memory setting for this virtual machine is 96 MB which is a little low. Follow the guide in section 8.2 to increase the memory to something between 256 MB and 1024 MB, depending on how much memory you have in your physical machine.
7. You might also need to add a different keyboard layout since the default layouts are German and U.S. English. Follow the guide in section 9.3.2 to change the default keyboard layouts.


DebianEtch - DebianEtch

This virtual machine may have been moved or copied.

In order to configure certain management and networking features VMware Workstation needs to know which.

Did you move this virtual machine, or did you copy it?
If you don't know, answer "I copied it".

I moved it [✓ I copied it]

Figure 44 VMware dialog

12.2 Install Necessary Tools

This section describes how to install all the necessary tools needed to build the u-boot and µClinux.

1. Start a terminal application from the Applications menu.

   Application → Accessories → Terminal

2. Create a directory called "install" where you will store the tools you download.

```
$ mkdir install
```

3. Change to the super-user, enter "root" as password.

```
$ su
```

4. The package manager is needed to install some of the tools and for this reason we must make sure the package index files are resynchronized.

```
$ apt-get update
```

5. Install the build-essential package to get programs needed for building and compiling (note this is not the cross-compiler tools). Answer Y on any questions you may get while installing the package.

```
$ apt-get install build-essential
```

6. Install the ncurses library which is needed to run the graphical Linux kernel configuration tool. Answer Y on any questions you may get while installing the library.

```
$ apt-get install libncurses5-dev
```

7. Install the compression library zlib.

```
$ apt-get install zlib1g-dev
```

8. Install the MTD tools which are needed when creating a JFFS2 file system.

```
$ apt-get install mtd-tools
```

9. Create a symbolic link in /usr/local/bin to the mkfs.jffs2 utility and change the group permission of the utility so that it can be used by the user named “user”.

```
$ cd /usr/local/bin
$ ln -s /usr/sbin/mkfs.jffs2 mkfs.jffs2
$ chgrp users /usr/sbin/mkfs.jffs2
```

10. To be able to create a cramfs file system (compressed file system) the mkcramfs tool is needed. The mkcramfs tool must have mkfs.jffs2 compatible device table support. Because of this we don’t just install the cramfs tools (apt-get install mkcramfs), but instead download a prebuilt mkcramfs tool from Embedded Artists support site. If you would like to apply the patch that adds device table support you will find the patch at the location specified in ref [36].
11. Download the mkcramfs file and copy it to the `/usr/local/bin` directory. Make sure everyone can execute this file.

```
$ cd /usr/local/bin
$ cp /home/user/install/mkcramfs .
$ chmod a+x mkcramfs
```

12. Download the ARM Linux toolchain from SnapGear, see ref [37], and store the file (`arm-linux-tools-20061213.tar.gz`) in the `/home/user/install` directory.
13. Install the toolchain.

```
$ cd /
$ tar -xzvf /home/user/install/arm-linux-tools-20061213.tar.gz
```

14. Download the ARM ELF toolchain, see ref[38], and store it in the `/home/user/install` directory. This toolchain is needed to build the applications in the µClinux distribution. Usually you would only need one cross-compiler, but there were too many build related problems building the applications using the ARM Linux toolchain (gcc 3.4.4). Using the ARM ELF toolchain with gcc 2.95.3 didn't result in build problems.
15. Install the toolchain

```
$ sh /home/user/install/arm-elf-tools-20040427.sh
```

16. Exit as super-user

```
$ exit
```

12.3 Install and Build the u-boot and mkimage

This section describes how to install and build the u-boot and also how to copy the mkimage tool so that it is accessible when building µClinux.

1. Follow the guide in section 10.1 to install and build the u-boot. You can skip step 1 since there isn't any existing installation. Step 2 can be changed to download the u-boot distribution from directly from the u-boot website, see ref [30].
2. When u-boot has been built copy the mkimage tool from the u-boot to the `/usr/local/bin` directory and make sure everyone can execute the tool. The mkimage tool is used when creating a u-boot image of the µClinux image.

```
$ su
$ cd /usr/local/bin
$ cp /home/user/u-boot-1.1.6/tools/mkimage
$ chmod a+x mkimage
$ exit
```

3. You are now ready to install and build µClinux.

12.4 Install and Build µClinux

You can follow the guide described in section 11.1 to install and build µClinux. Skip step 1 since there won't be an existing installation of µClinux. Step 2 can be changed to "download the µClinux distribution from ref [33]". In step 4 you can download the Linux kernel from ref [34] instead of getting it from the DVD.

13 Resources

[1] Embedded System Designs Annual study of the Embedded market from 2007
http://www.embedded.com/design/opensource/201803499

[2] Free Software Foundation and GNU Project
http://www.gnu.org

[3] µClinux website
http://www.uclinux.org/

[4] Embedded Linux system design and development
P. Raghavan, Amol Lad, Sriram Neelakandan
ISBN: 0-8493-4058-6

[5] Real-time Preemption Patch
http://rt.wiki.kernel.org/index.php/Main_Page

[6] Real-Time Core for Linux from Wind River
http://www.windriver.com/products/platforms/real-time_core/

[7] Open RTLinux
http://www.rtlinuxfree.com/

[8] Adeos
http://home.gna.org/adeos/

[9] RTAI
https://www.rtai.org/

[10] Xenomai
http://www.xenomai.org

[11] The APEX boot loader
http://wiki.buici.com/wiki/Apex_Bootloader

[12] The RedBoot boot loader
http://sourceware.org/redboot/

[13] The MicroMonitor boot loader
http://microcross.com/html/micromonitor.html

[14] GRUB, the Grand Unified Boot loader
http://www.gnu.org/software/grub/

[15] Das U-Boot – the Universal Boot Loader
http://www.denx.de/wiki/U-Boot/

[16] Memory Tecknology Device (MTD) subsystem
http://www.linux-mtd.infradead.org/

[17] Linux Device Drivers, Third Edition
Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
ISBN: 0-596-00590-3

[18] Nano-X Window System
http://www.microwindows.org/

[19] PCA9532 data sheet
http://www.nxp.com/acrobat_download/datasheets/PCA9532_3.pdf

[20] JFFS2 file system
http://sources.redhat.com/jffs2/jffs2.pdf

[21] VMware Player
http://www.vmware.com/products/player/

[22] Understanding Full Virtualization, Paravirtualization, and Hardware Assit
http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[23] VMware Virtual Appliance Marketplace
http://www.vmware.com/appliances/

[24] DistroWatch.com
http://distrowatch.com/

[25] Debian Distribution
http://www.debian.org/

[26] File system Hierarchy Standard
http://www.pathname.com/fhs

[27] KDE Desktop environment
http://www.kde.org

[28] GNOME Desktop environment
http://www.gnome.org/

[29] 7-Zip
http://www.7-zip.org/

[30] U-Boot version 1.1.6 source code
ftp://ftp.denx.de/pub/u-boot/u-boot-1.1.6.tar.bz2

[31] Flash Magic
http://www.flashmagictool.com/

[32] Tera Term Terminal Application
http://ttssh2.sourceforge.jp/

[33] µClinux distribution, version 20070130
http://www.uclinux.org/pub/uClinux/dist/uClinux-dist-20070130.tar.gz

[34] Linux kernel, version 2.6.21
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.tar.gz

[35] Debian Etch as WMware Appliance
http://www.visoracle.com/download/debian/v/vmware-debian-etch-40r0.zip

[36] Patch for mkcramfs that adds mkfs.jffs2 compatible device table support
http://sourceforge.net/tracker/?func=detail&aid=660651&group_id=18351&atid=318351

[37] ARM Linux Toolchain
http://www.snapgear.org/downloads2.html
http://ftp.snapgear.org/pub/snapgear/tools/arm-linux/arm-linux-tools-20061213.tar.gz

[38] ARM ELF toolchain
http://opensrc.sec.samsung.com/download/arm-elf-tools-20040427.sh

[39] Install VMware Tools
http://www.vmware.com/support/ws55/doc/new_guest_tools_ws.html

[40] Embedded Artists U-Boot patch for the LPC24xx OEM Board. Log in to the support page and go to the patches section for the LPC24xx OEM Board.
http://www.embeddedartists.com/support/