DIPLOMARBEIT

# Embedded Linux for Signal Processing Platform

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

o. Univ. Prof. Dipl.-Ing. Dr. techn. Dietmar Dietrich
und
Univ. Ass. Dipl.-Ing. Dr. techn. Stefan Mahlknecht
als verantwortlich mitwirkendem Universitätsassistenten am
Institutsnummer: 384
Institut für Computertechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Thomas Tamandl
9825868
Reiherweg 30
1220 Wien

Wien, im Oktober 2005 _____

**Abstract**

In recent years the acceptance of using embedded systems in many fields of daily life is growing. Actually processors used for embedded systems are getting more and more powerful while their prices are getting cheaper and cheaper. For these systems, there is the need of a universal operating system which allows simple and fast application development. uClinux is an operating system that is able to fulfill these requirements. It is a derivate of Linux adapted to work with resource limited microcontrollers. uClinux is freely available and there are already a lot of usable applications, which means that companies can save a lot of development costs.

This thesis presents how to port uClinux and an accompanying bootloader in order to develop user applications on a custom designed hardware. As a result, a lot of basic information about Linux and uClinux is provided. For a better understanding of application development a simple demo application is designed. This application is added to the user applications in order to show how to integrate a new program into uClinux.

To help developers to start developing device drivers for a custom made hardware, an introduction to the possibilities of integrating device drivers to uClinux is given. As a consequence a simple character device driver is designed step by step in order to describe the different stages of developing a device driver.

The development hardware used for porting uClinux is based on the Analog Devices ADSP-BF533 Blackfin processor. It is shown how to add a custom designed board to the uClinux project and to the bootloader source code. Hence, software developers may be able to configure their uClinux without knowing details about the hardware. This will allow them to fully concentrate on developing user applications.

**Kurzfassung**

In den letzten Jahren werden immer mehr Aufgaben aus dem täglichen Lebens mit Hilfe von embedded Systems bewältigt. Der stete Anstieg dieser Geräte beruht auf dem andauernden Preisverfall der Prozessoren welche für diese Aufgaben eingesetzt werden. Trotzdem werden diese Prozessoren immer leistungsfähiger und können immer komplexere Tätigkeiten vollführen. Daher wird es immer wichtiger für embedded Systems über ein universell einsetzbares Betriebssystem, welches ein einfaches und schnelles Entwickeln von Anwendungen erlaubt, zu verfügen. Alle diese Anforderungen können durch uClinux erfüllt werden. Dieses Betriebssystem ist ein Derivat von Linux, welches für den Einsatz auf Prozessoren mit eingeschränkter Funktionalität optimiert ist. uClinux ist kostenlos erhältlich, trotzdem ist bereits eine hohe Anzahl an nützlichen Anwenderprogrammen enthalten. Dadurch können Entwicklungskosten für neue Geräte gespart werden.

Diese Diplomarbeit beschäftigt sich mit der Portierung von uClinux und dem zugehörigen Bootloader auf ein noch nicht von uClinux unterstütztes Entwicklungsboard. Für diesen Vorgang wird einiges an Wissen über Linux bzw. uClinux benötigt, dieses wird am Anfang dieser Arbeit präsentiert. Da durch uClinux eine einfache und schnelle Anwenderprogrammentwicklung möglich ist, wird der Ablauf der Programmerstellung bzw. die Integration eines Anwenderprogramms in uClinux, an Hand eines einfachen Beispielprogramms präsentiert.

Da für kundenspezifische Hardware meist auch speziell angepasste Gerätetreiber benötigt werden, wird aufgezeigt wie Gerätetreiber in das System integriert werden können. Um Entwicklern den Einstieg in die Treiberentwicklung zu erleichtern, wird ein einfacher, zeichenorientierte Gerätetreiber entworfen. Dieser Treiber wird Schritt für Schritt entwickelt um näher auf die benötigten Entwicklungsstufen eingehen zu können.

Das verwendete Entwicklungsboard basiert auf dem Analog Devices ADSP-BF533 Prozessor. Diese Arbeit zeigt die nötigen Vorgänge um dieses Board zu uClinux oder dem Bootloader hinzu zu fügen. Nach der Integration des Entwicklungsboards in uClinux, können Entwickler uClinux als Basis für Anwenderprogramme verwenden, ohne über Detailkenntnisse der darunter liegenden Hardware zu verfügen.

**Danksagung**

Hiermit möchte ich mich bei meinen Eltern bedanken welche auf Grund meines Studiums sicher auf einiges verzichten mussten. Ohne ihre Unterstützung wäre mein Studium nicht möglich gewesen.

Ein herzliches Dankeschön auch an meine Freundin die mich während meines Studiums immer unterstützt hat, obwohl sie sich die letzten Jahre sicher etwas anders vorgestellt hat.

Ganz besonders möchte ich meinem Betreuer Dipl.-Ing. Dr. Stefan Mahlknecht für seine konstruktive Unterstützung danken.

„Last but not least“ danke ich allen, die bereit waren mich mit ausführlichen Diskussionen zum Thema zu unterstützen, und die mir bei der Korrektur der Arbeit geholfen haben.

# Abbreviations

| | |
|---|---|
| BASH | Bourne Again Shell |
| BIOS | Basic Input/Output System |
| CF-CARD | CompactFlash-Card |
| CFI | Common Flash Interface |
| EXT2 | Second Extended File System |
| EXT3 | Third Extended File System |
| FSF | Free Software Foundation |
| GNU | recursively for GNU’s Not UNIX |
| GPIO | General Purpose Input Output |
| GPL | GNU General Public License |
| GUI | Graphical User Interface |
| IRQ | Interrupt Request |
| JFFS2 | Journalling Flash File System 2 |
| JTAG | Joint Test Action Group |
| LED | Light Emitting Diode |
| MAC | Multiply Accumulate |
| MMU | Memory Management Unit |
| MSA | Micro Signal Architecture |
| MTD | Memory Technology Devices |
| OSI | Open Source Initiative |
| PPI | Parallel Peripheral Interface |
| PWM | Pulse Width Modulator |

| | |
|---|---|
| RAM | Random Access Memory |
| RFS | Root File System |
| RISC | Reduced Instruction Set Computing |
| ROM | Read Only Memory |
| RTC | Real Time Clock |
| SD-CARD | Secure Digital-Card |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SPI | Serial Peripheral Interace |
| SPORT | Serial Port |
| TFTP | Trivial File Transfer Protocol |
| UART | Universal Asynchronous Receiver/Transmitter |
| USB | Universal Serial Bus |
| VFS | Virtual File System |
| VM | Virtual Memory |

# Table of contents

# Chapter 1 Introduction

## 1.1 Linux, uClinux and embedded systems

Linux is a free, widely used operating system. A lot of people think that Linux is an operating system with a nice, fully featured graphical user interface which they have seen on some workstation. What they have seen was a graphical user interface like KDE or Gnome[1] used for easy interaction with the workstation. Usually when speaking about Linux, the whole system including the kernel and a lot of other utilities is meant. When being stricter Linux refers only to the Linux kernel! A lot of further information on Linux will be provided in Chapter 2 .

Today we will find embedded systems in a lot of devices. A high range of complexity is possible when using embedded systems. For example a relatively simple device can be a washing machine, on the other end of the scale there are mobile phones and so on. On the contrary on general purpose computers, like simple workstations, the software installed on an embedded system is adapted to the special requirements of the system. Often the user of an embedded system is not able to change the operating system or some user application. On your workstation you can do all that. An embedded system also has to work under different conditions than a work station. On a work station normally a very fast processor is used to provide high computational power. Embedded systems have to meet other major criteria, for example cheap processors or less energy consumption. The computational power also is not very important on a lot of embedded systems. A very important factor on embedded systems is the amount of system- and not volatile memory. On high end workstations we are used to have a hard disc drive with hundreds of gigabytes as not volatile memory and a system memory with a few gigabytes. In an embedded system normally only few megabytes are used for system- and non volatile memory.

---

[1] KDE and Gnome are free Desktop environments; there are also a lot of similar projects. A Desktop environment offers a complete Graphical User Interface (GUI) for easy interaction with the work station. Depending on the different projects more or less features are supplied when installing the GUI.

Why should anyone use Linux on such a restricted system? An operating system helps to manage your hardware and resources. It provides an API (Application Programming Interface), so programmers need not know details about the hardware details. Furthermore a lot of utilities are available and tested, this helps to save development costs. If your chosen system is Linux, it is available for free and there are no royalty costs. Therefore you have to provide the source code of your system to everybody for free. Further information to the Linux license agreements will be found in Chapter 2 .

How should Linux with its memory requirements fit on an embedded system, especially when there is a relatively simple microcontroller working on it? The answer to all these questions is uClinux. Further information on uClinux will be provided in later chapters.

## 1.2 The Blackfin processor family

The Blackfin processor family is based upon the Micro Signal Architecture (MSA) jointly developed by Analog Devices and Intel. Based on this architecture the Blackfin processor offers a 32-bit RISC (Reduced Instruction Set Computing) instruction set with a dual 16-bit mac (Multiply Accumulate) digital signal processing functionality. So the Blackfin processor can be used for fast digital signal processing as well as for control applications. Furthermore the Blackfin processor family supports special instructions for video processing.

Low power consumption is provided by using Dynamic Power Management. Dynamic Power Management supports the independent adjustments of operating frequency and voltage. So the computational power can be adjusted to meet the requirements of the application. This helps to save a lot of energy and will help to increase the lifetime of battery powered systems. [WANA1]

Because of the high complexity of the Blackfin processor core it would be outside the scope of this introduction to provide a more detailed view. A short overview on the Blackfin processor core basics can be found on [WANA1]. A very detailed explanation of the MSA architecture can be found in an IEEE Xplore article “High Performance Dual-MAC DSP Architecture” [KOL02].

Because some samples shown in the following chapters will use the Blackfin ADSP-BF533, a short overview describing the ADSP-BF533 peripherals will be given.

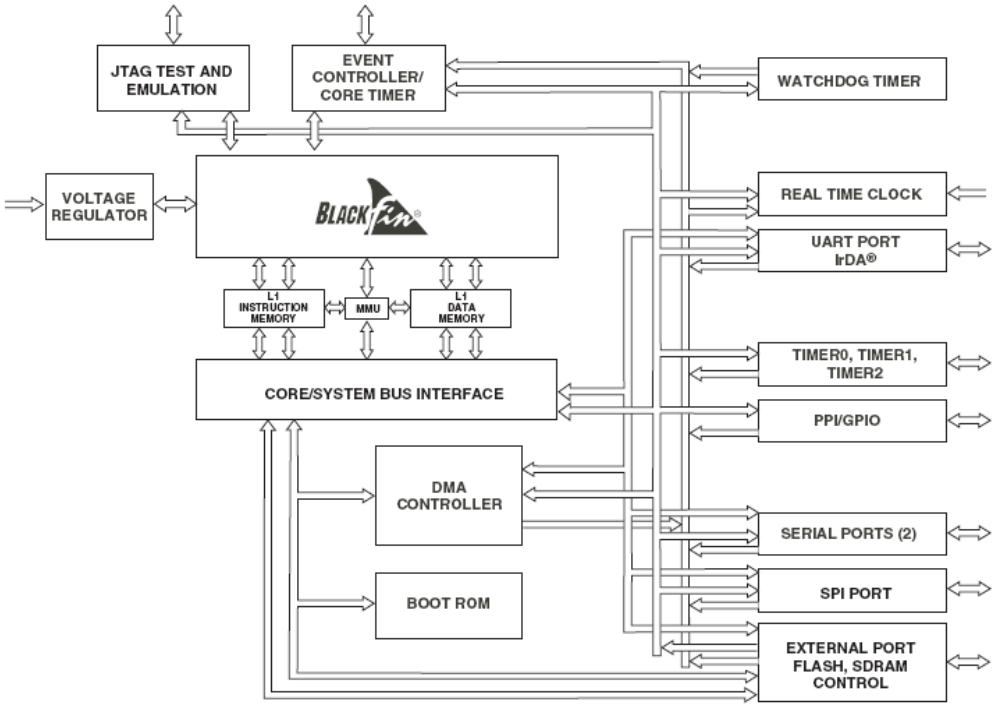


**Figure 1.1 Processor Block Diagram [ANA04]**

On the right hand side of Figure 1.1 the processor's peripherals are shown, they are connected to the processor's core components (left hand side of Figure 1.1) via several high bandwidth busses. [ANA04]

**Watchdog Timer:**

A 32-bit timer is used to count down from an initial value to zero. The software has to update this timer periodically. If the timer is expired, something goes wrong and some actions will be needed to get the processor into a defined state. This action can be to do a hardware reset or to generate an interrupt.

**Real Time Clock (RTC):**

Beside the function of a watch the RTC can be used as stop watch or to wake up the processor from power down or sleeping mode. For providing these functions the RTC has dedicated power supply pins. This allows the RTC to continue working even when the system is powered down.

**UART Port (Universal Asynchronous Receiver/Transmitter):**

Used for serial transmissions, compatible to a standard PC Interface.

**Timer 0…3:**

Programmable times provide a lot of different functions. For example they can be used for the PWM (Pulse Width Modulator) or to generate interrupt events.

**PPI/GPIO (Parallel Peripheral Interface / General Purpose Input Output):**

The PPI interface is used to connect devices with a parallel peripheral interface to the processor, for example analog/digital converters. When configured as GPIO, pins can be used to control, for example, the state of a LED or to read some input value.

**SPORTs (Serial Ports):**

Used for synchronous transmission of serial data, for serial and multiprocessor transmissions.

**SPI (Serial Peripheral Interface):**

SPI is a special type of serial communication. There are, for example, SPI compliant Flash-memories supporting this type of communication.

All provided information can be found in [ANA04]. No further information on the Blackfin ADSP-BF533 will be provided because the Blackfin processor hardware is not the major topic of this documentation. Furthermore in later chapters a lot of different Linux and uClinux topics will be discussed. Therefore no more detailed information on the Blackfin processor is needed, because uClinux is already running on this processor, even if uClinux has to port on the used development platform.

## 1.3 Problem description

As mentioned in section 1.2 uClinux is already running on an ADSP-BF533 processor. There are a handful of different development boards which are powered by this processor. Bluetechnix has also produced a development board powered by this processor. The major topic of this document is to describe how to port uClinux on a new platform using a Blackfin processor. Anyone who is not familiar with uClinux should be able to read in quickly.

**Figure 1.2 Bluetechnix CM-BF533 Core Module [WBLU1]**

Figure 1.2 shows the Bluetechnix CM-BF533 Core Module powered by an ADSP-BF533 processor. All pins used for the peripheral units described in section 1.2 are connected to the two connectors downside the Core Module.

Beside the shown peripheral modules the CM-BF533 offers the following features:

- Clock: up to 600MHz
- 32 MB SDRAM
- 4 MB Flash memory

Two development boards for this core module are supplied by Bluetechnix. First the “EVAL-BF5xx Blackfin Evaluation Board” (see Figure 1.3) with the following key features, as explained by [WBLU1]:

- Sockets for one core board: CM-BF533 or CM-BF561
- 1 Connector for ITU-656 camera
- SD-Card socket
- USB-UART Interface
- All other pins on expansion connector

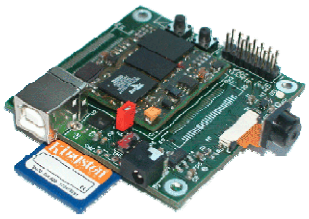

**Figure 1.3 EVAL-BF5xx Blackfin Evaluation Board [WBLU1]**

The second development board is the “DEV-BF5xx Development KIT”, see Figure 1.4. This board provides the following features, as explained by [WBLU1]:

- Sockets for 2 coreboards
- 6 Connectors for ITU-656 cameras (stereo vision supported with Camera Kit)
- Compact Flash and SD-Card sockets
- UART on RS232 Port or fast UART on USB interface
- 10/100 Mbit Ethernet
- 16 bit Dualported RAM interconnects two core boards
- Software selectable multiple JTAG[1] interface
- Buttons and LEDs for evaluation

[1] JTAG (Joint Test Action Group) is an interface to JTAG compliant Hardware for testing and debugging issues. Have a look at the IEEE 1149.1 Standard for more details.

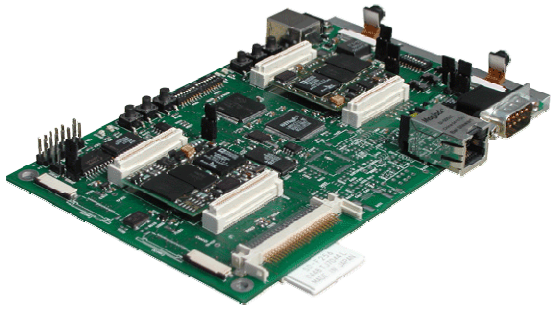

**Figure 1.4 DEV-BF5xx Development KIT [WBLU1]**

## 1.4 Structure of this document

Chapter 2:

A lot of information based on operating system basics and Linux basics will be discussed.

Chapter 3:

In this chapter the work that has to be done will be defined.

Chapter 4:

While developing new software a boot loader is very comfortable. It helps to get the uClinux image into the Flash memory via Ethernet, works with a serial console and supplies a lot of other useful features. This chapter provides the information on how to port the bootloader to the development hardware.

Chapter 5:

uClinux, the operating system chosen for the development hardware, will be ported to the CM-BF533 in this chapter. Information on how to adapt uClinux to the new board is provided. This means adapting configuration files and adding board specific options to the kernel configuration utility. Finally a simple user application will be added to uClinux.

Chapter 6:

Some device driver samples and samples on how to add ISR’s (Interrupt Service Routines) to uClinux are provided.

Chapter 7:

A short summary of chapters 1-6 will be given. The possibilities for further extensions will be discussed.

# Chapter 2 Operating Systems and Linux

## 2.1 History of Linux

Linux is a UNIX like operating system, so a short overview of the history of UNIX will be given. There is a very good overview in [LOV04] on page 1: “Since the creation of UNIX in 1969, the brainchild of Dennis Ritchie and Ken Thompson has become a creature of legends, a system whose design has withstood the test of time with few bruises to its name.

UNIX grew out of Multics, a failed Bell Laboratories multiuser operating system project. With the Multics project terminated, members of Bell Laboratories’ Computer Sciences Research Center were left without capable interactive operating system. In the summer of 1969, Bell Lab programmers sketched out a file system design that ultimately evolved into UNIX. Thompson implemented the new system an otherwise idle PDP-7. In 1971, UNIX was ported to the PDP-11, and in 1973 the operating system was rewritten in C, an unprecedented step at the time, but one that paved the way for future portability.”

In the first days of UNIX AT&T (Bell Telephone was taken over by AT&T) it was not allowed to sell UNIX. So they licensed it to Universities and research institutes without service but for marginal costs. Hence a lot of different versions of UNIX have become available. More detailed information can be found in [HEA03].

Linux is a UNIX like operating system. The first release of Linux was developed by Linus Torvalds, and a few programmers who helped him, in 1991. Today there are a lot of programmers all over the world developing the Linux kernel. Up to now Linus Torvalds is responsible for the kernel. Ideas from UNIX were used when developing Linux, but Linux was written up from scratch. Linux is also a POSIX[1] and Single UNIX

[1] As explained in [LOV04]:”One of the more popular application programming interfaces in the Unix world is based on the POSIX standard. Technically, POSIX is comprised of a series of standards from the IEEE that aims to provide a portable operating system standard roughly based on Unix. Linux is POSIX compliant”

Specification[1] compliant operating system.

A good overview of what has happened since 1969 is provided by Figure 2.1.

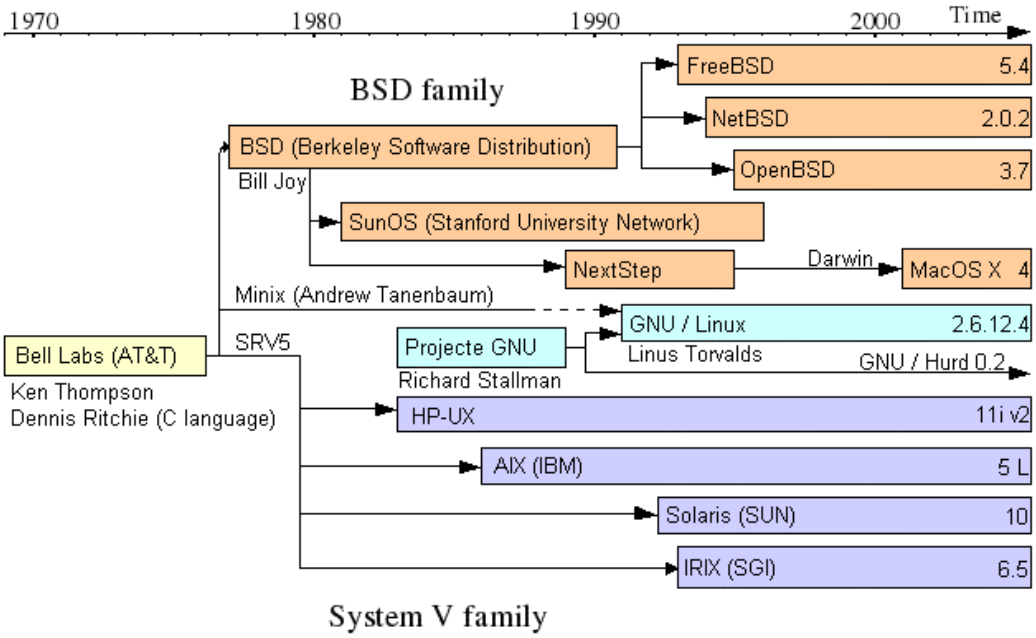


**Figure 2.1 History [WWIK1]**

## 2.2 Terminologies and licenses

There often arises a lot of confusion when reading open source, free software, GNU, GPL, GPL/Linux and so on. This section will shortly explain the most frequently used terms. A lot of other terminology and licenses are used in the open source / free software world. It will be out of the scope of this document to explain them all.

**Free software, Free Software Foundation:**

The Free Software Foundation (FSF), founded in 1985 by Richard Stallman and others, is responsible for saving the rights of the users using free software and the rights of the software itself. After founding the Free Software Foundation it earned its money selling tapes with Emacs[2] or printed manuals.

[1] The aim of the single UNIX specification is a standardized interface for application software. Further information can be found on http://www.unix.org/version3/.

[2] One of the first GNU projects done by Richard Stallman was the GNU Emacs text editor.

Isn’t there any discrepancy in selling free software? No, when speaking of free software “free” is not meant in terms of free from any costs or fees. Free means, that you are free to do with the software whatever you want. It can be used without restrictions; it can be modified and redistributed. [WFSF1]

**Open source, Open Source Initiative:**

Second, there is the Open Source Initiative (OSI), an idea of Eric S. Raymond and Bruce Perens in 1998. Open source is not interchangeable with free software, even if lots of people think so. There is a philosophical difference between the FSF and the OSI, free software is always open source. [WOPE1] Furthermore, when comparing the Open Source Software Definition [WOPE1] and the Free Software Definition [WFSF1], the Free Software Definition is stricter. Since the free software versus open source discussion is not a topic of this document, the interested reader can find a lot of information on [WFSF1] and [WOPE1].

**GNU, GNU’s Not UNIX:**

The GNU (recursively for GNU’s Not UNIX) project started in 1984 initialized by Richard Stallman. The goal of this project was to develop a free, UNIX like operating System. Because of the high amount of work for distributing and managing the GNU projects the FSF was founded in 1985. Today there can be found a lot of highly used tools in the GNU project, for example the GNU Compiler Collection (Gcc) or the GNU Emacs text editor. [BON99]

**GNU/Linux:**

When Linux Torvalds had written the Linux kernel there was no kernel available by the GNU project. Nonetheless a lot of system tools were available by the GNU project. So the two projects were put together and the first free operating system GNU/Linux was born. Often only Linux is used.

**GPL, GNU General Public License:**

The GPL met the requirements for free software placed by the Free Software Foundation. This means that the source code is available, can be edited and redistributed. When redistributing software, which was originally distributed under GPL, always GPL must be used. It is not allowed to restrict the rights of software distributed under the GPL. Every program, even if it includes only parts of code distributed using GPL, has to use GPL. The GPL itself is not distributed under GPL; it is not allowed to change anything in the GPL. Every project distributed under the GPL has to include a copy of the license. [BON99]

**LGPL, GNU Lesser General Public License**

When using software distributed under GPL in seperate projects, it is not allowed to distribute this projects under a stricter license, also if using library functions distributed under GPL. For libraries, for example the GNU C library, the LGPL can be used. In contrast to the GPL it is allowed to build proprietary programs using library functions distributed under LGPL.[WGNU1]

**BSD, Berkeley System Distribution:**

With programs distributed with their licenses everything can be done. There is no need to redistribute changed source codes, even if the program will be sold. The only thing you have to do is to mention that the software was developed by the University of California.

## 2.3 Inside the Linux kernel

Don’t be afraid, this section will not explain the whole, complex kernel internals. This would be out of the scope of this work anyway. But to understand all the practical work starting in Chapter 4 , a lot of detailed knowledge on some topics will be necessary. uClinux will be compared to standard Linux in Chapter 5 , also for this purpose some information is required. This information should be provided now. All provided information given in this section is based on the kernel version 2.6.x.

### 2.3.1 Kernel, user-space, kernel-space

For which purpose and where do we need the Linux kernel in the system? The major function of the kernel is to manage hardware and resources and share them to user applications if needed, as can be seen in Figure 2.2.

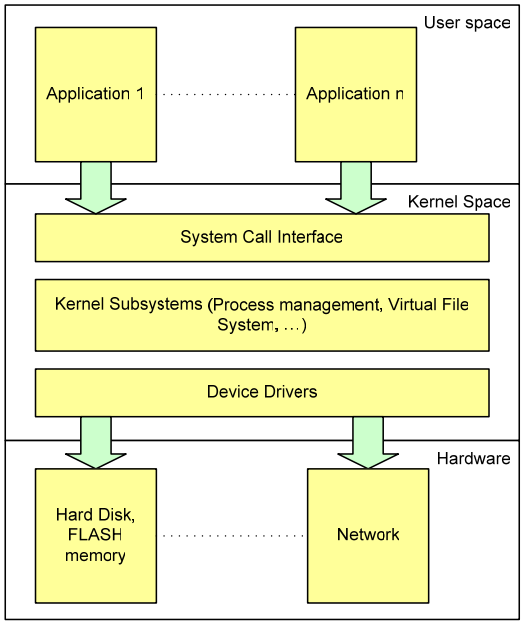


**Figure 2.2 Relationship between applications, the kernel, and hardware**

The kernel is the connection between hardware and user applications. The kernel itself communicates with the hardware using device drivers. Applications cannot directly

communicate with the hardware, for example to open a file. They have to use the System Call Interface provided by the kernel. If the kernel is executing something on behalf of an application, the kernel is running in process context.[LDK05]

Be careful when programming in kernel-space. Crashed applications in user-space can be killed without effects on the system. They are also very restricted when trying to get access, for example, to a wrong memory address. But in kernel-space, for example when writing a device driver, the system can be killed if the driver is not carefully tested. There cannot be restrictions to a device driver, the kernel has to thrust itself.

### 2.3.2 Processes, Threads, Scheduling

As explained in [LOV04]:" A process is a program (object code stored on some media) in execution. It is, however, more than just the executing program code (often called the text section in UNIX). Processes also include a data section containing global variables, a set of resources such as open files and pending signals, an address space, and one or more threads of execution."

With this abstraction it is possible to give each process its own virtual processor and virtual memory. This has the following advantages: On the view of the process it holds its own processor and memory. Because of these abstractions the system is able to share the physical processor to more than one process. Furthermore, because every process has its own address space, no misrouted memory access can affect other processes. This will not be mentioned by the programmer.

In Linux a thread (this is the short form for thread of execution) is a process which is able to share resources, for example data or memory. This means that many threads can be included into one process.

In this document scheduling strategies will not be introduced. Only be different process states which will be needed for scheduling are shown. Even if scheduling strategies will not be explained, the process states are needed anyway. For example, when writing an application that has to wait for some event and busy waiting is not allowed, the application has to sleep.

As shown in Figure 2.3 there are five process states:

**Running:**

The process is able to run, even if it is scheduled in the moment.

**Interruptible:**

The process is sleeping. It is waiting for a signal or an event.

**Uninterruptible:**

Like the Interruptible state the process is sleeping. But it cannot be woken up by a signal.

**Zombie:**

When the process has terminated the process descriptor (that held the data about the terminated process, for example the exit code) is held in memory until the parent process cares about the data.

**Stopped:**

The process execution can be stopped (and restarted) with a signal.

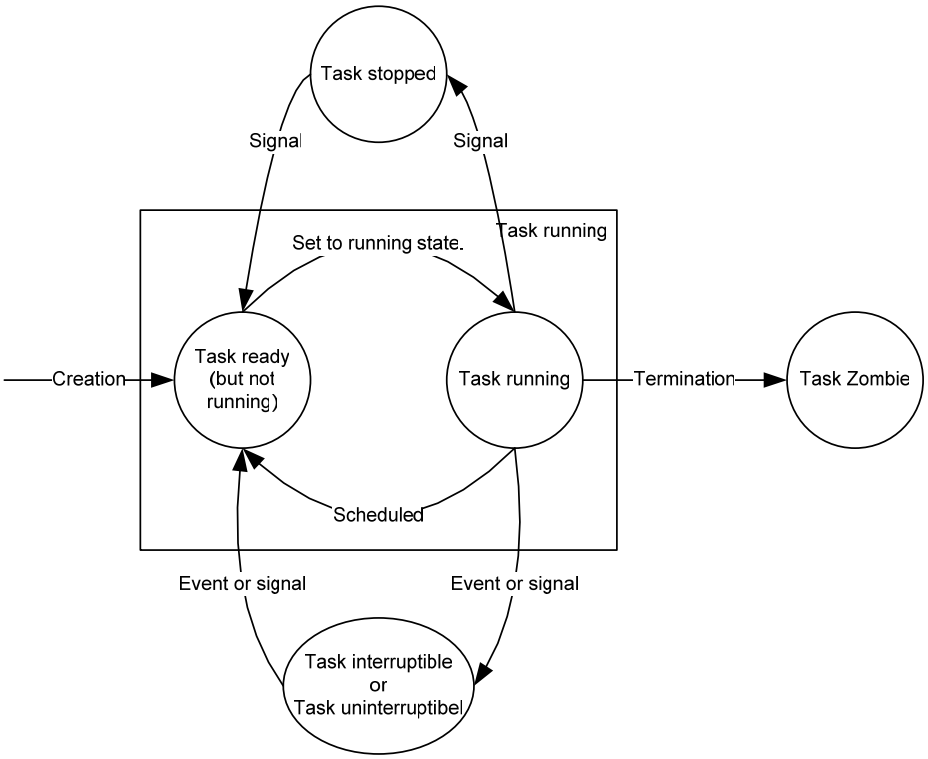


**Figure 2.3 Process states**

If a new process shall be created, the fork() command can be used. When fork() was executed a nearly identical copy (except the process identification value PID) of the parent process will be created in the memory. Then the exec() command can be called, this will load the new executable into the created address space.

To lower the overhead when fork() is called, Linux uses copy-on-write. This means that the parent and the child processes will share the address space until no data will be written. If the exec() command will be called immediately after the fork() command copy-on-write can save a lot of work. If there was no exec() command executed, it is possible that the child process never needs to write data. So there was no need to duplicate all data.

### 2.3.3 Linux file system and Virtual File System

In Linux nearly everything is (or can be) interpreted as a file. This means that a set of commands used for files (typical read, write, create or delete) can be used to do a lot of different things. For example process information, placed in the /proc directory, can be read out like a file. Even if there is no physical representation on some storage media, they will be generated on the fly when needed.

The Virtual File System (VFS) is a very important abstraction layer inside the Linux kernel. With help of this abstraction layer it is possible to use a lot of different file systems on Linux, see Figure 2.4.

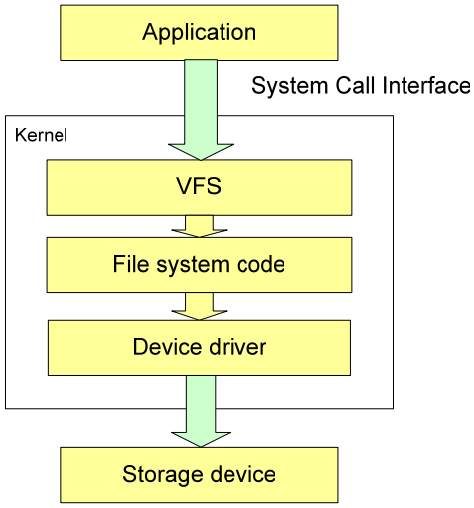


**Figure 2.4 VFS**

The VFS provides a standard interface to the kernel for file operation, for example create, open or delete a file. There is no need to know the specific instructions for a specific file system. Therefore, for each supported file system, the file system code is adapted to the actual file systems needs. Furthermore, through this abstraction, it is possible to use different file systems at the same time. They are also able to interoperate among themselves, for example copying files from one to the other. Even if there is a none UNIX compliant file[1] system it is often possible to mount[2] such a file system. Therefore the file system code has to provide a UNIX like interface to the VFS.

---

[1] A Unix compliant file uses, for example, superblocks to store information of the mounted file system and inodes which represents a specific file. Further information can be found in [SYS02].

[2] Linux uses a hierarchically organized file system. All directories are organized in a big tree structure. If a file system is placed into this structure, to get access to the stored data, this is called to mount a file system.

### 2.3.4 Memory Technology Devices (MTD)

Memory Technology Devices are primary used by embedded systems. Embedded systems are often transportable, so for example a hard disk drive used in desktop computers, wouldn't have a long life time. And they often have to be as small as possible. In comparison to hard disk drives solid state memories, such as Flash memory devices, are very small and easier to fit into embedded systems. Furthermore the high amount of storage space supported by hard disks is often not necessary in embedded system, so it can help to save costs when using a fitting solid state memory.

What are Memory Technology Devices? There is no further definition of MTDs, they are already Memory Technology Devices. Linux normally uses character devices or block devices. A character device, like a keyboard, delivers one character after the other. A block device, like a hard disk drive, reads or writes always a block of data (usually 512 bytes) and is able to seek for data on different storage positions. A Flash memory will not fit to this approach. So a new abstraction layer was introduced, see Figure 2.5.

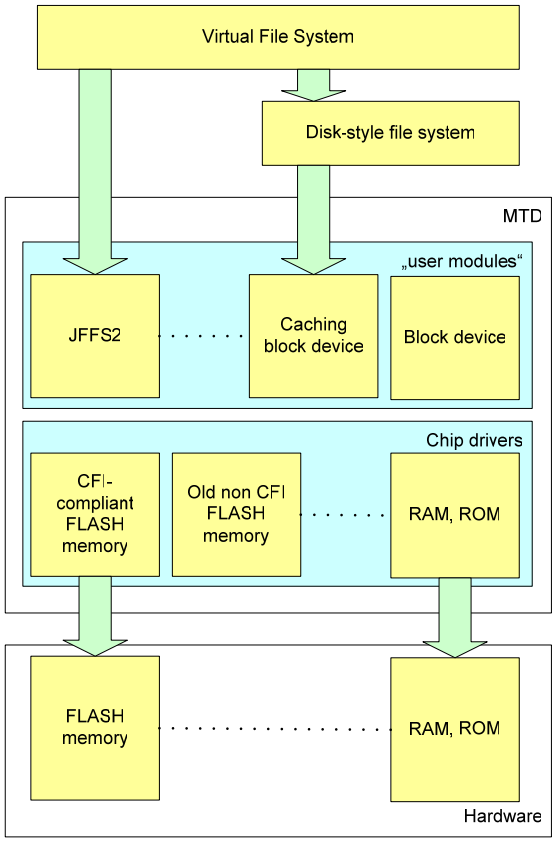


**Figure 2.5 MTD devices**

The MTD subsystem provides an interface to the hardware devices. From bottom up, first the MTD chip drivers provide the low level features needed for communicating with the

hardware. The “MTD user modules” (they have nothing to do with kernel or user space, they are already inside the kernel) will provide the higher level features to the system.

What is to be done when interfacing a MTD?

A suitable MTD chip driver is needed. This means that there are for example drivers for CFI-compliant Flashes[1], old non-CFI Flash memories and also for conventional RAM or ROM. Furthermore there are some other MTD chip drivers.

To get access to the available storage place a “MTD user module” is needed. For example the JFFS2 (Journalling Flash File System Version 2) or the “caching block devices” “MTD user module” can be used. The JFFS2 is a file system developed especially for Flash memory devices. It supports:

- Power down reliability: When system power goes down while writing some data, only the just written data can get corrupted. The file system itself will not be damaged.
- Wear leveling: Flash memories are organized in memory blocks, these blocks allow only a limited number of erase cycles. So data will be written uniformly over all available blocks.
- Data compression: To lower the costs for Flash memory, the data will be stored compressed on the Flash memory. If the data will be needed, they will be extracted into RAM first.

Furthermore no disk style file system is needed. The file system structure itself will be generated in the system RAM.

When using for example the “caching block device” “MTD user module” a block device interface will be provided. Therefore a disk style file system such as ext2[2] will be needed to provide a capable user interface.

Last but not least the kernel has to know about the MTD device. Therefore some settings inside the kernel configuration have to be done. An entry for the MTD device has to be generated in the /dev directory. The information provided in this subsection is based on [YAG03] chapter 7.

[1] The Common Flash Interface (CFI) is a specification developed by AMD and Intel. This provides a standardized interface to CFI compliant Flash devices.

[2] The second extended file system (ext2) is file system supported by the Linux kernel. These days its successor the third extended file system (ext3) is often used as file system for hard disks.

### 2.3.5 Kernel modules

Kernel modules are used to extend the functionality of the kernel. When configuring the kernel it is possible for some parts, for example a device driver, to compile the driver into the kernel or to build a module. Building a module means, that the kernel will not supply the functionality added by this module after starting up. Nevertheless, the functionality not now provided is packed into a module. Assumed that the kernel is configured to support “Enable loadable module support”, then modules can be loaded during runtime. When a new hardware is connected to the computer, and there is a module which enables the access to the hardware, this module can be loaded without rebooting the system.

Dealing with modules can help to get a smaller, better fitted kernel and nonetheless the possibility to add functionality to an existing and running kernel. Modules can be loaded automatically when starting up. So the advantages of kernel modules can be used without loading them manually. Even if the modules are loaded at start up they can be unloaded during run time.

As mentioned before, modules are running in kernel space. Therefore a painstaking programming and testing of modules is required. If there is a bug in the module, the whole system can die. Furthermore there are some restrictions when programming software running in the kernel space. There is for example no C-library, or no own stack for each module. This means that functions provided by the kernel itself have to be used, for example printk() instead of printf(). In user space it is possible to store huge data structures on the stack because every process holds its own stack, not so when programming in kernel space.

There are some special requirements when programming modules. They have to use the module_init(your_init_function) and the module_exit(your_exit_function) macro. These macros define the behavior of the module when loading or unloading it.

Further information on modules and samples can be found in chapter 7 or in [COR05].

### 2.3.6 Linux device drivers

The Linux device driver model is a very complex structure. So this section will not be able to discover the secrets of the whole device driver model. Very good explanations of the model can be found in [COR05] or [WLWN1] in the section driver model. Even if the device driver model itself will be outside the scope of this document, it is possible to write a device driver without knowing details on the device driver model.

First of all, what is a device driver? A device driver is used to get access to some hardware through a defined interface. Assumed that all necessary components are initialized, for example sending data via serial interfaces is only a write command to some file. Behind the file there is the device driver, like a black box, which is concerned to get access to the hardware.

Linux knows three classes of device derivers:

**Character devices:**

Character devices get or send one byte after the other, normally jumping around in the received data is not possible. Character devices are represented by a file in the /dev directory. For example, a serial port is a character device.

**Block devices:**

In contrast to character devices block devices read or write a block (normally 512 bytes) of bytes. Furthermore jumping around and getting access to data stored on different locations are possible. For example, a hard disc drive is a block device.

**Network interfaces:**

A network device uses interfaces to exchange data with other hosts. This device is completely different to the first two devices.

When developing device drivers, the following things have to be known, chosen or provided:

- The name for the device driver
- The type of the driver (char, …)
- A major number
- Operations supported by the driver

What is the driver major number? The device major number is used for associating the driver to a special device. Furthermore, different instances of one driver can be associated with different devices of the same type. When more than one device uses the same major number, a so called minor number is used to identify one device. Traditionally some major device numbers are defined and can be found in Documentation/devices.txt. In modern Linux it is also possible to get a dynamic major number by the kernel. When developing drivers for different systems, a dynamically assigned major number should be preferred to avoid conflicts between already existing major numbers. When developing special adapted systems, such as embedded systems, there will be no disadvantage in using fixed major numbers.

Furthermore the driver needs to know what to do when receiving commands. Therefore, in case of a character device driver, a so called file_operations structure (fops) is used. This structure includes a field for every specific function, and will be filled with pointers to the functions which implement the supported functions.

Further information on device drivers can be found in [COR05].

### 2.3.7 Interrupts

An interrupt is a signal from a device to the processor that indicates that the device needs some attention. When there is a pending interrupt the processor informs the operating system

about it, further work has to be done by the operating system. For example, a serial port receives some data over the serial line. The managing device for this serial port can issue an interrupt when data arrives. So the operating system knows that there is some data in the input buffer. There are the following advantages when using interrupts: It is not necessary to poll[1] the buffer of the serial port to see if there is some data. Furthermore it is possible to react very fast on an interrupt request.

For knowing the device which has issued the interrupt there are processor specific approaches. For example the processor itself offers some input lines for interrupts and stores the action to do in a specific interrupt table. It is also possible to use an external interrupt controller.

A function which defines what to do after a specific interrupt is called an interrupt handler. When the processor works on an interrupt handler it is running in interrupt context, because the interrupt can occur every time the processor currently works on an unknown process. This process will be interrupted and the processor will change to the interrupt context. There will be no separate process for the interrupt handler. Hence, the following things are important. The interrupt handler uses the stack of the interrupted process, be careful to clean up correctly. An interrupt handler is a time critical function; it should be as short as possible. Furthermore, as mentioned before an interrupt handler is not associated with a process, it cannot sleep. Hence there are some functions that can not be used inside the interrupt handlers.

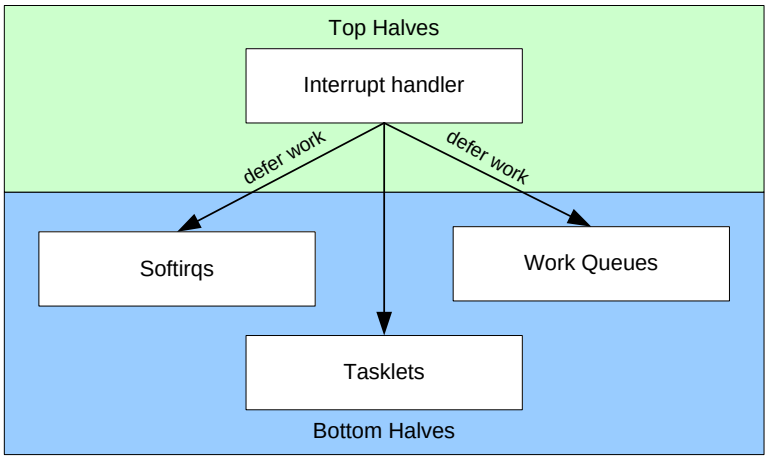


**Figure 2.6 Interrupt handling**

As mentioned before, an interrupt handler is a time critical function. Furthermore, while processing an interrupt the assigned interrupt level or all interrupts will be disabled. The

[1] Polling is defined as the periodic checking of a device's status. In order to avoid wasting time doing software loops interrupts could be used.

interrupts should be enabled again as soon as possible. What can a programmer do when receiving a lot of data requiring further work? A network interface device driver will be a good example for that. Figure 2.6 shows the solution. The work is split into the bottom and top halves. The interrupt handler (top halves) itself, for example copies only data from the device memory to the system memory. This could be very easy to do and will not take a lot of time. The “real work” is done by the bottom halves. Work that is really time critical has to be done in the top halves. The programmer has to know which work can be deferred to the bottom halves. When using bottom halves the following is very important. It is unknown when the bottom halves will be executed, even if it is immediately after the interrupt handler.

There are three different types of bottom halves. Historically there were also other types of bottom halves, but the kernel version 2.6.x supports only the following:

**Softirqs:**

There are only 32 softirqs available. In the actual kernel 2.6 series only 6 softirqs are used. They are used for really time critical bottom halves. Softirqs will be checked for execution in a suitable way, for example after processing a hardware interrupt. The disadvantages of softirqs are that there is only a limited number and they have to be defined on compile time.

**Tasklets:**

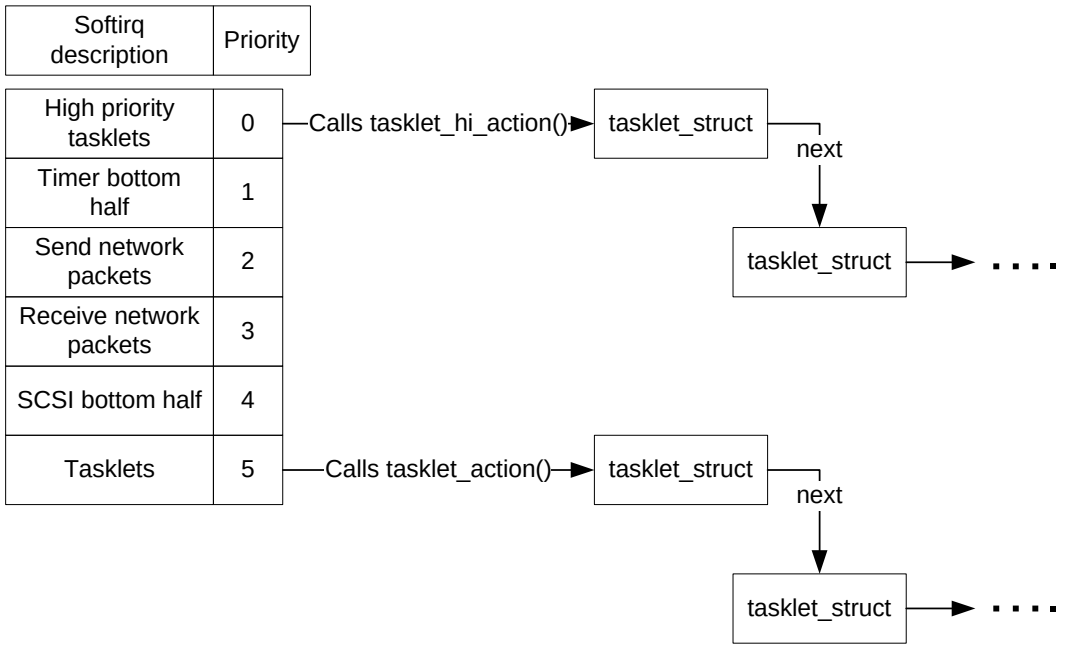


**Figure 2.7 Softirqs and Tasklets**

Even if the name is ambiguous they have nothing to do with tasks. Tasklets are based on softirqs. Usually tasklets should be used. As seen in Figure 2.7 there are two priority levels of tasklets, even if these levels only define the order they will be processed. Scheduled tasklets

(they are waiting for execution) are stored (more exactly, a structure including a pointer to the handler function is stored) in two linked lists, depending on their priority. So the big advantage of tasklets is, that they can be dynamically created during runtime and there is theoretically no maximum number of them.

**Work Queues:**

In contrast to softirqs and tasklets (they are running in interrupt context) work queues are running in process context. Hence, work queues are able to sleep. This will make the decision what to use, tasklets or working queues. The deferred work will be processed by so called worker threads; they are kernel threads. Generally speaking, work queues provide an interface to defer work to a kernel process. Similar to tasklets the information on the work to do is stored in a structure. All the deferred work (represented by work_struct structure) is stored in a linked list and will be processed one after the other by a worker thread.

The information about bottom halves is a summary of [LOV04] Chapter 6.

## 2.4 User interfaces

### 2.4.1 Basics

A lot of information on Linux was provided in this chapter until now. The kernel itself provides interfaces for some software but no human usable interface. There are special programs to provide a user interface. Sure, not all embedded systems must have a user interface when they are installed. Furthermore when hearing the words user interface, a lot of people think of a standard screen or a keyboard. Using a screen is not really impossible, but normally embedded systems use little displays and some buttons like a mobile phone, or merely some buttons, and if there is any, a non graphical display like a washing machine.

Moreover this document will discuss how to develop embedded systems, not how to build a user interface. Furthermore the user interface during developing can be different from the finally used interface. A standard user interface during developing will be a text based shell.

### 2.4.2 The shell

A shell provides a command line interface to the system. It uses kernel functions to provide the functionality needed by the user, even if this will be hidden to the user. Basically a shell is nothing else than a “simple” program running in user space.

Shells offer a lot of different features. For example there are functions provided by the shell itself, the shell could start programs and it is possible to write shell scripts. More information about a specific shell, for example the BASH (Bourne Again Shell), can be found at

[WGNU1]. As there are a lot of different shell implementations it would be out of the scope to explain them all. For embedded systems a shell called Busybox is used.

The Busybox project provides a complete environment (including a text based user interface as described above), have a look at [WBUS1]. In contrast to standard shells the Busybox includes a lot of commands within one file. For example the ls command (prints out a directory listing) is not a dedicated program executed by the shell, the Busybox includes the ls command. Nevertheless the Busybox places soft links[1], named like the corresponding functions, in a directory. All soft links placed by the Busybox, point to the Busybox itself (because all functions are included in the Busybox). Based on the called soft link, the Busybox will decide what to do.

The advantage of using Busybox in embedded systems is that it was programmed with the view to use as little storage space as possible.

In later chapters the Busybox will be used as interface to the hardware described in section 1.3.

### 2.4.3 Terminal programs

As shown in section 1.3 the used development hardware has nothing like a keyboard or screen. As explained in subsection 2.4.2 the Busybox will be on the system. To communicate with the Busybox the development hardware has to be connected to a computer. Hence a so called terminal program is used to show the text received from the development hardware or send the keyboard input to the development hardware.

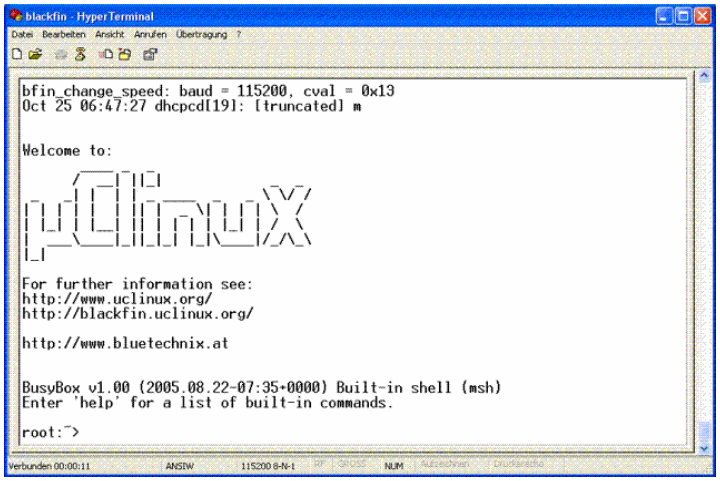


**Figure 2.8 Hyper Terminal**

[1] Soft links are special directory entries. They only point to the physical representation of a file.

If using Microsoft Windows on your developing machine a terminal program called hyper terminal is included, have a look at Figure 2.8. When Linux is installed on your developing machine for example Kermit [WCOL1] or minicom [WDEB1] can be used.

## 2.5 The kernel boot procedure

This section should provide a short overview of the kernel boot up procedure. There are no major differences between a standard Linux kernel and the kernel used in embedded systems.

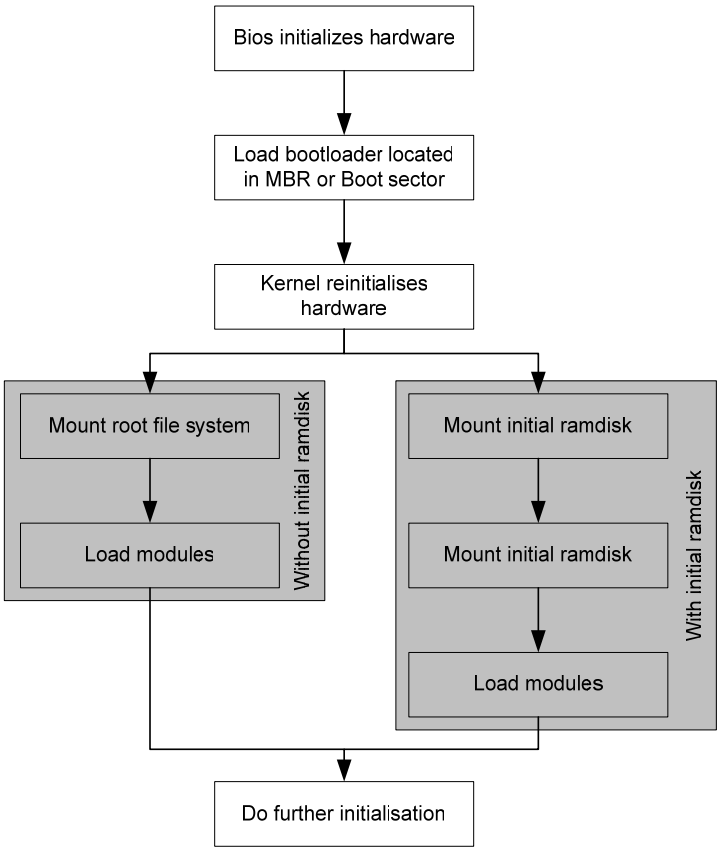


**Figure 2.9 Linux boot process**

Figure 2.9 shows the standard Linux/Kernel boot process, based on a standard work station. On an embedded system the bootloader will start immediately. After the BIOS[1] has finished the initialisation it loads the bootloader into RAM and executes it. The tasks of the bootloader are to load the kernel, and if used to load also the initial ramdisk into RAM from a non

[1] The Basic Input/Output System (BIOS) does the necessary work to initialise a standard work station after power up. If the work of the BIOS is done, the machine should be able to boot some boot loader or operating system.

volatile storage device. The initial ramdisk can be used to store kernel modules, in this way a tightened up kernel can be used. Once the initial ramdisk is mounted, all eventually needed modules can be loaded. This can be used, for example, if it is unknown what file system is needed to mount the root file system. So drivers for all supported file systems can be placed into the initial ramdisk, and the one which is needed will be loaded dynamically. After that the real root file system can be mounted.

If no initial ramdisk is used, all drivers to mount the root file system have to be compiled into the kernel. So the kernel can directly mount the root file system. All further modules can be loaded directly from the mounted root file system.

When the “Do further initialisation” box is reached, the kernels is already up and running. In this step, if using an initial ramdisk, the real root file system can be mounted. Furthermore the first real process will be created (the parent of all other processes) and some startup scripts will be invoked.

## 2.6 The Root File System

As already mentioned in Linux nearly everything is represented like a file. So the root file system is a very important part of the operating system. Even if the kernel is not stored within the root file system it is impossible to get a running system without any root file system. The kernel will run its start up sequences and when it tries to mount the root file system, it will stop working with the message: “Kernel Panic: VFS: Unable to mount root fs on …”

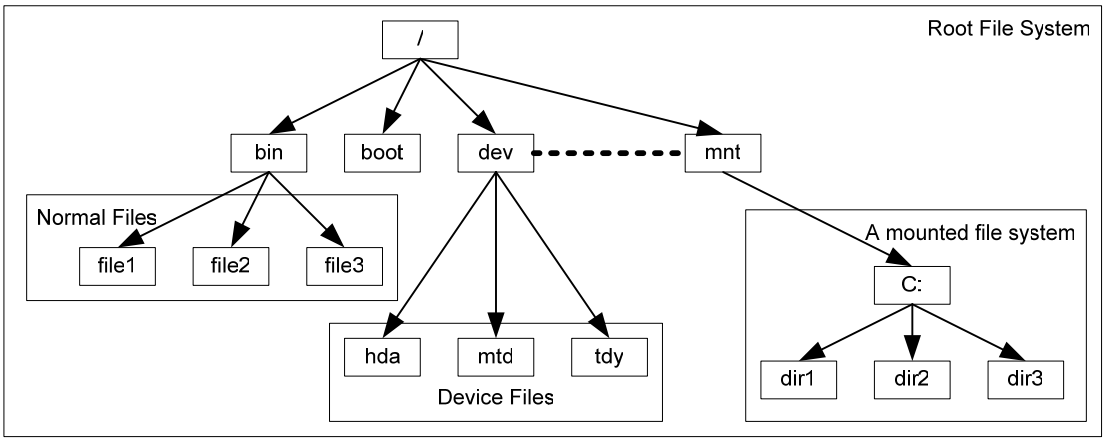


**Figure 2.10 Root file system**

The root file system itself can be based on different hardware devices. On a standard workstation it will be stored on the hard disk drive, in embedded systems on some solid state memory. There are also a lot of different file systems suitable for the root file systems. On

embedded systems often some kind of compressed root file systems is used. This helps to save Flash memory space.

The root file system uses a hierarchical tree structure. The top level is called root, represented by the "/" character, have a look at Figure 2.10.

The Figure 2.10 provides more information than only the structure of the file system. The /bin directory holds 3 "normal" files. When using the ls command to show a directory entry this looks like:

```
root:/bin> ls -l date
-rwxr--r--  110 1000     100          406468 date
```

As already discussed in 2.6.3 there are some device entries in the /dev directory:

```
root:/dev> ls -l mtd1
crw-r-----    1 0        0          90,   2 mtd1
```

MTDs were discussed in subsection 2.3.4. The leading 'c' character indicates that this device is used as a character device. Also the major number (90) and minor number (2), discussed in subsection 2.3.6, will be displayed.

Furthermore Figure 2.10 shows a mounted file system, this could be any file system supported by Linux. If the file system was mounted, there is no need to know where the file system is physically stored. It is fully integrated into the root file system. If the mounted file system is a non UNIX compliant file system, the driver has to supply a UNIX compliant interface.

| Directory | Description |
|---|---|
| bin | Essential command binaries |
| boot | Static files of the boot loader |
| dev | Device files |
| etc | Host-specific system configuration |
| lib | Essential shared libraries and kernel modules |
| media | Mount point for removeable media |
| mnt | Mount point for mounting a filesystem temporarily |
| opt | Add-on application software packages |
| sbin | Essential system binaries |
| srv | Data for services provided by this system |
| tmp | Temporary files |
| usr | Secondary hierarchy |
| var | Variable data |

**Table 2.1 Description of directories [WPAT1]**

Table 2.1 shows mandatory directories as explained by the "Filesystem Hierarchy Standard" [WPAT1]. There is the description of which requirements have to be met by a file system for UNIX like operating systems. Nevertheless not all Linux distributions are using these

guidelines. Furthermore in embedded systems a root file system adapted to meet the special requirements of the application will be used.

# Chapter 3 Problem analysis

## 3.1 Feasibility of Linux on the Blackfin processor family

Some information on Linux was provided in Chapter 2 . Comparing the resources of a standard work station with the used development board, described in section 1.3, it looks impossible to run Linux on such a minimized system. But, thinking back a few years ago, Linux has already run on a personal computer having 4 MB RAM. The development hardware uses 32 MB, so there should be no problem to run Linux on it.

Maybe the 2 MB Flash memory is too small to store the whole program code needed? If the Linux is stored uncompressed in the Flash memory, there definitely will be not enough free space. On embedded systems normally a compressed image is stored into the Flash memory, so 2 MB of Flash memory should be enough free space.

Linux needs a CPU that supports virtual memory and memory protection. The MMU offered by the Blackfin processor family is not able to do that. The MMU supports only memory protection. So it is impossible to run a standard Linux kernel.

As already mentioned in the introduction, there is a uClinux port available for the Blackfin processor family. Further information on the Embedded Linux/Microcontroller project can be found at [WUCL1].

uClinux was initially built to run on MMU (Memory Management Unit) less microcontrollers. Today uClinux has grown up to a fully featured operating system with a lot of useful utilities. It is able to run a large range of different processors and it is still requiring a much smaller memory footprint than standard Linux. This is reached by, among other things, using other system libraries.

As there is a uClinux port available for the Blackfin processor family, it should be possible to adapt uClinux to the Bluetechnix development hardware.

## 3.2 The U-Boot and the uClinux on the CM-BF533

Figure 3.1 shows all parts that have to be fitted together for booting up uClinux. Even if uClinux doesn't need a bootloader to boot up, a bootloader is recommended while developing. Once the bootloader is copied to the Flash memory, the hardware is able to start up and to connect to the development workstation. If the bootloader is up and connected to the workstation, the uClinux image can be easily placed into the available Flash memory.

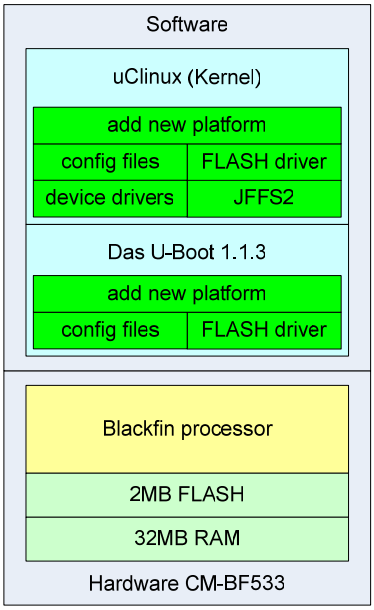


**Figure 3.1 System overview**

If the developing process is finished, only uClinux can be placed into the Flash memory. Regardless of this uClinux feature, it is recommended to use the bootloader anyway, so the system will be easily serviceable for all its lifetime.

As can be seen in Figure 3.1 there are two major parts to start with. First the bootloader "Das U-Boot" has to be adapted to the development hardware. The U-Boot is needed to handle the uClinux images. For example the U-Boot can download the uClinux image from the development workstations, store it in the Flash or start it. Once the U-Boot is running on the CM-BF533, no further hardware for programming the Flash memory is needed anymore. While developing the U-Boot a JTAG interface is needed to program the Flash memory.

First of all it is important to get familiar with the development environment, the U-boot source code and the uClinux source code. Therefore the Analog Devices STAMP board can be used, there is already a running uClinux port available for this board. Getting familiar with the development environment means setting up the toolchain, building the U-Boot and building uClinux images. These images can be tested with the STAMP board.

Once there is enough information on the processes described above, the work to adapt the U-Boot and the uClinux to the Bluetechnix development hardware can start. It is recommended to start with porting the U-Boot in order to be able to work without the need for a JTAG interface. There are three major tasks when porting the U-Boot the first time:

- Bringing up the U-Boot on the CM-BF533
- Setting up the Ethernet connection to enable a fast image download.
- Being able to store these images into the Flash memory.

To allow an easy configuration for the U-Boot on the CM-BF533 the board has to be integrated in the U-Boot source code. This means to generate board specific configuration files and to be able to build the U-Boot with a CM-BF533 specific command.

Once the U-Boot is running, the porting of uClinux can start. The steps to do are similar to the steps required for porting the U-Boot:

- Bringing up the uClinux on the CM-BF533
- Setting up the Ethernet connection.

It seems, as if this would be less work than porting the U-Boot. But when comparing the U-Boot with the uClinux kernel, the U-Boot is a relatively small project. Inside uClinux there are a lot of different configuration files that need some attention. For example, Bluetechnix has to be added to the vendors' directory and special kernel settings have to be provided for the CM-BF533. That means that all CM-BF533 related configurations can be done using the uClinux kernel configuration utility.

Finally a software package that includes a tested toolchain, the U-Boot for the CM-BF533 and the adapted uClinux has to be built. This package should be the basis for further development on this hardware.

Last but not least this diploma thesis should help getting familiar with uClinux. Therefore an introduction on Linux and uClinux has to be provided. Based on this instruction the development of user space program and device drivers will be shown.

# Chapter 4 Das U-boot 1.1.3

As mentioned before a bootloader has several advantages while developing uClinux. This chapter will explain how to set up a running the U-boot. As explained in Chapter 1 the used development hardware is a CM-BF533 Core Module on a DEF-BF5xx Development Kit.

## 4.1 Basics

The U-Boot is used to decompress the Linux image stored in the Flash memory. Because there are only 2 MB of Flash memory available, this is not enough space to store the uClinux image uncompressed. The uClinux image includes the kernel and the root file system. Furthermore the U-Boot is able to download uClinux images over the serial port and the Ethernet. The downloaded image is stored into the RAM. The U-Boot can start these images directly out of the RAM, or flash it to the Flash memory. It should also be mentioned, that the U-Boot can boot up other applications than uClinux too. This can be used to test the U-Boot and simple board functions.

While developing the U-boot itself a connection to the development workstation via serial port or USB port is needed. A terminal program, mentioned in subsection 2.4.3 is used to communicate with the U-Boot. For downloading the U-Boot binary file into the Flash memory a JTAG interface is needed.

### 4.1.1 The source code, cross compiling and the toolchain

This subsection provides some information on the necessary software on a development work station to build programs that are able to run on the target platform. With target platform the development hardware, the CM-BF533 is meant.

This document will not provide any information on how to setup the development work station. It is assumed that a work station running Linux is available. Even if there is a

Cygwin[1] port available for developing uClinux, no information about developing under Microsoft Windows will be given. For example, while writing this document, it is impossible to compile source codes checked out from CVS[2] without changes under Cygwin.

The following requirements have to be fulfilled by the work station to develop the U-Boot and uClinux:

- A serial port or a USB port
- An Ethernet interface
- Not less than 1GB of free hard disk space
- A terminal program
- A development environment or a simple text editor
- A TFTP[3] server (optional, only when using Ethernet)
- Support for CVS (optional)

As the development work station is based on different hardware architecture than the target hardware cross compiling is needed.

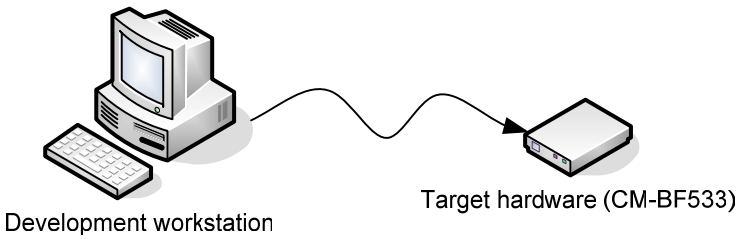


**Figure 4.1 Cross compiling**

Figure 4.1 illustrates the purpose of cross compiling. This means that the developing workstation compiles the source code for execution on a physically different hardware. Even

[1] Cygwin is a program that simulates a UNIX link environment under Microsoft Windows. Software that is POSIX compliant should be able to run with Cygwin.

[2] The Concurrent Versions System (CVS) is used to store source code files. Programmers will be able to work together using CVS. Everyone (assumed he has the needed permissions) can check in or out files. All changes to files are stored and can be monitored.

[3] The Trivial File Transfer Protocol (TFTP) is used to transfer data via a network. The TFTP protocol is very simple, so it can be integrated in embedded devices and systems with a small amount of memory.

if the work station is able to build programs for the target platform, it will not be able to execute the created code.

Figure 4.2 shows a standard procedure how to create executables when cross compiling.

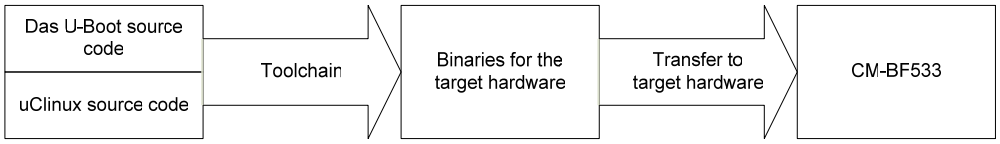


**Figure 4.2 Developing process**

A very important part in Figure 4.2 is the toolchain, so further information will be supplied. The toolchain includes all software which is necessary for cross compiling, this includes:

- Binutils: Collection of Binary utilities, for example an assembler or a linker is included.
- elf2flt: Converts ELF to flat binary format
- gcc: The GNU C compiler.
- gdb: The GNU debugger, will not be explained in this document.
- genext2fs: Generates the EXT2 file system for the RFS.

Figure 4.3 explains the way from the source code to the executable code for uClinux.

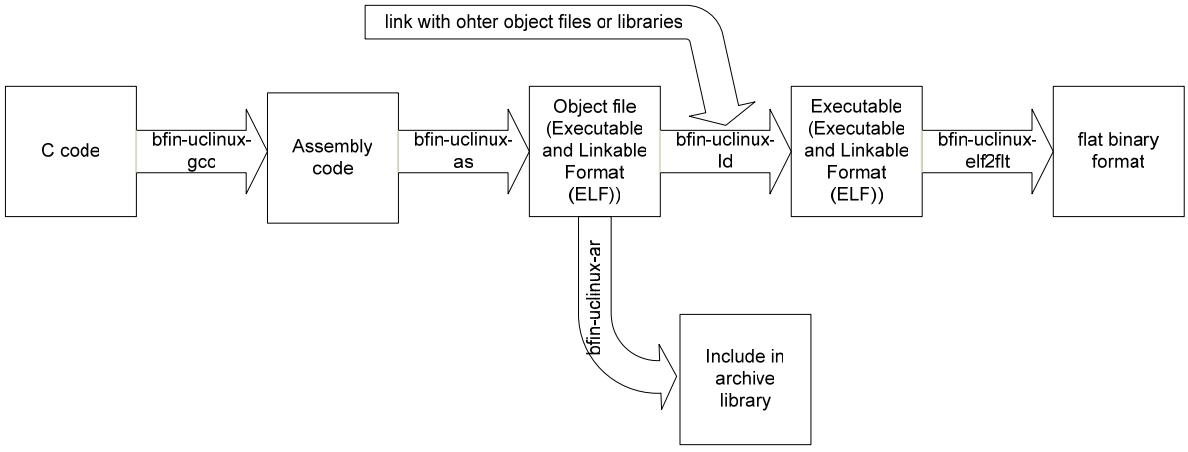


**Figure 4.3 Code to binary**

To set up a working toolchain several hardware packages are needed. As already mentioned, the toolchain is necessary to compile source code. For building the toolchain also the uClinux

package is necessary, because this package includes some header files needed by the toolchain build process. Hence, three packages are needed for developing the U-Boot und uClinux.

- The toolchain source code
- The Das U-Boot source code
- The uClinux source code

These three packages can be downloaded from http://www.bluetechnix.at. These files have to be copied to one directory holding the source code. Performing the following commands on the work station will set up the toolchain:

```
tar -xvzf toolchain.tar.gz [enter]

tar -xvzf u-boot_1.1.3.tar.gz [enter]

tar -xvzf uClinux-dist.tar.gz [enter]

./buildscript/BuildToolChain -s /source_code_directory/ -k
/source_code_direcotry/uClinux-dist -b /toolchain_target_directory/build -o
/toolchain_target_directory/out [enter]

export PATH=$PATH:/toolchain_target_directory/out-uclinux/bin
```

The last statement has to be entered after each reboot. To avoid this, the path to the toolchain must be included into the default environment variables.

Two types of toolchains will be created by the buildscript. The “bfin-elf-toolchain” is set up to build standalone applications (without operating system), and the bfin-uclinux-toolchain is optimised for the use with elf2flt.

### 4.1.2 The development board and the console settings

Figure 4.4 shows how to set up the development board, to avoid problems with the hardware set all jumpers and switches like this figure. Table 4.1 shows the address mapping of the development board. These addresses are needed to configure the U-Boot

As shown in Figure 4.4 for the connection to the development hardware there is a serial port and a USB port available. For the communication only one interface can be used at the same time, for the connection via USB a USB to serial converter is placed on the development board. In later texts only the “serial port” will be mentioned, this includes to use the USB to serial converter. Using USB port is recommended because no further power supply is necessary.

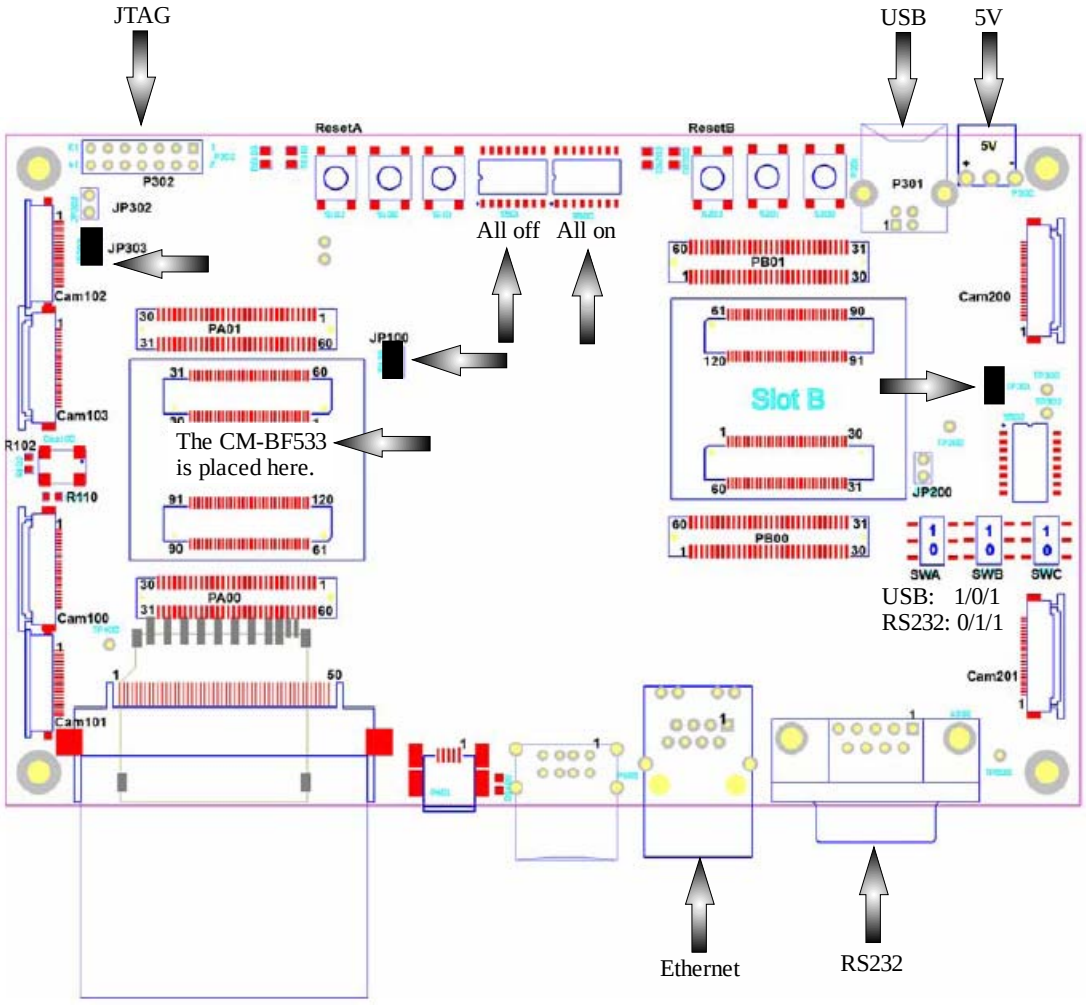


**Figure 4.4 DEV-BF5xx Development KIT [WBLU1]**

| CM_BF533 | | from | to |
|---|---|---|---|
| | SD-RAM | 0x 0000 0000 | 0x 01FF FFFF |
| AMS0 | Flash | 0x 2000 0000 | 0x 200F FFFF |
| AMS1 | Flash | 0x 2010 0000 | 0x 201F FFFF |
| AMS2 | Ethernet | 0x 2020 0000 | 0x 202F FFFF |
| AMS3 | NOT AVAILABLE | | |
| AMS4 | Dual ported RAM | 0x 2030 0000 | 0x 2030 3FFF |
| AMS5 | Dual ported Semaphore Register | 0x 2030 4000 | 0x 2030 7FFF |
| AMS6 | USB-OTG | 0x 2030 8000 | 0x 2030 BFFF |
| AMS7 | Compact Flash CS1 | 0x 2030 C000 | 0x 2030 CFFF |
| AMS8 | Compact Flash CS2 | 0x 2030 D000 | 0x 2030 DFFF |
| AMS9 | not used | 0x 2030 E000 | 0x 2030 EFFF |
| AMS10 | not used | 0x 2030 F000 | 0x 2030 FFFF |

**Table 4.1 Address mapping [WBLU1]**

On the development board shown in Figure 4.4 the CM-BF533 Core Module (Figure 4.5) is placed. The ADSP-BF533 processor supports four different boot modes. For the U-Boot the boot mode 00 has to be used. As described in [ANA04] the boot ROM will be bypassed and the execution will start from 16-bit external memory at address 0x20000000 when the boot mode 00 is set. Setting the boot mode 00 means that no resistors are placed on the two marked positions in Figure 4.5.

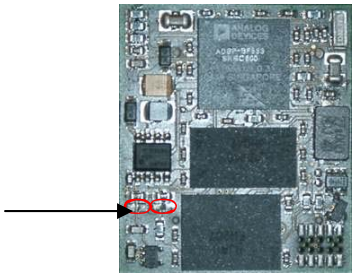

**Figure 4.5 CM-BF533 boot mode 00**

Assuming the use of Kermit [WCOL1] as Terminal program, for a stable communication the following settings should be applied. Theses settings can also be stored in ~/.kermrc.

```
set line /dev/ttyS0 or set line /dev/ttyUSB
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

The first line depends on how the hardware is connected to the development workstation. The second line depends on the setting of the communication speed done in the U-Boot and the uClinux configuration files.

When using another terminal program, the following settings are the minimum requirement for communicating with the hardware:

Speed: depends on hardware settings, usually 115200 bit/sec

Parity: none

Stop bits: 1

Flow control: none

## 4.2 Booting and porting the U-boot

### 4.2.1 Overview of booting up

Figure 4.6 shows the essential files for the boot process.

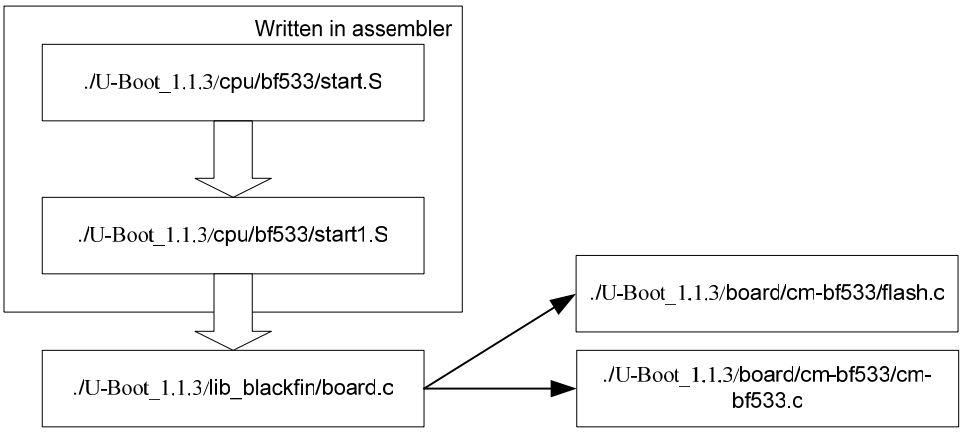


**Figure 4.6 Files included in the U-Boot boot process**

The first two files are written in assembler. They do the basic configurations such as initialising the processor registers or the external RAM. Furthermore the U-Boot itself is copied from the Flash memory into the internal RAM.

Then the function board_init_f() included in ./U-Boot_1.1.3/lib_blackfin/board.c is called. From now on, further initialisations will be done in high level C code. Because this function does a lot of very important hardware initialisations, a few lines of source code shall be shown:

```
------cut------

/* Initialize */
init_IRQ();
env_init();             /* initialize environment */
init_baudrate();        /* initialze baudrate settings */
serial_init();                  /* serial communications setup */
console_init_f();
display_banner();       /* say that we are here */
checkboard();
#if defined(CONFIG_RTC_BF533) && (CONFIG_COMMANDS & CFG_CMD_DATE)
        rtc_init();
#endif
timer_init();
printf("Clock: VCO: %lu MHz, Core: %lu MHz, System: %lu MHz\n", \
CONFIG_VCO_HZ/1000000,CONFIG_CCLK_HZ/1000000,CONFIG_SCLK_HZ/1000000;
printf("SDRAM: ");
```

```
print_size(initdram(0), "\n");

------cut------
```

When the board doesn't start up, the searching of the problem can be started at the source code provided above.

As shown above, a lot of functions do a lot of different work. It is out of the scope of this document to describe them all. Often the names of the functions are self explaining. When something goes wrong inside these functions, a step by step manual cannot be provided.

Another interesting point included inside this source code sample is that the functions checkboard() and initdram() are called. These functions are included in the board specific file ./U-Boot_1.1.3/board/cm-bf533/cm-bf533.c. A few lines below the provided piece of source code also the function flash_init() will be invoked, this function is included in ./U-Boot_1.1.3/board/cm-bf533/flash.c. All constants like CONFIG_RTC_BF533 and so on are defined in ./U-Boot_1.1.3/include/configs/bf-533.h. If everything works, the following lines should be seen on the terminal program:

```
U-Boot 1.1.3 (Jul 15 2005 - 08:36:17)

CPU:   ADSP BF533 Rev.: 0.3
Board: Bluetechnix CM-BF533 board
       Support: http://www.bluetechnix.at/
Clock: VCO: 594 MHz, Core: 594 MHz, System: 118 MHz
SDRAM: 32 MB
Flash: id = 16, manufacturer_id = 89
Device ID of the Flash is 890016
Flash Memory Start 0x20000000
Memory Map for the Flash
0x20000000 - 0x201FFFFF Single Flash Chip (2MB)
Please type command flinfo for information on Sectors
FLASH:   2 MB
```

What to do if there is no output will be shown later.

### 4.2.2 Adding the CM-BF533 to the U-Boot

This subsection should provide an overview on how to add a new board to the U-Boot.

It is recommended to create new files based on an existing board and to use these files as a template for the new board. In the case of the CM-BF533 the hardware is similar to the Analog Devices BF533 EZ-KIT Lite, except the incompatible Flash memory. So the settings and files used by this board are a good template to start with.

Further information on some specific files is provided in subsection 4.2.3, subsection 4.2.4 and subsection 4.2.5. The following files or directories need some attention when adding the CM-BF533 to the U-Boot source code. When files or directories are shown, the given path is relative to the development directory holding the U-Boot subdirectories.

**./U-Boot_1.1.3/Makefile:**

This file holds the initial settings for the U-Boot build process. Here the name for the new board has to be added.

Adding the CM-BF533 to the Makefile is simple. This file is located in ./U-Boot_1.1.3/Makefile. To generate a new entry for the CM-BF533, the following lines have to be added to the Makefile. The Makefile holds a separate section for the Blackfin processor family. That is where to place the following lines.

```
cm-bf533_config    :      unconfig
  @./mkconfig $(@:_config=) blackfin bf533 cm-bf533
```

First the “cm-bf533_config” term is used to identify the specific board by the make command. Second, the last term has to be fitted for the new board. This term defines the name of the configuration file (subsection 4.2.3) and the name of the directory holding the board specific files. The term “blackfin bf533” is used by the toolchain to set the right parameters for building the U-Boot.

**./U-Boot_1.1.3/board:**

Here a new directory for a new board has to be added. This directory holds some files; these files are called during the board initialisation process. All specific files for the board have to be placed there.

The board specific directory will be called ./U-Boot_1.1.3/board/cm-bf533/. It has to include the following files:

- A Makefile, config.mk and u-boot.lds; used for board specific compiling and linking
- cm-bf533.c and flash.c (including flash.h); the board specific files

Like above another existing board can be used as template. Assuming that the directory ./U-Boot_1.1.3/board/ezkit/ will be used as template, the following work has to be done. The first three files need less action for adapting to the CM-BF533 needs.

In the Makefile the object file which will be included has to be renamed to cm-bf533.o.

There are no changes in the config.mk file; the EZ-KIT lite also uses 32 MB SDRAM, so this address is also inside the RAM. The config.mk file defines the constant TEXT_BASE, this constant defines where to place the U-Boot text section in the RAM. In the U-Boot configuration file the constant CFG_MONITOR_BASE is defined, that also defines the physical start address of the U-Boot. Furthermore it is defined that the U-Boot uses the last 256 kB of RAM. As there are 0x2000000 Byte of RAM available, the last 256 Byte start at the address 0x1FC0000. This value is placed in the file config.mk.

The linker file u-boot.lds includes the board name and directory, this entry has to be set to ./U-Boot_1.1.3/board/cm-bf533/cm-bf533.o.

**./U-Boot_1.1.3/include/configs:**

The settings for each supported board are stored in one file. A configuration file for the CM-BF533 has to be placed there.

All not now mentioned files will be explained in detail in later sections.

### 4.2.3 The configuration file for the CM-BF533

It is recommended to use an existing configuration file as a template for the CM-BF533. The board specific configuration file can be found in ./U-Boot_1.1.3/include/configs/cm-bf533.h. The entries of this configuration file depend on the used processor and the additional hardware placed on the development board. Studying some config files will help to get familiar with the possible entries. For example not all config files include settings for an additional Ethernet controller.

The configuration file can be cut into the following pieces:

- Settings for essential hardware, such as CPU clock, RAM size, Flash memory size and so on.
- Settings for additional hardware, for example the Ethernet controller
- Settings for the U-Boot itself
- Settings to define the standard environment variables

Not all lines of the cm-bf533.h configuration file will be explained. A lot of define tags used in the configuration files are self explaining when reading their names. Furthermore an introduction to a lot of define tags can be found in ./U-Boot_1.1.3/README, especially all hardware independent functions will be explained there. Here only some special configuration tags are picked out and explained, this selection is based on the gained experience while porting the U-Boot.

**CONFIG_BAUDRATE:**

For communication with the hardware using the serial port it is necessary to set a baudrate for the communication. Possible values are 9600, 19200, 38400, 57600, 115200 bit/s.

**CONFIG_RTC_BF533:**

If there are problems with the RTC[1], or if there is the possibility that something is wrong with the RTC don't use this setting. If CONFIG_RTC_BF533 is defined and something goes wrong with the RTC the U-Boot will hang at the RTC initialisation routine. Then no output will be created on the console. As can be seen in the source code provided in the subsection 4.2.1, there the initialisation routine for the RTC will not be called if CONFIG_RTC_BF533 is not set.

**CONFIG_STAMP:**

Even if it is not oblivious CONFIG_STAMP has to be set for the CM-BF533. It is needed for the Flash memory initialisation. If it is not set, no Flash memory will be detected by the U-Boot.

**CONFIG_MEM_MT48LC16M16A2TG_75**

For the SDRAM initialisation a few SDRAM specific timing parameters will be needed. The supported SDRAMs can be found in ./U-Boot_1.1.3/ include/asm_blackfin/mem_init.h. Select a suitable memory and set the assigned CONFIG_MEM_XXX.

**CFG_ENV_ADDR, CFG_ENV_SECT_SIZE:**

One sector of the Flash memory can be used to store the U-Boot environment variables. Section 4.3 provides more information on the U-Boot environment variables. CFG_ENV_ADDR is used to select the sector where to store the environment's variables. CFG_ENV_SECT_SIZE configures how much environment data can be stored. While developing the U-Boot for the CM-BF533, a test with CFG_ENV_SECT_SIZE = 0x20000 (a whole sector) was done. The U-Boot seemed to operate in the right way. But no Linux kernel could be started with these settings. So be careful when changing these settings, there is no obvious correlation between these settings and booting the uClinux kernel.

**CONFIG_COMMANDS**

Here the commands provided by the U-boot can be selected. All available commands can be found in ./U-Boot_1.1.3/README. For example no commands for the RTC are provided for the CM-BF533.

**CONFIG_BOOTARGS**

CONFIG_BOOTARGS can be used to transfer some parameters to the Linux kernel. For example the Kernel can be informed where to find its RFS.

[1] A Real Time Clock (RTC) is a unit that provides the time in a suitable format like seconds, minutes or hours. Normally a RTC stay powered even if the system is switched off.

**CONFIG_BOOTCOMMAND,CONFIG_BOOTDELAY,CONFIG_AUTOBOOT_PROMPT**

These commands are needed when the U-Boot shall automatically start a uClinux image. CONFIG_AUTOBOOT_PPOMPT has to be set to enable the U-Boot autoboot functionality. Further information on booting images automatically can be found in section 4.3.

All hardware specification settings should be understandable with a little look at the processor datasheet or Table 4.1.

It is highly recommended to keep in mind the suggestions below.

- Try to get an output on the console. This means changing only the processor specific parameters to get a running system. There are commands to test the hardware, for example to do a memory test.
- Don’t change too many settings a time. There can be curious side effects when changing a parameter.

### 4.2.4 Board specific initialisation file

This is the first really board specific file including a source code that is able to do some board specific work. This file can be found in ./U-Boot_1.1.3/board/cm-bf533/cm-bf533.c. It includes the following functions. All these functions are invoked by the file ./U-Boot_1.1.3/lib_blackfin/board.c as introduced in subsection 4.2.1.

**int checkboard(void)**

This function only has the basic function to print a board specific hello message. The standard return value is 0; regardless it will not be checked after invoking this function.

**long int initdram(int board_type)**

The name for this function is a little bit confusing. It doesn’t physically initialize the SDRAM. This function sets up some pointers needed for addressing the SDRAM. The return value of this function is the physically available amount of SDRAM in MB.

**int misc_init_r(void)**

This function can be used to initialize additional hardware. It is called only if CONFIG_MISC_INIT_R is set in the board configuration file. No further hardware will be initialized on the CM-BF533, so this function is not used. While writing this document, the return value will not be checked.

The source code for the described functions can be found on the CD included to this document.

### 4.2.5 The Flash memory driver

As shown in Table 4.1, the Flash memory is placed on address 0x2000 0000 to 0x201F FFFF. Due to its physical limitations of the Flash memory written data can only be erased block by block. When reading the Flash memory random access will be possible. How to write data to the Flash memory will be explained soon.

The used Flash memory offers 32 Mb (=4 MB) of available storage space. On the CM-BF533 there are only 2 asynchronous memory banks used to get access to the Flash memory. Each of these two banks is able to address 1 MB of external memory. So there is only 2 MB of the Flash memory available. The 2 MB of available Flash memory is organized in 16 blocks, each holding 128 kB of data, see Table 4.2.

| Block number | Content | Address |
|---|---|---|
| Block 15 | | 0x201F FFFF<br>0x201E 0000 |
| . . . . . | | |
| Block 1 | Optional environment variables | 0x2003 FFFF<br>0x2002 0000 |
| Block 0 | Das U-Boot | 0x2001 FFFF<br>0x2000 0000 |

**Table 4.2 Flash memory sectors**

If the Flash memory is erased, all data bits are set to logic one. While programming the Flash memory, bits can only be set to zero. It is not possible to set one single bit from zero to one. Only a whole block can be erased (that means setting all bits to one).

The sector boundaries for read, write or erase commands will be checked by the U-boot itself. The functions concerned with the Flash memory hardware are placed in ./U-Boot_1.1.3/board/cm-bf533/flash.c. The used Flash memory is an MT28F320J3 from Micron Technology. Until now files from the EZ-KIT lite were used as templates. The memory used by the CM-BF533 is incompatible to the Flash memory used by the EZ-KIT lite, so a complete rewriting of the driver is necessary.

Some further information on the used Flash memory should be provided. The used Flash memory supports different operating modes. When powered up the Flash memory uses the READ ARRAY mode. This means that the Flash memory can be accessed (for reading) like a normal SDRAM. For a better understanding when reading the Flash memory driver source

code, a few Flash memory commands will be shown, see Table 4.3. For more detailed information have a look at the Flash memory datasheet [MIC05].

| Command | First cycle | Second cycle |
|---|---|---|
| word program | 0x40 | apply address and data |
| clear status register | 0x50 | |
| clear block lock bits | 0x60 | write 0xD0 to an address inside the block |
| read identifier code | 0x90 | read manufacturer and device code |
| block erase | 0xd0 | |
| read array | 0xff | |

**Table 4.3 Flash memory commands**

flash.c has to include the following functions:

**unsigned long flash_init(void):**

This function, it is called by ./U-Boot_1.1.3/lib_blackfin/board.c, has to check the Flash memory device ID and manufacturer ID. Combining these two numbers will allow an identification of the used chip. All supported manufacturer IDs are listed in ./U-Boot_1.1.3/include/flash.h. To enable the used Flash memory to work, this number has to be added to the mentioned list first. Furthermore in ./U-Boot_1.1.3/include/flash.h a structure holding data on the Flash memory is defined. This structure will be filled with data by flash_init(). After all initialisations are finished an overview on the found Flash memory will be given to the user. The return value of the function (flash_init) is the size of the Flash memory in MB.

**void flash_print_info(flash_info_t * info):**

When invoking the console command flinfo, the U-Boot prints a summary on the used Flash memory. flash_print_info will be invoked by ./U-Boot_1.1.3/common/cmd_flash.c. This summary includes the Flash memory type and an overview of the Flash memory sectors.

**int write_buff (flash_info_t *info, uchar *src, ulong addr, ulong cnt):**

This function will be called from ./U-Boot_1.1.3/common/cmd_flash.c. write_buff will be invoked for writing data to the Flash memory. Keep in mind that the Blackfin processor stores the data in little endian byte order! The parameters supplied to the functions are:

- flash_info_t *info: info points to information about the Flash memory, it will not be used for the CM-BF533
- uchar *src: Points to the data to write
- ulong addr: The start address to copy the data to

- ulong cnt: the number of bytes to write

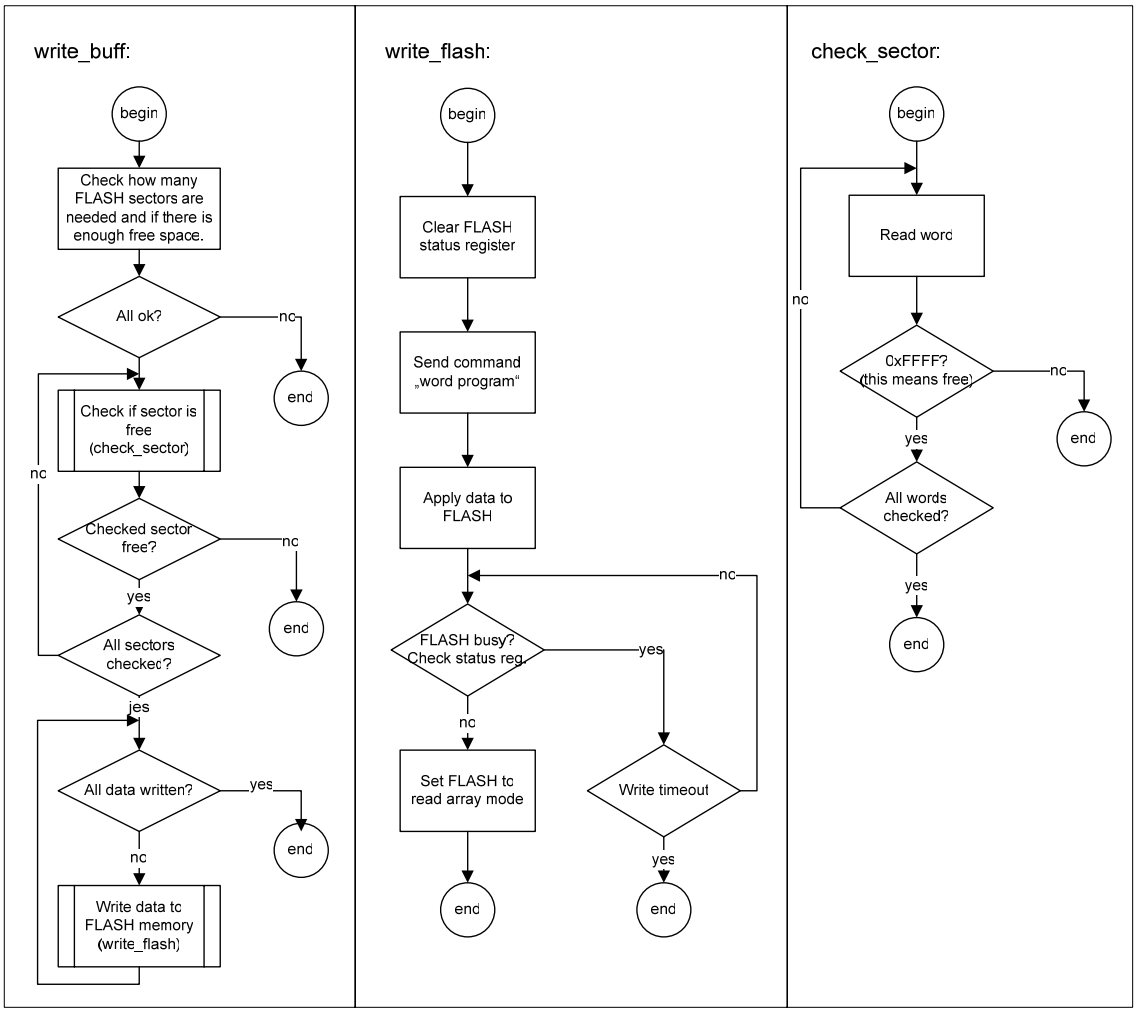


**Figure 4.7 Function write_buff flow chart**

Figure 4.7 illustrates what happens when data are copied to the Flash memory. First of all it is checked, if enough free space is available. Therefore the parameter addr and cnt are used. If enough free space is available, it has to be checked if the sectors are erased. This is done by reading out a specific sector and comparing the data with the data stored in an erased sector. Finally the data are written to the Flash. Therefore the internal state machine of the Flash memory is set to the programming mode. After applying the data and programming the Flash memory, it is set back to the read array mode again.

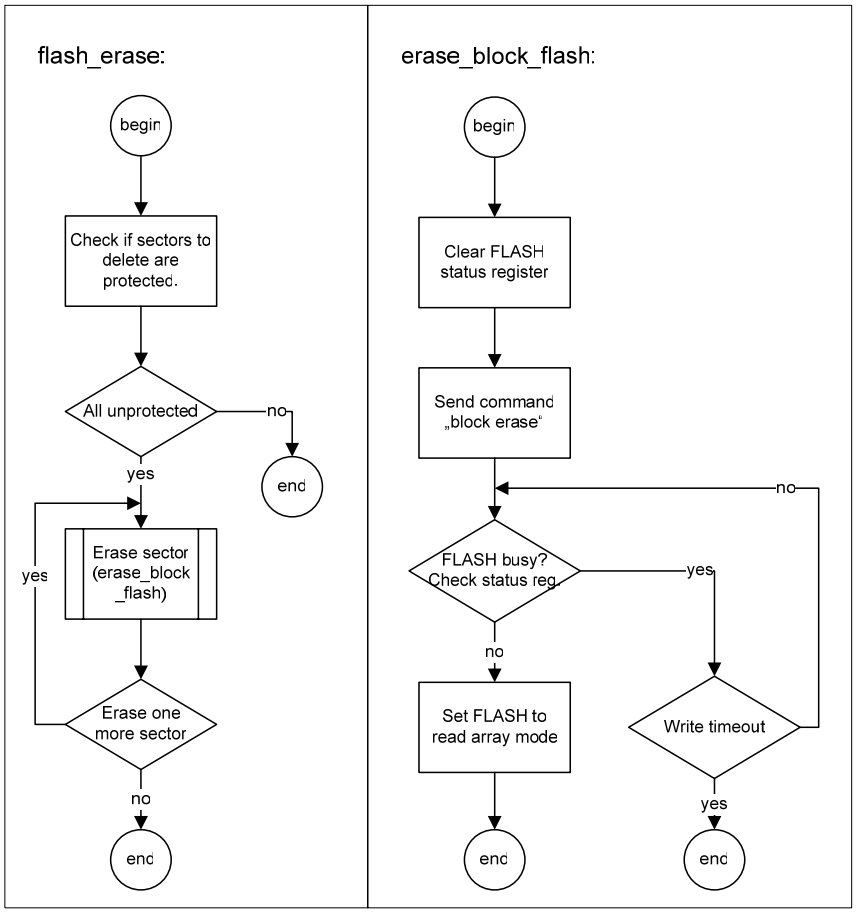


**Figure 4.8 Function flash_erase flow chart**

**int flash_erase(flash_info_t * info, int s_first, int s_last):**

As the name of the function already says, it is used to erase the Flash memory. The provided function to the function is:

- flash_info_t *info: info points to information about the Flash memory, it will be used to determine if the sector is protected
- int s_first, int s_last: Defines the first and the last sector for erasing.

Sectors can be marked as protected. Use the flinfo command to see which sectors are protected; they will be marked with (R0). This means that is not possible to erase them. Regardless that the used Flash memory chip will support sector protection by hardware, this feature will not be used by the CM-BF533. The first sector of the Flash memory will be protected automatically by the U-Boot. This avoids the unintended erase of the U-Boot.

Figure 4.8 shows what to do to erase a Flash sector. First there is a check, if all sectors are unprotected. In case of the CM-BF533 only the first sector will be

protected by the U-Boot itself. Even if the used Flash memory supports the locking of sectors, this feature is not supported from the Flash driver. As the parameters “first” and “last” define the sectors that will be erased the function erase_block_flash, responsible to erase one sector, is called many times. To erase a sector, the block erase command is applied to the internal state machine of the Flash memory.

## 4.3 Environment variables

Environment variables are used to configure the U-Boot. They can be preconfigured in the configuration file, as explained in subsection 4.2.3, or they can be set dynamically. When the environment variables were set in the configuration file, there is no need for an additional Flash memory sector because they are already compiled into the U-Boot. When environment variables are set at runtime, they have to be saved to the Flash memory in order to store them permanently.

The dynamically assigned environment variables are stored in the second Flash memory sector. All available environment variables are introduced in ./U-Boot_1.1.3/README. So in this section only the usage of them on the CM-BF533 will be described.

Figure 4.9 shows the U-Boot when it has booted up correctly.

```
CPU:   ADSP BF533 Rev.: 0.3
Board: Bluetechnix CM-BF533 board
       Support: http://www.bluetechnix.at/
Clock: VCO: 594 MHz, Core: 594 MHz, System: 118 MHz
SDRAM: 32 MB
Flash: id = 16, manufacturer_id = 89
Device ID of the Flash is 890016
Flash Memory Start 0x20000000
Memory Map for the Flash
0x20000000 - 0x201FFFFF Single Flash Chip (2MB)
Please type command flinfo for information on Sectors
FLASH:   2 MB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   SMC91111 at 0x20200300
Hit any key to stop autoboot:  0
CM-BF533> _
```

**Figure 4.9 The U-Boot**

Have a look at the line with the “*** Warning – bad CRC…” This means that no valid environment data was found in the second Flash memory sector. When no valid environment data are stored in the second Flash memory sector, the standard environment variables defined in the CM-BF533 configuration file will be used. The printenv command will show all currently used environment variables, as shown in Figure 4.10.

```
CM-BF533> printenv
bootargs=root=/dev/mtdblock0 ip=192.168.0.32:192.168.0.33:192.168.0.1:255.255.25
5.0:CMBF533:eth0:on
bootcmd=bootm 20020000
bootdelay=5
baudrate=115200
loads_echo=1
ethaddr=02:80:ad:20:31:b8
ipaddr=192.168.0.32
serverip=192.168.0.33
netmask=255.255.255.0
hostname=CM_BF533
stdin=serial
stdout=serial
stderr=serial

Environment size: 313/65532 bytes
```

**Figure 4.10 Environment variables used on the CM-BF533**

For setting, editing or deleting environment variables the setenv command can be used. Enter “help setenv” to get information on how to do. The meaning of nearly all environment variables shown in Figure 4.10 is self explaining or it has already been explained on the corresponding configuration tags in subsection 4.2.3. The only one, not yet explained variable is bootcmd. Because this command is essential for booting up automatically, a short introduction on it will be given. When the U-Boot starts up it waits for a defined number of seconds. The time to wait is defined with CONFIG_BOOTDELAY (in the configuration file) or with help of the environment variable. When the time to wait is expired, the command stored in bootcmd will be executed. As shown in Figure 4.10 the command bootm 20020000 will be invoked. bootm boots an application image out of memory.

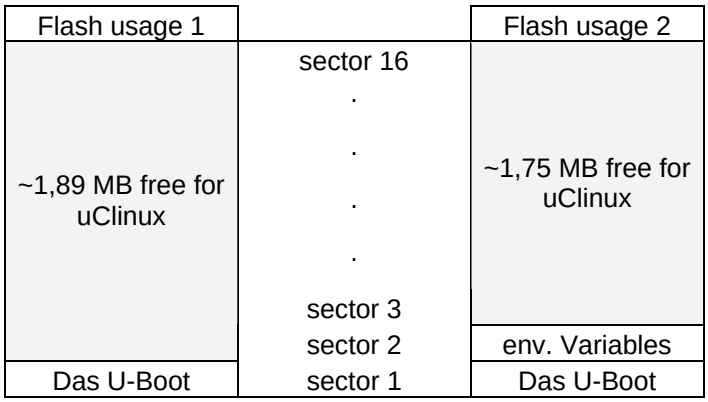


**Figure 4.11 Possibilities to use the Flash**

Figure 4.11 shows the possible Flash usage.

**Flash usage 1:**

This is the recommended solution because the maximum amount of free space will be reached (when using the U-Boot). All predefined environment settings will be used. This means that the U-Boot, after the autoboot timer has expired, tries to load an image stored on address 0x2002 0000.

**Flash usage 2:**

The user could change the environment variables, for example to use a different IP address, a user defined delay for autobooting and so on. If the changes shouldn't be lost after a reboot, they have to be stored into the first Flash memory sector. So the user has to spend one more Flash memory sector to the U-Boot. Furthermore the command used for auto booting has to be corrected. So the following has to be done on the console:

```
Set all needed variables, e.g. setenv ipaddr 192.168.002.003
setenv bootcmd bootm 20040000
saveenv
```

Until now the user defined environment variables should be used. If the defined variables are not needed anymore, or "Flash usage 1" should be used, only clearing of the second Flash memory sector will be needed. Don't forget, when the Flash memory sector 2 has been cleared the U-Boot tries to boot an image beginning at sector 2.

## 4.4 Building and flashing the U-Boot

Assuming that the toolchain has been built correctly and the PATH variable has been set there should be no problems to build the U-Boot. In order to build the U-Boot, the following commands have to be invoked:

```
make cm-bf533_config
make
```

Now there can be found a file named "u-boot.bin". This file can be directly used to update the U-Boot by itself, or it can be converted to the Intel Hex Format. This file can be used to flash the U-Boot using Visual DSP.

### 4.4.1 Updating the U-Boot

When transferring the binary file to the U-Boot via a serial port, a terminal program which supports the Kermit file transfer protocol is needed. The transfer was successfully tested with

the Microsoft Windows Hyper Terminal and Kermit. Figure 4.12 and Figure 4.13 shows how to update the U-Boot. The commands to enter are marked in grey.

When using Kermit as terminal program, after the loadb command disconnect by pressing Strg+\ c. Then send the file to the CM-BF533 using: “send u-boot_1.1.3/u-boot.bin”. After the file has been transferred to the CM-BF533 connect again. All other steps are shown in Figure 4.12.

When using the Microsoft Windows Hyper terminal for sending the file, the protocol has to be set to Kermit!

```
CM-BF533> loadb
## Ready for binary (kermit) download to 0x01000000 at 115200 bps...
## Total Size      = 0x000133d4 = 78804 Bytes
## Start Addr      = 0x01000000
CM-BF533> protect off all
Un-Protect Flash Bank # 1
CM-BF533> erase 1:0
Erase Flash Sectors 0-0 in Bank # 1
Erasing Flash locations, Please Wait
CM-BF533> cp.b 1000000 20000000 133d4
Copy to Flash... Bytes for programming: 78804
First sector: 0
Sectors needed:1
Checking sector 0 ... sector empty
[                    ]
[....................]
done
CM-BF533> cmp.b 1000000 20000000 133d4
Total of 78804 bytes were the same
CM-BF533> _
```

**Figure 4.12 Updating the U-Boot via a serial line**

```
CM-BF533> tftp 1000000 u-boot.bin
Using MAC Address 02:80:AD:20:31:B8
TFTP from server 192.168.0.33; our IP address is 192.168.0.32
Filename 'u-boot.bin'.
Load address: 0x1000000
Loading: ##################
done
Bytes transferred = 78804 (133d4 hex)
CM-BF533> protect off all
Un-Protect Flash Bank # 1
CM-BF533> erase 1:0
Erase Flash Sectors 0-0 in Bank # 1
Erasing Flash locations, Please Wait
CM-BF533> cp.b 1000000 20000000 133d4
Copy to Flash... Bytes for programming: 78804
First sector: 0
Sectors needed:1
Checking sector 0 ... sector empty
[                    ]
[....................]
done
CM-BF533> cmp.b 1000000 20000000 133d4
Total of 78804 bytes were the same
CM-BF533> _
```

**Figure 4.13 Updating the U-Boot via Ethernet and a serial line**

Figure 4.13 shows how to update when for the communication with the CM-BF533 the serial port, and for the data transfer Ethernet is used. This assumes that the development workstation is correctly configured, there is a TFTP server running. The file u-boot.bin has to be placed to the TFTP server source directory.

A short explanation to the commands used in Figure 4.12 and Figure 4.13 should be provided:

**loadb:**

The U-Boot waits for a serial transmission that has to be initiated by the terminal program.

**tftp:**

This command tries to load the file u-boot bin to the address 0x1000000. Therefore a TFTP server has to run on the development work stations.

**protect off all:**

As mentioned in subsection 4.2.5 the U-Boot will protect itself. So the first sector has to be unprotected before it can be deleted.

**erase 1:0 :**

This command will erase the sector number 0 on the first memory bank. To erase more than one sector, for example all erase 1:0-15 can be used to erase all sectors.

**cp.b:**

Data are copied byte by byte from address 0x1000000 to 0x20000000. Take care to set the right number of bytes to copy. The count of transferred bytes will be displayed after the tftp or loadb command.

**cmp.b:**

This command does a byte by byte compare of the provided addresses with the given length. Always use this command, so wrong written data can be detected before pressing the reset button.

### 4.4.2 Loading the U-Boot the first time

For loading the U-Boot for the first time a JTAG device has to be used to program the Flash memory. For the communication with this device, Visual DSP from Analog Devices is needed, this program is only available for Microsoft Windows. While porting the U-Boot to the CM-BF533, there was no Linux compatible JTAG available.

Visual DSP will not support the binary files created by the U-Boot make file. So they have to be converted to the Intel Hex Format. This can be done with the bfin-uclinux-objcopy command:

```
bfin-uclinux-objcopy -I binary -O ihex u-boot.bin u-boot.hex
```

The now created u-boot.hex can be used by Visual DSP to program the Flash memory. Figure 4.14 to Figure 4.16 provide a step by step manual how to flash the U-Boot to the Flash memory.

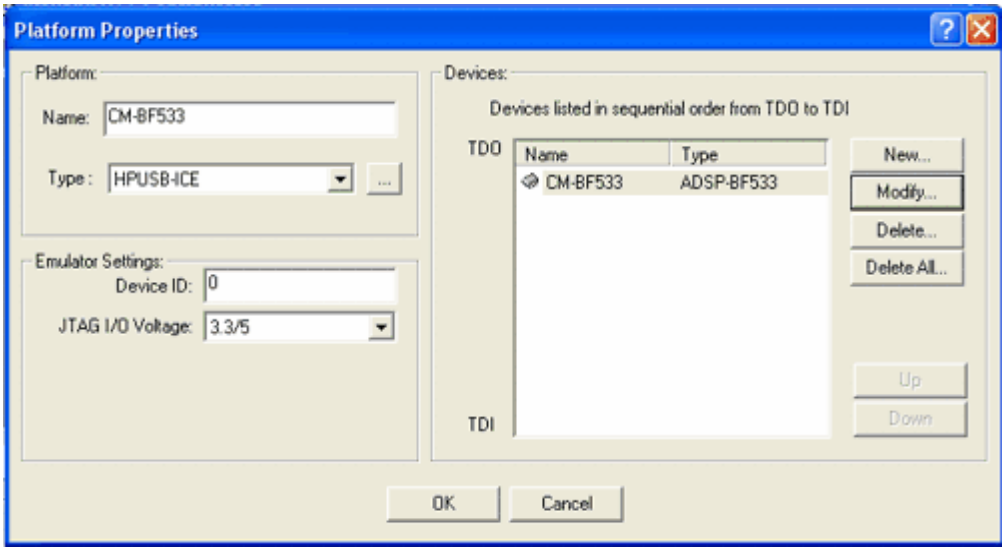


**Figure 4.14 Visual DSP configurator**

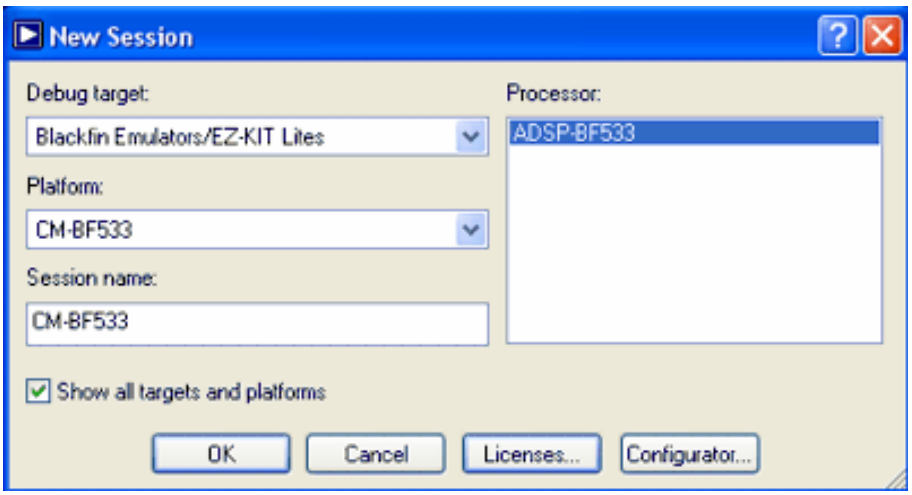


**Figure 4.15 Start new Visual DSP session**

After the first start of Visual DSP start the Configurator and create a new platform like Figure 4.14. Next start a new session, how to configure the new session can be seen in Figure 4.15. When the new session has been started, the Visual DSP Flash Programmer can be found in the tools menu. For the flash programmer a driver for the used Flash memory is needed, it can be obtained from Bluetechnix. Select the BF533EzFlash.dxe driver (provided by Bluetechnix) and click Load Driver. Then the file u-boot.hex can be selected as source file and copied to the Flash memory by clicking the Load File button. Now the U-Boot should work on the CM-BF533.

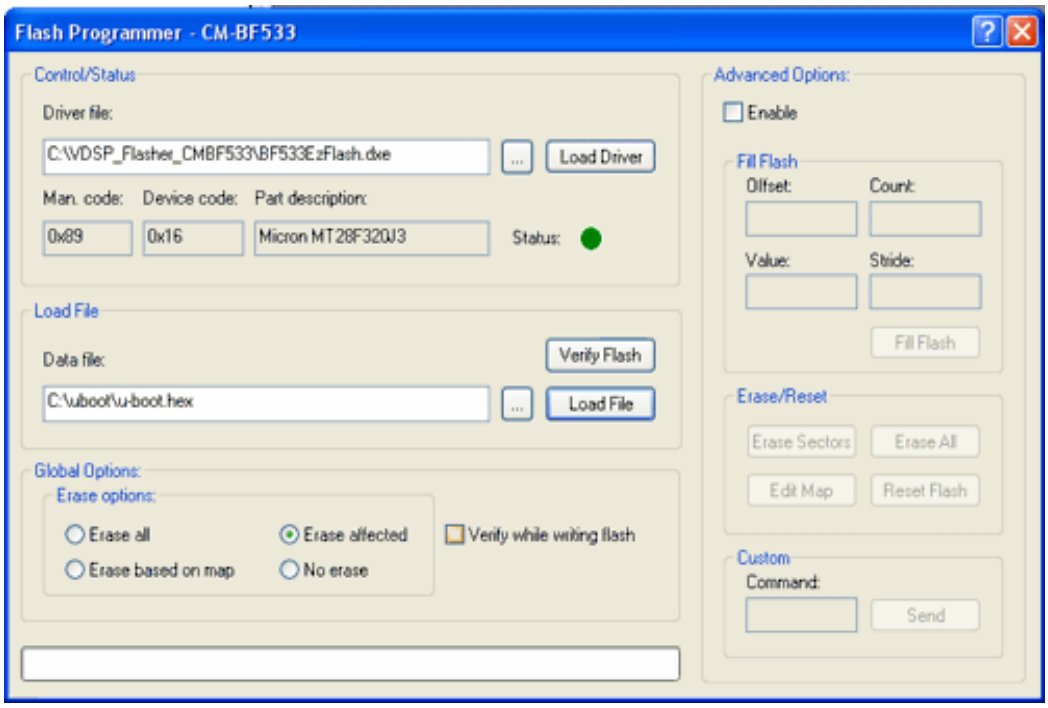


**Figure 4.16 The visual DSP flash programmer**

## 4.5 Troubleshooting

When the U-Boot doesn't start, this means there can be seen no output on the terminal program, what can be done? The U-Boot start up process is already known from subsection 4.2.1. As there is no readable output, the LEDs placed on the development hardware can be used to generate some status messages:

```
#define LEDS_INIT asm("[--SP]=P0;\
                       [--SP]=R0;\
                       P0.h = 0xFFC0;\
                       P0.l = 0x0730;\
                       R0.l=W[P0];\
                       bitset(R0,6);\
                       bitset(R0,7);\
                       w[P0]=R0.l;\
                       P0.h = 0xFFC0;\
                       P0.l = 0x0700;\
                       R0.l = W[P0];\
                       bitclr(R0,6);\
                       bitclr(R0,7);\
                       w[P0]=R0.l;\
                       R0=[SP++];\
                       P0=[SP++];");
```

```
#define LED_red_on asm("[--SP]=P0;\
                       [--SP]=R0;\
                       P0.h = 0xFFC0;\
                       P0.l = 0x0700;\
                       R0.l = W[P0];\
                       bitset(R0,6);\
                       w[P0]=R0.l;\
                       R0=[SP++];\
                       P0=[SP++];");

#define LED_red_off asm("[--SP]=P0;\
                       [--SP]=R0;\
                       P0.h = 0xFFC0;\
                       P0.l = 0x0700;\
                       R0.l = W[P0];\
                       bitclr(R0,6);\
                       w[P0]=R0.l;\
                       R0=[SP++];\
                       P0=[SP++];");
```

The shown labels LEDS_INIT, LED_red_on and LED_red_off can be used to initialize and to switch on or off the LEDs. These labels can be placed anywhere in the C source code, the used registers are stored on the stack and recovered afterwards. To use this functions in assembler source code, don't use the surrounding asm("…") and the '\'es.

LEDS_INIT configures port PF6 and PF7 as output. The other two defines can be used to switch the LED connected to PF6 on or off. To be able to control another LED simply change the number 6 in the bitset or bitclr statement to the right port.

## 4.6 Testing the U-Boot, simple hardware tests

Once the U-Boot is flashed it can be tested with some example applications located in /U-Boot_1.1.3/examples. These applications will be built automatically with the U-Boot. To rebuild these applications, rebuild the U-Boot.

The easiest test for the U-Boot is to print out a "hello world" message. All applications are built to run on address 0x1000. Assumed that the Ethernet is used for transferring data and all configurations are correct, the following lines will load and start the "hello world" application:

```
tftp 1000 hello_world.bin

go 1000
```

The go command jumps directly to address 0x1000 and executes the program stored on this position, even if there is no valid data stored. Also some hardware test can be performed with some simple test programs. To add a seperate test program, named test.c, add the following lines to examples/Makefile after the commands to build hello_world:

```
SREC      = hello_world.srec
BIN       = hello_world.bin hello_world

SREC      += led.srec
BIN       += led.bin led
```

Adding these lines will build led.c and create a run able led.bin. The following program can be used to switch on or off the LEDs placed on the CM-BF5xx Development board.

```
#include <common.h>
#include <exports.h>

int led (int argc, char *argv[])
{
  char x;

  /* recommended for stand alone applications*/
  app_startup(argv);
  if (get_version() != XF_VERSION)
  {
        return 1;
  }

  /*configure LED Ports */
  *pFIO_INEN  &= ~( PF6 | PF7);
  *pFIO_DIR   |=  ( PF6 | PF7);

  printf ("LED switch example, press one of the following keys:\n");
  printf ("1...switch on\n");
  printf ("2...switch off\n");
  printf ("3...exit\n");

  while(1){
        x=getc();
        switch (x)
              {
                    case '1':
                          *pFIO_FLAG_S = PF6;        //LED on PF6 on
                          *pFIO_FLAG_C = PF7;        //LED on PF7 off
                          break;
                    case '2':
                          *pFIO_FLAG_C = PF6;        //LED on PF6 off
                          *pFIO_FLAG_S = PF7;        //LED on PF7 on
                          break;
                    case '3':
                          printf("Good bye!\n");
                          return(0);
                          break;
                    default:
                          return (0);
                          break;
              }
  }
  return (0);
}
```

## 4.7 Working with images

All information needed to load images to the RAM and Flash memory is already provided in section 4.4.

Once the image is copied to the RAM it can be started without flashing it. To start uClinux images the following commands can be used:

**go 1000:**

This command has already been used in section 4.6. It will directly jump to the address 0x1000 and start executing. How to convert the image to binary, will follow.

**bootelf 1000000:**

The elf image has to be loaded to address 0x1000000. The command booelf 1000000 analyses the elf image, stored at address 0x1000000, and extracts the data sections stored in the elf image. Then it starts uClinux.

**bootm 1000000:**

The compressed image, stored at address 0x1000000, will be expanded and started.

Usually all images are loaded to address 0x1000000 before they are started or copied to the Flash memory. Depending on the address where the image is stored, the provided address has to be adapted. Binary files are usually copied to 0x1000, this depends on the settings at compile time.

An uncompressed uClinux image will be too big to fit in the Flash memory, so it has to be compressed. Compressing and decompressing an image are features provided by the U-Boot. The command to generate a compressed image is provided from the U-Boot. So, even if no development on the U-Boot is planned, it has to be built.

It is assumed, that no environment variables will be used, so the uClinux image will be placed on address 0x20020000. Furthermore loading the uClinux image via the serial port will take a long time. Using the DEV-BF5xx Development Kit, it is recommended to use the Ethernet for the image transfer.

To compress a uClinux image the following steps have to be done in the directory holding the uClinux image:

```
bfin-elf-objcopy -O binary linux linux.bin
gzip -9 linux.bin

u-boot_1.1.3/tools/mkimage -A Blackfin -O Linux -T kernel -C gzip -a 0x1000
-e 0x1000 -n "CM-BF533 uClinux Kernel" -d linux.bin.gz uImage
```

The first command will convert the uClinux image from the ELF file format into a binary file format. That file format can be compressed using gzip. The mkimage command (the fourth line is part of the third line) builds an image that can be expanded by the U-Boot. Finally, when no environment variables are used, this image has to be copied to address 0x20020000, this means it starts at the second Flash memory sector.

## 4.8 Problems

Unfortunately there were some unpleasant moments while developing the U-Boot.

- When porting the U-Boot 1.1.1 the first time, there was no CONFIG_RTC_BF533. This means, that is was not possible to disable the RTC only by not defining CONFIG_RTC_BF533. As on some CM-BF533 there was a problem with the RTC the U-Boot was not starting up. This was the first action to port the U-Boot, find and disable the RTC initialisation. Actually the RTC is disabled in the configuration file to avoid such problems.
- First the U-Boot version 1.1.1 was ported to the CM-BF533. After a change in the start up routines of uClinux it was necessary to port the U-Boot 1.1.3. So a lot of time was spent on the U-Boot 1.1.1 which is not needed anymore.
- It took a lot of time to find out why the U-Boot 1.1.3 was not able to boot when the first port was done, furthermore the memory test failed. As mentioned in subsection 4.2.3 a mistake in the configuration file was made. As it was not obvious, that an unusual value of an environment variable bared the U-Boot from booting up uClinux images, it was really hard to find that mistake.

# Chapter 5 uClinux

## 5.1 Basics

As already described in 3.2 the uClinux project was initially founded to use the Linux kernel on MMU less microcontrollers. The first port of Linux (to avoid ambiguities, the Kernel is meant) to a Palm PDA was based on the Linux 2.0 kernel series. This port was done by Kenneth Albanowski and D. Jeff Dionne. They released this port under the GPL in January 1998. Since the initial release the project has grown up and today uClinux is a synonym for a whole operating system running on MMU less microcontroller. [WUCL1]

In October 2003 Analog Devices released the first uClinux port, based on kernel 2.4.6, for the ADSB-BF535 processor. While writing this document, the kernel version 2.6.12 was used for the Blackfin uClinux project [WUCL2].

## 5.2 uClinux vs. Linux

### 5.2.1 The Memory Management Unit

Linux and uClinux use memory pages for managing the system memory. Most 32-bit architectures use 4 kB page tables. As mentioned in subsection 2.3.2 in Linux every process will get its own memory area. Each program can start at the same address; there will be no conflict between two programs. Figure 5.1 illustrates the mapping from virtual memory to physical memory. For this mapping, indicated by the arrows, Linux uses a so called page table. As shown in Figure 5.1 the memory pages of one process can be scattered over the whole system memory, or even swapped to a secondary storage medium. For doing this memory mapping the Memory Management Unit is needed. A more detailed view of the virtual memory management will be out of the scope of this document, a lot of further information can be found in [SYS02].

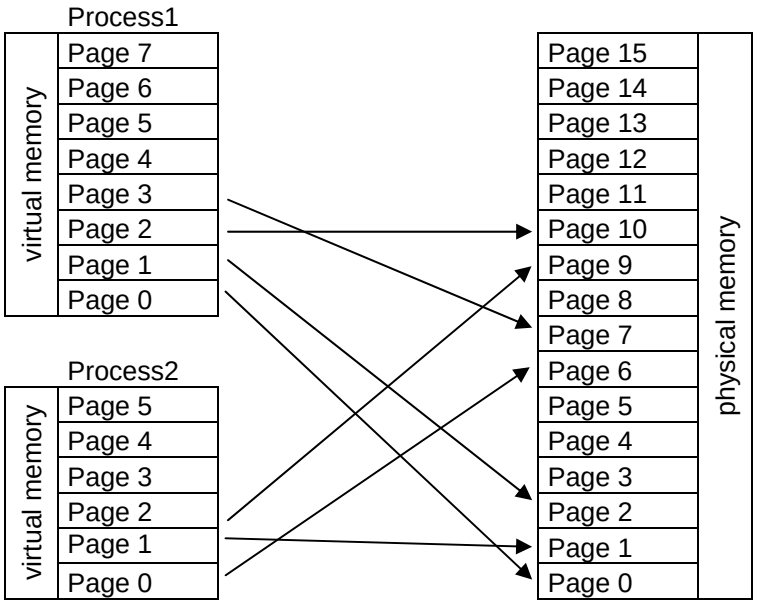


**Figure 5.1 virtual to physical memory mapping**

Regardless of that very high level view to virtual memory, it is obvious what happens when a processor offers no suitable MMU. No virtual memory will be available. Each process has to be loaded into the memory using a contiguous memory segment.

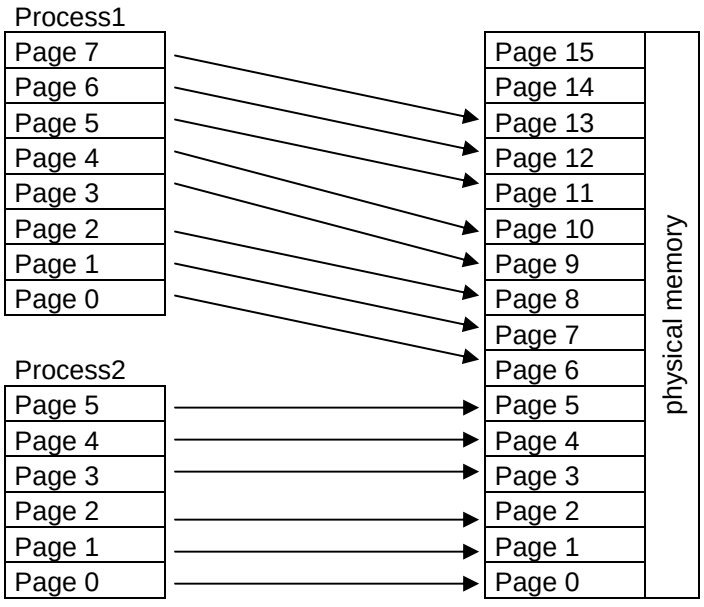


**Figure 5.2 flat memory model**

As shown in Figure 5.2, the start address of a program loaded into the memory has to be adapted to the physical memory addresses. This has to be done with all addresses used by the process. This is called relocating, therefore a special file format is used, see subsection 5.2.2.

The following disadvantages occurred when using a MMU less processor:

- Because no virtual memory is available, no memory pages can be swapped out (for example to a hard disc drive) to enhance the amount of available memory. Swapping needs virtual memory because it is unknown where the swapped out memory page will be restored. No swapping is not really a problem on embedded system, there is often no secondary storage media.
- Memory protection is a problem. Because each process uses real addresses, a misrouted memory access can kill the whole system.

  As shown in Figure 1.1 the Blackfin processor offers a MMU that supports memory protection. Hence, it is possible to separate the uClinux kernel from the user space. It is impossible to protect one user space program from another. Therefore virtual memory is needed (each process needs its own address range).
- Some parts of the memory management are different. For example an auto growing stack is not possible. Once a process has been loaded into the memory it cannot change its size because there is no virtual memory. Other processes can be placed directly before and after the actual process. Because the size of an existing process can not be changed, the implementation of malloc[1] in uClinux has to be different to Linux. Further information on how malloc is implemented on MMU less processors can be found in [WCYB1].
- The creation of a new process is different to processors that offer a full featured MMU. For the creation of a new process fork command is used. As explained in subsection 2.3.2 the fork command will use copy-on-write. Remember, when a new process is created, the new process shares all data with the parent process. But if new data is written back to the memory, the new process gets its own address space for writing. Without an MMU copy-on-write does not work. Hence uClinux uses vfork instead of fork. If a new process was created using vfork, the parent process is blocked until the child process calls exec or exits. The exec command is used to load a new program into the process. So a new process, which can do different things as the parent process, is created.

### 5.2.2 Flat binary format, ELF format and relocation

As shown in Figure 5.2 the physical start address for a program is unknown at compile time. On machines that offer a MMU, zero is used as start address, as described in Figure 5.2. When a flat memory model is used, due the lack of a suitable MMU, loading all processes to

[1] malloc is used for the dynamically allocation of memory.

the same address will not work. While loading into the memory, the executable has to be relocated to the actual start address.

Due to the lack of a suitable MMU uClinux doesn't support the standard format for executable files. Hence the flat binary format has to be used, it will be described later. As shown in Figure 4.3 there are some steps needed to get from the source code to the flat binary format. Before the flat binary format can be created by bfin-uclinux-elf2flt a file format called ELF is used.

Here the ELF format is only an intermediate step to the flat format. Hence no detailed view on this format is provided. As [LEV99] explains: "ELF files come in three slightly different flavors: relocatable, executable, and shared object. Relocatable files are created by compilers and assemblers but need to be processed by the linker before running. Executable files have all relocations done and all symbols resolved except perhaps shared library symbols to be resolved at runtime. Shared objects are shared libraries, containing both symbol information for the linker and directly runnable code for runtime." Now it should be obvious why in Figure 4.3 the output of the assembler and the linker can be stored in the ELF format.

As already mentioned uClinux uses a flat binary format for executables. This format includes a header, a text section, a data section and relocation information. For uClinux on Blackfin processors a slightly different flat binary format is used. Have a look to the file flat.h placed in the elf2flat directory that comes with the toolchain. There are two kinds of flat binary files. First absolute addresses are used. This allows adding the load address to the absolute addresses, so the physical address can be calculated. Second, only the data section is copied to the RAM, the program is executed directly out of the Flash memory, called execute in place (XIP). Therefore the Global Offset Table (GOT), placed on the beginning of the data section, is needed. The GOT holds the necessary data for the relocation.

### 5.2.3 glibc, uC-libc and uClibc

**glibc:**

As already mentioned the glibc, also known as GNU C library, defines basical functions like printf. Furthermore the system call interface is provided by the glibc.

**uC-libc:**

uC-lib was the initial replacement for the glibc in uClinux. uC-libc only supports ARM m68k processors.

**uClib:**

uClib should be a replacement for the glibc with a different kind of improvement. uClib is optimized to generate a very small sized code, regardless it offers nearly all features as provided by the glibc. In its beginnings, uClib supported only MMU less controllers. Today it can be used on standard Linux as well as uClinux.

For the Blackfin processor the uClib is used. First it supports MMU less processors and second, it is optimized to save memory.

## 5.3 GNU make, kernel build, kernel configuration

The GNU make command is used for building executable and other files from the source code. It automatically determines which files have been updated and rebuilds them. All information needed to build a project is stored in the Makefile. The simple “make” command followed by an optional parameter is sufficient to build a project.

For the kernel configuration a more powerful tool is needed. For building the kernel the kbuild utility is used.

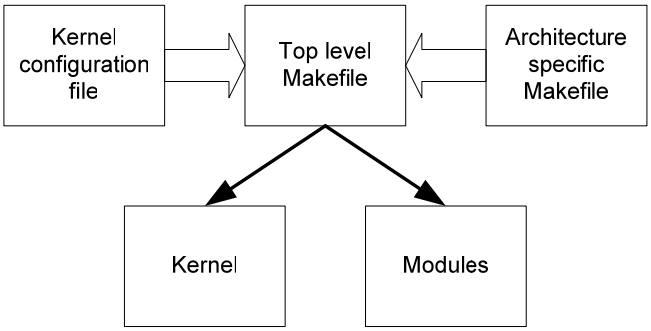


**Figure 5.3 kbuild**

The kbuild is responsible to build the kernel and the kernel modules out of the source code. As shown in Figure 5.3 the top level Makefile gets information on what to out of configuration files. These files are generated by the kernel configuration. Second, the top level makefile includes an architecture specific makefile.

The executable generated with help of the top level Makefile is generated by recursively walking through all subdirectories, depending on the kernel configuration file. Every subdirectory must have its own kbuild makefile. Adding a new program to compile is very simple.

**obj-y += cm_bf533.o**

This line means, that the object cm_bf533.o is built out of cm_bf533.c or cm_bf533.S.

**obj-$(CONFIG_CM_BF533) += cm_bf533.o**

This means that cm_bf533.o is built in as shown above, or built as module, depending on the kernel configuration file. If “obj-m += cm_bf533.o” is set, this means to build cm_bf533.o as module. Furthermore it is possible not to build the object when CONFIG_CM_BF533 is not y or m.

**obj-$(CONFIG_CM_BF533) += cm_bf533/**

Also here the CONFIG_CM_BF533 can be y or m, like above. Especially this line adds the cm_bf533 as new subdirectory. Furthermore a kbuild makefile has to be placed in this directory.

Have a look at the directory Documentation/kbuild/ inside the kernel directory for more details.

Above there is description on how to decide if a part of source code is built or not. To do these configurations a configuration interface is provided. Calling “make menuconfig” or “make xconfig” starts a more or less graphical configuration environment. For setting up this configuration environment Kconfig files are needed. They include all possible settings that can be done. A good documentation on the kconfig-language can be found in ./uClinux-dist/linux-2.6.x./Documentation/kbuild/ inside the kernel directory. How to add options, especially for the CM-BF533, to the configuration utility will be seen later.

## 5.4 The uClinux boot process

The start up process of uClinux is very complex. Hence, only a few files or functions will be mentioned. The selection of the files to present is based on the gained experience while porting uClinux to the CM-BF533. These files can be used for placing some led blinking code to see where the boot process stops, see section 4.5.

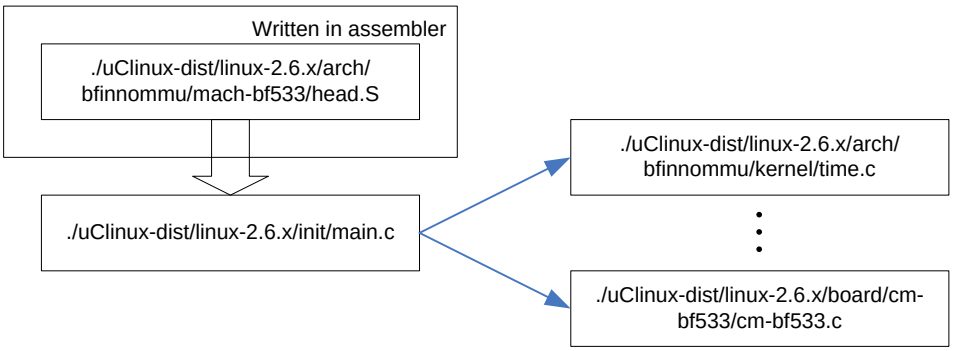


**Figure 5.4 Files included in the uClinux boot process**

As shown in Figure 5.4 the first file called is head.S. This file, written in assembler, does some basic initializations, afterwards the function start_kernel is called. The function start_kernel can be found in main.c, already written in C language. The architecture independent function start_kernel tries to bring up the system. Therefore a lot of other functions are called. One of the first called functions is setup_arch. As the name already says, this function does a lot of architecture specific initializations. Furthermore some early messages like “Blackfin support (C) 2004 Analog Devices, Inc.” are printed on the console. ATTENTION: This messages will be buffered, this means that they ae not printed

immediately. Before something can be printed out, the console has to be initialized, this happens later in main.c. Regardless, it is impossible to send status messages to the console at that moment, for example LEDs can be switched on or off.

Some functions after setup_arch, time_init will be called. As already mentioned, there were some problems with the RTC. That problem also affected the boot up process from uClinux. Hence, for a correct boot up of uClinux the initialization of the RTC has to be bypassed. When the boot up process stops on the RTC initialization this is a very bad issue. At this time a lot of initializations were correctly done, they also printed some status messages. But the console is initialized later than the RTC, so there is no output on the console. Only when dealing with LEDs, the boot up progress can be monitored in this case.

Once the kernel is up and running, the kernel tries to mount the root file system. Even if it is possible for the U-Boot to supply kernel boot parameters to the kernel, as already mentioned in subsection 4.2.3, normally compiled in kernel boot parameters are used to tell the kernel where to find the RFS. Once the kernel configuration is started, the option “Compiled-in Kernel Boot Parameter” can be found in the subsection Kernel hacking. The following parameter has to be defined there:”root=/dev/mtdblock0 rw” This option tells the kernel where to find the root file system. Moreover this kernel option indicates that a memory technology device is used to hold the root file system. Some further options have to be set, so that the kernel can mount the root file system when using MTDs. These options can be determined by having a look at the standard kernel settings for the CM-BF533.

Above there was described how the board start up process works. When the kernel is up and running, there are some further initializations that can be done automatically in the user space. Therefore some configuration files are placed in ./uClinux-dist/vendors/Bluetechnix/CM-BF533/. These files are included in the root file system. One essential file is the file rc (nun command). This file can be used for an automatical start up of user space programs. All programs that should be started have to be placed there. For example the configuration of the Ethernet address, by calling the ifconfig command, is done there.

There are also some other configuration files. They will be used like the corresponding files on a standard Linux workstation. Furthermore some of them are used for some user program. So the Linux manpages or the help of the specific program should be used to determine their meaning.

## 5.5 Configuring uClinux, building the image

Assumed the platform to use is already included in uClinux, the uClinux image should be easy to configure and build.

First of all, a whole uClinux image consists of two parts, the kernel image and the root file system. These two parts are built independently and concatenated to the uClinux image. So both parts have to be configured before building it. Therefore a graphical user interface can be used. Executing the following command inside the uClinux-dist directory starts the configuration.

| |
|---|
| make xconfig (for the graphical configuration) |
| or if it doesn't work |
| make menuconfig |

If the graphical configuration works the window on the top of Figure 5.5 should appear.

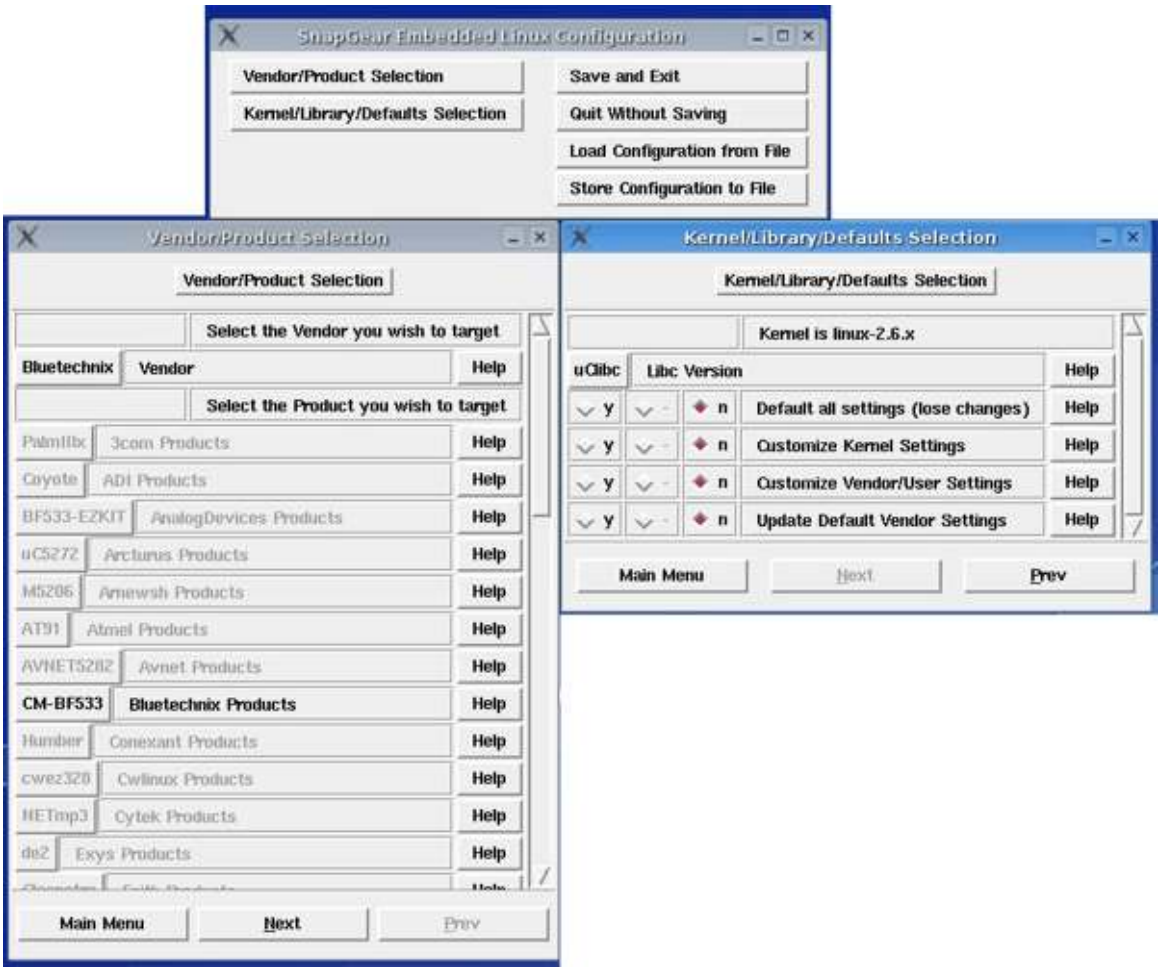


**Figure 5.5 The uClinux configuration interface**

To open the two windows shown at the bottom of Figure 5.5 "Vendor/Product Selection" or the "Kernel/Library/Default Selection" have to be selected. When the configuration is done, the button "Save and Exit" has to be clicked.

### Vendor/Product Selection

To support the CM-BF533 select Bluetechnix as Vendor. Then it should be possible to select the CM-BF533.

### Kernel/Library/Default Selection

First of all, there is the possibility to recall the standard settings stored in ./vendors/Bluetechnix/CM-BF533/ with setting “Default all settings”. In the other direction it is possible to update the settings stored in ./vendors/Bluetechnix/CM-BF533/ with the actually used settings when selecting “Update Default Vendor Settings”. If one of these options is used, close the window and click the “Save and Exit” in the main window to activate the selected option.

The two other options can be selected for further configuration of the kernel or the programs included in the user space. Set one or both options to yes. If the “Customize Kernel Settings” option was selected, a window like Figure 5.6 should pop up.

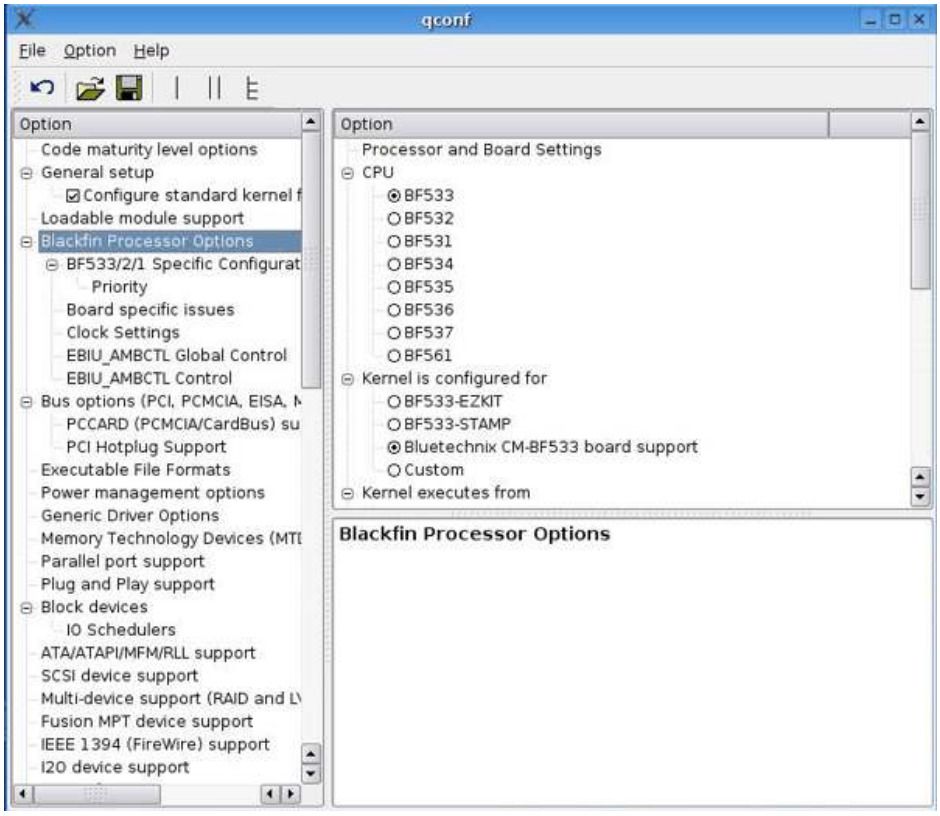


**Figure 5.6 Kernel configuration**

There all necessary kernel configurations, like the core clock frequency or the memory size, can be done. First of all the CM-BF533 have to be selected in the Blackfin Processor Options.

All further settings should be set according to the standard values for the CM-BF533 when starting the configuration the first time. Select File->Quit to end the configuration.

A similar window appears, when the user space configuration is done. There can be found all programs that can be included into the image.

Once uClinux is configured, a simple “make” inside the uClinux-dist directory will build the kernel and the root file system. The final elf image is placed in ./uClinux-dist/images/ and will be called linux. In section 4.7 it was already described how to start or compress this image. The file linux includes the kernel and the root file system. How the root file system is created and concatenated to the kernel image can be found in the board specific Makefile located in ./uClinux-dist/vendors/Bluetechnix/CM-BF533/.

## 5.6 Adding new platforms

### 5.6.1 Platform dependent files

Nearly all platform specific files are stored in ./uClinux-dist/vendors/Bluetechnix/CM-BF533. There are placed all standard configuration files and all platform specific files that should be included in the root file system. These files are:

**Some files with the prefix “config.”:**

These files include the standard settings for the board. The kernel specific files should not be edited manually. As described in section 5.5 theses files can be updated with the option “Update Default Vendor Settings”. Furthermore it is possible that the file config.uClibc has to be updated when a new uClibc is used. If the file config.uClibc was dated out it is possible that the build process fails.

**device_table.txt:**

This is a very important file. Based on this file the device entries in /dev are created. If a new device driver has been written and a device node for the new device should be created, the name for the device node and the major and minor number of the driver have to be added to this file. Furthermore file permissions can be set using this file.

**Makefile:**

The architecture specific Makefile has two major topics. First, as explained above the directory ./uClinux-dist/vendors/Bluetechnix/CM-BF533 holds some files that should be included into the root file system. This is done by the Makefile. Second, the Makefile is responsible for building the root file system and to concatenate it with the kernel image.

**Some uClinux specific files and others:**

As already explained in section 5.5 some configuration files for uClinux itself and other programs are placed in ./uClinux-dist/vendors/Bluetechnix/CM-BF533. The file rc was already explained in section 5.4, an example for a program specific configuration file is the file boa.conf. This file is the configuration file for the web server running on uClinux. As mentioned before, there can be placed also some other files in ./uClinux-dist/vendors/Bluetechnix/CM-BF533. For example the web pages for the web server can be stored in this directory. The Makefile is used to include these web pages into the root file system.

Furthermore one board depending file is placed in ./uClinux-dist/linux-2.6.x/arch /bfinnommu/mach-bf533/boards. In case of the CM-BF533 the file is called cm_bf533.c. It is essential to make an entry to the Makefile located within this directory. A proper entry will be:

```
obj-$(CONFIG_BFIN533_BLUETECHNIX_CM) += cm_bf533.o
```

The meaning of this entry is already described in section 5.3. Depending on the CONFIG_XXX entry the right file for actual hardware will be used. When a different filename is used, the CONFIG_XXX entry has to be adapted.

### 5.6.2 Adding a new board

Adding new platforms is relatively simple. Therefore only a new directory has to be placed in the ./uClinux-dist/vendors/ directory. The configuration utility will automatically detect the new vendor. The directory placed in ./uClinux-dist/vendors/ is used for the vendor name. This directory has to include, at least, one further subdirectory. These subdirectories stand for the different platforms supplied by one vendor.

It is recommended to copy the directory of a similar platform and adapt the files to the new board needs. Some of them are already explained. When adding files to this directory, they will not be automatically added to the file system for the CM-BF533. Therefore the Makefile has to be edited, to add an additional directory, add the directory to the ROMFS_DIRS = … list. To add an additional file insert an additional $(ROMFSINST)… entry. To see which options are supported when adding a file, call ./uClinux-dist/tools/romfs-inst.sh without options. As there are already a lot of such entries, which can be found in the Makefile, it should be obvious what to do. As explained there is no need to edit the configuration files. This can be done with help of the kernel configuration utility. Furthermore the file device_table.txt has to be edited for the CM-BF533. Some essential changes for this file will be provided in subsection 5.7. Finally all program specific files or for example the web pages have to be adapted.

A further architecture specific file can be found in ./uClinux-dist/linux-2.6.x/arch/bfinnommu/mach-bf533/boards. In case of the CM-BF533 it is used to configure the address and the IRQ used by the network chip. Finally it adds the driver to the system. This file has to be adapted for the CM-BF533.

### 5.6.3 Doing some kernel configurations

Some platform dependent configuration options or standard settings can be added to the kernel configuration. The kernel configuration utility was explained in section 5.5. As already mentioned in section 5.3 a special language is used for the configuration utility. A few samples should be provided now.

The Blackfin ADSP-BF533 specific Kconfig files (these files configure the appearance of the kernel configuration) are placed in ./uClinux-dist/linux-2.6.x/arch/bfinnommu and ./uClinux-dist/linux-2.6.x/arch/bfinnommu/mach-bf533. To add the CM-BF533, add the following lines to /uClinux-dist/linux-2.6.x/arch/bfinnommu/Kconfig:

```
choice
      prompt       "Kernel is configured for"
      default BFIN533_STAMP
      help
        Do NOT change the board here, please use the top level configuration to ensure all
        all the other settings are correct

[cut; other boards are placed here]

config BFIN533_BLUETECHNIX_CM
      bool "Bluetechnix CM-BF533 board support"
      depends on BF533
      help
        CM-BF533 support for EVAL- and DEV-Board

[cut; other boards are placed here]

endchoice
```

The gray marked lines have to be added to be able to select the “Bluetechnix CM-BF533 board support”. If the entry for the CM-BF533 is selected, CONFIG_BLUETECHNIX_CM will be set. The prefix CONFIG_ is added by the kernel configuration utility. A short explanation of the used keywords should be given:

**choice, endchoice:** When config entries (only if their type is bool, that means that it is selected or not) are enclosed by choice and endchoice, only one of them can be selected.

**prompt:** Will print a message.

**config:** This keyword will start a new config entry

**bool:** Defines the type of the configuration option.

**depends on:** The availability of a configuration option can be based on other options.

**help:** A help text for the configuration entry can be displayed.

When adding a new platform, there can be some options needed that will not be displayed by the kernel configuration utility, for example:

```
config MEM_MT48LC16M16A2TG_75
    bool
    depends on (BFIN533_EZKIT || BFIN533_BLUETECHNIX_CM)
    default y
```

CONFIG_MEM_MT48LC16M16A2TG_75 will be set automatically when BFIN533_EZKIT or BFIN533_BLUETECHNIX_CM was selected. This entry defines the used RAM, depending on this setting the time settings for the RAM will be determined. The settings for the specified RAM can be found in ./uClinux-dist/linux-2.6.x/include/asm-bfinnommu/mach-bf533/mem_init.h . When the adoption of this entry has been forgotten, the build process will stop with a linker error because some essential information about the RAM timings is missing.

Furthermore a value can be used for input, for example:

```
config CLKIN_HZ
    int "Crystal Frequency in Hz"
    default "11059200" if BFIN533_STAMP
    default "27000000" if BFIN533_EZKIT
    default "25000000" if BFIN537_STAMP
    default "27000000" if BFIN533_BLUETECHNIX_CM
    help
      The frequency of CLKIN crystal oscillator on the board
in Hz.
```

This will allow the input of a value. Different default settings for different boards can be used. The last keyword that should be introduced here will be used to generate a new submenu:

```
menu            'Board specific issues'

[cut; other options are placed here]


config BFIN_ALIVE_LED
    bool "Enable Board Alive"
    depends on ( BFIN533_STAMP || BFIN533_BLUETECHNIX_CM )
    default n
    help
      Blinks the LED you select when the kernel is running. Helps detect a hung kernel.

[cut; other options are placed here]


endmenu
```

This code will place an entry which will open a new submenu when selected. As shown, between menu and endmenu standard configuration options can be placed. More than the provided basical introduction would be out of the scope of this document. For further information on the kbuild system have a look to Documentation/kbuild/.

### 5.6.4 Testing the new kernel configurations

In subsection 5.6.3 new options were added to the kernel config. Assuming that a new option was added using “config MY_OPTION” to some Kconfig file. Furthermore CONFIG_MY_OPTION is not used in the whole kernel source code, remember the prefix CONFIG_ is added automatically. Regardless if this option is set or not, there are no effects. Some source code which depends on this option has to be added to the kernel source code. A good example is the initialisation of the LEDs placed on the development hardware. This source code was taken out of ./uClinux-dist/linux-2.6.x/arch/bfinnommu/kernel/time.c:

```
#if defined(CONFIG_BFIN_ALIVE_LED)
inline static void do_leds(void)
{
	static unsigned int count = 50;
	static int	flag = 0;
	unsigned short tmp = 0;

	if( --count==0 ) {
			count = 50;
			flag = ~flag;
	}
	tmp = *(volatile unsigned short *)CONFIG_BFIN_ALIVE_LED_PORT;
	__builtin_bfin_ssync();

	if( flag )
			tmp &=~CONFIG_BFIN_ALIVE_LED_PIN;		/* light on */
	else
			tmp |=CONFIG_BFIN_ALIVE_LED_PIN;	/* light off */

	*(volatile unsigned short *)CONFIG_BFIN_ALIVE_LED_PORT = tmp;
	__builtin_bfin_ssync();

}
#else
inline static void do_leds(void) {}
#endif
```

Whether the alive LED (indicates a running kernel by blinking) is used or not, the function shown above will be compiled in or not. If a function should be compiled in or not depending on the kernel configuration, the line before the last should be used. Now, independently if CONFIG_BFIN_ALIFE_LED is set or not, the function do_leds(void) will be available. If this is done in that way, some code can always call do_leds().

The provided source code is responsible to toggle a specified LED. The pin where the LED is connected is also defined by a Kconfig file. So this short source code sample highly depends on the configuration done by the kernel configuration utility. A lot of similar examples can be found in the kernel source code.

It is recommended not to use platform dependent code inside a platform independent source code. Initially the code that does something with the LEDs was specified for the Analog Devices STAMP board. Hence, while uClinux was ported to the CM-BF533 the first time, a lot of source code was added only to toggle the LEDs, because there was no platform independent code. A lot of elif statements were used to select the right code.

## 5.7 Adding user programs and the file device_table.txt

User programs are placed in the user space. They are placed in the root file system structure when building the image. All user programs can be found in ./uClinux-dist/user/. This subsection describes how to get access to the board LEDs out of the user space.

First of all, the kernel has to be configured to get access to the port pins of the ADSP-BF533. As already mentioned above, device entries are created with help of the file device_table.txt. Regardless if the driver specified within the file device_table.txt (due to its major and minor number) is available, the device entry is created in the /dev directory. The necessary driver to get access to the Blackfin port pins can be found in kernel configuration in the subsection “Character devices” and is called “Blackfin BF53x Programable Flags Driver”. As the development platform is the DEV-BF5xx Development Kit the available ports are:

- PF4 is connected to the button.
- PF6 is connected to the LED1.
- PF7 is connected to the LED2.

To get access to these pins, the file device_table.txt has to be edited in the following way:

```
/dev/pf4  c    664  0  0  253  4  0  0  -
/dev/pf6  c    664  0  0  253  6  0  0  -
/dev/pf7  c    664  0  0  253  7  0  0  -
```

This means, for example, the device node /dev/pf4 is created with the following settings. The file permission is 664 and the driver used for this device node has the major number 253 and the minor number 4. The port used by this driver is determined by the minor number of the driver. Hence the port PF4 should be interfaced, the minor number 4 has to be used.

Now the ports PF4, PF6 and PF7 should be accessable through the corresponding device nodes.

To create a new user program, a new directory in ./uClinux-dist/users has to be created. For this example the directory led is created. Then, first of all a Makefile for the new program is needed. This Makefile could be very simple, like the following:

```
EXEC = led
OBJS = led.o

all: $(EXEC)

$(EXEC): $(OBJS)
  $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)

romfs:
  $(ROMFSINST) /bin/$(EXEC)

clean:
  -rm -f $(EXEC) *.elf *.o
```

This Makefile builds the program led based on led.c and adds it to the root file system. Furthermore the program led.c is needed:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/ioctl.h>

#define SET_FIO_DIR                    1  // Peripheral Flag Direction Register
#define SET_FIO_INEN                   5  // Flag Input Enable Register

int pf4,pf6,pf7;

void close_pf(int sig)
{
  if (close(pf4) != 0) printf ("Can not close PF4\n");
  if (close(pf6) != 0) printf ("Can not close PF6\n");
  if (close(pf7) != 0) printf ("Can not close PF7\n");
  exit(0);
}

int main(int argc, char *argv[])
{
  unsigned char buf[2];

  (void) signal (SIGQUIT, close_pf);

  if ((pf4=open ("/dev/pf4",O_RDWR)) == NULL) printf ("Can not open PF4\n");
  if ((pf6=open ("/dev/pf6",O_RDWR)) == NULL) printf ("Can not open PF6\n");
  if ((pf7=open ("/dev/pf7",O_RDWR)) == NULL) printf ("Can not open PF7\n");

  if ((ioctl(pf4, SET_FIO_INEN, 1 )) < 0) printf ("Can not configure PF4 as input\n");
  if ((ioctl(pf6, SET_FIO_DIR, 1 )) < 0) printf ("Can not configure PF6 as output\n");
  if ((ioctl(pf7, SET_FIO_DIR, 1 )) < 0) printf ("Can not configure PF7 as output\n");
```

```
  printf ("LED switch example, press the switch on the development board to toggle the
LEDs.\n");
  printf ("Press Strg+C to exit\n");

  while(1){
        if ((read(pf4, buf, 2)) == -1) printf ("Error reading PF4: %s \n",strerror(errno));
        if ((write(pf6, buf, 2)) == -1) printf ("Error writing PF6: %s \n",strerror(errno));
        buf[0]= (buf[0]=='0' ? '1' : '0');
        if ((write(pf7, buf, 2)) == -1) printf ("Error writing PF7: %s \n",strerror(errno));
  };
  exit(
```

This program does the following:

1. To get access to the pins PF4, PF6 and PF6 the open command is used.
2. With the ioctl command the pins are configured as input or output. This configuration is a standard function of a character device driver. With help of the ioctl command, extended functions can be added to a driver. The error message if something goes wrong can be used to determine what the single ioctl commands do.
3. An infinite loop will set the state of the two LEDs depending on the state of the button. The driver returns the character ‘0’ odr ‘1’ depending on the state of the button. The LEDs can be set in the same way. The loop can be cancelled when pressing Strg+c. Therefore a standard Linux signal handling routine is used.

Now the user program is able to work. The last step is to fit it into the userland configuration utility. Before that can be done, the directory ./uClinux-dist/user/led has to be added to the Makefile placed in /uClinux-dist/user/. Therefore the following lines have to be added to ./uClinux-dist/user/Makefile:

```
dir_$(CONFIG_USER_LED)                  += led
```

Depending on the settings for the userland the directory ./uClinux-dist/user/led is included when building the user applications or not.

Last an option for the program led has to be added to the userland configuration. All possible options are placed in the file ./uClinux-dist/config/config.in. Adding the following lines to this file creates a new Bluetechnix submenu. In this submenu an option for the led program can be found. If this option is activated, the constant CONFIG_USER_LED is set.

```
mainmenu_option next_comment
comment 'Bluetechnix demo applications'

bool 'LED'                                      CONFIG_USER_LED

endmenu
```

Now, as described in the section 5.5, the user should be able to select in the user space configuration if the program led should be compiled in or not. If everything works, the

program should be placed in the directory /bin and can be started by executing the program led. Figure 5.7 shows how it should work.

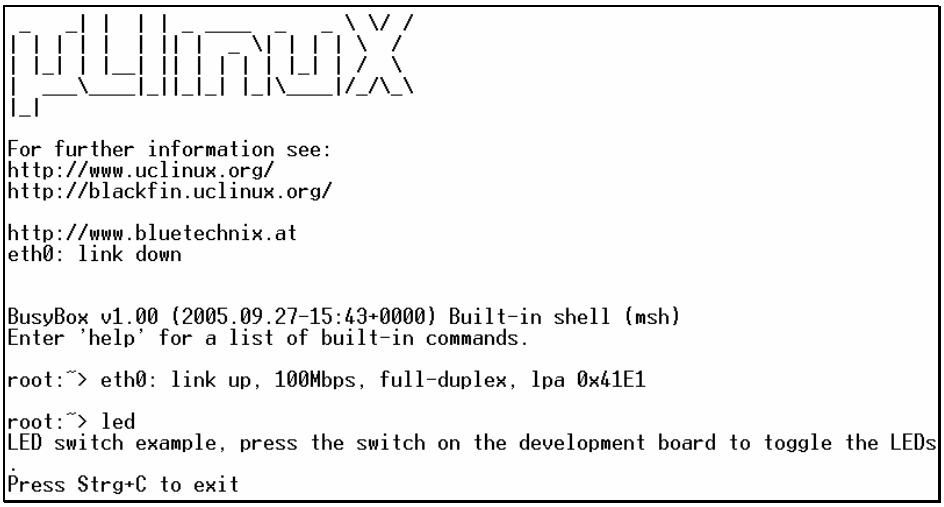


**Figure 5.7 LED example program**

## 5.8 The diff and patch command

To distribute the changes done on the source code, the diff and patch commands are used. These two commands only work with text base source code. Except the creation of the board specific directory all other changes described in this document can be distributed with diff and path. The whole board specific directory can not be distributed with these commands because there are also pictures, like bluetechnix.gif, inside the board directory.

To create a patch a copy of the original source code is needed. Then one of the two source codes has to be edited. When all work is done, the following command has to be used to create a patch.

```
diff -urN ./original_source ./edited_source > name_of_the_patch
```

Once a patch was created only the source code and the patch is needed to patch the source code. This can be done with the following command:

```
patch -p1 < name_of_the_patch
```

Depending on how the patch was created the p flag can be used to stripe the leading directories. So the patch can be adapted to the directory structure to patch.

## 5.9 Problems

When starting the work to port uClinux to the CM-BF533 in spring 2005, there were some problems with the assembler files that are responsible for the first initialization of the processor. In this case it was not obvious if there is a problem with the hardware, the U-Boot or the adapted uClinux. These problems were often solved by doing an update of the source code. It is very interesting to work on a project with a lot of other individuals. But it can be very unpleasant not to know where to look for a mistake.

Similar to the U-Boot a lot of time was used to find out if the RTC is needed for the uClinux and afterwards to disable the not working RTC. Nowadays an option to the kernel configuration is added to disable the RTC.

As there are a lot of undocumented parts of source code in the Blackfin uClinux project a lot of time is needed to find some information. Mostly there are two possible ways: First an intensive study of the source code or second to look for the needed information in the forums placed on the project homepage. A lot of questions are answered there.

# Chapter 6 Modules, Sample Device Drivers and Interrupts

This chapter should provide an easy to understand overview on how to get a working device driver. Therefore first a new kernel module is introduced to the kernel. To this module further functionality will be added step by step.

A sample device driver to control one pin of the processor is used to explain how to write a driver. As there is already a driver for the GPIOs of the processor, the major topic of the following driver is to provide a framework on how to build up a device driver from scratch. There are a lot of different device drivers included in uClinux, so a basical knowledge on device driver is needed when dealing with hardware.

For all examples the focus is set to show the principals. There is, for example, no error management implemented to provide an easy to understand source code.

## 6.1 Modules

Modules have already been explained in subsection 2.3.5. Now it will be explained how to integrate a new module to the kernel source tree. Furthermore the necessary changes the kbuild files will be done.

The module introduced in this section will only be able to print out a “Hello World” message. For the new module the directory ./uClinux-dist/linux-2.6.x/drivers/Bluetechnix has to be created. Now as already explained, the new directory must be added to the Makefile based one directory above, the new directory has to include a Makefile and the new module has to be added to the kernel configuration.

Adding the new directory to the Makefile placed in ./uClinux-dist/linux-2.6.x/drivers can be easily done by adding:

```
obj-$(CONFIG_LED)                += Bluetechnix/
```

The constant CONFIG_LED has to be added to the kernel configuration. Now if CONFIG_LED is set the kbuild system is looking for a Makefile in ./uClinux-dist/linux-2.6.x/drivers/Bluetechnix. Depending on CONFIG_LED the module is compiled into the kernel (so the functionality provided by this module is added to the kernel) or as a module. Assuming that the filename for the module is led.c, only the line

```
obj-$(CONFIG_LED)                    += led.o
```

has to be inserted into the Makefile placed in ./uClinux-dist/linux-2.6.x/Bluetechnix. Last the module has to be added to the kernel configuration. The Kconfig file for the top level of the kernel configuration is placed in ./uClinux-dist/linux-2.6.x/arch/bfinnommu. For the configuration of device drivers normally Kconfig files included in the driver directory are used. So in the file ./uClinux-dist/linux-2.6.x/arch/bfinnommu/Kconfig a lot of lines similar to

```
source "drivers/Bluetechnix/Kconfig"
```

can be found. This means that the file ./uClinux-dist/linux-2.6.x/drivers/Bluetechnix/Kconfig is included in the top level Kernel configuration. Hence, the needed Kconfig file placed in ./uClinux-dist/linux-2.6.x/drivers/Bluetechnix looks like:

```
menu "Bluetechnix"

config LED
  tristate LED
  default y
  help
    Select this option if you don't have magic firmware for drivers that
    need it.
    If unsure, say Y.

endmenu
```

This file must be included into the top level kernel configuration. After that a submenu with the name Bluetechnix can be found in the kernel configuration. Important is the tristate statement. Only when using this statement for the LED option, it can be chosen if the module should be built in, should be built as module or not to build the module. Now the source code for the module itself has to be written and stored in the file led.c:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

static char *hellomsg="default";

static int __init led_init(void)
{
  printk (KERN_INFO "Hello, my %s module was loaded. \n", hellomsg);
  return 0;
}
```

```
static void __exit led_exit(void)
{
  printk (KERN_INFO "Goodbye\n");
}


MODULE_AUTHOR("Thomas Tamandl thomas.tamandl@bluetechnix.at");
MODULE_LICENSE("GPL");

module_init(led_init);
module_exit(led_exit);
module_param(hellomsg,charp,0);
```

These few lines of source code define the framework for a module. The heart of the module is the module_init and the module_exit macro. These macros define what to do when loading or unloading the module. If the module is compiled into the kernel the function defined with the module_init macro is called anyway. It is recommended to use the macros MODULE_AUTHOR and MODULE_LICENSE, but there is no error if they are missing. It is possible to supply some parameters to the module when loading it. Therefore the last line, module_param(parameter_name,parameter_type,permission) is used. If this line is missing, no parameters can be transferred to the module. ATTENTION, when passing parameters from the user space to the kernel space these parameters have to be carefully tested. A wrong, unchecked parameter can injure the module. As the corrupt module is running in the kernel space, it is possible that there are effects to the whole kernel.

In the source code provided above, the following commands or macros are used:

printk(log_level message):

> Due to the lack of the C library no printf() command is available in the kernel space, instead printk() has to be used. printk() can be used similar to printf(), except the log_level. These log_leves are used by the kernel to decide if the message is print to the console or not. In the source code above KERN_INFO is used as log_level. In the case of uClinux this log_level prints a message to the console.

module_param(var_name, type, permission):

> To be able to receive parameters the module_param macro has to be used. It is also possible to load this module without parameters, therefore a default value has to be set. The type of the variable can be set to bool, charp, int, ushort and so on. All module parameters can be represented by an entry placed in the /sys directory. The permission field controls the permissions for these entries. In the example above 0 is used to define the permissions for the hellomsg parameter, this means that this parameter is not visible in the /sysfs directory. The /sys directory is not mounted in uClinux, that directory can be mounted using “mount -t sysfs sysfs /sys”

A lot of information on modules can be found in [LOV04]. Especially some parameter types which are not described here, log_levels or permissions are described in this book.

Now the module has to be loaded, the compiled modules are placed in the /lib/modules/2.6.12.1/kernel/drivers directory. To load the demo module insmod has to be used in the following way:

```
insmod /lib/modules/2.6.12.1/kernel/drivers/Bluetechnix/led.ko

insmod /lib/modules/2.6.12.1/kernel/drivers/Bluetechnix/led.ko hellomsg=”my_module”
```

The second line shows how to pass parameters to the module. The module will be unloaded when using the rmmod command:

```
rmmod led.ko
```

## 6.2 Sample device driver

In this section a simple example for a character device driver will be explained. The provided information is based on section 6.1. To avoid the need for a new module, the file led.c is used for the sample driver. So no adoption of kernel build utility is needed.

As mentioned in [COR05] a device driver should not access the hardware using pointers. In the kernel 2.6.x there are functions like ioread8 or iowrite16 to get access to the memory. These functions are defined in ./uClinux-dist/linux-2.6.x/include/asm-bfinnommu/io.h. While writing this document, these functions were not implemented in this file. So there is no fully implemented interface for the kernel 2.6.x programming style available. As all other functions defined in io.h are using direct hardware access with pointers, all later presented driver samples are using direct memory access with pointers. Furthermore a lot of drivers supplied with uClinux do so. The interface to declare the driver to the kernel uses the recommended kernel 2.6 functions.

Explaining all details that can be found in the provided source code, would be outside the scope of this document. All of the not provided information can be found in [COR05].

The character driver that will be presented is able to switch on or off the LEDs and to read the state of the button.

```
static int __init led_init(void)
{
  int result;
  dev_t dev = MKDEV(led_major,0);
  struct cdev *my_cdev = cdev_alloc();

  result = register_chrdev_region(dev, 1, "CM-BF533 button and LEDs driver");
  if (result<0) return result;
  my_cdev -> owner = THIS_MODULE;
```

```
  my_cdev -> ops = &led_fops;
  if (cdev_add (my_cdev, dev, 1)) printk(KERN_INFO "Could not add device\n");

  create_proc_read_entry("led",0,NULL,led_read_proc,NULL);

  *pFIO_DIR |= 0x00C0;              //set PF4 to input and PF6+PF7 to output
  *pFIO_INEN |= 0x0010;             //enable input pin PF4
  *pFIO_FLAG_C |= 0x00C0;           //switch off the LEDs

  printk (KERN_INFO "CM-BF533 button and LEDs driver was loaded. \n");
  return 0;
}

static void __exit led_exit(void)
{
  dev_t dev = MKDEV(led_major,0);
  *pFIO_FLAG_C |= 0x00C0;           //switch off the LEDs
  *pFIO_DIR &= ~0x00C0;             //set PF6+PF7 to input
  *pFIO_INEN &= ~0x0010;            //disable input pin PF4
  unregister_chrdev_region(dev, 0); //unregister the character device
  remove_proc_entry("led",NULL);    //remove the /proc device entry
  printk (KERN_INFO "All configured as input again, goodbye.\n");
}
```

Like the demo module the functions led_init and led_exit are called when the module is loaded or unloaded. All work to register a character device to the system is placed in led_init. First a new character device is registered to the system. Therefore the command register_character_device(dev_t first, unsigned int count, char *name) is used. The dev_t structure is used to identify a device. It can be created out of the major and minor number using the MKDEV macro. The parameter count is the number of contiguous device numbers needed by the driver. For the sample driver one device is needed. The last parameter specifies which name for the device is used. This name represents the device in the /proc/devices directory. Now the kernel knows that there is a new character device, but at the moment, there are no functions that can be done from that device.

Inside the kernel files are represented through a so called inode structure. This inode contains the actual device number and the cdev structure. The cdev structure is used for the kernel internal representation of a character device. It holds a pointer to the file_operations structure that defines the work that can be done by the device. This new structure will be explained soon. As shown in the source code the cdev structure has to be initialized and some pointers have to be set. Now the function cdev_add (struct cdev *dev, dev_t num, unsigned int count) is used to register the cdev structure to the kernel. The parameters for that are a pointer to the explained cdev structure, the device number and the number of devices.

The file_operations structure is necessary for a driver. It defines what the driver can do and how to do it. For the sample driver this file_operations structure is used:

```
struct file_operations led_fops = {
  .owner = THIS_MODULE,
  .read =   pin_read,
```

```
  .write = pin_write,
  .open = pin_open,
  .release = pin_release,
};
```

There are a lot of fields defined by this structure. Here only four of them are used. For example, when the file representing the device is opened the function pin_open is called, and so on.

The functions used by this sample driver are:

```
int pin_open (struct inode *inode, struct file *filp)
{
  printk (KERN_INFO "CM-BF533 button and LEDs driver; file opened.\n");
  return 0;
}


int pin_release (struct inode *inode, struct file *filp)
{
  printk (KERN_INFO "CM-BF533 button and LEDs driver; file closed.\n");
  return 0;
}

ssize_t pin_read (struct file *flip, char __user *buf, size_t count, loff_t *f_pos)
{
  char buffer[4];
  if (count < 4) return -EMSGSIZE;
  buffer[0]=(*pFIO_FLAG_D & 0x0010) ? '0' :  '1';    //state of button
  buffer[1]=(*pFIO_FLAG_D & 0x0040) ? '1' :  '0';    //state of LED1
  buffer[2]=(*pFIO_FLAG_D & 0x0080) ? '1' :  '0';    //state of LED2
  return (copy_to_user (buf, buffer, 4) ? -EFAULT : 2);
}


ssize_t pin_write (struct file *flip, const char __user *buf, size_t count, loff_t *f_pos)
{
  if (count < 4) return -EMSGSIZE;
  if (buf[1] == '0')  *pFIO_FLAG_C = 0x0040;          //switch on LED1
  else *pFIO_FLAG_S = 0x0040;                         //switch off LED1
  if (buf[2] == '0')  *pFIO_FLAG_C = 0x0080;          //switch on LED2
  else *pFIO_FLAG_S = 0x0080;                         //switch off LED2
  return count;
}
```

In case of the CM-BF533 there is nothing to do when a file is opened or closed. The whole initialization has been done in the led_init function. As explained in section 5.7 read and write command can be used to communicate with the driver. For the communication with the driver three characters are used. When writing the character at position 0, representing the button, is ignored. The used data format can be seen in Table 6.1.

| position | pin_write | pin_read |
|---|---|---|
| char[0] | ignored | button |
| char[1] | LED1 | LED1 |
| char[2] | LED2 | LED2 |

**Table 6.1 Sample driver data format**

Before having a look on the functions required to operate with files, another data structure should be mentioned. As already shown all functions used to handle the device file includes a parameter with a pointer to a so called file structure. This structure represents an open file and is used only by the kernel. The file structure has nothing to do with the file pointer used in user space programs. To avoid confusion when using the file structure and a pointer to this structure, the pointer to the file structure is called filp (“file pointer”).

**int pin_open (struct inode *inode, struct file *filp) or**

**int pin_release (struct inode *inode, struct file *filp):**

These functions print a message to indicate that the device file has been opened or closed. The inode structure and the filp structure were explained above. The return value has to be 0 if there were no errors.

**ssize_t pin_read (struct file *flip, char __user *buf, size_t count, loff_t *f_pos):**

When a user space program calls the read command, this function is invoked. The character pointer buf points to a memory page that is used for the data transfer from or to the user space. Furthermore the number of characters to read is stored in the parameter count. The parameter f_pos can be used to define a special position to read, it is not supported here.

First the pin_read function checks that a valid number of characters will be read by the user space program. Then it fills up the buffer with the states of the GPIO pins. To transfer data to the user space, the buffer is copied into a specific memory page. This is done by the function copy_to_user(destination_address, source_address, number of bytes). The return value has to be set to the number of transferred bytes.

**ssize_t pin_write (struct file *flip, const char __user *buf, size_t count, loff_t *f_pos):**

This function is used when a user space program executes the write command. pin_write works similar to the pin_read function. It reads data from a memory page, assigns the data to the LEDs and returns the number of the read bytes.

Now the user space programs should be able to use the new device. A lot of devices create an entry in the /proc directory. This entry helps the user to get more information about the device. In the case of the sample driver, the actual state of the button and the LEDs will be displayed if someone reads the device file. To create such an entry the function create_proc_read_entry(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read, void *data) is used:

```
create_proc_read_entry("led",0,NULL,led_read_proc,NULL);
```

This function is executed out of the init_led function. The name for the file created in the /proc directory is led. The second parameter defined the protection mask, 0 is the default value. base indicates the subdirectory where to place the file, NULL means that the file is placed directly in the /proc directory. The fourth parameter defines the function to call when the file is read, and with the last parameter data can be passed to the led_read_proc function.

The led_read_proc function itself uses the parameters buf, count and eof:

```
int led_read_proc (char *buf, char **start, off_t offset, int count, int *eof, void *data)
{
  int len=0;
  len = sprintf(buf, "CM-BF533 button and LEDs driver\n");
  len += sprintf(buf+len, "The button is %s.\n", (*pFIO_FLAG_D & 0x0010) ? "not
                                                              pressed" :  "pressed");
  len += sprintf(buf+len, "The LED1 is %s.\n", (*pFIO_FLAG_D & 0x0040) ? "on" :
                                                                        "off");
  len += sprintf(buf+len, "The LED2 is %s.\n", (*pFIO_FLAG_D & 0x0080) ? "on" :
                                                                        "off");
  *eof=1;
  return len;
}
```

This function works in the following way: A parameter holding a pointer to a buffer is supplied to the function. The led_read_proc function puts its output into this buffer. When all output is transferred to the buffer, the end of file indicator (eof) is set. The return value of this function is the number of bytes stored into the buffer.

In section 5.6 and section 5.7 the file device_table.txt is described. This file is used to create the entries for device drivers placed in the /dev directory. To get access to the device driver described above, an entry for the driver is needed. This entry can be created by adding

```
/dev/led  c       664     0       0       252     0       0       0       -
```

to the device_table.txt file. Now it is possible to get access to the button and the LEDs by opening the file /dev/led and writing or reading some data to it. This can be done similar to section 5.7.

To read out the device file /dev/led the command

```
cat /dev/led
```

can be used, the output can be seen in Figure 6.1.

```
root:~> cat /proc/led
CM-BF533 button and LEDs driver
The button is not pressed.
The LED1 is off.
The LED2 is off.
```

**Figure 6.1 cat /proc/led**

## 6.3 Interrupts

Based on the sample driver, described in section 6.2, the feasibility to react on an external interrupt is discussed now.

Before implementing an interrupt handler, a brief introduction on the interrupt management system for the GPIO pins of the Blackfin processor will be given. Each GPIO pin can be configured to generate an interrupt. The Blackfin processer provides two different interrupt channels, these channels are called Flag Interrupt A and Flag Interrupt B. All pins can be configured to generate an interrupt on one ore both channels. Inside the uClinux kernel the two interrupt channels are represented by the name IRQ_PROG_INTA or IRQ_PROG_INTB. Before adding an interrupt handler to the system, the Blackfin processor has to be configured to generate an interrupt for a selected GPIO pin. Therefore changes to a few registers have to be done.

While writing this document the interrupt IRQ_PROG_INTB is used by the network controller, so for the sample driver IRQ_PROG_INTA has to be used. The button on the DEV-BF5xx Development Kit is connected to PF6 (programmable flag 6 accords to the 6$^{th}$ GPIO pin) pin. So the Blackfin processor has to be configured to generate an interrupt when the button is pressed. Therefore the following steps have to be performed:

1. PF4 has to be configured as input. Therefore the fourth bit of the Flag Direction Register (FIO_DIR) has to be configured as 1 and the fourth bit of the Flag Input Enable register (FIO_INEN) has to be configured as 1.
2. PF4 has to be configured to generate an interrupt on the IRQ_PROG_INTA channel. Therefore setting the fourth bit inside the Flag Mask Interrupt A Data Register (FIO_MASKA_D) has to be set to 1. To avoid the previous reading of FIO_MASKA_D, setting the bit to one and then write the register back, the register Flag Mask Interrupt A Set Register (FIO_MASKA_S) is used to set the bit inside of FIO_MASKA_D. Now PF4 is assigned to the IRQ_PROG_INTA. It is possible to assign further pins to this interrupt channel or to assign PF4 to the other channel, too.
3. Now it has to be decided when the interrupt should be generated. It is possible to configure a level- sensitive or an edge- sensitive interrupt. Therefore the following registers are needed. The Flag Polarity register (FIO_POLAR) is used to configure

the polarity of the input source. This means, when writing a 0 to this register an interrupt will be generated if a logic 1 or a rising edge is detected on the input corresponding input pin. When set to 1, logic 0 or a falling edge will generate an interrupt.

To configure if an interrupt source is level or edge sensitive the Flag Interrupt Sensitivity register (FIO_EDGE) is used. A 0 inside this register indicates that the interrupt source is level sensitive, or if configured to 1, it is edge sensitive. When a PFx pin has set to be edge sensitive, also both edges can be used for interrupt generation. This can be achieved by setting the corresponding bit inside the Flag Set on Both Edges register (FIO_BOTH).

Once the processor is configured to generate an interrupt, the interrupt channel has to be registered to the kernel. Therefore the function request_irq is used. All the work described above will be done in the function led_init (described in section 6.2):

```
static int __init led_init(void)
{
  ---cut---

  create_proc_read_entry("led",0,NULL,led_read_proc,NULL);

  *pFIO_DIR |= 0x00C0;              //sets PF4 to input
  *pFIO_INEN |= 0x0010;             //enable the input buffer for PF4
  *pFIO_FLAG_C = 0x00C0;            //clear pending interrupt

  *pFIO_MASKA_S = 0x0010;           //assign PF4 to Flag Interrupt A
  *pFIO_POLAR |= 0x0010;            //edge-sensetive
  *pFIO_EDGE |= 0x0010;             //generate interrupt on falling edged


  if (request_irq (IRQ_PROG_INTA, led_irq_handler, SA_INTERRUPT,"CM-BF533
                                                        button interrupt",NULL))
            {
               printk(KERN_INFO "CM-BF533: can not get IRQ
                                                          %d.\n",IRQ_PROG_INTA);
               return -EIO;
            }

  printk (KERN_INFO "CM-BF533 button and LEDs driver was loaded. \n");
  return 0;
}
```

As shown in the source code above, PF4 is configured to generate an interrupt on channel A when a falling edge was detected. The function request_irq works in the following way. The first parameter defines the requested interrupt number. led_irq_handler defines the interrupt handler function. SA_INTERRUPT indicates that a “fast” interrupt handler is used. This includes that all other interrupts are disabled while executing the interrupt handler. The third parameter defines a device name. This name is used to show the owner of an interrupt line in /proc/interrupts. The last parameter is used to define a device id when using shared interrupts.

Now the demo driver is able to detect a falling edge on PF4. When the interrupt is generated, the interrupt handler function is called:

```
static irqreturn_t led_irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  *pFIO_FLAG_C = 0x0010;          //clear the interrupt
  *pFIO_FLAG_T = 0x0040;          //toggle the LED connected to PF6
  return IRQ_HANDLED;
}
```

The parameters used by the interrupt handler are the interrupt number, some client data (described in [COR05]) and a pointer to a snapshot of the processor register before the processor entered the interrupt code. The handler itself clears the pending interrupt and toggles the LED connected to PF6 to indicate that the interrupt handler was executed. If there was an interrupt to handle, in case of the CM-BF533 there is only one, IRQ_HANDLED has to be returned.

An interrupt handler used in “real life” has to take care of the bouncing of the button. When the button is pressed once, often more than one edge will be detected. The demo driver provides no functions for debouncing the switch.

## 6.4 Tasklets and Work Queues

Up to now the demo driver is able to control the LED’s, to read out the state of the button and to toggle one LED with help of an interrupt handler triggered by PF6. As explained in subsection 2.3.7 work can be deferred to speed up an interrupt handler. In this chapter the ability to use tasklets or work queues will be added to the sample driver. Remeber, one of the major differences between tasklets and work queues is, that work queues are able to sleep. Even if both bottom halves use a similar interface, there are a lot of differences how they are used by the kernel.

First the functionality to defer work will be added to the demo driver using a tasklet. The tasklet has to be declared with the DECLARE_TASKLET(name, function,data) macro, the demo driver uses

```
DECLARE_TASKLET(cm_bf533_tasklet, tasklet_handler, 0);
```

to declare a tasklet with the name cm_bf533_tasklet. tasklet_handler is the name of the function that will be called if the tasklet is executed and data can be used to pass data to the tasklet_handler. To defer work to the cm_bf533_tasklet the interrupt handler has to be changed in the following way:

```
static irqreturn_t led_irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  *pFIO_FLAG_C = 0x0010;                  //clear the interrupt
  tasklet_schedule(&cm_bf533_tasklet);    //schedule the tasklet
  return IRQ_HANDLED;
}
```

As shown in the provided source code, there is no code for toggling the led. This work is deferred to the tasklet, so the tasklet has to be scheduled from the interrupt handler.

In case of the demo driver the tasklet only toggles the state of the LED. In reality nobody will use a tasklet in this way. For the demo driver it is the easiest way to indicate that the tasklet was executed.

```
void tasklet_handler (unsigned long data)
{
  *pFIO_FLAG_T = 0x0040;          /toggle the led
}
```

The parameter data is defined when declaring the tasklet, see above.

As already mentioned workqueues use a similar interface. Work that has to be run out of a workqueue has to be declared using the following macro:

```
int data = 0;
DECLARE_WORK(cm_bf533_work_queue, work_queue_handler, &data);
```

The macro DECLARE_WORK(name, function, data) uses three parameters. The first parameter is the name of the workqueue, the second is the function called to do some work and the third parameter is a pointer to some data that can be provided by the function.

The interrupt handler used for deferring work to a workqueue has to look like:

```
static irqreturn_t led_irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  *pFIO_FLAG_C = 0x0010;                    //clear the interrupt
  schedule_work(&cm_bf533_work_queue);      //schedule work
  return IRQ_HANDLED;
}
```

Similar to the interrupt handler described for tasklets, this interrupt handler deferes the work to toggle the led. Therefore it has to clear the pending interrupt and to schedule the workqueue.

The worker function looks like:

```
void work_queue_handler (int *data)
{
  *pFIO_FLAG_T = 0x0040;
}
```

# Chapter 7 Summary and further projects

uClinux is a very interesting but also very complex topic. Initially it was built to provide a simple UNIX compliant operating system on MMU less microcontrollers, nowadays uClinux has grown up to a full operating system. While writing this document a patched kernel version 2.6.12 was used by uClinux. Besides providing a multitasking operating system uClinux provides a standardised interface for application software. Furthermore there are a lot of drivers and application software that can be used.

The aim of this work was to port uClinux to the Bluetechnix CM-BF533 Core Module. While developing uClinux a bootloader, named “Das U-Boot”, is used to download the uClinux image from the development workstation to the target platform.

The CM-BF533 is based on an Analog Devices ADSP-BF533 Blackfin processor. A Blackfin processor specific port of the U-Boot and the uClinux can be found at http://blackfin.uclinux.org. Based on the sources for the Blackfin U-Boot project and the Blackfin uClinux project the porting of the two projects to the CM-BF533 was started. First the U-Boot was ported to the target hardware. This had to be done first because without the U-Boot a JTAG interface is needed to write data to the Flash memory. A JTAG device is a very expensive device, so a lot of uClinux developers prefer to use the U-Boot for handling the Flash memory. Although a JTAG interface was available it could not be used for debugging because no Linux driver was available for the workstation. Once the U-Boot is ported to the CM-BF533 no additional hardware is used to load the uClinux image to the development hardware and to start it.

Porting the U-Boot to the CM-BF533 means adapting all development hardware related parts to the new platform. The primary goals while porting the U-Boot were to bring up the Network and to get access to the Flash memory. The Flash memory driver is used for writing and erasing the Flash memory. Therefore Flash specific commands had to be applied to the Flash memory. The Flash memory driver, supplied by the uClinux project, is not suitable for the CM-BF533 because the used Flash memory is incompatible to the originally used driver. So a new driver to get access to the Flash memory was written. To bring up the network some changes in the board specific configuration files had to be done.

After the porting of the U-Boot had been finished, the porting of the uClinux was started. In contrast to the U-Boot, there were fewer problems caused by the hardware but it took a lot of time to find some specific configuration files. As the uClinux project contains about 77000 files in approximately 5000 directories it sometimes is very tricky to find a specific file.

The work that has been done can be summarised by presenting the following results:

- The U-Boot has been successfully ported to the CM-BF533. If the CM-BF533 is used with the DEV-BF5xx Development Kit the Ethernet interface can be used to

download a uClinux image. Afterwards it can be stored into the Flash memory. The CM-BF533 is fully integrated in the U-Boot source code. So it can be built by using a single command.

- The uClinux has been successfully ported to the CM-BF533. The driver for the Ethernet controller has been adapted to work with the DEV-BF5xx Development Kit. A new vendor directory has been created and the CM-BF533 has been added to the kernel configuration. So it is possible to configure all CM-BF533 specific options with the graphical kernel configuration utility.
- The CM-BF533 has been added to the official uClinux CVS sources. Thanks to Robin Getz!
- A framework on how to generate user programs, modules and a character device driver with different features has been designed.
- The U-Boot and the uClinux have been successfully tested when using the CM-BF533 on the EVAL-BF5xx Blackfin Evaluation Board.

Now a short overview of the problems while porting the U-Boot to the CM-BF533 will be given:

- uClinux is similar to standard Linux. It takes a lot of time to figure out the differences because the information on uClinux is scattered on the project homepage, a lot of other related homepages, mailing lists and forums. As a lot of individuals are working on uClinux a lot of programs are available for uClinux, unfortunately this projects are roughly documented.
- When the porting of the U-Boot and the uClinux was started the RTC didn’t work. At that time the RTC was needed for a successful boot up, so a workaround had to be found. These days options for not to use the RTC are added.
- There was no documentation on how to add new development hardware to the U-Boot or the uClinux. So a lot of time had to be used to find out how to add Bluetechnix specific configurations to the source code and how to configure them.

As already mentioned uClinux to the CM-BF533 has been successfully ported. So uClinux can be used for further software development. The next steps should be to adapt or program all drivers needed to access the SD-card, the CF-Card or to connect a CMOS camera. For uClinux a simple web server called BOA is available. This web server can be used to publish the pictures made by the CMOS camera.

# List of Figures

# List of Tables

# References

[ANA04] ADSP-BF533 Blackfin Processor Hardware Reference
Revision 3.0, September 2004

[BON99] Chris Di Bona: Open Sources: Voices from the Open Source Revolution.
O'Reilly, Sebastopol, 1999

[COR05] Jonathan Corbet,Alessandro Rubini,Greg Kroah-Hartman:Linux device driver 3.ed.
O'Reilly, Sebastopol, 2005

[HEA03] Steve Heath: Embedded Systems Design.
Newnes, Newnes, 2003

[KOL02] Kolagotla, Fridman, Aldrich, Hoffman, Anderson, Allen, Witt, Dunton, Booth:
"High performace dual-MAC DSP architecture",
IEEE Signal Processing Magazine, Volume 19, Issue 4, July 2002

[LEV99] John Levine: Linkers and Loaders.
Morgan Kaufmann, San Francisco, 2000

[LOV04] Robert Love: Linux Kernel Development.
Sams Publishing, Indianapolis, 2004

[MIC05] Datasheet Micron MT28F320J3 Q-Flash Memory
Revision 3/05

[OS99] Übersetzung und Bearb. von Snoopy und Martin Müller: Open Source kurz&gut.
O'Reilly, Köln, 1999

[SYS02] zsgest. von Wilfried Elmenreicht:
Systemnahes Programmieren - C Programmierung unter Unix und Linux.
Institut für Technische Informatik, Wien, 2002

[YAG03] Karim Yaghmour: Building embedded systems.
O'Reilly, Sebastopol, 2003

Internet References:

[WANA1] http://www.analog.com/, 16.Aug. 2005

[WBLU1] http://www.bluetechnix.at/ 10.Oct. 2005

[WBUS1] http://www.busybox.net/, 14.Sep. 2005

[WCOL1] http://www.columbia.edu/kermit/, 14.Sep.2005

[WCYB1] http://www.cyberguard.info/snapgear/tb20020530.html, 21.Sep. 2005

[WDEB1] http://alioth.debian.org/projects/minicom/, 14.Sep. 2005

[WFSF1] http://www.fsf.org/ 30.Aug. 2005

[WGNU1] http://www.gnu.org/, 14.Sep. 2005

[WLWN1] http://lwn.net/Articles/driver-porting/, 12.Sep. 2005

[WOPE1] http://opensource.org/, .8.2005

[WPAT1] http://www.pathname.com/, 15.Sep. 2005

[WUCL1] http://www.uclinux.org, 15.Sep. 2005

[WUCL2] http://blackfin.uclinux.org, 22.Sep. 2005

[WWIK1] http://en.wikipedia.org, 26.Aug. 2005