INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR G. FÄRBER

Operating System Components for an Embedded Linux System

Martin Hintermann

Studienarbeit

Operating System Components for an Embedded Linux System

Studienarbeit

Executed at the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

Advisor: Prof.Dr.rer.nat.habil. Thomas Bräunl

Author: Martin Hintermann
Kirchberg 34
82069 Hohenschäftlarn

Submitted in February 2007

Acknowledgements

At first, i would like to thank my supervisor Prof. Dr. Thomas Bräunl for giving me the opportunity to take part at a really interesting project.

Many thanks to Thomas Sommer, my project partner, for his contribution to our good work.

I also want to thank also Bernard Blackham for his assistance by email and phone at any time.

In my opinion, it was a great cooperation of all persons taking part in this project.

Abstract

Embedded systems can be found in more and more devices. Linux as a free operating system is also becoming more and more important in embedded applications. Linux even replaces other operating systems in certain areas (e.g. mobile phones).

This thesis deals with the employment of Linux in embedded systems. Various architectures of embedded systems are introduced and the characteristics of common operating systems for these devices are reviewed.

The architecture of Linux is examined by looking at the particular components such as kernel, standard C libraries and POSIX tools for embedded systems. Furthermore, there is a survey of real-time extensions for the Linux kernel.

The thesis also treats software development for embedded Linux ranging from the prerequisites for compiling software to the debugging of binaries.
More precisely it describes the assembly of a cross-compiling toolchain for developing embedded software on a Windows system. Additionally means for cross-platform remote debugging of embedded software from both Linux and Windows hosts were implemented and are introduced in this thesis.

In addition to that some software for EyeBot M6 was developed in this context. A Linux framebuffer library and several user interface routines were implemented and are presented in the following.

Contents

List of Figures

List of Tables

List of Symbols

ARM	Advanced RISC Machine or Acorn RISC Machine
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DDD	Data Display Debugger
DSP	Digital Signal Processor
DVD	Digital Versatile Disc or Digital Video Disc
EMC	ElectroMagnetic Compatibility
EULA	End-User License Agreement
FPGA	Field-Programmable Gate Array
FS	FileSystem
gcc	GNU Compiler Collection
gdb	GNU DeBugger
GNU	GNU's Not Unix
GPL	GNU General Public License
I/O	Input/Output
I^2C	Inter-Integrated Circuit (bus)
IC	Integrated Circuit
IPC	Inter-Process Communication
kB	kiloByte
LCD	Liquid-Crystal Display
LED	Light-Emitting Diode
LGPL	GNU Library General Public License or GNU Lesser General Public License
LIN	Local Interconnect Network
MB	MegyByte
MinGW	Minimalist GNU for Windows
MIPS	Microprocessor without Interlocked Pipeline Stages
MIPS	Million Instructions Per Second
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
POSIX	Portable Operating System Interface for uniX

POWER	Performance Optimization With Enhanced RISC
RAM	Random Access Memory
RCS	Lehrstuhl für Realzeit-Computersysteme
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RT	Real-Time
RTOS	Real-Time Operating System
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SPI	Serial Peripheral Interface (bus)
SRAM	Static Random Access Memory

Chapter 1

Introduction

"I think there is a world market for maybe five computers."
Thomas Watson, chairman of IBM, 1943

"There is no reason anyone would want a computer in their home."
Ken Olson, president, chairman and founder of Digital Equipment Corp., 1977

Several years ago, computers were huge expensive machines doing things that were uninteresting for almost everyone. Only very few people were able to operate them and even fewer were able to access such a machine.

Nowadays almost everyone has a desktop computer at home and/or at work. Most people even have more than one computer without knowing about it. They are hidden in all kinds of devices as embedded devices. You can find them for instance in TV sets, washing machines or even in microwave ovens.
Every new car is equipped with a multitude of embedded systems, they control the engine (to reduce emissions and fuel consumption while maximising the output power), the anti-lock brakes, headlights, power windows and so on.

What Is an Embedded System?

An embedded system is a combination of computer hardware and software residing in a bigger device. It is completely encapsulated by that device, in several cases you do not even recognise that there is an embedded system in an appliance. Mostly the embedding device is controlled by these systems. Other tasks are doing the communication between multiple pieces of equipment or forming a man-machine interface for machines. Embedded systems perform only one or a few strictly predefined tasks, quite contrary to a desktop-computer.

The manufacturer of a general purpose computer (desktop) does not know, what the customer will do with it, possibly it is used as a server, a workstation or simply for playing. In contrast to that, both hardware and software of an embedded system are usually designed for only one specific application.
Surprisingly a desktop-system consists of several embedded systems. In almost every component works a small computer - keyboard, mouse, (hard-)drives, even in the monitor.

Embedded systems, which are used for controlling purposes, typically have stringent timing constraints. If an important calculation is not completed within a certain deadline, the controlled device (machine) or the part it is working on can be damaged or even people can get harmed. For example, it can end in disaster if the real-time system of an aeroplane's flight control system misses a deadline.
So real-time systems must reliably meet timing constraints within random asynchronous events (e.g. unpredictable values of environmental sensors). The more severe the consequences, the "harder" are these constraints (deadlines) and the device requires a hard real-time system. In reverse, weak timing constraints allow a soft real-time system.

By definition, embedded systems contain a processor and software. But there are more components that are mandatory. The software has to be stored permanently (e.g. in a ROM or a flash chip) and there has to be some RAM for storing runtime data. Depending on storage size requirements of the system, Memory can be on chip (microcontroller) or in additional circuits or external devices (hard-disk). Small devices may have less than 1kB of RAM and ROM, whilst more extensive systems may have several megabytes of these.
Besides memory, an embedded system needs in- and outputs as seen in Figure 1.1, e.g. to get sensor data, to interact with the user or to trigger actuators.

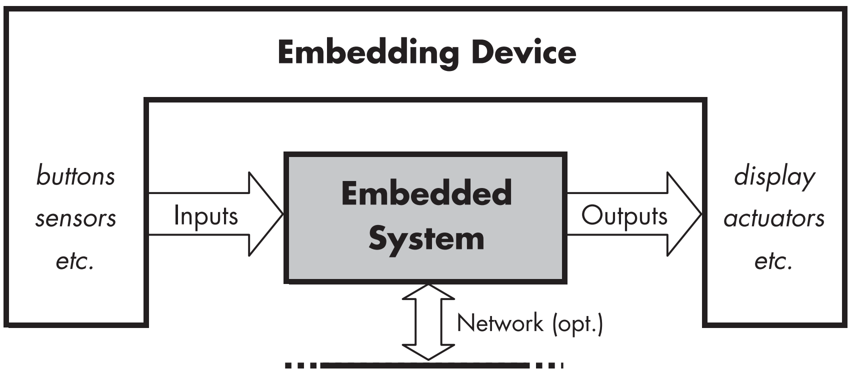


Figure 1.1: Setup of an embedded system *(based on [25] Fig. 1.1))*

Additional to these common features, the rest of the embedded hardware is usually unique to meet the particular requirements of the appliance. Some embedded systems have to communicate with other embedded systems or, for instance, with a control centre. This is typically done by using some kind of network. Automotive devices often use CAN, LIN and FlexRay in newer systems, in a industrial domain, there are field buses like LON (LonWorks) or Ethernet applied amongst others.

Other design requirements are:

- **Processing power:** Not only the number of operations per second (MIPS) are important. The register width also has to fit the requirements. Processors from 4 to 64 bit register width are currently used in embedded systems. For example a window lift controller is satisfied with 4 bit, an electronic engine control unit needs more, usually 16 or 32 bit.

- **Development cost and number of units:** These are costs of the hardware and software design process. As it is a fixed cost, it is not very important for high-volume products, on the other hand it may be the biggest expense factor if only a small number of units is produced.

- **Expected lifetime and reliability:** How long does the product have to continue to function - only some days or weeks or several years? How reliable does it have to be during this period?
  These questions have a big influence on all sorts of design decisions - from the selection of the components to the amount of expenses for development and production.

- **Electromagnetic compatibility (EMC):** Due to the close connection to the embedding device, an embedded device has particular requirements to the emission and susceptibility of electromagnetic disturbances.

- **Environmental requirements:** Embedded systems in so-called extreme hostile environments have to be resistant against several harmful influences. These may be humidity, dust, temperatures, vibration and shock. Particular in automotive uses, these factors are very important. You can see some temperatures that can occur in an ordinary car in figure 1.2.

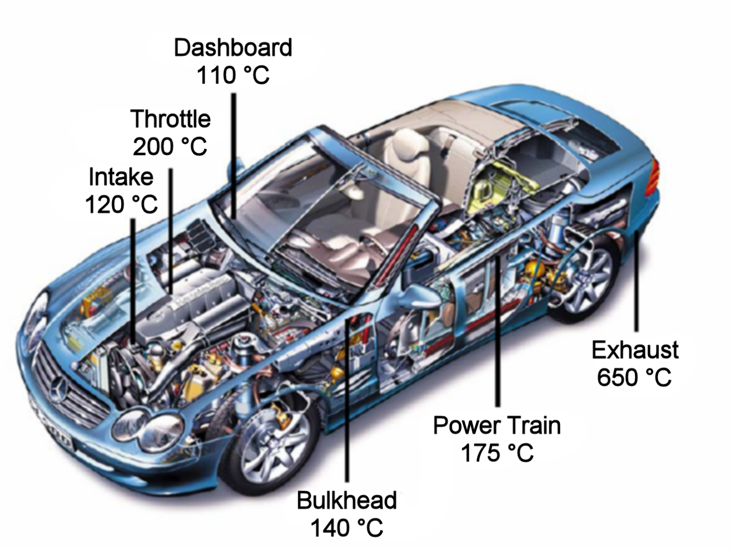


Figure 1.2: Temperature requirements in cars *(source: Daimler Chrysler AG)*

These general Requirements are expanded by detailed functional requirements of the system itself. For example portable devices have strong constraints concerning size, weight and electric power consumption (if it has to run on battery).

Chapter 2

Common CPU-Architectures for Embedded Systems

Normally the CPU for an embedded system is selected to meet the specific needs of the application. A processor's architecture has great influence in the aforementioned requirements - particularly in computing power and costs.

Some characteristic attributes of the architectures are listed in table 2.1.

2.1 x86

The x86 family was introduced by Intel and is typically known as architecture for desktop computers. But there are several x86-CPUs especially designed for embedded systems. A "standard" x86-CPU needs an external chipset that typically consists of North- and Southbridge in order to communicate with the memory or peripheral devices. This is too much dissipation of resources like space and power to use these standard x86 CPU for an average embedded device.
But there are special embedded x86 processors that combine the CPU, the chipset and some additional glue-logic in a single IC-package (cf. Figure 2.1). These devices are called

	x86	**PowerPC**	**MIPS**	**ARM**	**Cell**
Instruction Set	CISC	RISC	RISC	RISC	RISC
Max. Clock Rate	> 3.2 GHz	2.7 GHz	1 GHz	1.25 GHz	> 3.2 GHz
Implementation	16 - 64 bit	32 - 64 bit	32 - 64 bit	32 bit	64 bit (PPE) 128 bit (SPE)
Endianess	little	big *)	arbitrary *)	little *)	arbitrary *)

*) *switchable*

Table 2.1: CPU Architectures - Key Data

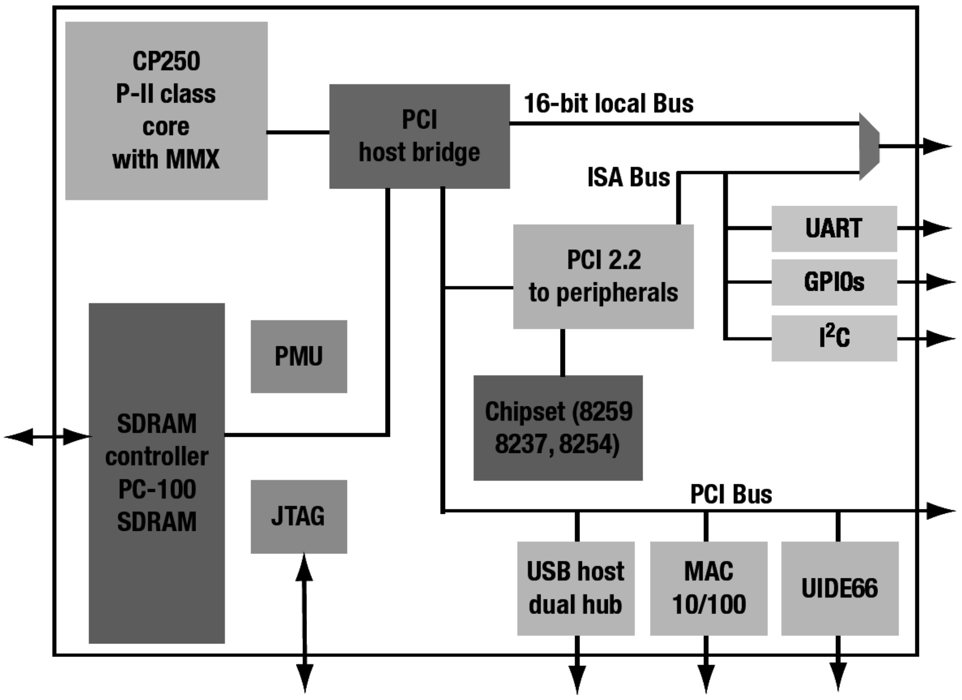


Figure 2.1: x86 System-on-Chip *(source: STMicoreletronics)*

System-on-Chip (SoC). Using a SoC, you will need almost no external circuitry to set up an embedded computer.

The x86 architecture is the most widely documented architecture around [30]. There are many books and online documents about the intricacies of this architecture. Development for x86 is rather easy - no cross development is needed when working on an Intel PC (like most developers do). Almost everyone has a computer of this architecture at work or at home.
However this architecture represents only a small fraction of the traditional embedded system market. One reason for this is the high cost of the x86-CPUs, another one is the relatively high power consumption compared to other architectures.

2.2 PowerPC

The PowerPC architecture is the result of collaboration between IBM, Motorola and Apple. While it has its origins in personal computers (Apple computers using Motorola 68k CPUs) and mainframes (IBM's POWER Architecture), PowerPC CPUs were also intended for the use as embedded and high-performance processors as well. [29]
PowerPC is largely based on IBM's earlier POWER architecture, and retains a high level of compatibility with it. These two architectures have remained close enough that the same programs and operating systems will run on both - if some care is taken in

preparation. Newer chips in the POWER series implement the full PowerPC instruction set.

PowerPC processors are known as CPU for Apple Macintosh computers, but they can also be found in many other devices, for example in the first edition of the TiVo hard-disk recorder or in several set-top boxes.

Similar to x86, this architecture is also very well documented and supported by various operating systems. [30]

2.3 MIPS

The MIPS Platform is the result of the Stanford "Microprocessor without Interlocked Pipeline Stages" project. The CPU core is licensed by MIPS Technologies Inc. as intellectual property to several manufacturers [30]
This architecture has many instruction set implementations, hence operating systems and software programs have to be built for a specific target board.

MIPS CPUs can be found in Sony's PlayStation 1 and 2, Nintendo 64, servers and workstations from SGI and many other devices like routers and other networking equipment.
According to MIPS technologies, MIPS products can be found in 72% of VoIP applications, 76% of cable set-top boxes, 70% of DVD recorders and 95% of cable modems *(source http://www.mips.com/)*.

2.4 ARM

The ARM (Advanced RISC Machine - formerly Acorn RISC Machine) architecture is sold as intellectual property to chip manufacturers just like the MIPS architecture. [30]
ARM Ltd. only designs CPU cores and licenses them to manufacturers such as Samsung, Atmel and many others. Intel sold its XScale family quite recently to Marvell Technology Group.

In difference to the MIPS architecture all ARM CPUs have the same instruction set, so all processors are fully software compatible (assembly and binary code).
Almost all current ARM CPUs are based on ARMv5TE and ARMv4 architectures. They only differ in clock rate, cache and additional features. [1]

Due to their power saving features, ARM CPUs are dominant in the mobile electronics market, where low power consumption is a critical design goal.
ARM Limited states that ARM's market share of the embedded RISC microprocessor Market is approximately 75%. Until January 2005, ARM Partners have shipped more than 2.5 billion ARM core-based microprocessors. [19]

Contrary to other vendors, ARM does not provide free manuals for its chips. There are only reference manuals available, which are sufficient in most cases. As individual chip manufacturers are responsible for the development of the chips, some information is available from them.

2.5 Cell Architecture

The Cell Architecture was developed by an alliance formed by Sony, Toshiba and IBM (STI) to provide power-efficient and cost-effective high-performance processing for a wide range of applications, especially the most demanding consumer appliance: game consoles [18]. For instance Sony employs Cell-processors for its new PlayStation 3 console.

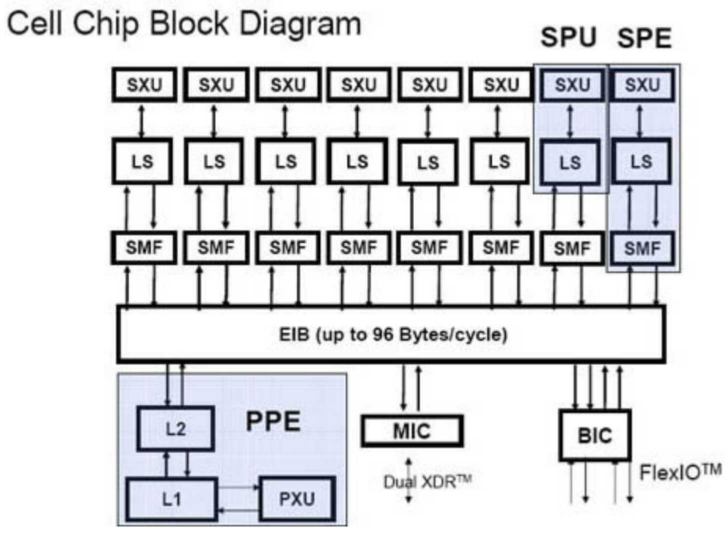


Figure 2.2: The Cell architecture *(source: IBM Research [18])*

As seen in Figure 2.2, a current cell CPU consists of an 64-bit Power Architecture core (Power Processor Element - PPE) and 8 specialised co-processors based on a novel single-instruction multiple-data (SIMD) architecture called Synergistic Processor Unit (SPU). These cores are connected to each other and to the outside (RAM, I/O) through an on-chip bus (EIB).
This configuration combines the flexibility of a normal CPU core with the functionality and performance of parallel optimized SIMD cores.

2.6 Microcontrollers

There are reams of different microcontroller-architectures available on the market. Some of them are based on one of the architectures mentioned above, but typically a microcontroller has a small CPU core with 4, 8 or 16 bit register width, only a few MHz clock rate and only some kilobytes of RAM and ROM.

Due to these limits, microcontrollers are usually programmed directly, without the assistance of an operating system. Only in very special appliances a small real-time OS is used, e.g. if multi-tasking is needed.

These chips are virtually always integrated in embedded systems, as there are no real applications for stand-alone microcontrollers.

Microcontrollers are usually equipped with additional peripheral devices on chip such as signal converters (A/D, D/A), timers and counters (that may be used for PWM), watchdogs and various interfaces (e.g RS-232, SPI, CAN and I^{2}C).
Due to these extensions the chip-count in an embedded appliance and thus the system's size may be reduced.

A very popular microcontroller family is the Freescale MC68300 (formerly from Motorola). The CPU core is based on the 32-bit Motorola 68k architecture. For instance EyeBot M5 utilises an MC68332 which is equipped with 2kB internal SRAM and numerous I/O-facilities; non-volatile memory has to be added externally.
An interesting feature of this controller is "fully static operation" [7], i.e. it can operate on any clock frequency from 0 Hz up to the maximum of about 20 MHz. This feature may be used for debugging purposes - you may run your program slowly (or step through the program) and show the status of the system with a few LEDs or a numeric LED display connected to the microcontroller.

2.7 DSPs or FPGAs as Co-processor

In some cases, more processing power is needed than a single embedded CPU can deliver. Simply using more than one processor in parallel possibly bears conflicts with the requirements of an embedded system. Two identical processors on one board doubles the processing power in ideal case, but also doubles power consumption and increases the size of the board.
Using even more processors certainly increases computing power but also increases the disadvantages of this concept.

A different approach is to support the main processor of an embedded system with a specialised processing unit such as a DSP or a FPGA. The combination of a "normal" and a "specialised" CPU has several benefits. While the DSP or FPGA is used for signal processing, the main CPU is available for other tasks like communicating with the embedding hardware, other embedded systems or the user.

In contrast to a DSP that usually can only be used for sequential signal processing, a (re)configurable FPGA can be used in many different ways. [23, 22]
You may load a CPU core into the FPGA, so it can act as a full-featured CPU. Such cores (also referred to as "softcores") are available as intellectual property and may be

bought for varying amounts of money (depending on the complexity). Another resource is OPENCORES (opencores.org), an open source project providing numerous CPU cores for many different applications.

Digital image processing is an ideal task for parallelisation. Common Procedures of image processing like filtering and resizing involve the same simple mathematical operation repeatedly on every pixel. By doing these operations in parallel, the time needed for the computation can be heavily reduced.
While a DSP or a normal CPU fetches and processes one part (pixel) of the image per time unit, a FPGA can do several of these operations in parallel by using several processing units. The number of parallel "processing units" depends on the number of logic cells in the FPGA and the complexity of the operation which affects size of each "softcore".
Employing a FPGA allows complex image processing in real-time without generating any CPU load on the main processor of the system.

The Eyebot M6 makes use of a common Xilinx FPGA from the Spartan 3E family for supporting the main processor in image processing tasks.
Another FPGA family from Xilinx is called Virtex-4. These devices combine the benefits of "usual" processors and FPGAs by combining one or two PowerPC CPU "hardcores" and a FPGA in the same package. [13]

Chapter 3

Common Operating Systems for Embedded Systems

Embedded systems are becoming increasingly complex. Simple microcontroller circuits may be programmed easily with an assembly language or C. But if you wish to use advanced features like multitasking, you will soon face the limitations of this approach. Implementing more advanced features by hand is elaborate and maybe error-prone. So using an operating system that provides these features is reasonable. Most operating systems have already absolved comprehensive tests, because typically they are already in use in many other projects.

In the following sections, there is a survey of the most common operating systems for embedded systems.

3.1 Linux

Almost everyone in the computer business knows the history of Linux - started in 1991 by Linus Torvalds as a simple hobby project, grown-up to a full-featured UNIX-like operating system.
The name Linux is interchangeably used in reference to the Linux kernel, a Linux system or a Linux Distribution. Strictly speaking, Linux refers only to the kernel, but in colloquial language use Linux means usually a Linux system. Such systems may be custom built (from the sources) or can be based on an already available binary distribution such as Debian or Novell SUSE.

Linux developed as a (more or less) POSIX-conform general purpose operating system. Due to the immense cost if a POSIX certification, only very few distributions are certified [6]. There are still some issues that do not fully comply POSIX standards such as threading.
Linux is a multi-user system, which is suitable for any kind of application (multi-func-

tional). Thus it is a big system that needs lots of resources in terms of memory and processing power and the scheduler is based on ”fairness” instead of real-time aspects. It seems to be the direct opposite of an operating system for embedded systems.

A typical desktop Linux installation usually needs several hundreds of megabytes of disk space and at least 32 MB RAM. Embedded targets are often limited to very few megabytes of flash or ROM and only some megabytes of RAM. But due to the modularity and scalability of Linux it can be adapted to fit almost any embedded system.
Much of the several hundred megabytes of the desktop distribution are composed of documentation, desktop utilities etc. and can be omitted as they are unnecessary for an embedded target. It is absolutely possible to build a fully-functional Linux system needing less than 2 MB of non-volatile memory.
Even the kernel itself is highly configurable and it is possible to remove unneeded kernel functionality with the assistance of several built-in frontends.

Linux is available for almost every 32-bit architecture and many 64-bit architectures. For a list you may look in the directory `arch/` in the Linux sources.
Even some 16-bit x86-processors (e.g. 8086 and 286) are supported by a project called ELKS *(http://elks.sourceforge.net/)*.

Traditional embedded systems operating systems have three components of software cost:

- **initial development setup** - purchase of development licenses from the OS vendor (often ”per seat” - one license for each developer)
- **additional tools** - if the tools provided by the basic developer package are insufficient
- **runtime royalties** - per-unit royalty when deploying the system

With Linux, all development tools and OS components (including the sources) are available free of charge and any royalties are prevented by the licenses.

3.2 Microsoft Windows Systems

At present, there are two operating systems for embedded systems from Microsoft - Windows CE and Windows XP Embedded. [5, 15, 16]
Windows CE (WinCE) is an operating system for minimalistic computers and embedded systems. It is not a smaller version of a desktop Windows, instead, it is a distinctly different kernel. It supports Intel x86 and compatibles, MIPS, ARM, and Hitachi SuperH processors.
Windows CE is optimized for devices that have minimal storage - a Windows CE kernel may run in under a megabyte of memory.
Windows CE conforms to the definition of a real-time operating system, with deterministic interrupt latency. It supports 256 priority levels and uses priority inheritance for dealing with priority inversion.
Similar to Linux, Windows CE forms only the kernel of the OS. By adding extra software

such as a graphical user interface, it becomes a ”complete” operating system called e.g. Windows Mobile.

Windows XP Embedded, or XPe, is a modularised variant of Microsoft Windows XP Professional. It runs normal Windows (resp. Win32) applications and device drivers. It is only compatible to x86 processors and needs at least 32MB Flash, 32MB RAM and a 200 MHz CPU.
XPe is not related to Windows CE. They target completely different devices. Windows XPe will not run on other architectures or with as little resources as CE. But the latter does not have the Win32 API and needs its own software and drivers.

Windows XPe is used for instance in ATMs and Thin Clients, whilst CE is normally used in mobile phones and PDAs.

3.3 Symbian

Symbian OS is the successor of 32-bit EPOC Platform from Psion. Symbian is currently owned by Ericsson (15.6%), Nokia (47.9%), Panasonic (10.5%), Samsung (4.5%), Siemens AG (8.4%), and Sony Ericsson (13.1%). All of the owners are (or were) manufacturers of mobile phones. [12]

Symbian is structured like many desktop operating systems with pre-emptive multitasking, multithreading and memory protection. Its kernel is a microkernel architecture, which means that only the minimum necessary is within the kernel. Things like networking or file system support have to be provided by another layer called base layer. Between base layer and user software are system libraries.
The most important user interfaces based on Symbian are S60 (Nokia) and UIQ (Sony Ericsson).

The major advantage of this operating system is the fact that it was built for handheld devices with limited resources that may be running for months or years. It has programming idioms such as descriptors and a cleanup stack and other techniques in order to conserve RAM and avoid memory leaks. There are similar functions to save disk space (flash memory).

All Symbian OS programming is event-based and the CPU is switched off when applications are not directly dealing with an event. This is achieved through a programming idiom called active objects. Correct use of these techniques helps ensure longer battery life.

Today, Symbian OS is solely employed in mobile phones (i.e. smartphones).

3.4 Real-Time Operating Systems

Real-time operating systems (RTOS) are operating systems intended for real-time applications. That is to say, a RTOS guarantees deadlines to be met generally (soft real-time) or deterministically (hard real-time).

An RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time - this requires correct development of the software. An RTOS will typically use specialised scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behaviour in the final system.

There are numerous proprietary and free RTOSes available. See the following list for some examples.

- **VxWorks** is the most popular commercial RTOS [4]. Like most other RTOSes it includes a multitasking kernel with pre-emptive scheduling and fast interrupt response, extensive inter-process communications, synchronization facilities and a file system. [14]
  Major distinguishing features of VxWorks include efficient POSIX-compliant memory management, multiprocessor facilities, a shell for user interface, symbolic and source level debugging capabilities and performance monitoring.
- **QNX** is a commercial POSIX-conform RTOS [10]. It uses a microkernel, enabling the user (developer) to turn off any functionality he does not require. It offers features such as fault tolerance, pre-emptive multitasking and runtime memory protection. The system is quite fast and small, in a minimal fashion it fits on a single floppy disk.
- **RTEMS** is a free RTOS designed to support various open API standards including POSIX and uITRON. It was originally planned to be used for missiles and other military systems. RTEMS closely corresponds to POSIX Profile 52 which is "single process, threads, file system" [9] - it does not provide any form of memory management or processes. There is only a single process with multiple threads running on an RTEMS system. No services like memory mapping, process forking, or shared memory are offered.
- **RTAI, Xenomai and RTLinux** are extensions to the Linux kernel in order to allow Linux to meet real-time requirements. These systems run the Linux kernel as a low-priority task, so higher priority real-time tasks can interrupt the execution of the Linux kernel. The architecture of these real-time extensions is described in 4.3.

Chapter 4

Embedded Linux - Kernel and Standard Libraries

You may ask yourself: "What's the difference between Linux and embedded Linux?" Paradoxically there is none. It is the same Linux - the difference is founded in the configuration. An embedded Linux is typically aligned to the hardware and the application it is intended to be used with, while a "normal" Linux system is commonly more generic. In most cases, a "normal" Linux kernel is able to run on different machines; an embedded Linux kernel may get into difficulties when doing so.

Linux has been developed under the philosophy of *Open Source* software pioneered by the Free Software Foundation *(http://www.fsf.org/)*. So it is freely available to use, to modify and to copy. Free does not necessarily free of copyrights and free of charge. There are several companies that sell packaged Linux distributions. The customer pays for the services of the vendor (e.g. technical support, data mediums) and maybe for some vendor specific closed-source Software, but not for Linux itself.

When released under the terms of the GNU General Public License (GPL), Open Source code is copyrighted by its author who has released it. Unlike most End User License Agreements (EULA) that restrict rights, the GPL guarantees rights to the users and developers.
The onliy restriction is, if you modify and distribute software covered by the GPL, you have to make public the modified source code. This also applies if your code is based on GPL'ed software. Your code becomes "derivative work", which is also covered by the GPL.
If you want to write a proprietary application using GPL libraries, you will have to use libraries, which are subject to the "Library GPL" (LGPL - also "Lesser GPL" termed). Programs linked to a LGPL library are not considered as derivative work, so there is no requirement to publish the source of the program, but the library's source has to be distributed.

One benefit of freely available sources is the possibility to analyse erroneous software.

That eases correcting the code or finding workarounds. With traditional embedded operating systems, the code is not available or has to be purchased for large sums of money. Developers have to wait for fixes or spend a lot of money.

More reasons to employ Linux are:

- **Hardware support** - Linux supports most different types of hardware platforms of all operating systems. Linux also supports all kinds of devices such as controllers, network interfaces or graphics cards. There are only few manufacturers not providing Linux drivers, but there are a number of developers caring about that by reverse engineering drivers from other operating systems.
- **Communication** - Linux can interact with a huge variety of different systems off-the-shelf (Windows and UNIX systems via Ethernet; I^{2}C, SPI etc.)
- **Numerous freely available tools**
- **Community support** - if you have issues with some Linux software or driver, you can get support by posting in the development and support mailing lists. Often the developers themselves are present in these mailing lists and the level of expertise found there often surpasses what can be found over expensive support hotlines of vendors.
- **Vendor independence** - you are not bound to a specific vendor of an operating system.

4.1 Structure of Linux (UNIX)

Linux has a protected mode architecture using the protected mode memory implementation in contemporary Intel processors since 80386. The processor can operate in four privilege levels. A program running at the highest level (0) can do anything it wants (I/O Instructions, enable and disable interrupts, modify descriptor tables). Lower privilege levels prevent programs from performing operations that might be "dangerous".
Linux uses the highest and the lowest level (0 and 3). In Linux context, level 0 is called "kernel space" and level 3 is called "user space".

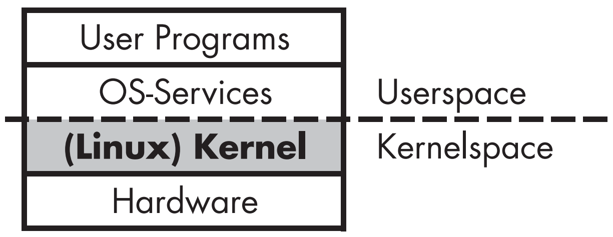


Figure 4.1: Architecture of a UNIX system

Figure 4.1 shows the typical architecture of a UNIX system. The two aforementioned layers are subdivided each into two different layers.
The bottom layer consists of the hardware controllers. This subsystem is comprised of all the possible physical devices in a Linux/UNIX installation. For instance, CPU, memory hardware, hard disks and network hardware are all parts of this subsystem).
The layer above is the kernel (e.g. Linux, BSD, etc.), which abstracts and mediates access to the hardware resources including the CPU. Possible kernels are Linux, BSD and so on.
The user space is composed of the OS-services and user applications. The OS- services are services that are typically considered part of the operating system (a windowing system, command shell, etc.). Also, the programming interface to the kernel (compiler tool and C-library) is included in this subsystem.

4.1.1 Kernel

The Linux kernel is a monolithic kernel; the tight internal integration of components make a monolithic kernel highly efficient. It runs solely in kernel space in supervisor mode. It defines a high-level virtual interface for accessing the computer's hardware.

The kernel's primary purpose is to manage the computer's resources and allow other programs to run and to use these resources. Typically, these resources consist of:

- The **CPU** is the most central part of a computer system, responsible for running or executing programs. The kernel takes responsibility for deciding which of the running processes should be allocated to the processor. Because Linux is a preemptive multitasking system, a scheduling algorithm is used for task selection.
- **Memory** is used to store program instructions and data. Both need to be present in memory in order for a program or process to execute. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough memory is available (swapping).
- Any **Input/Output (I/O)** device that is present in the computer, such as disk drives, printers, displays, etc. The kernel allocates requests from applications to perform I/O to an appropriate device and provides convenient methods for using the device.

To enable processes to access the services provided by the kernel, the C-Library of the kernel offers routines for the kernel's system calls. All kernel functions are being invoked by using these system calls.
The kernel also provides methods for synchronization and communication between processes called inter-process communication or IPC.

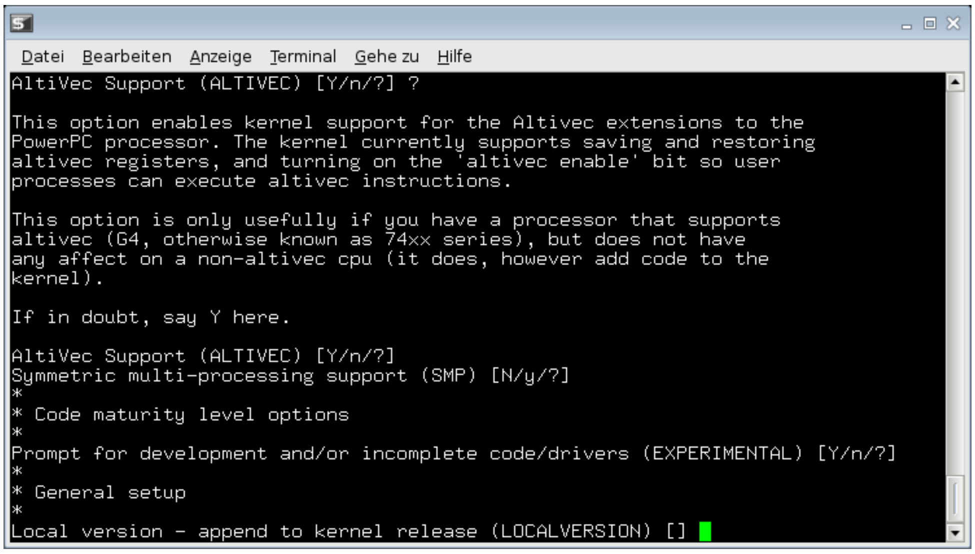
It is possible to dynamically load and unload executable kernel modules at runtime. So it is possible to easily extend the kernel's capabilities when required. For instance hardware modules may be loaded just before the hardware is required.
Kernel modules add a small overhead as opposed to code being directly built into the kernel. Despite that fact, only loading modules when they are needed helps to keep the

amount of code running in kernel space to a minimum.
Typically kernel modules can only be loaded into kernels they have been built for. So it is necessary to have the sources of all kernel modules that are needed for your system in order to build a stable Linux kernel.
Some manufacturers only offer closed-source drivers for their hardware making it difficult to build a kernel that is perfectly adapted to your system. Pre-built kernel modules often cause stability issues, though there is a kernel feature allowing you to load modules that were built for a different kernel.

4.1.2 Kernel Configuration

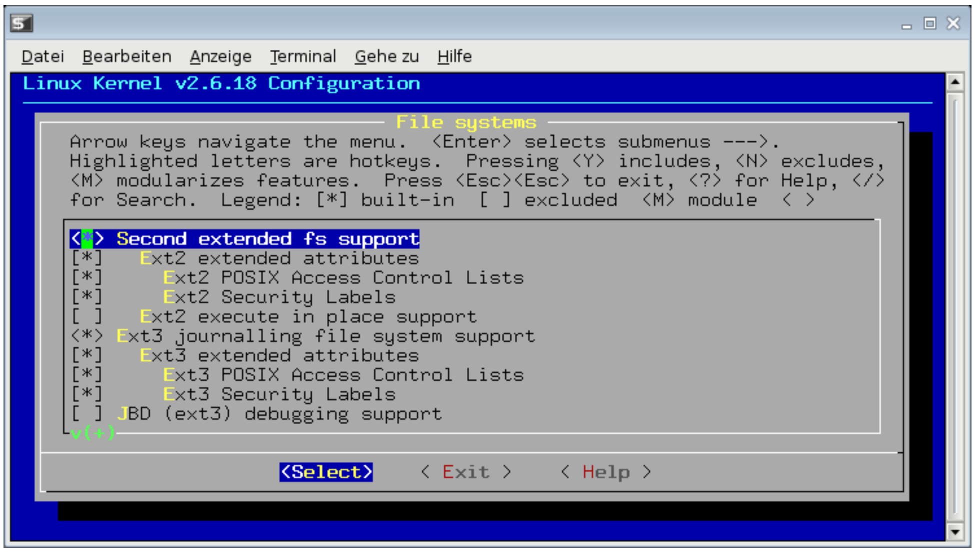
The Linux kernel is highly configurable. It is possible to select the features supported by your processor, device drivers, file systems, buses and so on. In this way, you can build a kernel that is ideally customised for your appliance.

In most cases you can decide whether a kernel component should be integrated into the kernel or if it should be compiled as a module.
There are several user-friendly tools to ease the configuration process. They can be invoked by executing `make config` (Figure 4.2), `make menuconfig` (Figure 4.3 - ncurses based), `make xconfig` (Figure 4.4 - QT frontend) or `make gconfig` (Figure 4.5 - GTK frontend) in the kernel source top directory.



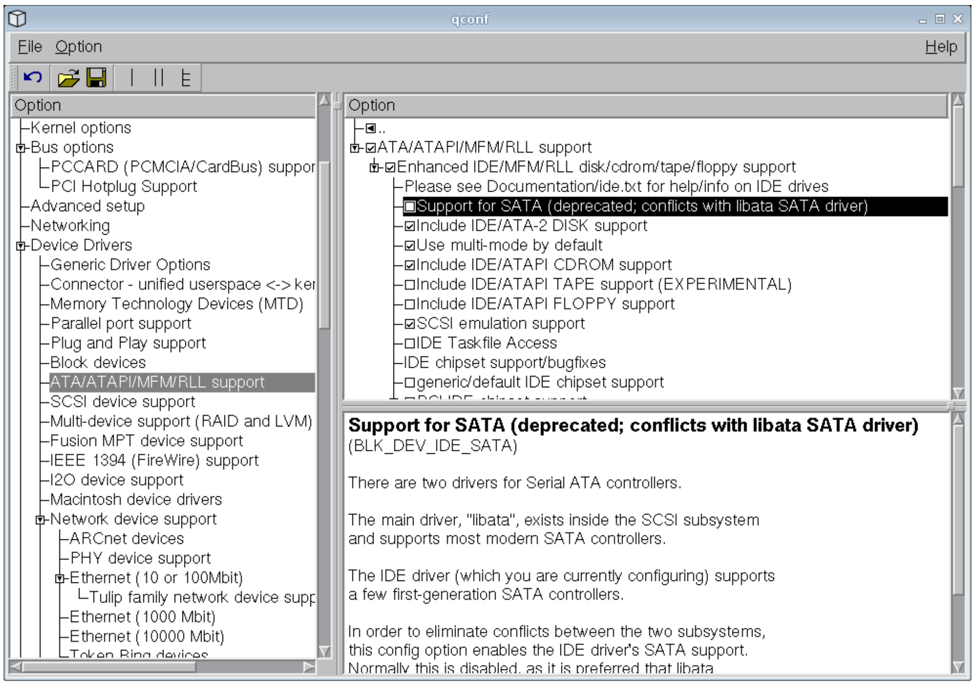
```
AltiVec Support (ALTIVEC) [Y/n/?] ?

This option enables kernel support for the Altivec extensions to the
PowerPC processor. The kernel currently supports saving and restoring
altivec registers, and turning on the 'altivec enable' bit so user
processes can execute altivec instructions.

This option is only usefully if you have a processor that supports
altivec (G4, otherwise known as 74xx series), but does not have
any affect on a non-altivec cpu (it does, however add code to the
kernel).

If in doubt, say Y here.

AltiVec Support (ALTIVEC) [Y/n/?]
Symmetric multi-processing support (SMP) [N/y/?]
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?]
*
* General setup
*
Local version - append to kernel release (LOCALVERSION) []
```

Figure 4.2: Configuration of the Linux kernel - `make config`



```
Linux Kernel v2.6.18 Configuration
                          File systems
  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, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

  <*> Second extended fs support
  [*]   Ext2 extended attributes
  [*]     Ext2 POSIX Access Control Lists
  [*]     Ext2 Security Labels
  [ ]   Ext2 execute in place support
  <*> Ext3 journalling file system support
  [*]   Ext3 extended attributes
  [*]     Ext3 POSIX Access Control Lists
  [*]     Ext3 Security Labels
  [ ] JBD (ext3) debugging support
  v(+)

              <Select>    < Exit >    < Help >
```

Figure 4.3: Configuration of the Linux kernel - `make menuconfig`



```
Option
├Kernel options
⊞Bus options
│ ├PCCARD (PCMCIA/CardBus) suppor
│ └PCI Hotplug Support
├Advanced setup
├Networking
⊞Device Drivers
│ ├Generic Driver Options
│ ├Connector - unified userspace <-> ker
│ ├Memory Technology Devices (MTD)
│ ├Parallel port support
│ ├Plug and Play support
│ ├Block devices
│ ├ATA/ATAPI/MFM/RLL support
│ ├SCSI device support
│ ├Multi-device support (RAID and LVM)
│ ├Fusion MPT device support
│ ├IEEE 1394 (FireWire) support
│ ├I2O device support
│ ├Macintosh device drivers
│ ⊞Network device support
│ │ ├ARCnet devices
│ │ ├PHY device support
│ │ ⊞Ethernet (10 or 100Mbit)
│ │ │ └Tulip family network device supp
│ │ ├Ethernet (1000 Mbit)
│ │ ├Ethernet (10000 Mbit)
│ │ └Token Ring devices
```

```
Option
├..
⊞☑ATA/ATAPI/MFM/RLL support
  ⊞☑Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support
    ├Please see Documentation/ide.txt for help/info on IDE drives
    ├☐Support for SATA (deprecated; conflicts with libata SATA driver)
    ├☑Include IDE/ATA-2 DISK support
    ├☑Use multi-mode by default
    ├☑Include IDE/ATAPI CDROM support
    ├☐Include IDE/ATAPI TAPE support (EXPERIMENTAL)
    ├☐Include IDE/ATAPI FLOPPY support
    ├☑SCSI emulation support
    ├☐IDE Taskfile Access
    ├IDE chipset support/bugfixes
    ├☐generic/default IDE chipset support
```

Support for SATA (deprecated; conflicts with libata SATA driver)
(BLK_DEV_IDE_SATA)

There are two drivers for Serial ATA controllers.

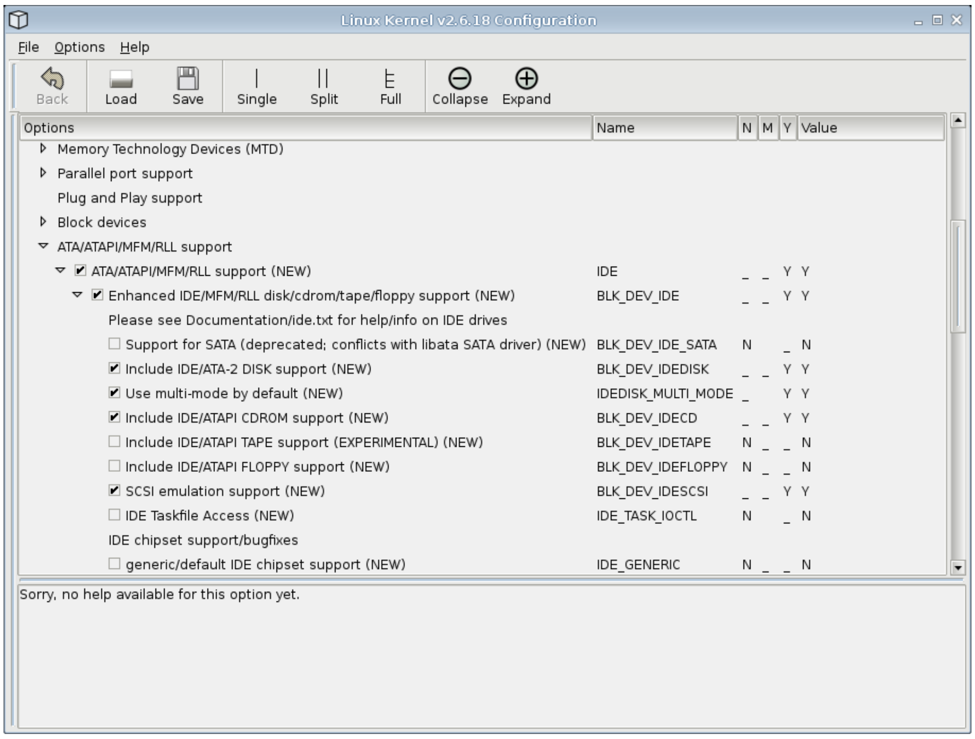
The main driver, "libata", exists inside the SCSI subsystem
and supports most modern SATA controllers.

The IDE driver (which you are currently configuring) supports
a few first-generation SATA controllers.

In order to eliminate conflicts between the two subsystems,
this config option enables the IDE driver's SATA support.
Normally this is disabled, as it is preferred that libata

Figure 4.4: Configuration of the Linux kernel - `make xconfig`

When using `make config`, there are three or four choices for each kernel option. They are "y" (yes - build the option into the kernel image), "n" (no - do not build the option), "m" (build as loadable module - not for all options) and "?" (print help text). The default selection is printed in a capital.



Options	Name	N	M	Y	Value
▷ Memory Technology Devices (MTD)					
▷ Parallel port support					
Plug and Play support					
▷ Block devices					
▽ ATA/ATAPI/MFM/RLL support					
▽ ✓ ATA/ATAPI/MFM/RLL support (NEW)	IDE	_	_	Y	Y
▽ ✓ Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support (NEW)	BLK_DEV_IDE	_	_	Y	Y
Please see Documentation/ide.txt for help/info on IDE drives					
☐ Support for SATA (deprecated; conflicts with libata SATA driver) (NEW)	BLK_DEV_IDE_SATA	N		_	N
✓ Include IDE/ATA-2 DISK support (NEW)	BLK_DEV_IDEDISK	_	_	Y	Y
✓ Use multi-mode by default (NEW)	IDEDISK_MULTI_MODE	_		Y	Y
✓ Include IDE/ATAPI CDROM support (NEW)	BLK_DEV_IDECD	_	_	Y	Y
☐ Include IDE/ATAPI TAPE support (EXPERIMENTAL) (NEW)	BLK_DEV_IDETAPE	N	_	_	N
☐ Include IDE/ATAPI FLOPPY support (NEW)	BLK_DEV_IDEFLOPPY	N	_	_	N
✓ SCSI emulation support (NEW)	BLK_DEV_IDESCSI	_	_	Y	Y
☐ IDE Taskfile Access (NEW)	IDE_TASK_IOCTL	N		_	N
IDE chipset support/bugfixes					
☐ generic/default IDE chipset support (NEW)	IDE_GENERIC	N	_	_	N

Sorry, no help available for this option yet.

Figure 4.5: Configuration of the Linux kernel - `make gconfig`

With `make config` you can only change one option after the other; there is no possibility to go back. So usually several passes are needed - that is tedious.
Fortunately the other methods are menu-based and at least one of them runs on almost every computer.
All of these tools create or change the *.config* file in the kernel-source's top-level-directory.

After the configuration is completed the new kernel can be built by invoking the `make` command.

When you are building a kernel for a different target platform (than your host system), you will have to pass additional arguments such as the name of the architecture and the prefix of the toolchain programs to the make commands.
For instance to build a kernel for an ARM-based *gumstix* board the command to run is: `make ARCH=arm CROSS_COMPILE=arm-linux-` ***`target`***, where target is either one of config, menuconfig, xconfig, gconfig or nothing when starting the build process.

4.1.3 The Linux Process Model

The basic structural element in Linux is a process. It consists of executable code and resources like data, file descriptors and so on. These resources are fully protected; one process cannot access or manipulate the resources of another. So a erroneous process cannot harm any other running processes.

Due to that, communication between processes can only be done by using inter-process communication (IPC) mechanisms provided by the kernel as seen in Figure 4.6. IPC means are for instance shared memory regions and named pipes. The creation of processes and using inter-process communication bares an excessive overhead compared to the threading model.

Threads are an alternative to processes - threading is also called "lightweight multitasking". In contrast to a process, a thread is only code; it only exists within the context of the invoking process and shares its resources. All threads of a process have equal access to data memory and file descriptors. Communication between threads is much more efficient (e.g. simply by using global variables), but the drawback is: Any thread can damage another thread's data.

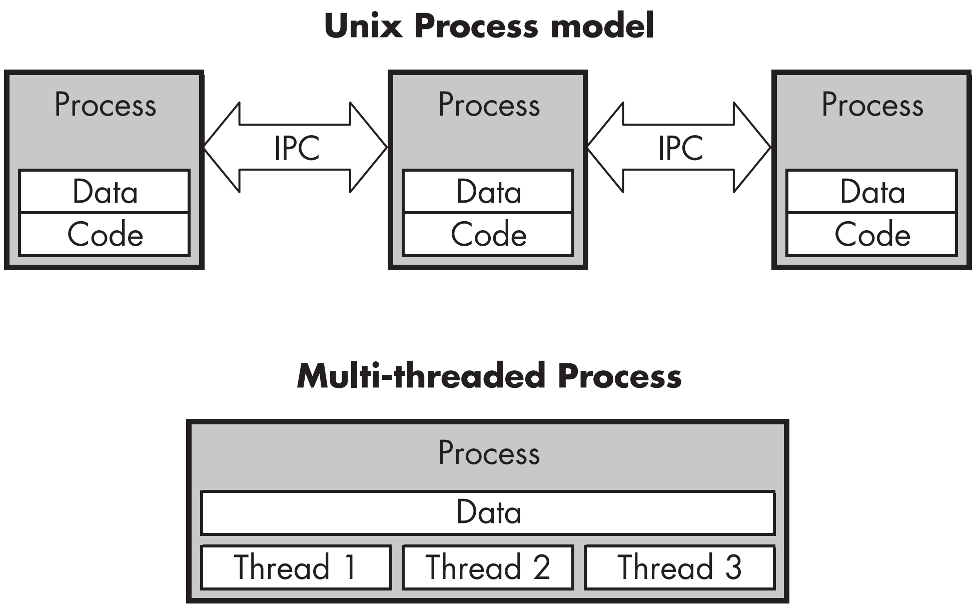


Figure 4.6: Processes and threads *(based on: [20] Figure 2-5)*

Linux starts with one process, the init process, which is created at boot time. Every other process in the system is a child of that process. The creation of a child process is done by the system call `fork()`.
`fork()` makes a copy of the parent process and gives a new process id (PID) to the child process. To save memory and to reduce overhead, the data of the processes is only duplicated when it is accessed.

In most cases, the child process invokes a new program by calling `execve()` to load an executable file image from disk. `execve()` overwrites the calling process's code, data and stack segments. If `execve()` succeeds it does not return, as control is transferred to the newly loaded program.

If the child process is a foreground process, the parent process (usually the command interpreter) has to wait for its completion. This is accomplished with waitpid() which

blocks the calling process until the child process has completed.
If the child process is a background process, nevertheless waitpid() has to be invoked by the parent process after completion. When omitted, the child process will reside in the system as a "zombie" process.

4.1.4 Connections Between Kernel Space and User Space

Because of the absolute detachment of kernel Space and User Space, there are communication means necessary in order to connect user processes to the kernel.
Therefore the kernel provides virtual filesystems and special files. These can be found on any Linux system (anyway since Linux version 2.6). Device nodes are usually located in /dev/, the proc filesystem is typically mounted in /proc and the sys filesystem in /sys.

Device-Nodes

A device node is a special file facilitating communication between user space applications and computer hardware. A device node corresponds to hardware resources that have already been allocated by the kernel. The resources are identified by a major number representing the device driver and a minor number representing the device (e.g. major number 4 → serial port; minor number 64 → first port, 65 → second port and so on).
A device node is treated like any other file, and is accessed by using standard system calls. There are three kinds of device files corresponding to different device types.

- **Character devices** transmit only single characters per time unit. These device nodes are used e.g. for serial ports, modems or virtual terminals. Character devices are usually unbuffered.
- **Block devices** transmit data in data blocks. For instance hard drives are represented by block device nodes. The kernel allocates input and output buffers for this kind of device. User programs write into and read out of these buffers. When a buffer is full, the complete buffer (block) is transmitted and the buffer is cleared.
- **Pseudo devices** are devices that do not represent hardware. They are used for various functions handled by the kernel. The best known pseudo-devices are `/dev/null`, `/dev/random` and `/dev/zero`.

For the Linux 2.6 kernel series the device nodes are handled by the device manager udev. It invokes all necessary actions when adding or removing devices. In a traditional Linux system there is a static set of device nodes in the `/dev` directory. With udev these special files are dynamically created (and removed) according to predefined rules. So there are only device nodes for all devices that are actually present on the system.

Proc - Filesystem

The `/proc` directory contains virtual files that are windows into the current state of the running Linux kernel. The proc filesystem is a pseudo-filesystem residing in system memory usually mounted at `/proc`. It supplies information on the kernel, the modules and the running processes (in `/proc/`*`PID`*). In `/proc/sys` dynamically configurable kernel options can be accessed (e.g. enable network packet forwarding). Some additional information about the system state is also available (e.g. in `/proc/meminfo` or `/proc/cpuinfo`).

Sys - File System

SysFS is a virtual file system provided by the Linux 2.6 kernel. Information about devices, drivers and relations between both are exported to this file system. It is used by several utilities such as udev to collect information about the hardware and its drivers (kernel modules).

4.2 Standard C-Library

The standard C library is a collection of header files and library routines that are used to implement common operations like input, output or string handling and it provides an interface to the kernel's system calls.
Almost all C programs need a standard C library to function.

4.2.1 GNU C Library

The GNU C library (or glibc) is a standard C library released in subject to the LGPL. It provides functionality required by the Single UNIX Specification, POSIX, ISO C99 and some additional extensions for the GNU development. It supports many different kernels and numerous hardware architectures.
It is the most common libc for Linux systems.

Unfortunately it is very large and considered being slow. So there are several approaches implementing a small more effective libc, especially in the view of embedded Linux systems.

4.2.2 uClibc

The uClibc (or μClibc) is a C library developed especially for embedded Linux systems (uC stands for microcontroller). It is has its origins in the uCLinux project that provides a Linux distribution for processors without an MMU. It was composed from parts of libc4,

glibc and many other sources.
The library has become a project of its own and supports many architectures with or without MMU and FPU. The developers go for maintaining compatibility with C89, C99 and SUSv3 (Single UNIX Specification Version 3).

The library provides most of the functionality the glibc does. Although it is not as complete as the glibc, most applications that compile with glibc also run in combination with uCLibc.

For instance, a "Hello World" program linked with uClibc uses only 4 kB of disk space, while it uses more than 200 kB when linked with glibc. [24]

Like glibc, uClibc is released in subject to the GNU LGPL. So it is possible to publish proprietary software that uses this library.

4.2.3 Dietlibc

Dietlibc is another C library for embedded systems. It is released under the GNU General Public License Version 2, so there is nor room for using it in proprietary applications. It aims to be compliant to SUSv2 and POSIX standards.

Dietlibc was developed from scratch with an emphasis in minimizing size and optimizing performance. Thus it implements the most important and commonly used functions - "implement stuff because it's needed, not because it's there". [27]
For example all modules (e.g. stdio and unistd) are strictly separated and most functions are compiled into separate object files, allowing the linker to reduce the size of an application by only linking the used functions into the binary. Another very important point is that the dietlibc developers reused code wherever possible. Because of these techniques the resulting binaries are about 17 to 20 times smaller and contain almost no redundancy compared to binaries produced with glibc.

But there are several drawbacks using this library besides the licensing issue. Once, there are some known, but still unfixed bugs in this library. Furthermore, there is a lot of functionality missing compared to the aforementioned libraries and thus it is not an all-purpose library at all.

4.3 Real-Time Extensions for the Linux Kernel

In embedded systems, real-time capabilities are required in many cases. Unfortunately Linux is everything but a real-time operating system.
There are two different approaches to provide real-time performance when using Linux as operating system [28]:

- Improving the Linux kernel preemption.

- Adding a new software layer beneath Linux kernel with full control of interrupts and processor key features (cf. Figure 4.7).

"The Linux kernel preemption project" is working on the first item. The developers are still faced wit "a few" unacceptable latencies. [4]
A commercial project called TimeSys Linux is also working on this approach. TimeSys adverts with needing only a very small size of 500 kB - 1.2 MB for its operating system, the scheduler offers more priority levels and enforced CPU reservations and a better timer resolution than standard Linux does.

The other approach has been taken by RTAI, Xenomai and RTLinux. These projects differ in licensing issues, supported hardware platforms and kernel versions, the development status and the API. For instance RTAI and Xenomai are being actively developed, while the free variant RTLinux stalled at version 3.1 in 2001.

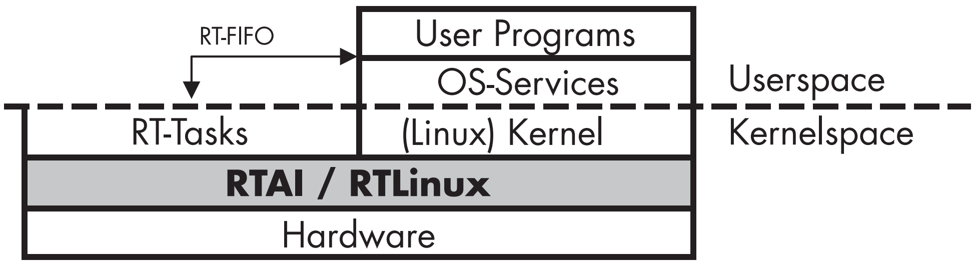


Figure 4.7: Architecture of a Linux system with real-time extensions

Within the additional layer that can be seen in Figure 4.7, there is a real-time kernel with its own scheduler. The Linux kernel is started as lowest priority real-time task - it is safe to say, it runs as idle process. The real-time layer is inactive until the corresponding module is loaded into the Linux kernel. So real-time functionality can be started and suspended as needed.

In the following RTAI is taken as an example, but Xenomai and RTLinux are very similar. The execution of tasks in RTAI is controlled by a timer which can operate in two modes. In the periodic mode, the timer is restarted automatically after generating an interrupt. In the one-shot mode the timer has to be set manually. The latter is more extensive but the timer can be set to alternating intervals.
RTAI provides all usual means for inter-task communication such as mailboxes, semaphores, messages and remote procedure calls. Mailboxes allow asynchronous communication; a task can send a message to another task's mailbox. The other task can read out the messages after the FIFO-principle when it is ready to do so. The other means are used for synchronous communication.

Interaction with Linux processes is possible through a FIFO buffer. Real-time tasks can write into or read from this buffer by invoking `rtf_put()` or `rtf_get()`. Linux processes can access this buffer through ordinary device nodes.

RTAI also allows the usage of shared memory between real-time tasks and Linux processes. Hence it is possible that multiple processes and tasks communicate with one another at the same time.

4.4 BusyBox - User Space Utilities for Embedded Linux

BusyBox melds small versions of numerous UNIX-tools together to one single binary. It provides minimalist replacements for most of the utilities you usually find in GNU coreutils, util-linux, and so on. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins. But the options that are included provide the expected functionality. [11]
In this way a fairly complete POSIX environment is provided by only one quite small file. Particular commands are represented by symbolic (or hard) links to the BusyBox binary. BusyBox determines the function to be invoked by checking the name of the calling link.

The developer may select, which utilities the BusyBox-binary should contain. Several utilities (e.g. `vi`) can even be configured in their functionality.
Thus it is possible to create an ideally adapted set of commands for an embedded system and a lot of disk space or ROM can be saved.

An entire list of the commands, BusyBox supports, can be found at
`http://www.busybox.net/downloads/BusyBox.html`.

Chapter 5

Cross Development

Like mainstream software developers, embedded system developers need compilers, linkers, interpreters and other development tools. Embedded developing tools are different from normal developing tools, as they typically run on one platform while building applications for another. For this reason these tools are often called cross-platform development tools, or in short ”cross development” tools. As there is always a bunch of tools required and these tools are used one after the depending on each other, a set of those tools is also called ”toolchain”.

There are several commercial cross-development environments that cost depending on their complexity and debugging facilities a lot of money.
In this context we are developing for a free platform - so why pay for the development tools. All of the tools needed are available as free software.

Any modern PC will work as development host. Using a Linux system for development is the easiest way to develop software for embedded Linux targets. The tools used in a cross-development toolchain are virtually the same tools that already reside on the system - they only differ in the target for which code is compiled for.

5.1 Components of a Cross-Compilation-Toolchain

In order to build software for an embedded Linux system, there are several tools and a standard library needed besides the compiler. All of these components are specific for both target (the embedded system) and host (the computer used for development).

5.1.1 Binary Utilities (binutils)

The binutils package includes the utilities that are most often used to manipulate binary object files (cf. table 5.1).

as	The GNU assembler
ld	The GNU linker
gasp	The GNU assembler pre-processor
ar	creates and manipulates archive content
nm	Lists the symbols in an object file
objcopy	Copies and translates object files
objdump	Displays information about the content of object files
ranlib	Generates an index to the content of an archive
readelf	Displays information about an ELF format object file
size	Lists the sizes of sections within an object file
strings	Prints the strings of printable characters in object files
strip	Strips symbols from object files
c++filt	Converts low-level mangled assembly labels resulting from overloaded c++ functions into their user-level names
addr2line	Converts addresses into line numbers within

Table 5.1: Tools in the Binutils Package

5.1.2 C/C++ Library

As mentioned before, almost every C program needs a standard C library to be functional; so this library is also needed for our toolchain. Due to the fact that this library is an interface to the running kernel, it has to be built based on the current kernel headers. Typically if you develop an embedded system, you compile your own kernel and thus you already have the headers for the kernel you want to use.

You have to deliberate, which libc is best for your application. When size does not matter the glibc might be best choice as it is the most compatible one.
uCLibc is possibly the best choice for an ”average” embedded system. In my opinion, it is an ideal trade-off between compatibility and size.
Even smaller C libraries (dietLibc or stripped down variants of the two libraries above) sometimes will cause issues with some programs. They should only be employed, when size and speed do really matter.
For instance if you plan to build a cgi-enabled web server that might run several thousand instances of a program, size and speed really matter, but you have to handle possible incompatibilities.

5.1.3 Compiler

The most important program in a toolchain is the compiler. It translates the source code to executable binary code for our target architecture.

Typically gcc is a good choice. ”gcc” is an abbreviation for GNU Compiler Collection - previously it was GNU C Compiler, but since more than just the C language is supported

the name was changed.
It supports more than 60 platforms (more than any other compiler) and several programming languages such as C, C++, Objective-C, Java, FORTRAN and Ada. It is possible to build gcc to run on one architecture while generating code for another architecture - i.e. it can be used as a cross-compiler.
It is the standard compiler for most Linux distributions as well as for BSD, Mac OS X, BeOS and so on.

5.2 Cross-Development on a Linux-Host

The easiest way doing cross-development is of course using a pre-built toolchain. There is a project called Buildroot that is simply a set of Makefiles and patches that allow to easily generate both, a cross-compilation toolchain and a root filesystem for the target.
There are many target platforms being supported by Buildroot, such as x86, PowerPC, MIPS and ARM. It utilises the BusyBox package in order to generate the root filesystem.

Surely you can build your own toolchain by compiling gcc, binutils, uClibc and all the tools by hand. But dealing with all configuration options, with all problems of every gcc or binutils version is very time-consuming and frustrating.
Buildroot automates this process through the use of Makefiles, and has a collection of patches for each gcc and binutils version it supports to make them work.

Buildroot is configured with an ncurses-based menu giving you the possibility to customise the toolchain and the root system in an all embracing way. You can choose between many versions of every part of the toolchain. You can select which packages shall be in your target's root filesystem.
Almost everything you can configure when building the toolchain "by hand" is also modifiable in the Buildroot environment.

5.3 Developing on a Windows Machine

Unfortunately there is no development environment running on Windows freely available that is as flexible as Buildroot. Although there are several cross-compiling toolchains allaround in the Internet. Most of them are for a very specific platform (e.g. Ipod-Linux) or outdated (gcc version < 3.0) or both of them.

If you want to omit buying an expensive cross-development environment, you still have some alternatives. One of them is running Linux in a virtual machine (e.g. VMware) and using the Buildroot environment within this VM. But this has a big disadvantage; it is rather circumstantial and very slow. You can find other possibilities in the following.

5.3.1 CygWin

CygWin is an emulation of the Linux-API for several Microsoft Windows operating systems. CygWin was developed by Cygnus Solutions, which was taken over by RedHat. Recently, it is subject to the GNU GPL. It enables you to port applications for POSIX operating systems (e.g. gcc) to Windows in an easy way.

Unfortunately there is a compatibility layer required to run applications that are compiled with/for CygWin. This layer is formed by a dynamic link library called `cygwin1.dll`. Without that file in the same path like the program or in the search path, the program will not run. So you always have to bundle this dll with your Software. If there is more than one application using CygWin on the same system, compatibility issues are in all probability.

A big benefit of CygWin is offering an X11 emulator enabling you to build and run graphical applications like DDD (Data Display Debugger - see 5.4.1).

5.3.2 MinGW

MinGW is a native port of gcc to Microsoft Windows Systems. The name is an abbreviation for "Minimalist GNU for Windows".
Theoretically it is possible to build a MinGW cross-compiler, but that seems to be rather difficult. I have not seen any functional one in my enquiries.
A MinGW based compiler would be a great convenience, as in contrast to CygWin it does not need any compatibility layer.

Usually MinGW comes with MSYS that supplies a UNIX-like shell making it possible to run shell-scripts like configure-scripts or execute native UNIX-Makefiles.

5.3.3 Building a Toolchain for a Windows-System

For building a toolchain running on a Windows machine with CygWin, first of all you need a Linux system and CygWin of course.

The components of a toolchain have to consort. As they are available in numerous versions, lots of experiments have been necessary to find matching versions. In my toolchain, gcc has version 3.4.6, binutils are version 2.17 and the kernel headers are from the Linux kernel version 2.6.18. The latter have been taken from a Linux Buildroot environment.

Installing CygWin

Setting up a CygWin environment is rather simple. You have to download the setup program from `http://cygwin.com/setup.exe` and start it. After selecting a mirror near your location, just follow the suggestions of the setup routine.

When finished, you can find a CygWin icon on your desktop. When invoking this icon, a bash shell is started offering almost the same functionality a bash on Linux does.

Building The Kernel Headers

Setting up the kernel headers is the first step in building a cross-development toolchain. Unfortunately this step fails in most cases on a windows machine. So you need to have a Linux system.
If there is only Windows running on your machine, Linux may be installed to a virtual machine using VMware or VirtualPC. VMware player is available for free and there are several pre-built virtual machines running Linux freely available on the internet (e.g. at http://www.thoughtpolice.co.uk/vmware/).

After selecting a kernel, a kernel package has to be downloaded (e.g. from www.kernel.org or a mirror site) and extracted to an arbitrary directory. In this case a common directory `$PROJECTROOT` with a subdirectory kernel is used.

```
$ mkdir $PROJECTROOT/kernel
$ cd $PROJECTROOT/kernel
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.17.tar.bz2
$ tar xjfv linux-2.6.18.tar.bz2
$ ln -s linux-2.6.18 linux
```

After these steps, the kernel has to be configured to meet your application's requirements. This is done by invoking one of the commands mentioned in 4.1.2.

```
$ cd linux
$ make ARCH=arm CROSS_COMPILE=$TARGET_PREFIX- menuconfig
```

The values of ARCH and CROSS_COMPILE are dependent on the target's architecture. ARCH stands for architecture and CROSS_COMPILE names the target-prefix of the compiler with an additional "-" (e.g. `arm-linux-` or `arm-xscale-linux-uclibc-`).

You do not have to build the kernel as the headers are already generated during configuration. Now you may create the include directory in your target directory and copy the kernel headers to it.

```
$ cd $PROJECTROOT
$ mkdir -p $TARGET_PREFIX/include
$ cp -r kernel/include/linux  $TARGET_PREFIX/include
$ cp -r kernel/include/asm-arm $TARGET_PREFIX/include/asm
$ cp -r kernel/include/asm-generic $TARGET_PREFIX/include
```

It is possible to reconfigure the kernel later without having to change the headers for your toolchain. The toolchain only needs one valid set of headers for your target, which is provided by the procedure mentioned above.

Only changing processor architecture or kernel version will make it necessary to build a new toolchain.

The resulting include directory may be copied to your Windows computer. This can be done with
`$ cd $PROJECTROOT` and
`tar cjvf includes.tar.bz2 $TARGET_PREFIX/include/*` on the linux machine and
`$ cd $PROJECTROOT` and
`tar xjvf includes.tar.bz2` within the CygWin shell.

Alternatively the kernel headers may also be taken from a Linux Buildroot environment.

Binutils Setup

Binutils may be obtained from http://ftp.gnu.org/gnu/binutils/. The build process looks like the following.

```
$ mkdir -p $PROJECTROOT/build-tools
$ cd $PROJECTROOT/build-tools
$ wget http://ftp.gnu.org/gnu/binutils/binutils-2.17.tar.bz2
$ tar xjfv binutils-2.17.tar.bz2
$ mkdir $PROJECTROOT/build-binutils
$ cd $PROJECTROOT/build-binutils
$ ../build-tools/binutils-2.17/configure \
   --prefix=$PROJECTROOT/$TARGET_PREFIX --target=$TARGET_PREFIX \
   --with-cpu=armv5te --with-arch=armv5te --with-tune=xscale \
   --disable-nls --enable-interwork --disable-multilib
$ make
$ make install
```

After this process has been completed, the tools named in 5.1.1 can be found in `$PROJECTROOT/$TARGET_PREFIX/bin`.

Setting Up a Bootstrap Compiler

In contrast to the binutils package, the gcc package contains only one utility - the GNU compiler.
At this stage we will build a bootstrap compiler, which will support only the C language. This ”small” compiler is needed in order to compile the C library. After compiling the C library, we are able to build a gcc with full C and C++ support.

The build process is similar to the build process before.

```
$ cd $PROJECTROOT/build-tools
$ wget ftp://ftp.gnu.org/gnu/gcc/gcc-3.4.6/gcc-3.4.6.tar.bz2
$ tar xjfv gcc-3.4.6.tar.bz2
$ mkdir $PROJECTROOT/build-gcc-bootstrap
$ cd $PROJECTROOT/build-gcc-bootstrap
$ ../build-tools/gcc-3.4.6/configure \
   --prefix=/$PROJECTROOT/$TARGET_PREFIX --target=$TARGET_PREFIX \
   --disable-nls --enable-interwork --disable-multilib \
   --without-headers --with-newlib --enable-languages=c
$ make all-gcc
$ make install-gcc
```

Now you can find the gcc for your target in
`$PROJECTROOT/$TARGET_PREFIX/bin`.

Compilation of the C Library

The Next step is the compilation of the C Library. Probably uClibc is the best choice for an embedded system.

Like before, the build process consists of downloading, configuration, compilation and installation.

```
$ cd $PROJECTROOT/build-tools
$ wget http://uclibc.org/downloads/uClibc-0.9.27.tar.bz2
$ tar xjfv uClibc-0.9.27.tar.bz2
$ cd uClibc-0.9.27
$ make CROSS_COMPILE=$TARGET_PREFIX- menuconfig
```

Configure uClibc to your needs. You will have to select CPU type, endianess, destination path, etc.

```
$ make CROSS_COMPILE=$TARGET_PREFIX-
$ make PREFIX=\$PROJECTROOT/$TARGET_PREFIX install
```

Building The Final Compiler

Now we are ready to install the full compiler for the target with C and C++ support.

```
$ mkdir $PROJECTROOT/build-gcc-final
$ cd $PROJECTROOT/build-gcc-final
$ ../build-tools/gcc-3.4.6/configure \
   --prefix=/$PROJECTROOT/$TARGET_PREFIX --target=$TARGET_PREFIX \
   --with-arch=armv5te --with-tune=xscale --with-float=soft \
   --disable-nls --enable-interwork --disable-multilib \
   --enable-threads=posix --disable-__cxa_atexit \
   --enable-languages=c,c++
$ make all
$ make install
```

Building a Debugger

Building the GNU debugger gdb is relatively easy. You can find some more information about this tool in 5.4.1.

```
$ cd $PROJECTROOT/build-tools
$ wget http://ftp.gnu.org/gnu/gdb/gdb-6.6.tar.bz2
$ tar xjfv gdb-6.6.tar.bz2
$ mkdir $PROJECTROOT/build-gdb
$ cd $PROJECTROOT/build-gdb
$ ../build-tools/gdb-6.6/configure \
  --prefix=/$PROJECTROOT/$TARGET_PREFIX --target=$TARGET_PREFIX
$ make
$ make install
```

The Easier Way

The build process of the toolchain is relatively complex and error-prone. There is a way to simplify this process by taking a large part from a Linux Buildroot environment.

First of all, you need a Linux machine, where you configure and build a Buildroot environment as described in
`http://docwiki.gumstix.org/Buildroot` or in
`http://Buildroot.uclibc.org/Buildroot.html`.

Then you may take the system includes and libraries from
`$BUILDROOT_DIR/build_$PREFIX/staging_dir`
to build the final compiler. In this way, you can omit building the kernel headers, the bootstrap compiler and the C library.

Build the binutils like described above and then the final compiler using the following additional options:

```
--with-sysroot=$STAGING_DIR \
--with-headers=$STAGING_DIR/include \
--with-libs=$STAGING_DIR/libs
```

Replace $STAGING_DIR with the path to (the copy of) your Buildroot's staging_dir.

5.4 Debugging Tools

If all of the development tools mentioned above have been built successfully and you are able to compile your applications, a powerful debugging facility is necessary, because also the best program has bugs. A good debugging tool enables you to localise and eliminate at least some of them.

To simplify debugging, it is recommended running the application being analysed from a NFS mount. You can save a lot of time not having to copy binary files manually and you possibly make fewer errors (for instance: copy the wrong file). NFS also allows debugging output (like performance data) to be available on the host system instantly.

There are two kinds of debugging solutions - debugging using special software and using special debugging hardware. The latter is used commonly with microcontrollers not having an operating system, but there are also devices for debugging "bigger" embedded Systems running Linux or other operating systems.

For Linux software, there are several debugging solutions like Valgrind (Open Source) or UndoDB(commercial). The latter stands out from the crowd as it is able to step forward and *backwards* through a program. Thus it allows the programmer to view the program's state at any point in the program's execution history.
However, the GNU Debugger (gdb) is the most important debugging tool for any Linux system.

5.4.1 The GNU Debugger gdb

The Gnu Debugger is a very common debugging tool for Linux and non-Linux systems. Remote debugging is possible by the use of so called gdb stubs. These stubs are a set of hooks and handlers in the target's operating system kernel or firmware that allow interaction with the remote debugger.

There are no gdb stubs needed to debug Linux software remotely, because the system call `ptrace()` is implemented in the Linux kernel. For remote debugging gdb provides a server (`gdbserver`) that makes use of this system call. gdbserver is a small application running on the target that executes the commands it receives from the gdb debugger running on the host.

Any application can be debugged on the target without having the gdb debugger running on the target. This is very beneficial, as the gdb binary is pretty large.

Using gdb

Before an application can be debugged using gdb, it has to be compiled with the appropriate flags. With gcc, you need to add the `-g` option to the command line. This option makes gcc add some debugging information to the object files. Using `-ggdb` will add even more debugging information.
Though the resulting binary is larger, you can use a stripped version on the target with gdbserver. Only the binary to be used with gdb on the host has to contain the complete debugging information.
Even though these two gdb components are using different binary files, the gdb running on the host is able to find and use the debug symbols, because it has access to the unstripped binary.

There are two possibilities for a gdbserver to communicate with a gdb debugger on the host. The first one is through a serial connection between host and target, the other one uses a TCP/IP network connection. The gdbserver may be started by invoking one of the following commands (depending on the type of connection)

- `gdbserver localhost:1234` *`program`*
  Runs gdbserver with the application *program* to be debugged using a network connection. The server listens to port 1234, the hostname (localhost) is ignored.
- `gdbserver /dev/ttyS0` *`program`*
  Starts gdbserver using the first serial port /dev/ttyS0 as communication means.

Once the gdbserver is started on the target, you can connect to it from the gdb debugger on the host using the target remote command. if you are connected to the target using TCP/IP networking, use the following command:

```
$ arm-linux-gdb program
(gdb) target remote target:1234
```

The first command starts the debugger (which was built for an arm-linux target) with the application *program* to be debugged. In this case the binary has to contain the debugging information mentioned above. The command invokes the gdb prompt.
At the gdb prompt the target is connected using the second command. The gdbserver that runs on the device with the hostname *target* and listens to port *1234*, will be connected. Instead of a hostname an IP-address can be stated too.
When connecting through a serial line, *target:1234* has to be replaced by the name of the host's serial port.

Graphical Frontends to gdb

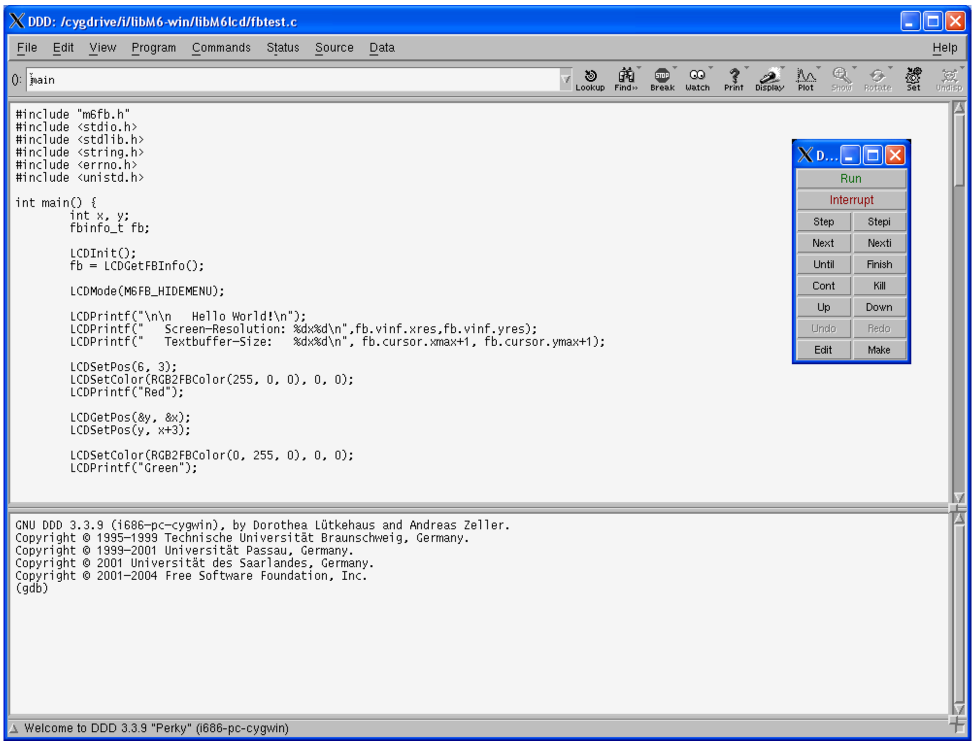
It is rather difficult to debug using the plain gdb command line. Fortunately there are some graphical frontends for gdb providing user-friendly mechanisms for doing most common debugging tasks such as setting breakpoints, viewing variables and so on.

The Data Display Debugger namely DDD is a free frontend for gdb provided by the GNU project. It is available in probably every Linux distribution and for Windows (using CygWin).
Besides usual front-end features such as viewing source texts, register contents and variables, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs (Figure 5.2).

You do not have to compile it yourself, because it is target independent. You only have to provide a gdb for your host/target combination by adding the `--debugger` option to the ddd command.

You may start DDD to debug your application *program* by executing:

```
$ ddd --debugger arm-linux-gdb program
```


DDD: /cygdrive/i/libM6-win/libM6lcd/fbtest.c
File Edit View Program Commands Status Source Data Help
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
(): main
```

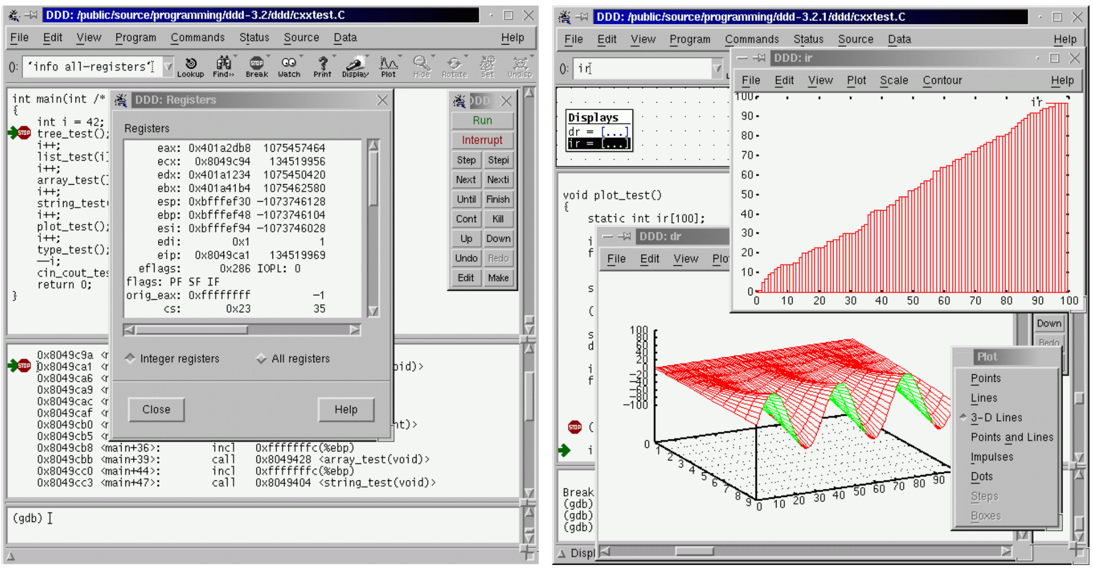
```
#include "m6fb.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

int main() {
        int x, y;
        fbinfo_t fb;

        LCDInit();
        fb = LCDGetFBInfo();

        LCDMode(M6FB_HIDEMENU);

        LCDPrintf("\n\n   Hello World!\n");
        LCDPrintf("   Screen-Resolution: %dx%d\n",fb.vinf.xres,fb.vinf.yres);
        LCDPrintf("   Textbuffer-Size:   %dx%d\n", fb.cursor.xmax+1, fb.cursor.ymax+1);

        LCDSetPos(6, 3);
        LCDSetColor(RGB2FBColor(255, 0, 0), 0, 0);
        LCDPrintf("Red");

        LCDGetPos(&y, &x);
        LCDSetPos(y, x+3);

        LCDSetColor(RGB2FBColor(0, 255, 0), 0, 0);
        LCDPrintf("Green");
```

```
GNU DDD 3.3.9 (i686-pc-cygwin), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(gdb)
```

Welcome to DDD 3.3.9 "Perky" (i686-pc-cygwin)

Figure 5.1: DDD - Data Display Debugger running on Windows XP

You can see a screenshot of DDD in figure 5.1. The largest part of the window is used by the source. Below is the debugger console, where you can enter gdb commands directly. For instance if you want to connect your target, you have to enter the corresponding command (target remote ...) in this console.


DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
DDD: Registers
Registers
Integer registers
All registers
Close
Help
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make
DDD: /public/source/programming/ddd-3.2.1/ddd/cxxtest.C
DDD: ir
File Edit View Plot Scale Contour Help
Displays
DDD: dr
Plot
Points
Lines
3-D Lines
Points and Lines
Impulses
Dots
Steps
Boxes

```
(): 'info all-registers'
```

```
int main(int /*
{
    int i = 42;
    tree_test();
    i++;
    list_test(i
    i++;
    array_test(
    i++;
    string_test
    i++;
    plot_test();
    i++;
    type_test();
    --i;
    cin_cout_tes
    return 0;
}
```

```
      eax: 0x401a2db8  1075457464
      ecx:  0x8049c94   134519956
      edx: 0x401a1234  1075450420
      ebx: 0x401a41b4  1075462580
      esp: 0xbfffef30 -1073746128
      ebp: 0xbfffef48 -1073746104
      esi: 0xbfffef94 -1073746028
      edi:        0x1           1
      eip:  0x8049ca1   134519969
   eflags:      0x286 IOPL: 0
flags: PF SF IF
orig_eax: 0xffffffff          -1
       cs:       0x23          35
```

```
0x8049c9a <r
0x8049ca1 <r                                           oid)>
0x8049ca6 <r
0x8049ca9 <r
0x8049cac <r
0x8049caf <r
0x8049cb0 <r                                           nt)>
0x8049cb5 <r
0x8049cb8 <main+36>:        incl    0xfffffffc(%ebp)
0x8049cbb <main+39>:        call    0x8049428 <array_test(void)>
0x8049cc0 <main+44>:        incl    0xfffffffc(%ebp)
0x8049cc3 <main+47>:        call    0x8049404 <string_test(void)>
```

```
(gdb)
```

```
(): ir
```

```
void plot_test()
{
    static int ir[100];
```

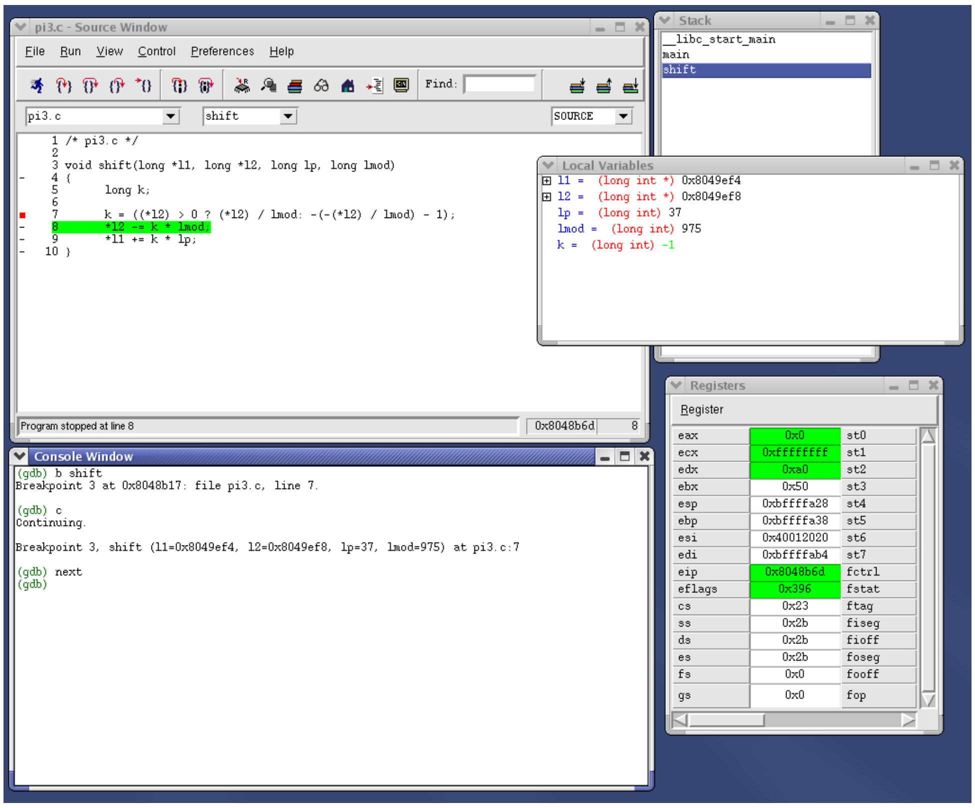
```
Break
(gdb)
(gdb)
(gdb)
```

Figure 5.2: DDD - register view and plots - *(source: gnu.org - Free Software Foundation, Inc.)*

Using DDD is rather simple, most functions you will need are self-explanatory and there are numerous tutorials available on the internet.

Another possibility avoiding gdb's console is Insight (Figure 5.3). It is not a frontend to gdb, but another version of GDB that uses Tcl/Tk to implement a graphical user interface. As it is not separable from gdb, it has to be built specifically for every target.

In my opinion, Insight's handling is much more complicated compared to DDD's. The documentation is not as extensive either.



```
 1 /* pi3.c */
 2
 3 void shift(long *l1, long *l2, long lp, long lmod)
 4 {
 5       long k;
 6
 7       k = ((*l2) > 0 ? (*l2) / lmod: -(-(*l2) / lmod) - 1);
 8       *l2 -= k * lmod;
 9       *l1 += k * lp;
10 }
```

```
Program stopped at line 8                0x8048b6d    8
```

```
__libc_start_main
main
shift
```

```
l1 =  (long int *) 0x8049ef4
l2 =  (long int *) 0x8049ef8
lp =  (long int) 37
lmod =  (long int) 975
k =  (long int) -1
```

```
(gdb) b shift
Breakpoint 3 at 0x8048b17: file pi3.c, line 7.

(gdb) c
Continuing.

Breakpoint 3, shift (l1=0x8049ef4, l2=0x8049ef8, lp=37, lmod=975) at pi3.c:7

(gdb) next
(gdb)
```

eax	0x0	st0
ecx	0xffffffff	st1
edx	0xa0	st2
ebx	0x50	st3
esp	0xbffffa28	st4
ebp	0xbffffa38	st5
esi	0x40012020	st6
edi	0xbffffab4	st7
eip	0x8048b6d	fctrl
eflags	0x396	fstat
cs	0x23	ftag
ss	0x2b	fiseg
ds	0x2b	fioff
es	0x2b	foseg
fs	0x0	fooff
gs	0x0	fop

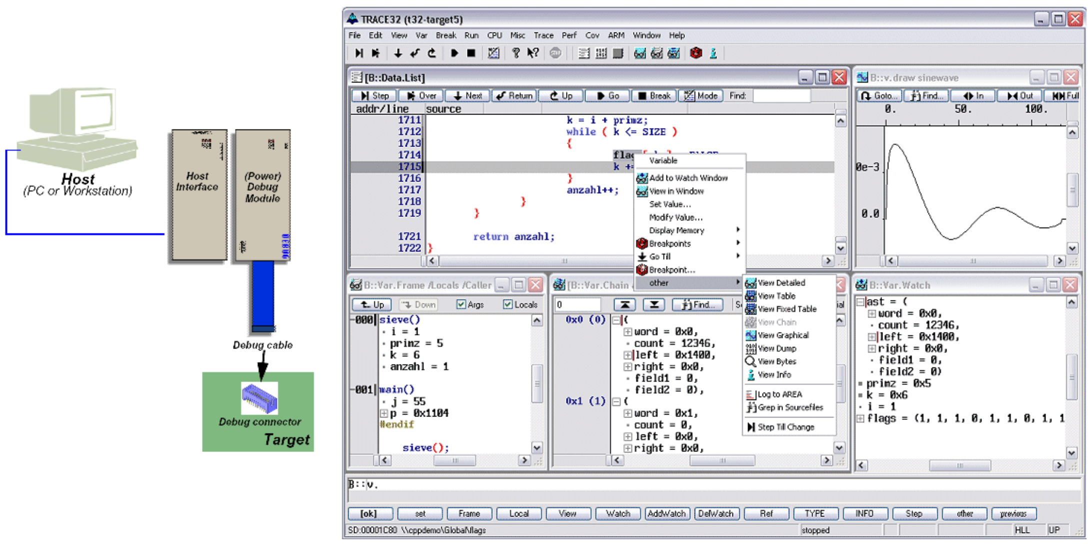
Figure 5.3: Insight - another gdb frontend - *(source: sourceware.org/insight)*

5.4.2 Hardware Tools

In addition to software tools, there are some hardware tools available for debugging embedded software. Hardware tools are sometimes more effective than software tools to eliminate software problems, but they have a great disadvantage: they are almost always really expensive.

There are several means of hardware debugging such as in-circuit emulation (ICE) or using BDM or JTAG interfaces.
For in-circuit emulation the CPU of an embedded system is replaced by an emulator,

which uses either a modified CPU of the same type where all needed signals can be picked up from the debugger (bond-out CPU) or it emulates the whole CPU (in another CPU or by the use of FPGAs). The latter emulation method is only available with slow CPUs.



```
addr/line  source
                              k = i + primz;
                              while ( k <= SIZE )
                              {
                                      flag
                                      k +=
                              }
                              anzahl++;
                      }
              }

              return anzahl;
}
```

```
-000|sieve()
     · i = 1
     · primz = 5
     · k = 6
     · anzahl = 1

-001|main()
     · j = 55
     ⊞p = 0x1104
     #endif

        sieve();
```

```
0x0 (0) ⊟(
          ⊞word = 0x0,
          · count = 12346,
          ⊞left = 0x1400,
          ⊞right = 0x0,
          · field1 = 0,
          · field2 = 0),
0x1 (1) ⊟(
          ⊞word = 0x1,
          · count = 0,
          ⊞left = 0x0,
          ⊞right = 0x0,
```

```
⊟ast = (
   ⊞word = 0x0,
   · count = 12346,
   ⊞left = 0x1400,
   ⊞right = 0x0,
   · field1 = 0,
   · field2 = 0)
■ primz = 0x5
■ k = 0x6
· i = 1
⊞flags = (1, 1, 1, 0, 1, 1, 0, 1, 1
```

```
B::v.
```

Figure 5.4: JTAG interface and debugging software - *(source: Lauterbach Datentechnik GmbH)*

The usage of BDM or JTAG interfaces is usually cheaper, but your target CPU has to provide one of these interfaces. Figure 5.4 shows a JTAG interface and the corresponding debugging software.

There was even an open source project developing a BDM debugger called BDM4GDB. The project offered hardware schematics and patches for gdb. But during the writing of this text, the project (formerly hosted at sourceforge.net) was not available anymore.

Chapter 6

Libraries and Binaries for EyeBot M6

One aim of this thesis was the implementation of some software routines to easily access the display of the EyeBot M6 (actually a Linux framebuffer device). Another task was building a user interface using display and touchscreen of this appliance in cooperation with Thomas Sommer.

6.1 API for the Display of EyeBot M6

The EyeBot M6 can be equipped with different displays connected via the Linux framebuffer driver. As the resolution of these displays may vary, the library functions must be flexible to adjust the size of its buffers, length and count of text lines and so on.

Due to the Linux framebuffer device being almost the same an different systems the library may be used on any system with a Linux-supported linear framebuffer device. Only few framebuffer devices do not work, for instance not-linear framebuffers (like some nvidia and older ATI devices) display wrong colours or odd patterns. Due to the multitude of implementations and the lack of documentation there is no plan to support non-linear framebuffer. Fortunately most framebuffer devices may be addressed linear - so the library is applicable in most cases.

6.1.1 The Linux Framebuffer Device

Unfortunately there is no documentation about programming the linux framebuffer device available. All of the facts mentioned below were collected by reverse-engineering existing framebuffer implementations such as DirectFB (*www.directfb.org/*) and EZFB (*http://www.akrobiz.com/ezfb/*).

Like most other hardware in Linux, the framebuffer device is accessed through a device node. It may be opened and written like any regular file using `open()`.

To use the framebuffer in a sensible way, we need some information about the hardware - such as screen resolution, colour-depth and size of the framebuffer memory. This information can be obtained by invoking:

```
ioctl(fbdev, FBIOGET_VSCREENINFO, *var_screeninfo);
ioctl(fbdev, FBIOGET_FSCREENINFO, *fix_screeninfo);
```

These functions return structures with different pieces of information.

- `var_screeninfo` contains data that is changeable by the user such as screen resolution (not needed with an LCD), virtual resolution, colour-depth and timing values.
- `fix_screeninfo` contains fixed screen information like starting address and length of the framebuffer in the memory.

The complete break down of these structures can be found in `linux/fb.h` in the include-directory of any Linux system.

If you want to change framebuffer settings, you have to obtain the information like above, then you can apply your changes to the data structure. Finally you have to send your changes to the device by invoking:

```
ioctl(fbdev, FBIOPUT_VSCREENINFO, *var_screeninfo);
```

After the initial setup, the framebuffer device is mapped to User Space memory using mmap:

```
mm_data = (unsigned short*) mmap(NULL, fix_screeninfo.smem_len, \
(PROT_READ|PROT_WRITE), MAP_SHARED, fbdev, 0);
```

Now you may write your data to the device using `memcpy()` or pixel-by-pixel by dereferencing mm_data.

`memcpy(mm_data, screenbuffer, bufsize);` or `mm_data[i] = color;`

In order to release the framebuffer device you should unmap the device from memory and close the file handle to the device node.

If you are using virtual terminals on your Linux system, you will have to add functions to change to a free terminal, as instead you will have a mix of your framebuffer-output and the console text overwriting each other.

6.1.2 Software Routines

The software routines for EyeBot M6 support virtual terminals, various resolutions, text and graphic output.

Before any of the operations can be used, `LCDInit()` has to be invoked. This function initialises the buffers and global variables, prepares the access to the framebuffer device and sets all default values (e.g. text colours).

Due to the use of several buffers, it is possible to change the text without destroying the graphics on the display and vice versa. You can see the buffers and their combination in Figure 6.1.

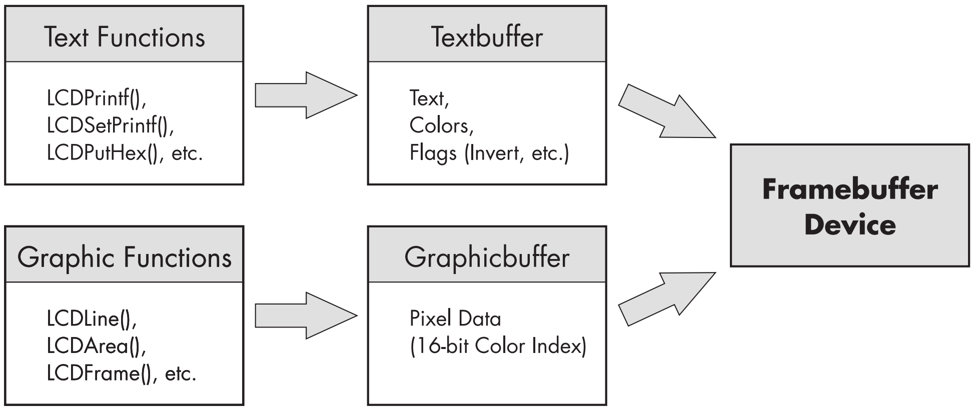


Figure 6.1: Architecture of the framebuffer library

When a text operation like `LCDPrintf()` is invoked, the provided string is copied to the textbuffer at the current cursor position. Additionally the current text-colours are stored to this buffer at the corresponding places as well. Graphic operations alter the content of the graphicbuffer.

Each text or graphic operation calls `LCDRefresh()` when it has finished - this may be switched off if you want to print/display multiple items and refreshing the screen amongst would only be a waste of time.
`LCDRefresh()` copies the contents of the graphic buffer to the framebuffer device, renders the textbuffer (with a given font) and outputs the rendered text to the framebuffer too.

During this thesis, all LCD Output-functions mentioned in the documentation of RoBIOS for EyeBot M5 have been implemented for EyeBot M6. You can find a complete list of the newly implemented routines in tables 6.1 to 6.4.

LCD Handling Routines	
LCDInit();	Initialize the framebuffer-device
LCDRelease();	Release the framebuffer-device
LCDGetFBInfo();	Get various information about the display
LCDClear();	Clear the LCD
LCDMode (...);	Set LCD mode (Autorefresh, Menu, etc.)
LCDGetMode();	Returns the current mode
LCDRefresh();	Refresh the screen (copy graphic- and textbuffer to screen)
RGB2FBColor(...);	Convert a 3x8-bit RGB value to 16-bit palette-index

Table 6.1: Routines in the Framebuffer Library - LCD Handling

Text Operations	
LCDPrintf(...);	Print text or numbers or combination of both onto LCD
LCDSetPrintf(...);	Print formatted string at given position
LCDPutChar(...);	Write character
LCDSetChar(...);	Set Cursor to given position, write character
LCDPutString (...);	Print string
LCDSetString (...);	Print string to given cursor-position
LCDPutHex (...);	Print hexadecimal number
LCDPutHex1 (...);	Print single hexadecimal digit
LCDPutInt (...);	Print an integer value
LCDPutIntS (...);	Print an integer value with leading spaces (if necessary)
LCDPutFloat (...);	Print a float value
LCDPutFloatS (...);	Print a formatted float valu
LCDSetPos (...);	Set the cursor to given position
LCDGetPos (...);	Returns current cursor position
LCDSetColor (...);	Set colours (Textcolour, Background colour and Flags)

Table 6.2: Routines in the Framebuffer Library - Text Output

Graphical Operations	
LCDSetPixel (...);	Set a pixel to the given color
LCDInvertPixel (...);	Invert the pixel at the given position
LCDGetPixel (...);	Return the colour of the addressed pixel
LCDLine(...);	Draw a line using the Bresenham Algorithm
LCDLineInvert(...);	Draw a line using the Bresenham Algorithm inverting the pixels
LCDFrame(...);	Draw a rectangular
LCDArea(...);	Fill rectangular area
LCDAreaInvert(...);	Invert rectangular area
LCDPutImage(...);	Output an image with 16-bit depth
LCDPutRGBImage(...);	Output an image with 24-bit depth (RGB - each 8 bit)
LCDPutGSImage(...);	Output an 8-bit greyscale image
LCDPutBWImage(...);	Output an 1-bit black-and-white image
LCDLoadFBImage(...);	Load an image in FB format and shows it

Table 6.3: Routines in the Framebuffer Library - Graphics

Menu Routines	
LCDMenu (...);	Fills the menu with the given entries and enables menu
LCDMenuI(...);	Changes a menu entry

Table 6.4: Routines in the Framebuffer Library - Menu

6.2 GUI for Running and Managing User Programs

This GUI is able to start user programs, terminate them and manage running programs. A pre-defined directory is scanned for executable files, afterwards, the list of files is displayed using a user-friendly menu. Any program in that directory may be selected and executed by simply touching the screen.

When a program is started, the menu line changes to show those possibilities (cf. Fig. 6.2:

- BACK returns to the selection screen.
- LIST shows a list of all user programs currently running.
- KILL terminates the program.

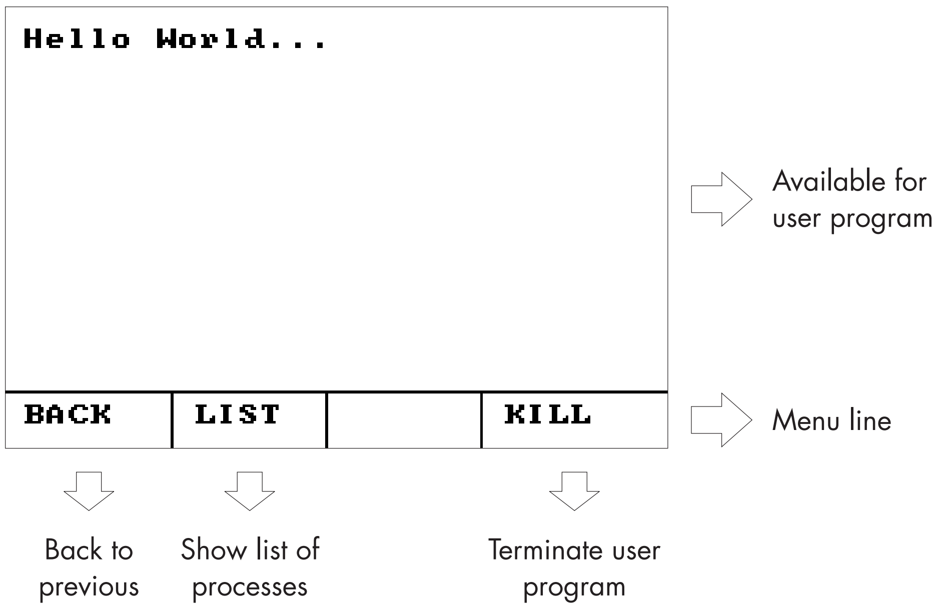


Figure 6.2: Execution screen with menu

If the "list" command is invoked and there are user programs running, a list similar to the one aforementioned is shown. But this time the menu shows PID and name of the user programs that are currently running. You have the possibility to select one and terminate it.

To enable running more than one program at the same time, a kind of process management is needed. When a program is started, this happens in a new process that uses `execv()` to run the binary. At the same time PID and name of this new process is enlisted in a process list and a new thread is started waiting for the termination of this process and removing it from the list afterwards. In the meanwhile, the main program is responsible for displaying the menu as seen in Figure 6.2.

Unfortunately, there is one problem in Linux' process-handling. Only the parent-process can use `waitpid()` to wait for a child-process to terminate. Another child-process or a thread cannot.
So we check whether the newly started process is still running by sending a signal 0 to the process using the `kill()` function and evaluate the response to see if the PID is still valid. But there is a huge disadvantage in this workaround: if a process terminates itself regularly, the parent is not able to call waitpid() instantly as it does not notice the termination. So the child-process becomes a "zombie" process and the PID remains valid. Thus the thread waiting for the termination of the child-process does not notice the termination either and the program remains in the process list. There is also no possibility to send a signal from the child process, because the `execv()` call does not return.
By doing a non-blocking `waitpid()` call to all processes that are currently in the process list, the list is cleaned up. This procedure is accomplished every time the user invokes the process list.

The only alternative that solves this issue would be the creation of a PID-file representing the status of the user program. This file has to be created when the user program is invoked. After its regular termination, the PID-file has to be deleted by the user program itself, as the termination is unobservable by the GUI routines.
The GUI could check for the existence of this file to evaluate whether the program is still running or not.
The disadvantage of this approach is that the user program has to delete the file itself. Thus the program cannot be an arbitrary binary an more; it has to be coded especially for this appliance.

Chapter 7

Conclusion

In conclusion the project EyeBot M6 was promoted one step forward. It was extended by a library to handle the display of the EyeBot enabling a programmer to

- print strings, characters and numbers (in numerous formats)
- set the printing position
- set text colours, transparency and colour-inversion
- enable/disable features like text-scrolling, automatic linefeed etc.
- draw pixels, lines and frames
- fill areas with colour
- load and display images
- draw a menu to the screen

in an easy way.

Besides this work, a frontend for the EyeBot M6 was developed, of course employing these library functions. The frontend allows the user to gather information about the current status of the EyeBot and to exert influence on this status.

The user may run arbitrary user programs in the foreground or in the background. There is also a possibility to manage all running programs. Additionally, there is a number of demo programs showing some of the functionality that was newly implemented.

Also new means for software development for the EyeBot were created. A cross-compiling toolchain that runs on a Windows host computer compiling binaries for Linux on ARM processors was built.

Furthermore remote debugging facilities were implemented. These enable the software developer to debug software running on the EyeBot M6 remotely from his development workstation running either Windows or Linux. Even graphical frontends are available now to ease this job.

Now it is possible to code, compile and debug software for the EyeBot M6 on both Windows and Linux machines.

Unfortunately the RoBios-library EyeBot M6 is still far away from completion, but it is well on the way to getting an even bigger functional range than the previous version for EyeBotM5 has.

Appendix A

The EyeBot M6 API

Basically the EyeBot M6 API is based on the previous version for EyeBot M5. Because of some additional functionality (e.g. color display) the syntax is sometimes slightly different, but it is preferably matched.

A.1 Finished Components

This section lists the RoBIOS routines that are currently implemented for EyeBot M6. Actually display, audio and touchscreen libraries are available. Also some OS and misc routines have been implemented.

A.1.1 Display Library

In the following, you can see the documentation of the RoBIOS display functions implemented during this thesis. Each function is listed with its specific input an output data and the semantics.

Framebuffer Handling

`int LCDInit();`

Input:	none
Output:	none
Semantics:	Initialise the framebuffer-device

`int LCDRelease();`

Input:	none
Output:	none
Semantics:	Release the framebuffer-device

`fbinfo_t LCDGetFBInfo();`

Input:	none
Output:	struct fbinfo_t
Semantics:	Gather information such as cursor, textcolors and LCDMode fb_var_screeninfo, fb_fix_screeninfo → see linux/fb.h cursor → actual position and max. values textattr → current fore-/background-colors, color-flags mode → see LCDMode()

`int LCDClear();`

Input:	none
Output:	none
Semantics:	Clear the LCD

`int LCDRefresh();`

Input:	none
Output:	none
Semantics:	Refresh the Screen (copy graphic- and textbuffer to screen)

`unsigned short RGB2FBColor(unsigned char R, unsigned char G, unsigned char B);`

Input:	(R) (G) (B) 8-bit RGB-values
Output:	(return value) 16-bit palette-index
Semantics:	Convert 3x8-bit RGB value to 16-bit palette-index

`int LCDMode (int mode);`

Input:	(mode) the display mode(s) you want multiple modes may be added or 'OR'ed)
Output:	none
Semantics:	Set the display to the given mode (defaults set by LCDInit() are marked with (*))

Modes:

M6FB_AUTOREFRESH	(*)	enable/disable automatic refresh of
M6FB_NOAUTOREFRESH		the screen after a command was invoked
M6FB_SCROLLING	(*)	enable/disable scrolling
M6FB_NOSCROLLING		when last row is exceeded
M6FB_LINEFEED	(*)	enable/disable automatic linefeed
M6FB_NOLINEFEED		when last column is exceeded
M6FB_SHOWMENU	(*)	show/hide menu
M6FB_HIDEMENU		
M6FB_ROOTMENU		color of menu (red-blue/yellow-blue)
M6FB_NORMMENU	(*)	
M6FB_FB_ROTATE	(*)	rotate screen by 180 degrees/don't rotate
M6FB_FB_NOROTATION		

`int LCDGetMode (void);`

Input:	none
Output:	(return value) current display mode
Semantics:	Gather current display mode

Text Routines

`int LCDPrintf(const char *format, ...);`

Input:	format string and parameters
Output:	none
Semantics:	Prints text or numbers or combination of both onto LCD (and refresh LCD if AUTOREFRESH is enabled)

`int LCDSetPrintf(int row, int column, const char *format, ...);`

Input:	print position, format string and parameters
Output:	none
Semantics:	Print formatted string at given position (and refresh LCD if enabled)

`int LCDPutChar(char c);`

Input:	(char) the character to be written
Output:	none
Semantics:	Write character to current cursor-position and increment cursor-position (and refresh LCD if enabled)

`int LCDSetChar(int row, int column, char c);`

Input:	the desired position and the character to be written
Output:	none
Semantics:	Set Cursor to given position, write character and increment cursor-position (and refresh LCD if enabled)

`int LCDPutCharBuffer (char c);`

Input:	character
Output:	none
Semantics:	Print character to current cursor-position (without refreshing the LCD)

`int LCDPutString (char *string);`

Input:	(string) the string to be written
Output:	none
Semantics:	Print string to the current cursor-position (and refresh LCD if enabled)

`int LCDSetString (int row, int column, char *string);`

Input:	(string) the string to be written
Output:	none
Semantics:	Print string to given cursor-position (and refresh LCD if enabled)

`int LCDPutHex (int val);`

Input:	(val) the number to be written
Output:	none
Semantics:	Print hexadecimal number to current cursor-position

(and refresh LCD if enabled)

`int LCDPutHex1 (int val);`

Input:	(val) the number to be written (single byte 0..255)
Output:	none
Semantics:	Print single hexadecimal digit to current cursor-position (and refresh LCD if enabled)

`int LCDPutInt (int val);`

Input:	(val) the number to be written
Output:	none
Semantics:	Print an integer value to current cursor-position (and refresh LCD if enabled)

`int LCDPutIntS (int val, int spaces);`

Input:	(val) the number to be written (spaces) the minimal number of spaces
Output:	none
Semantics:	Print an integer value with leading spaces (if necessary) to current cursor-position (and refresh LCD if enabled)

`int LCDPutFloat (float val);`

Input:	(val) the number to be written
Output:	none
Semantics:	Print the given number as floating point number to current cursor-position (and refresh LCD if enabled)

`int LCDPutFloatS (float val, int spaces, int decimals);`

Input:	(val) the number to be written (spaces) the minimal number of spaces (decimals) the number of decimals to be written
Output:	none
Semantics:	Print the given number as floating point number with leading spaces (if necessary) and specified number of decimals to current cursor-position (and refresh LCD if enabled)

`int LCDSetPos (int row, int column);`

Input:	(row) the number of the row (valid: 0 - fb->cursor.ymax) (column) the number of the column (valid: 0 - fb->cursor.xmax)
Output:	none
Semantics:	Set the cursor to the given position

`int LCDGetPos (int *row, int *column);`

Input:	none
Output:	(*row) the number of the row (valid: 0 - fb->cursor.ymax) (*column) the number of the column (valid: 0 - fb->cursor.xmax)
Semantics:	Return current cursor position

`int LCDSetColor (unsigned short fgcol, unsigned short bgcol, char flags);`

Input:	(fgcol) text foreground color (bgcol) text background color (flags) text color flags multiple flags may be added or 'OR'ed
Output:	none
Semantics:	Set given colors and flags for the following text operations (defaults set by LCDInit() are marked with (*))

Flags:

M6FB_BGCOL_TRANSPARENT		Set/unset transparent background
M6FB_BGCOL_NOTRANSPARENT	(*)	
M6FB_BGCOL_INVERSE		Set/unset background color to
M6FB_BGCOL_NOINVERSE	(*)	inversion of graphic buffer
M6FB_FGCOL_INVERSE		Set/unset foreground color to
M6FB_FGCOL_NOINVERSE	(*)	inversion of graphic buffer

Graphics Routines

`int LCDSetPixel (int x, int y, unsigned short color);`

Input:	(x, y) coordinates of the pixel (color) color to be assigned to pixel
Output:	none
Semantics:	Set the pixel at the given position to the given color

`int LCDInvertPixel (int x, int y);`

Input:	(x, y) coordinates of the pixel
Output:	none
Semantics:	Invert the pixel at the given position

`unsigned short LCDGetPixel (int x, int y);`

Input:	(x, y) coordinates of the pixel
Output:	(return value) color of the pixel
Semantics:	Return color value of selected pixel

`int LCDLine(int x1, int y1, int x2, int y2, unsigned short color);`

Input:	(x1, y1), (x2, y2) coordinates of the endpoints (color) color of the line
Output:	none
Semantics:	Draw a line from (x1, y1) to (x2, y2) with the given color using the Bresenham Algorithm

`int LCDLineInvert(int x1, int y1, int x2, int y2);`

Input:	(x1, y1), (x2, y2) coordinates of the endpoints
Output:	none
Semantics:	Draw a line from (x1, y1) to (x2, y2) with inverting the background using the Bresenham Algorithm

`int LCDArea(int x1, int y1, int x2, int y2, unsigned short color);`

Input:	(x1, y1), (x2, y2) coordinates of the corners (color) filling color
Output:	none
Semantics:	Fill a rectangular area from (x1, y1) to (x2, y2) with the given color

`int LCDAreaInvert(int x1, int y1, int x2, int y2);`

Input:	(x1, y1), (x2, y2) coordinates of the corners (color) filling color
Output:	none
Semantics:	Invert a rectangular area from (x1, y1) to (x2, y2)

`int LCDFrame(int x1, int y1, int x2, int y2, unsigned short color);`

Input:	(x1, y1), (x2, y2) coordinates of the corners
	(color) line color
Output:	none
Semantics:	Draw a rectangle (frame) from (x1, y1) to (x2, y2) with the given color and a line width of 1 pixel

`int LCDPutImage(int xpos, int ypos, int xsize, int ysize, unsigned short *data);`

Input:	(xpos, ypos) coordinates of the top-left corner
	(xsize, ysize) size if the image
	(*data) array of 16-bit values (image data)
Output:	none
Semantics:	Draw a 16-bit image data to the given position
	Image data format: 5 bit red, 6 bit green, 5 bit blue
	(large images will be cropped)

`int LCDPutRGBImage(int xpos, int ypos, int xsize, int ysize, char *data);`

Input:	(xpos, ypos) coordinates of the top-left corner
	(xsize, ysize) size if the image
	(*data) array of 24-bit values (image data)
Output:	none
Semantics:	Convert a 24-bit image to 16-bit and put it at given position
	Image data format: array of chars - alignment: RGBRGBRGB...

`int LCDPutGSImage(int xpos, int ypos, int xsize, int ysize, char *data);`

Input:	(xpos, ypos) coordinates of the top-left corner
	(xsize, ysize) size if the image
	(*data) array of 8-bit values (image data)
Output:	none
Semantics:	Display a 8-bit grayscale image at given position

`int LCDPutBWImage(int xpos, int ypos, int xsize, int ysize, char *data, char threshold);`

Input:	(xpos, ypos) coordinates of the top-left corner
	(xsize, ysize) size if the image
	(*data) array of 8-bit values (image data)
Output:	none
Semantics:	Display a black-and-white image at given position
	values: 0..(threshold-1) $\rightarrow$ black, threshold..255 $\rightarrow$ white

`int LCDLoadFBImage(char *filename);`

Input:	(*filename) filename of the image
Output:	none
Semantics:	Load and display a FB image (file of 16-bit values) see ppm2fb for conversion to this format

Menu

`int LCDMenu (char *string1, char *string2, char *string3, char *string4);`

Input:	(*string1) 1st menu entry (*string2) 2nd menu entry (*string3) 3rd menu entry (*string4) 4th menu entry valid values are - a string (will be word-wrapped if so - new-line-char is supported) - "" leave the menu entry untouched - " " clear the menu entry
Output:	none
Semantics:	Fill the menu line with the given menu entries and activate menu

`int LCDMenuI(int pos, char *string);`

Input:	(pos) number of menu entry to be changed (valid: 0..3) (*string) new entry
Output:	none
Semantics:	Change the selected menu entry to given string

A.1.2 Audio Library

`int AUPlaySample (char* sample, long length);`

Semantics:	Plays an audio sample.

`int AURecordSample(char* buf, long len);`

Semantics:	Records an audio sample.

`int AUPlayFile(char* file);`

Semantics:	Plays an audio file.

`int AURecordFile(char* file, long len, long freq);`

Semantics:	Records to a file.

`int AUCheckSample (void);`
Semantics: Checks for audio playback end.

`int AUCheckRecord (void);`
Semantics: Checks for audio recording end.

`int AUCheckTone(void);`
Semantics: Checks for beep or tone end.

`int AUTone(int freq, int msec);`
Semantics: Outputs a tone.

`int AUBeep(void);`
Semantics: Outputs a beep.

`int AUCaptureMic(void);`
Semantics: Get microphone input value

A.1.3 Key Library (Touchscreen)

`int8_t KEYGetBuf (keycode_t *buf);`
Semantics: Wait for a keypress and store the keycode into the buffer.

`keycode_t KEYGet (void);`
Semantics: Wait for a keypress and return keycode.

`keycode_t KEYRead (void);`
Semantics: Read keycode and return it. Function does not wait.

`keycode_t KEYWait (keycode_t excode);`
Semantics: Wait for a specific key

`coord_pair_t KEYGetXY (void);`
Semantics: Return coordinates of a keypress.

`int8_t KEYSetTM (int8_t mode, ...);`
Semantics: Setup the key region map (for the touchscreen)

`void *KEYGetTM (int8_t mode);`
Semantics: Get a string representation of the key region map

A.1.4 OS Library

`info_cpu_t *OSInfoCPU (void);`
Semantics: Collects infos about the CPU.

`info_mem_t *OSInfoMem (void);`
Semantics: Collects infos about the memory.

`info_proc_t *OSInfoProc (void);`
Semantics: Collects infos about processes.

`info_misc_t *OSInfoMisc (void);`
Semantics: Collects miscellaneous infos.

`char* OSVersion (void);`
Semantics: Returns a string containig the current RoBIOS version.

A.1.5 Misc Library

`int iround_div (int dividend, int divisor);`
Semantics: Divide integers and round the result.

`int scale_index (int old, int old_max, int new_max);`
Semantics: Scales an index (starting from 0).

A.1.6 FPGA Functions

`volatile u16* FPGA_map(FPGA_addr_t address, unsigned long bytes);`
Semantics: Map FPGA to given memory address.

`int FPGA_memcpy_from(void *dest, FPGA_addr_t src, int bytes);`
Semantics: Read data from FPGA

`int FPGA_memcpy_to(FPGA_addr_t dest, void *src, int bytes);`
Semantics: Send data to FPGA

A.1.7 GPIO Functions

`int GPIO_set_bank_direction(int bank, GPIO_dir_t direction);`
Semantics: Set GPIO direction

`int GPIO_set_state(int bank, int gpio, int state);`
Semantics: Set GPIO-pin to given value

`int GPIO_get_state(int bank, int gpio);`
Semantics: Get current state of GPIO pin

A.1.8 PSD Functions

`void PSD_enable();`
Semantics: Enable PSD sensors

`void PSD_disable();`
Semantics: Disable PSD sensors

`void PSD_set_update_period(int ms);`
Semantics: Set sampling rate of the PSD sensors

`int PSD_read(int number);`
Semantics: Read value from PSD sensor [number]

A.2 Components to Be Implemented

There are many routines of the RoBios Library still to be implemented. Unfortunately, there is adequate programming of the FPGA needed, as many of these routines use hardware connected to this device. The features of Robios for EyeBot M5 still to be developed for EyeBot M6 are listed below.

- Camera, Image Processing
- Servos and Motors
- Parallel Port
- A/D Converter
- TV Remote Control

Appendix B

EyeBot M6 Hardware Issues

There were several problems getting the hardware of the EyeBot M6 platform running with all its components. Some of them were founded in undocumented or poorly documented connections on the board and GPIO-pins being untimely in the wrong state.

Some problems appear, due to the setting of GPIO-pins to the wrong state in some modules while loading.

The following list contains the GPIO-pins to be set ideally just before the corresponding module is loaded:

- For most parts the 5V-Enable line has to be set
  ```
  echo out set > /proc/gpio-ac97/UCB1400-0-2
  ```
- LCD shutdown-line
  ```
  echo GPIO out set > /proc/gpio/GPIO84
  ```
- Ethernet needs toggling of its reset line
  ```
  echo out clear > /proc/gpio-ac97/UCB1400-0-8
  echo out set > /proc/gpio-ac97/UCB1400-0-8
  ```
- USB reset line
  ```
  echo GPIO out set > /proc/gpio/GPIO52
  ```
- FPGA
  ```
  echo out set > /proc/gpio-ac97/UCB1400-0-0
  ```
- Camera 2
  ```
  echo out set > /proc/gpio-ac97/UCB1400-0-5
  ```

The USB-gadget module for emulating a serial port (g_serial) still has some problems due to errors in the module. A terminal session using this port hangs when too many data is transmitted in a very short time. Setting the baud rate to a minimum did not put things right.

Appendix C

Contents of the CD

The contents of the CD belonging to this thesis are listed below:

`/code`	code.tgz - tarball of the sources in the subdirectories
`/code/filemenu`	user program management routines for the main program
`/code/libm6fb`	EyeBot M6 framebuffer library
`/code/libm6fb/test`	EyeBot M6 framebuffer library test program for all routines
`/code/ppm2fb`	tool to convert PPM-images to FB-format
`/cygwin`	CygWin setup
`/gserial`	serial gadget driver for Win32 platforms
`/RobLin`	Windows cross-compiling environment
`/sources`	sources.tgz - tarball of the sources used to compile the Windows-toolchain in CygWin
`/thesis`	this thesis as pdf file
`/VMware`	VMware Player setup and virtual Linux machine with a buildroot - user: vmware - pass: vmware - root access with ”`sudo bash`” and user password

Bibliography

[1] *ARM architecture*. Wikipedia.org, http://en.wikipedia.org/wiki/ARM_architecture. 7

[2] *Embedded Linux*. Wikipedia.org, http://en.wikipedia.org/wiki/Embedded_Linux.

[3] *Embedded Systems*. Wikipedia.org, http://en.wikipedia.org/wiki/Embedded_Systems.

[4] *Homepage*. The Linux kernel preemption project, http://kpreempt.sourceforge.net/. 14, 25

[5] *"I will NEVER use Windows CE or Windows XP Embedded!"*. msdn, http://msdn2.microsoft.com/en-us/embedded/aa731327.aspx. 12

[6] *Kapitel 3 - Kompatibilitätsfragen*. Debian GNU/Linux-FAQ, http://www.us.debian.org/doc/manuals/debian-faq/ch-compat.de.html. 11

[7] *MC68832 Product Summary Page, ff.* Freescale Semiconductor, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC68332. 9

[8] *Notable CPU architectures ff.* Wikipedia.org, http://en.wikipedia.org/wiki/Notable_CPU_architectures.

[9] *Open Group pilots embedded real-time POSIX conformance testing.* LinuxDevices.com, http://www.linuxdevices.com/news/NS3992179355.html. 14

[10] *QNX*. Operating System Documentation Project, http://www.operating-system.org/betriebssystem/_german/bs-qnx.htm. 14

[11] *The Swiss Army Knife of Embedded Linux*. BusyBox, http://www.busybox.net/downloads/BusyBox.html. 26

[12] *Symbian OS*. Wikipedia.org, http://en.wikipedia.org/wiki/Symbian. 13

[13] *Virtex-4 Capabilities*. Xilinx, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/index.htm. 10

[14] *VxWorks.* Wikipedia.org,
http://en.wikipedia.org/wiki/VxWorks. 14

[15] *Windows CE.* Wikipedia.org,
http://en.wikipedia.org/wiki/Windows_CE. 12

[16] *Windows XP Embedded.* Wikipedia.org,
http://en.wikipedia.org/wiki/Windows_XP_Embedded. 12

[17] *First Steps with Embedded Systems.* Byte Craft Limited, Waterloo, Canada, 2002.

[18] *The cell architecture.* The Cell Project at IBM Research,
http://www.research.ibm.com/cell/,
http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html,
2005. , 8

[19] *Product backgrounder.* ARM,
http://www.arm.com/miscPDFs/3823.pdf, 2005. 7

[20] Abbott, Doug: *Linux for Embedded and Real-time Applications.* Newnes - Elsevier Science, Burlington, USA, 2003. , 21

[21] Barr, Michael: *Programing Embedded Systems in C and C++.* O'Reilly, Waterloo, Canada, 1999.

[22] Blackham, Bernard: *The Development of a Hardware Platform for Real-time Image Processing.* The University of Western Australia, Final Year Project Auflage, 2006. 9

[23] Chin, Lixin: *FPGA Based Embedded Vision Systems.* The University of Western Australia, Final Year Project Auflage, 2006. 9

[24] Engel, Michael: *Klein, aber Linux.* Linux Community,
http://www.linux-community.de/Neues/story?storyid=889, 2001. 24

[25] Färber, Univ.-Prof. Dr.-Ing. Georg: *Eingebettete Systeme - Manuskript zur Vorlesung.* Lehrstuhl für Realzeit-Computersysteme - TU München, 2005. , 2

[26] Hallinan, Christopher: *Embedded Linux Primer: A Practical, Real-World Approach.* Prentice Hall, 2006.

[27] Leitner, Felix von: *diet libc.* Linux Kongress 2001,
http://www.fefe.de/dietlibc/talk.pdf, 2001. 24

[28] Ripoll, Ismael: *Rtlinux versus rtai.* RTLinux Portal at Valencia,
http://rtportal.upv.es/comparative/rtl_vs_rtai.html, 2002. 24

[29] Stokes, Jon: *Powerpc on apple: An architectural history.* Ars Technica,
http://arstechnica.com/articles/paedia/cpu/ppc-1.ars/1, 2004. 6

[30] Yaghmour, Karim: *Building Embedded Linux Systems.* O'Reilly, 2003. 6, 7