MASTERARBEIT

Embedded Web Radio

Ausgeführt am
Institut für Computertechnik
Institutsnummer: 384
der Technischen Universität Wien

unter der Anleitung von
O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
und
Univ.Ass. Dipl.-Ing. Dr.techn. Stefan Mahlknecht
als verantwortlich mitwirkendem Universitätsassistenten

durch

Bakk.techn. Harald Krapfenbauer
Matr.Nr. 0125556
Waldrandsiedlung 126, 3910 Zwettl

April 2007

Kurzfassung

Ein *embedded web radio* ist ein autonomes Gerät, das es ermöglicht, Radiosender aus dem Internet ähnlich wie mit einem herkömmlichen terrestrischen Radioempfänger zu hören.
Internetradio bezeichnet eine Technik, die über das Internet fortlaufende Datenströme mit Audiosignalen überträgt. Verglichen mit traditionellem Rundfunk, sind dabei die zwei wesentlichen Vorteile, dass Internetradiosender überall in der Welt empfangen werden können und die Anzahl der Sender nicht von vornherein begrenzt ist.
Das Ziel dieser Arbeit ist es, ein vielseitiges Gerät zu entwickeln: Außer seiner Hauptfunktion – dem Abspielen von Radiosendern aus dem Internet – soll es die Wiedergabe von Audiodateien sowohl im lokalen Netzwerk als auch von einem Massenspeicher ermöglichen. Die Unterstützung all dieser Funktionen unterscheidet das Webradio von anderen kommerziell erhältlichen Produkten. Der Fokus der Arbeit liegt dabei rein auf der Funktionalität, das Design des Geräts wird nicht berücksichtigt.
Die für das Webradio ausgewählte Hardwareplattform basiert auf dem Blackfin-Prozessor ADSP-BF537 von Analog Devices und besteht aus mehreren Entwicklungsplatinen, die am Institut für Computertechnik entworfen wurden.
Das Betriebssystem *uClinux*, das für die Blackfin-Architektur verfügbar ist, wird auf der Zielhardware eingesetzt. Diese Linux-Distribution enthält den für eingebettete Prozessoren angepassten Linux-Kernel und darüber hinaus eine Fülle an bereits vorhandenen Werkzeugen und Anwendungen. Für das Webradio bietet es viele Softwarekomponenten wie Gerätetreiber für Audio und Netzwerk, Audiodecoder und Unterstützung für Netzwerk-Dateisysteme.
Die folgenden Softwarekomponenten wurden neu implementiert: ein Gerätetreiber für ein farbiges TFT-Display, welches zur Anzeige der grafischen Benutzeroberfläche dient, die zentrale Anwendung für die Steuerung und eine Webanwendung zur einfachen Konfiguration des Geräts. Weiters wurden der vorhandene Gerätetreiber für Secure-Digital-Speicherkarten sowie der Audiospieler verbessert.
Das Ergebnis dieser Arbeit ist ein voll funktionsfähiges Webradio.

Abstract

An *embedded web radio* is a stand-alone device that is capable of playing web radio stations from the Internet. It is similar to a traditional radio receiver.
Web radio is a technique that broadcasts continuous audio streams over the Internet. The main advantages compared to terrestrial radio are that web radio stations are accessible from anywhere in the world and the number of stations is not limited.
The aim of this work is to develop a versatile product: Apart from its main function – the playback of web radio stations – listening to audio files from the local network as well as from some mass storage shall be enabled. Supporting these functions differentiates the embedded web radio from available commercial products. The focus thereby solely lies on the functionality, design issues are neglected.
The target hardware platform that is chosen for the embedded web radio is based on the Blackfin processor ADSP-BF537 from Analog Devices and is comprised of several development boards developed at the Institute of Computer Technology.
The *uClinux* operating system which is available for the Blackfin architecture is employed onto the target. The uClinux distribution contains a Linux kernel that is adapted to the needs of embedded processors and a great many tools and applications. It provides a lot of necessary software components for the embedded web radio, like device drivers for audio and network, an audio player and support for network file systems.
The following software components are implemented from scratch: A device driver for a color TFT-LCD which provides a graphical user interface, the main control application, and a web application for easy configuration of the embedded web radio. Furthermore, an available device driver for secure digital (SD) memory cards is enhanced as well as the audio player.
The result of this work is a full-functional embedded web radio device.

Acknowledgements

Special thanks go to my parents who supported me in all respects during my studies in Vienna.

I want to thank my supervisor Dipl.-Ing. Dr.techn. Stefan Mahlknecht as well as Dipl.-Ing. Dr.techn. Gregor Novak for their helpful pieces of advice.

Furthermore, I'm grateful to the "Blackfin expert" Dipl.-Ing. Roland Oberhammer and Martin Hausner for his hardware support.

Contents

Abbreviations

ADI	Analog Devices, Inc.
ADC	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
BFLT	Binary Flat
CFI	Common Flash Interface
CGI	Common Gateway Interface
CVS	Concurrent Versioning System
DAC	Digital-to-Analog Converter
DIP	Dual In-line Package
DMA	Direct Memory Transfer
DSP	Digital Signal Processor
ELF	Executable and Linking Format
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPIO	General Purpose Input/Output
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
LAN	Local Area Network
LCD	Liquid Crystal Display
MMC	MultiMedia Card
MMU	Memory Management Unit
MPEG	Motion Picture Expect Group
PCM	Pulse-Code Modulation
PPI	Parallel Peripheral Interface
PWM	Pulse-Width Modulation
RAM	Random Access Memory
ROM	Read-Only Memory
SD	Secure Digital
SDRAM	Synchronous Dynamic Random Access Memory

SPI	Serial Peripheral Interface
TFT	Thin Film Transistor
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver/Transmitter
URL	Uniform Resource Locator
USB	Universal Serial Bus

Chapter 1

Introduction

1.1 Web radios

Web radio, also referred to as *Internet radio*, denotes a technique that broadcasts continuous audio streams over the Internet, much like traditional terrestrial radio. So the purpose and the use of traditional radio and web radio are nearly identical. Since the number of users that have broadband access to the Internet available increases, the web radio technique becomes more and more attractive as alternative to terrestrial radio. Although it is not widespread today, an increase of popularity can be forseen due to a couple of essential advantages which will be discussed later.

There are some major attributes common to web radio, terrestrial radio and satellite radio as well:

- A continuous audio stream is delivered to the listener.
- Listeners have no control over this audio stream as opposite to *on demand* services.
- The concept of stations is applied to all techniques: A radio station of terrestrial or satellite radio corresponds to a dedicated channel that is denoted by a frequency. With web radio, the term “web radio station” will be used throughout this document. Such a station is uniquely described by its Internet address, commonly known as URL [Ber94].
- There are web radio stations that correspond to terrestrial radio stations to reach more listeners. However, most web radio stations broadcast by means of the Internet only.

In the following, the differences between web radio and traditional radio shall be specified, together with advantages of the former:

- It was already mentioned that the medium of transmission is different: Terrestrial radio is broadcasted via radio waves through open space, whereas web radio is transported by means of the Internet.
- Traditional terrestrial radio is transported with analogue technique, though digital terrestrial radio will take over in the future. Web radios can only be transported in a digital form due to the inherent nature of data transfers in the Internet.

- Whereas terrestrial radio stations may only be received locally within their transmission range, web radio stations are accessible from anywhere in the world, provided that Internet access is available. This makes the latter a very popular service for travelers and people who spend a length of time abroad and want to tune in to radio stations from home.
- With conventional radio, the amount of stations that can be broadcasted simultaneous is limited. This is not true for web radio, where theoretically an infinite number of stations spread over the Internet may exist.
- The previous point leads to the advantage that every listener is likely to find web radio stations that suit one's taste exactly, because there is simply a much greater supply available. Hence web radio is popular for listeners who's interests are not adequately served by local radio stations provided by terrestrial or satellite radio.
- Finding a radio station is traditionally accomplished by means of a manual or automatic channel search. However, this cannot be applied to web radio stations because that would require scanning the whole Internet. The task of channel searching is undertaken by special search engines. Popular ones are *SHOUTcast* (`http://www.shoutcast.com`), *Live365* (`http://www.live365.com`) and Icecast (`http://www.icecast.org`) [WWIb].

The technique that is used for web radio is called "audio streaming" or "audio broadcasting". To be able to transmit such streams with a reasonable data rate, audio signals are encoded by means of an audio codec. The MP3 codec is most popular, followed by Ogg Vorbis, Windows Media Audio and Real Audio [WWIb]. The digital stream containing audio data is mostly transferred over the network by means of HTTP[1] which is the most common one used in the Internet.

To be able to listen to web radio stations, software or hardware is needed that is capable of receiving HTTP data streams and decoding the audio signal for playback. Examples of software players that run on a personal computer are *Winamp* (`http://www.winamp.com`) for Microsoft Windows, *iTunes* (`http://www.apple.com/itunes`) for Macintosh and Windows PCs, and *XMMS* (`http://www.xmms.org`) for Unix/Linux operating systems.

1.2 Problem description

Today, listening to web radio stations is mostly done by means of a personal computer with a connection to the Internet and appropriate software. Furthermore, loudspeakers must be attached to the computer. While this is a convenient way for web radio playback if the listener anyway works at the PC, the need of a PC to be able to listen to web radio stations is a big drawback in general. It should be as easy as listening to traditional radio stations with a radio receiver present in every household.

The aim of this project is to fill this gap by creating a stand-alone device that is independent from a PC and allows playing web radio stations by means of an Internet connection. This is the most important function of the proposed device. However, to develop an interesting and versatile product, additional features shall be integrated: Playback of music stored on

[1]HyperText Transfer Protocol

other devices in the local network shall be possible. To make the product portable, some mass storage containing audio files shall be able to be connected to the device. This allows employment even if Internet access is not available.

Actually, this idea is not completely new. There are a few vendors that already sell such products. The intended use of these devices vary. Some are part of complete systems that aim for professional use in restaurants or companies where the same music shall be transferred to many different locations. Others are consumer products and hence target the same class as the embedded web radio which will be developed here. In the following, 3 of these devices are picked out and shortly introduced:

Squeezebox is a product from the company *Slim Devices* [WSL]. It primarily aims at listening to music from the local network. Therefore it requires a proprietary software installed on each PC to function as a source for audio files. It also supports playback of web radio stations. Therefore it connects to the *SqueezeNetwork* web service to get a list of channels. Connecting a mass storage device is not supported.

LUKAS-EV is sold by the company *Streamit* [WST]. It is aimed for professional use in large buildings and optimized for autonomous operation. Configuration with a PC over a USB cable is necessary prior employment. It has two simple operation modes: Either a fixed web radio station is programmed that is played or a special web server delivers a list of stations, whereof the first is chosen and played. Playback of music files from the local network is not supported, neither is the connection of some mass storage.

Acoustic Energy Wi-Fi Radio was developed by the company *Reciva* [WREb]. It supports several audio codecs. Streaming of web radio stations is supported, whereby the list of available stations is downloaded automatically from the manufacturer's homepage. Playback of audio files from the local network is also possible. No mass storage can be connected to the device.

The embedded web radio that shall be developed in this project differentiates from the above ones by the following features:

- As already mentioned earlier in this section, the device shall be versatile by offering playback of web radio stations, audio files from the local network, and audio files from some mass storage, e.g. a memory card.
- The list of available web radio stations is not downloaded from a central server, but is totally configurable by the user.
- For data exchange within the LAN, common protocols shall be supported. There must be no need for additional software on a PC to transfer audio data to the embedded web radio.

The aim of this work is to create a fully-functional stand-alone device which typically consists of hardware and software. Questions of design, which would be inevitable to make a sellable end product, are totally neglected. As an example, such a question would be whether loudspeakers are integrated into the case or the device is connected to an external audio system. The focus of this project lies solely on the functionality.

1.3 Structure of this document

This section will give an overview about the work that was done. The chapters this document is divided into correspond to phases of development.

Chapter 2 deals with requirements analysis and system architecture for the embedded web radio. Basic inputs and outputs are determined, functional and non-functional requirements are worked out, and a list of required hardware and software components is made. The chapter ends with detailed use cases for the device.

Chapter 3 describes the composition of the target platform. The hardware components are introduced and configured. The selected hardware builds upon a Blackfin processor from ADI[2]. Afterwards, software issues are treated. Thereby, uClinux comes out as best suited operating system for the purpose of this project, because lots of required software components are already included.

Chapter 4 gives an introduction to the uClinux operating system for the Blackfin processor. The exercised development environment is described and the 3 main software packages are explained that are needed for uClinux: The toolchain for cross-compiling, the bootloader "Das U-Boot", and the uClinux distribution itself containing the Linux kernel and user space applications.

Chapter 5 deals with configuration of available software components, notably audio, Ethernet and flash memory device drivers and network applications.

Chapter 6 builds the main chapter of this document because it describes the software development phase. Device drivers are implemented first: For an LCD (liquid crystal display) one has to be implemented for the uClinux kernel. Furthermore, the available SD card device driver needs to be enhanced regarding stability to be usable for the embedded web radio. Thereafter, 2 user space applications are developed: The main control application and the web application for configuration of the device. Furthermore, the audio player is slightly modified.

Chapter 7 describes hardware modifications that are necessary for proper collaboration of all components.

Chapter 8 deals with testing and final work. Accurate testing is conducted with the main application. Finally, the work is completed by deploying the software onto the target.

Chapter 9 includes a summary of the work. Problems are pointed out and a list of imaginable future enhancements of the embedded web radio is presented.

[2]Analog Devices, Inc.

Chapter 2

Requirements analysis and system architecture

This chapter presents various sections about the planning phase of the embedded web radio device. To support effectiveness, some techniques from the field of software engineering are applied and adopted to this project's specific needs. The requirements elicitation process starts with a brainstorming where a general idea about the capabilities of the system is found. Afterwards, functional and non-functional requirements are determined. During the system architecture phase a hardware/software component analysis is accomplished and a detailed use cases document is worked out.

2.1 Brainstorming

Brainstorming is a technique to generate new ideas and possible solutions [Mac05] and can be used for requirements elicitation at the beginning of system design.

Since an *embedded web radio* per se implies nothing than being able to play audio broadcasts from the Internet, in this phase additional features are found and a basic concept of the device is created. Two questions are important during this phase:

1. What features should be supported by the system?
2. What are the inputs and outputs of the system?

The brainstorming process leads to the following unordered set of features which should be supported by this device:

- The web radio device should be a stand-alone device similar to a traditional radio.
- Web radio stations (audio streams) from the Internet should be able to be played. This is the main functionality of the device. Since there exist various audio stream encodings, it has to be analyzed which encodings should be supported. Furthermore, the type of network connection has to be discussed (cable Ethernet, wireless Ethernet, etc.).

- Audio data from the local network should be able to be played. There may be audio broadcasts on the local network too, but it should also be possible to play audio files from a personal computer or another network storage connected to the local network.
- Audio playback should also be possible from some mass storage. A mass storage device should be able to be connected to the embedded web radio, which plays audio files from there.
- The list of web radio stations should be able to be organized by the user. Depending on the user input facility, this may happen on the device itself (if e.g. a touchscreen is available) or through a web interface (more comfortable).
- Web radio stations should be organized in categories.
- The device should provide easy interaction with the user. Therefore, a display is required. For user operation buttons or a touchscreen are/is necessary.
- The playback volume should be adjustable.
- Audio output should be in stereo mode.
- The network connection should be able to be configured by the user.
- Whenever a list of audio files is provided (from mass storage, from personal computers, etc.), navigating through the directory structure should be possible.
- Since the device will not be an end product ready for mass production, components should be chosen with respect to future enhancements wherever possible.

Figure 2.1 points out the main input and output components that will be needed, based on the following input/output analysis of the system:

Inputs:

- Electrical power
- Digital audio stream data through network connection
- Digital audio file data from network device
- Digital audio file data from a mass storage device
- User operation (operation facility, possibly web browser)

Outputs:

- Analogous audio signals
- Visual output on display
- possibly configuration web pages

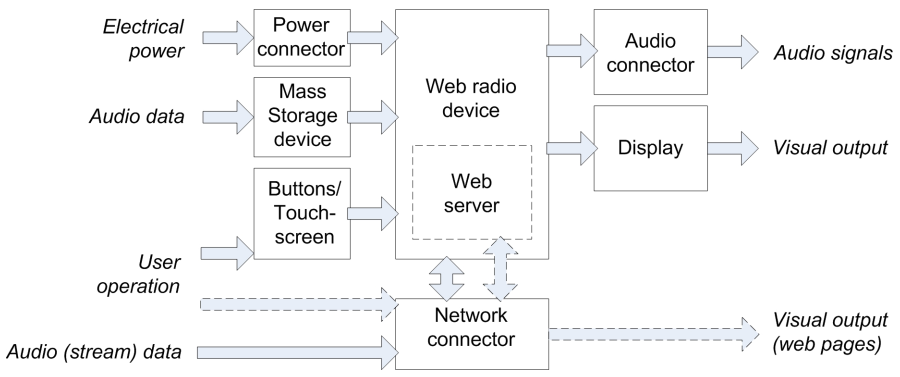


Figure 2.1: Basic inputs and outputs of the system (optional elements are marked by a dashed line)

2.2 Requirements engineering

Based on the results of the brainstorming phase, the next step is to build a complete list of requirements. These are subdivided in functional and non-functional requirements. A decision will be made at every point that was left unspecified up to now due to several different possibilities. A rationale is given accordingly.

To allow easy referencing, each requirement gets assigned a unique number, preceded by an "F" for a functional requirement or an "N" for a non-functional requirement.

2.2.1 Functional requirements

No. F1 – Web radio streams
SUMMARY:
The system shall be able to play stereo audio streams provided in the MP3 format.
DESCRIPTION:
The system connects to an Internet server which broadcasts a digital audio stream. An audio stream is specified by a URL common in the Internet world. Most URLs of audio streams contain a port number. Example: `http://www.someserver.com:1234/stream`
RATIONALE:
A rationale has to be given regarding the selected audio codec: The MPEG1 Layer III codec, short *MP3*, is the most widespread one for web radio streams. This was determined during an investigation on three popular Internet sites which offer search engines for web radio stations. Hence it was decided to support the MP3 codec. The investigation result is presented in table 2.1.

	total	MP3	AAC	Ogg/Vorbis
www.shoutcast.com	16700 (100%)	15000 (90%)	1700 (10%)	–
www.live365.com	> 1000 (100%)	> 1000 (100%)	–	–
www.icecast.org	950 (100%)	680 (72%)	50 (5%)	220 (23%)

Table 2.1: Spread of various audio codecs at popular web radio search engines (as of March 2, 2007)

No. F2 – Web radio stations listing

SUMMARY:
The system shall provide organization of web radio stations into categories and appropriate listings.
DESCRIPTION:
The system allows categorization of web radio stations. It is able to provide a listing of categories and listings of respective contents. A web radio station is defined by its name and URL. A default set should be provided.
RATIONALE: —

No. F3 – Network connection

SUMMARY:
The system shall be able to connect to an Ethernet network via a network cable.
DESCRIPTION:
The system connects to a server over an Ethernet network. The physical connection is made with an Ethernet cable (IEEE 802.3 standard).
RATIONALE:
Ethernet is the most widespread protocol for home and office networks. Cable Ethernet is used in favour of Wi-Fi[1] because the former is more common, requires less configuration and provides higher security per se in comparison to Wi-Fi (see requirement no. N3).

No. F4 – Network shares

SUMMARY:
The system shall be able to play MP3 stereo audio files from shared directories on devices on the local network.
DESCRIPTION:
The system accesses shared directories on the local network called *network shares* and plays MP3 audio files from there. The network shares are exported by the host system (e.g. a desktop PC) via SMB (server message block) or NFS (network file system) protocols and do not require any authentication. The URLs which identify the network shares must be configured before (see requirement no. F10).
RATIONALE:
The SMB protocol, described in [Her03], provides shared access to files on machines running Microsoft's Windows operating system. Most network hard disks provide SMB access too. The NFS protocol ([CPS95], [Sun89]) is common in the Unix/Linux world and also available for Apple's MacOS. By supporting these two protocols it is guaranteed that most users can easily set up a network share.

No. F5 – Mass storage

SUMMARY:
The system shall be able to play MP3 files from an attached SD (secure digital) memory card.
DESCRIPTION:
The system accesses an SD memory card and plays MP3 audio files from there. The memory card is formatted with the VFAT[2] file system.

[1]Wireless Fidelity
[2]Microsoft's *F*ile *A*llocation *T*able file system with long file name support

RATIONALE:
There are many different mass storage devices available. The two most common are USB[3] flash drives and SD memory cards. Latter are chosen here because they have a way simpler hardware interface and hence hardware is cheaper. Furthermore, SD memory cards itself are very cheap nowadays.

No. F6 – Directory listing
SUMMARY:
The system shall be able to provide listing and navigating through directories.
DESCRIPTION:
If the system accesses directories from network shares or SD card, it provides the ability to list and navigate through directories to the user. Only MP3 audio files that can be played by the system and subdirectories are shown in a directory listing.
RATIONALE: —

No. F7 – Display
SUMMARY:
The system shall be able to provide a graphical user interface on a display.
DESCRIPTION:
An LCD[4] with a size of 320 x 240 pixels is a key feature of the system. Contents are displayed in landscape format.
RATIONALE:
To be able to show directory listings on the display, a reasonable size is required.

No. F8 – Menu structure
SUMMARY:
The system shall provide a hierarchical menu structure.
DESCRIPTION:
A hierarchical menu structure will be provided by the system and shown on the LCD. Refer to the sequence diagram in figure 2.2 for the proposed structure.
RATIONALE: —

No. F9 – Buttons
SUMMARY:
The system shall be operated by the user with 4 multi-purpose buttons and a reset button.
DESCRIPTION:
The user will operate the system with 4 buttons. These buttons don't have a fixed function assigned, but their functions depend on the respective submenu. This is accomplished by placing the buttons near the bottom of the LCD and depicting their dynamic functions on the LCD. Moreover there will be a reset button, which does not need to be placed near the LCD.
RATIONALE:
Buttons will be used here in favour of a touch screen because of the following rationales:
1. Button operation is less complicated for users in combination with the proposed button concept.

[3]Universal Serial Bus
[4]Liquid Crystal Display

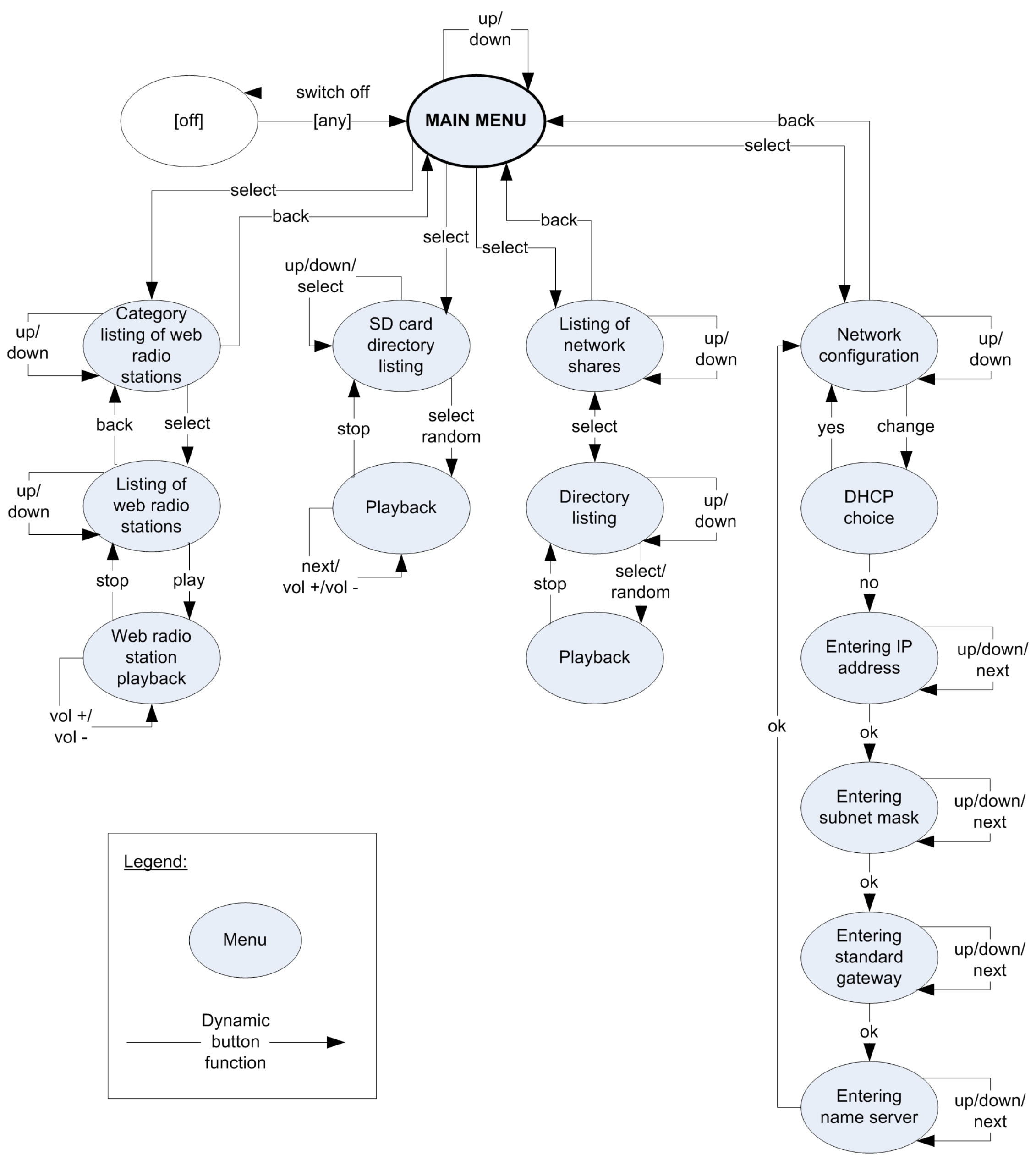


Figure 2.2: Hierarchical menu structure and dynamic button functions

2. The LCD content does not need to be attuned to touch screen operation. Hence smaller graphic items can be displayed. (Operation with a stylus is out of the question.)
3. Sophisticated configuration (like web radio stations configuration) is done by means of a web service, hence simple operation controls for the device itself are sufficient.

The number of buttons is investigated in combination with the menu structure shown in figure 2.2. Four buttons are needed to handle all of these proposed operation modes.

No. F10 – Web service for configuration
SUMMARY:
The system shall provide a web service for configuration of web radio stations and

network shares.
DESCRIPTION:
A web service will be provided by the system. Functions regarding web radio stations are: Adding and deleting categories, adding and deleting web radio stations. Functions regarding network shares: Adding and deleting network shares. Changes are stored in the system.
RATIONALE: —

No. F11 – Volume control
SUMMARY:
The system shall provide volume control during playback.
DESCRIPTION:
The volume can be decreased and increased while the system plays audio data from any source. The volume values are displayed in percent, i.e. "100" means maximum volume, "0" means mute. One step should decrease/increase the volume by a value of 5.
RATIONALE: —

No. F12 – Network configuration
SUMMARY:
The system shall provide a network configuration menu.
DESCRIPTION:
The system provides an extra menu for network configuration. An Ethernet network setup needs the IP address and the subnet mask. For accessing the Internet through a router, the standard gateway address is necessary. Furthermore, a name server[5] is required for resolving Internet domain names. Network configuration is done either by DHCP[7] or by manual step-by-step entry.
RATIONALE:
Network configuration is essential for the system. DHCP is a widespread standard.

No. F13 – Play modes
SUMMARY:
The system shall be able to play audio files in alphabetical or in random order.
DESCRIPTION:
If the system plays audio files from SD card or network shares where directory structures are possible, two play modes are supported: Standard and random mode. In standard mode, audio files in the current subdirectory are played in alphabetical order. If no more files are left, playback stops. In random mode, files in the current subdirectory are played in random mode. Playback continues until stopped by the user.
RATIONALE: —

2.2.2 Non-functional requirements

No. N1 – Usability
SUMMARY:
The system provides usability by means of ease of use.

[5]More precisely: DNS[6] server
[7]Dynamic Host Configuration Protocol

DESCRIPTION:
There are various points the system shall provide regarding usability:
1. A clearly arranged user interface shall be provided. This point will be supported by an LCD with reasonable size.
2. The system shall provide meaningful information to the user if an error occurs.
3. The system shall provide network configuration via the DHCP protocol, which is the simplest method to set up a network connection.
4. The system shall provide a web interface for advanced configuration, i.e. web radio stations and network shares configuration.

No. N2 – Performance
SUMMARY:
The system shall have reasonable response times during user interaction.
DESCRIPTION:
For user interaction it is very important that the system gives a response to button presses within 250 ms. If the requested operation cannot be completed within this interval, the system shall meanwhile display feedback to the user, i.e. a "Please wait" message.

No. N3 – Security
SUMMARY:
The system shall provide security mechanisms.
DESCRIPTION:
Since the system is part of a network, security issues must be of concern. Cable Ethernet provides a higher security level than wireless networks per se. The proposed web service needs special consideration regarding security: Both the embedded web server and the web application for processing user input have to be thoroughly tested. Moreover, the web service shall be protected by a password.

No. N4 – Network access
SUMMARY:
Network access is required.
DESCRIPTION:
For full functionality, the system requires network access for both playback of web radio stations from the Internet and of audio files from the local network. Limited functionality shall be available without network access by means of SD card playback.

2.3 System architecture

2.3.1 Architectural design

Architectural design is the description of a system in terms of its modules [Mac05]. The system will now be examined from the design perspective. For the embedded webradio device, this step is accomplished by a component-driven approach: Starting from the requirements defined in the previous section, a thorough analysis of required components is done, splitted into hardware and software parts. The results of this phase are a document listing hardware and software components and a component diagram providing information about the interaction between components. Technical information will be given accordingly to support the hardware components selection.

2.3.1.1 Hardware components

Embedded processor
The processor is the core of the whole system. The two most important aspects are: Execution speed and manifoldness of hardware interfaces. The required execution speed primarily depends on the available MP3 decoder but also influences the selection of software components, e.g. if it is possible to deploy an embedded operating system. Hardware interfaces must be available to connect all other hardware components, e.g. display, DAC, RAM, etc.

Memory
Two types of memory are required in the system:
1. RAM is mandatory for program execution on the processor and for buffering the audio stream. Again, the amount of available RAM influences selection of software components.
2. Non-volatile memory is required in the system – usually flash memory is used. This is generally important for stand-alone devices to store its firmware. Additionally, personal settings like web radio stations are stored here.
Memory devices and processors are mostly interconnected through the data/address bus.

Display
The central part of user interaction is the LCD. It should have a size of 320 x 240 pixels (standardized size). The interconnection of display and processor can either be direct or an additional display controller is required. The former method assumes that the processor itself has an appropriate interface (which usually includes at least eight data lines and several control lines with timing signals) for this kind of display.

Buttons
The system requires 5 buttons for user input, whereof one button is solely for resetting the device. So there remain 4 general-purpose buttons with dynamic function assignment. The reset button is usually connected with a dedicated input pin of the processor which triggers a reset. To connect the general-purpose buttons to the processor, there are various options:
1. Each button is connected directly to a processor input pin.
2. A serializer chip is deployed, which gets button inputs and outputs binary data to the processor in a UART-like manner.

Digital analog converter
A dual DAC for stereo audio output is required. This is the main part which influences quality of the device. Hence if possible, an enhanced chip shall be used which is specifically designed for audio output. There may be additional functionality like volume control or mute. The hardware interface must be supported by the processor. Usually, digital PCM audio data is transferred over a serial connection.

Ethernet controller
For connection to a local network, an IEEE 802.3 compatible Ethernet chip is required. It should support at least the 10BASE-T standard which allows a maximum transfer speed of 10 Mbit/s. Ethernet controllers are mostly connected to processors via the data/address bus.

SD card controller
SD cards provide two interfaces [San03]: SD bus mode and the simpler SPI mode. SPI is a common protocol used in embedded devices and hence many processors support it. Since the SD card is used only for playback of audio data, transfer speed comes second here.

Power supply
The device needs a stable power supply according to the requirements of used hardware components.

Connectors
Three hardware components listed above need connectors to the outside world:
1. A power connector.
2. Stereo audio connector: A 3.5 mm stereo jack plug or two RCA connectors (commonly known as CINCH/AV connectors) shall be deployed.
3. A network connector is required. The most common standard is an 8P8C[8] modular connector, usually referred to as *RJ45* [WWIa].

2.3.1.2 Software components

Device drivers
Device drivers are needed for all hardware components that must be dealt with:
1. Display device driver
2. Button device driver
3. DAC device driver
4. SD card device driver
5. Ethernet chip device driver, including a TCP/IP network stack
6. Flash memory device driver

Network protocol drivers
For the following network protocols software modules must exist or be implemented: SMB and NFS protocols for accessing files over the network; DHCP protocol for dynamic network configuration.

MP3 decoder/player
An MP3 decoder is required to decode audio streams. The decoded PCM audio data is transferred to the DAC device driver.

Main control application
The web radio application is the main software part that has to be implemented. It is started automatically when the device boots, manages all hardware with the aid of device drivers and steers audio decoding.

Graphics library
A library with basic graphics support is required to show information on the display. It must include font support.

Web server
To provide a web service to the user, an embedded web server is required. For dynamic generation of web pages support of the CGI[9] protocol is mandatory.

[8] 8P8C stands for "Eight positions, eight conductors".
[9] Common Gateway Interface

Web application
The configuration application allows manipulation of web radio stations and network shares by the user. It is dependent on a web server and is realized with dynamically generated web pages (CGI protocol).

Figure 2.3 presents a diagram of all components and their proposed interactions.

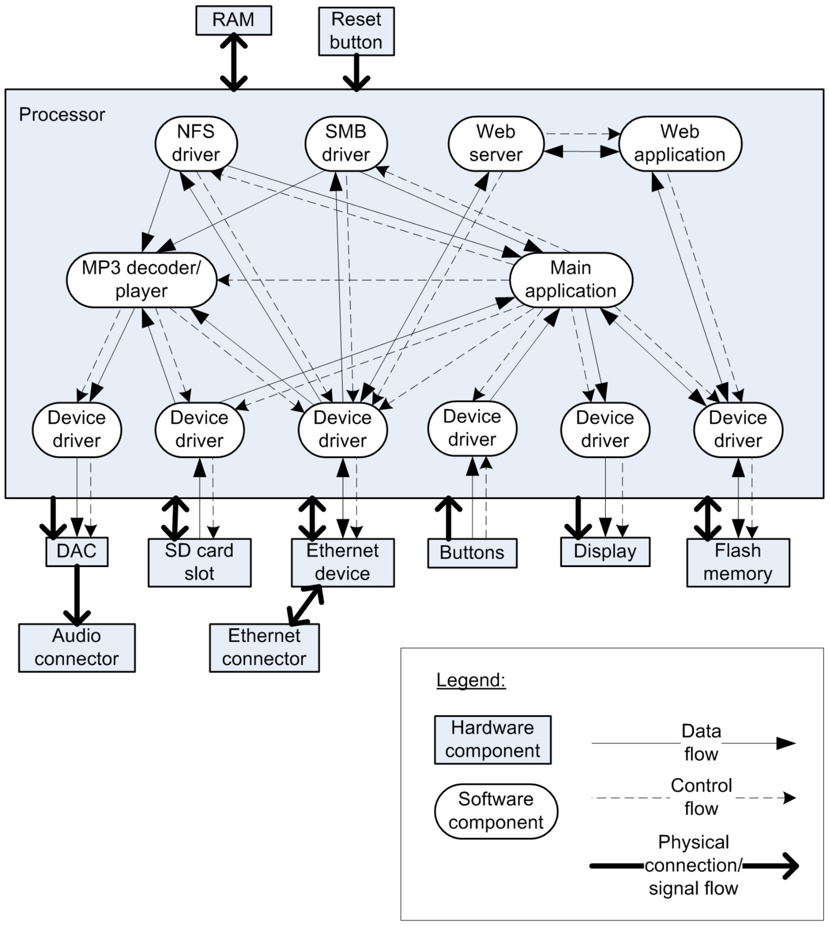


Figure 2.3: Hardware and software components and their interactions

2.3.2 Use cases

This section presents a set of detailed use cases. Use cases are text descriptions of event flows. According to [Mac05], the implementation of programs strictly follows the specifications in the use case document. Hence, use cases shall be precise enough to answer most questions a programmer will have. To be able to refer to a specific use case, each one gets assigned a unique number, preceded by the letter “U”. Each use case has a typical main flow of numbered events. There may exist one or more alternative flows. Their events are numbered by lowercase letters, preceded by the number where event flows start to diverge.

No. U1 – Power up

SUMMARY:
The user powers up the device.
PRECONDITIONS: —
MAIN FLOW:
1. User connects device with power supply.
2. Device starts up, main menu is shown on display.
3. Data previously stored to flash memory is restored.
4. Network is set up according to last setting.
NOTES: —

No. U2 – Switch on

SUMMARY:
The user switches on the device.
PRECONDITIONS:
1. Device is connected to power supply.
2. Device is in off mode, i.e. display is off.
MAIN FLOW:
1. User presses any button except reset button.
2. Device wakes up. Main menu is shown on display.
NOTES: —

No. U3 – Web radio station

SUMMARY:
The user selects a web radio station to play.
PRECONDITIONS:
1. Device is on.
2. Main menu is shown on display.
3. Device is physically connected to network.
4. Device has valid network configuration.
MAIN FLOW:
1. User selects function *web radio* with buttons.
2. Device shows list of categories on display.
3. User selects a category with buttons.
4. Device shows list of web radio stations within selected category on display.
5. User selects a web radio station with buttons.
6. Device connects to web radio station and plays audio stream. Device shows name and URL of web radio station and volume on display.
ALTERNATIVE FLOWS:
2a. No categories are stored in device. Device displays error message and returns to main menu.
4a. No web radio stations are stored in category. Device displays error message and returns to category list.
6a. Device cannot connect to web radio station. Device displays error message and returns to web radio stations list.
NOTES:
Default list of categories and web radio stations must be stored in device. Categories and web radio stations can be modified through a web interface (see use case no. U11)

No. U4 – Mass storage

SUMMARY:
Playing audio files from an SD card which is connected to the device.
PRECONDITIONS:
1. Device is on.
2. Main menu is shown on display.
MAIN FLOW:
1. User inserts SD card into device and selects function *mass storage* from main menu with buttons.
2. Device shows top directory contents (i.e. playable audio files and subdirectories) of SD card on display.
3. User selects a subdirectory with buttons as often as required.
4. Device shows current directory contents on display.
5. User selects audio file with buttons.
6. Device plays audio files in alphabetic order starting from selected one. Name of audio file and current volume are shown on display.
7. After last file is played, device returns to listing of current directory.
ALTERNATIVE FLOWS:
5a. User selects function *random* with buttons.
5b. Device repeatedly plays all audio files in current directory in random order until playback is stopped by user.
7a. User stops playback with buttons.
7b. Device stops playback and returns to listing of current directory.
NOTES: —

No. U5 – Network share

SUMMARY:
Playing audio files from a network share.
PRECONDITIONS:
1. Device is on.
2. Main menu is shown on display.
3. Device is physically connected to network.
4. Device has valid network configuration.
5. Network share is available and offers audio files.
MAIN FLOW:
1. User selects function *network share* with buttons.
2. Device shows list of network shares on display.
3. User selects a network share with buttons.
4. Device connects to network share and shows top directory content on display.
5. Remaining steps similar to use case no. 4, starting from step 3
ALTERNATIVE FLOWS:
2a. The list of network shares stored in the device is empty. Device displays error message and returns to main menu.
4a. Connection to network share fails. Device displays error message and returns to list of network shares.
NOTES:
Default list of network shares stored in device is empty. Network shares can be configured through a web interface (see use case no. U12). Supported protocols for file

exchange are SMB and NFS.

No. U6 – Volume
SUMMARY:
The user adjusts the volume while the device is in playback mode.
PRECONDITIONS:
1. Device is playing audio data from any source.
MAIN FLOW:
1. User presses button for *volume down* or *volume up*.
2. Device decreases/increases volume by 5 percent. Current volume is updated on display.
NOTES:
Maximum value for volume is 100. Minimum value for volume is 0 (mute).

No. U7 – Next song
SUMMARY:
The user skips the current audio file during playback mode.
PRECONDITIONS:
1. Device is playing audio data from network share or SD card.
MAIN FLOW:
1. User presses button for *next song*.
2. System stops playing current song and starts playing next song in alphabetic order.
ALTERNATIVE FLOWS:
2a. Current playback mode is *random*. Device stops playing current song and starts playing next song in random order.
NOTES:
In normal playback mode there is no button for *next song* if no more songs are left.

No. U8 – Main menu
SUMMARY:
The user returns to main menu.
PRECONDITIONS:
1. Device is on.
2. The device is showing any submenu or is in playback mode.
MAIN FLOW:
1. If device plays audio data, user stops it with a button.
2. Device stops playback and returns to the corresponding submenu.
3. While a submenu is shown on display, user goes up one level by a button press.
4. Device goes up to a submenu or to main menu and shows it on display.
NOTES:
A hierarchical menu structure is used. Refer to figure 2.2.

No. U9 – Network configuration
SUMMARY:
The user configures network parameters so that the device can connect to the local network and the Internet.
PRECONDITIONS:
1. Device is on.
2. Main menu is shown on display.

3. Device is physically connected to the network.
MAIN FLOW:
1. User selects function *configure network* with buttons.
2. Current configuration is shown on display (DHCP or manual configuration, IP address).
3. User selects function *change settings* with buttons.
4. Device offers manual or DHCP configuration on display.
5. User selects DHCP configuration.
6. Device gets configuration via DHCP protocol, then returns to network submenu and shows information.
ALTERNATIVE FLOWS:
5a. User selects manual configuration.
5b. Device prompts user to enter IP address.
5c. User enters IP address with buttons.
5d. Device prompts user to enter subnet mask.
5e. User enters subnet mask with buttons.
5f. Device prompts user to enter standard gateway.
5g. User enters standard gateway with buttons.
5h. Device sets up network, saves configuration, returns to network submenu and shows entered information.
6a. DHCP request fails. Device displays error message and returns to network submenu.
NOTES: —

No. U10 – Switch off
SUMMARY:
The user turns off the device.
PRECONDITIONS:
1. Device is on.
2. Main menu is shown on display.
MAIN FLOW:
1. User presses *off* button.
2. Device stores all configurations in non-volatile memory, shuts off display.
NOTES:
Configuration includes web radio stations and categories, network shares, current volume, network settings.

No. U11 – Configuration of web radio stations
SUMMARY:
The user manages web radio stations and categories via a web interface.
PRECONDITIONS:
1. Device is powered up.
2. Device is physically connected to network.
3. Device has valid network configuration.
4. PC with web browser is available and connected to same network as device.
MAIN FLOW:
1. User enters IP address of device into a web browser on the PC.
2. Main configuration web site is loaded from device.
3. User selects *configure web radio stations* option.
4. Web site with all web radio stations, grouped into categories, is loaded from device.

5. User adds a category by entering name in a form field and clicking corresponding *add* button.
6. Device sends updated web site with new category. The category is empty.
7. User adds a web radio station to a category by entering URL and name in form fields and clicking corresponding *add* button.
8. Device sends updated web site with new web radio station.
9. User deletes a category (including web radio stations) by clicking corresponding *delete* button.
10. Device sends updated web site without deleted category.
11. User deletes a web radio station by clicking corresponding *delete* button.
12. Device sends updated web site without deleted web radio station.
13. User clicks *save settings* button to store changes on device.
14. Device sends web site asking for password.
15. User enters correct password.
16. Device saves settings to non-volatile memory and sends main web page.

ALTERNATIVE FLOWS:
1a. User does not know current IP address of device.
1b. User reads out IP address from network configuration submenu (see use case no. 9).
15a. User enters wrong password.
15b. Device sends web site again asking for password. If wrong password is entered 5 times, device discards user's changes and sends main web page.

NOTES:
Default list of web radio stations and categories must be provided. Default password must be provided.

No. U12 – Configuration of network shares

SUMMARY:
The user manages network shares via a web interface.

PRECONDITIONS:
1. Device is powered up.
2. Device is physically connected to network.
3. Device has valid network configuration.
4. PC with web browser is available and connected to same network as device.

MAIN FLOW:
1. User enters IP address of device into a webbrowser on the PC.
2. Main configuration web site is loaded from device.
3. User selects *configure shares* option.
4. Web site with network shares is loaded from device. The list may be empty.
5. User adds a network share by entering URL and name in form fields and clicking the *add* button.
6. Device sends updated web site with added network share.
7. User deletes a share by clicking corresponding *delete* button.
8. Device sends updated web site without deleted web radio station.
9. User clicks *save settings* button to store changes on device.
10. Device sends web site asking for password.
11. User enters correct password.
12. Device saves settings to non-volatile memory and sends main web page.

ALTERNATIVE FLOWS:
1a. User does not know current IP address of device.
1b. User reads out IP address from network configuration submenu (see use case no. 9).
11a. User enters wrong password.
11b. Device sends web site again asking for password. If wrong password is entered five times, device discards user's changes and sends main web page.
NOTES:
Form of URL determines protocol: SMB URLs start with "`//`", NFS URLs do not.

Chapter 3

Target platform composition

This chapter describes the process of building up the target platform. Hardware decisions have to come at first here, but also software-related issues must be considered. Regarding the hardware side, it was drawn on available components from the Institute of Computer Technology and the company *Bluetechnix*. Processors from Analog Devices based on the Blackfin architecture are often used, and a lot of hardware extension modules which were specifically designed for development and evaluation of embedded systems are available. These products are also commercially distributed with the brand name “Bluetechnix tinyboards”. More information, data sheets etc. can be obtained on the web page `http://www.bluetechnix.at`.

Determination of hardware components is certainly not made without respect to available software for a specific processor and peripherals. The progress of this project will rely on already available software parts or programs to a rather large extent, so the more software can be relied on, the less has to be implemented from scratch.

3.1 Hardware components

3.1.1 Core module CM-BF537E V1.1

The core module CM-BF537E comprises an ADSP-BF537 Blackfin processor from ADI, 32 MByte of SDRAM and 4 MByte of flash memory. In addition, a chip for the Ethernet physical layer is mounted, because the processor integrates an Ethernet controller. With the aid of two 60-pin expansion connectors it can be connected to a variety of extension modules. By building upon this core module, three hardware components important for the web radio device have been found. The hardware user manual for the core module is available from [WBLj]. Figure 3.1 shows its top and bottom sides.
Starting with the processor, some detailed information about the comprised components will be given now.

ADSP-BF537 processor
The Blackfin architecture offers both microprocessor features and DSP capabilities. The former is achieved by a 32-bit RISC[1] architecture, a basic MMU[2] which allows memory

[1]Reduced Instruction Set Computer
[2]Memory Management Unit

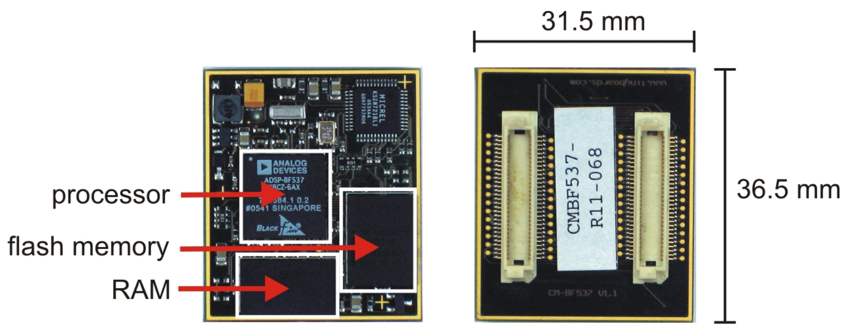


Figure 3.1: The CM-BF537E core module

protection, data and instruction caches, and support for a variety of hardware peripherals. Moreover, it offers different modes of operation: In supervisor mode all resources are available, whereas in user mode access to some of them is restricted. This is accomplished via the MMU and provides basic functionality for various embedded operating systems.
To provide DSP functionality, the Blackfin architecture is based on a SIMD[3] architecture typical for DSPs, comprising two hardware multiply-accumulates, two ALUs, and a barrel shifter. Up to three instructions per clock cycle can be executed. Moreover, some extensions, e.g. for video and image processing, are included in the instruction set.

The ADSP-BF537 processor supports a maximum speed of 600 MHz. It comprises a total on-chip memory (L1 memory) of 132 kbyte: 64 kbyte instruction memory, whereof 16 kbyte can be used as instruction cache; another 64 kbyte data memory, whereof 32 kbyte can serve as data cache; 4 kbyte of scratchpad memory. External SDRAM is supported by a PC-133 compliant controller which allows up to 512 MByte of SDRAM being interfaced. A 32-bit address space is used for addressing memory as well as I/O devices. Furthermore, a DMA controller is available both for memory and I/O transfers. All peripheral functions are tightly integrated with the DMA by means of DMA channels with fixed priorities. More information can be found in the datasheet [Ana06] and the hardware reference manual [Ana05].

The processor's key features regarding this project are summarized now:

- Execution speed: Due to its maximum speed it is adequate for fulfilling the web radio device's needs and additionally it provides space for later enhancements where more processor power is needed.

- A parallel peripheral interface (PPI) is available, with a dedicated clock input, 16 data pins and 3 frame synchronization pins. DMA can be used for transfers. Several general-purpose modes make the PPI suitable for a wide variety of applications. With the web radio device, PPI is used to transfer data to the LCD (see section 3.1.3).

- It offers eight 32-bit general purpose timers with dedicated output pins. PWM timer signals are mandatory for generating clock signals for an LCD device and hence will be used with the web radio device (see section 3.1.3). The timer units are able to be clocked from various sources and are tightly integrated with the parallel peripheral interface (PPI).

[3]Single Instruction Multiple Data

- It offers two synchronous serial ports (SPORTs) with bidirectional operation, adjustable word length, frame synchronization support and multichannel capability. SPORTs are commonly used to transfer digital data to DACs, so one SPORT will be used to output PCM audio signals (see section 3.1.5).
- The Serial Peripheral Interface Bus (SPI) is supported by the Blackfin processor. This is a simple serial data link with full duplex support that was originally invented by Motorola. For more information on SPI refer to [Fre03]. With the web radio device, SPI will be used primarily to communicate with an SD card (see section 3.1.2). Since SPI data can be transferred with DMA, the required data rate for an SD card can be reached by the Blackfin processor. Beside that, SPI will be used for configuration of the DAC (see section 3.1.5).
- It contains an on-board embedded Fast Ethernet MAC[4] peripheral which supports operation in 10BASE-T and 100BASE-T modes and is compliant to the IEEE 802.3 standard. The MAC needs an external Ethernet PHY (physical layer of the OSI model) chip, which is integrated onto the CM-BF537E core module. Network access is essential for the web radio device.
- With the aid of general purpose inputs/outputs (GPIOs) the buttons for user operation are linked to the processor.

Main memory
The core module comprises an SDRAM chip with a size of 32 Mbyte. The maximum clock speed is 133 MHz. It is connected to the processor via the data and address bus.

Flash memory
A non-volatile CFI[5]-compatible flash memory is integrated onto the core module. Though its size of 4 Mbyte is not fully addressable, a GPIO pin is used to switch between lower and upper 2 Mbyte banks.

3.1.2 Blackfin Evaluation Board EVAL-BF5xx V3.1

The core module introduced in section 3.1.1 is plugged into the evaluation board EVAL-BF5xx by means of an expansion slot. With the evaluation board and the core module, it is able to set up a basic embedded environment which can be acted upon by a connection to a personal computer. The evaluation board comprises an RJ45 Ethernet plug, an SD card slot, a UART-to-USB converter (CP2101), a JTAG plug, two expansion connectors for various extension boards, and a voltage regulator together with a power connector. Figure 3.2 shows a picture, the hardware user manual and schematic are available from [WBLj].

Two mandatory hardware components are included onto the evaluation board regarding the web radio device: The Ethernet plug and the SD card slot. The UART-to-USB converter connects the first UART port of the ADSP-BF537 processor to a USB port of a personal computer. This connection will be used to communicate with the device during application development.

[4]Media Access Controller
[5]Common Flash Interface

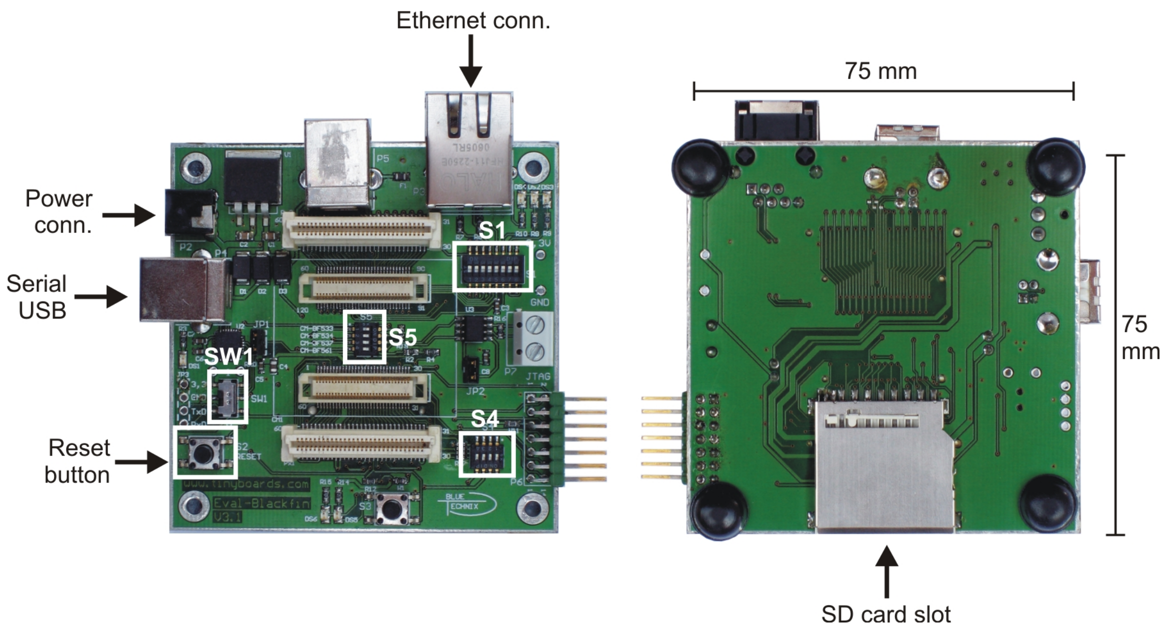


Figure 3.2: The EVAL-BF5xx evaluation board

3.1.3 Blackfin Extender Board EXT-BF5xx-Camera V1.0

The camera extender board can be connected with the evaluation board by means of its expansion connectors. There are practically no microchips on it, it just provides connectors for one LCD and two CMOS[6] cameras. The LCD device is interconnected with the PPI of the Blackfin processor.

The extender board is used within this project to attach the LCD. Figure 3.3 has a picture of the board. The hardware user manual and the schematic are available from [WBLj].

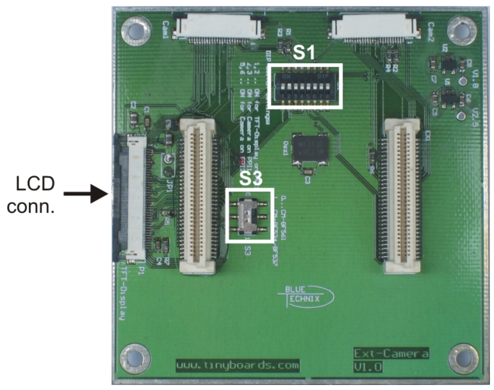


Figure 3.3: The EXT-BF5xx-Camera extension board

[6]Complementary metal-oxide-semiconductor

3.1.4 Hitachi color display TX09D70VM1CDA

The color TFT-LCD manufactured by Hitachi has an effective display area of 53 x 71 mm and a resolution of 240 x 320 pixels. It is able to display $2^{18} = 262144$ different colors, i.e. 6 bit per color. The display is compatible with the LCD connector of the camera extender board. The connection is done by a ribbon cable. Refer to the display datasheet [Kao06] for more information.

The ability of displaying colors is a “nice-to-have” feature for the web radio device. Though this is not mandatory to display the proposed content, it allows for various future enhancements, e.g. viewing pictures. Due to the fact that the Blackfin processor’s PPI has a maximum of 16 data lines, only $2^{16} = 65536$ colors can be differentiated.

3.1.5 Blackfin Extender Board EXT-BF5xx-Audio V0.1

The audio extender board is compatible to other Blackfin extender boards and can be connected through the expansion connectors. It comprises the powerful single-chip DAC/ADC AD1836A from Analog Devices, necessary amplifier logic and 5 dual-RCA connectors. Because the AD1836A chip and the Blackfin processor are developed by the same vendor, optimal compatibility is guaranteed. Figure 3.4 shows a picture of the extender board.

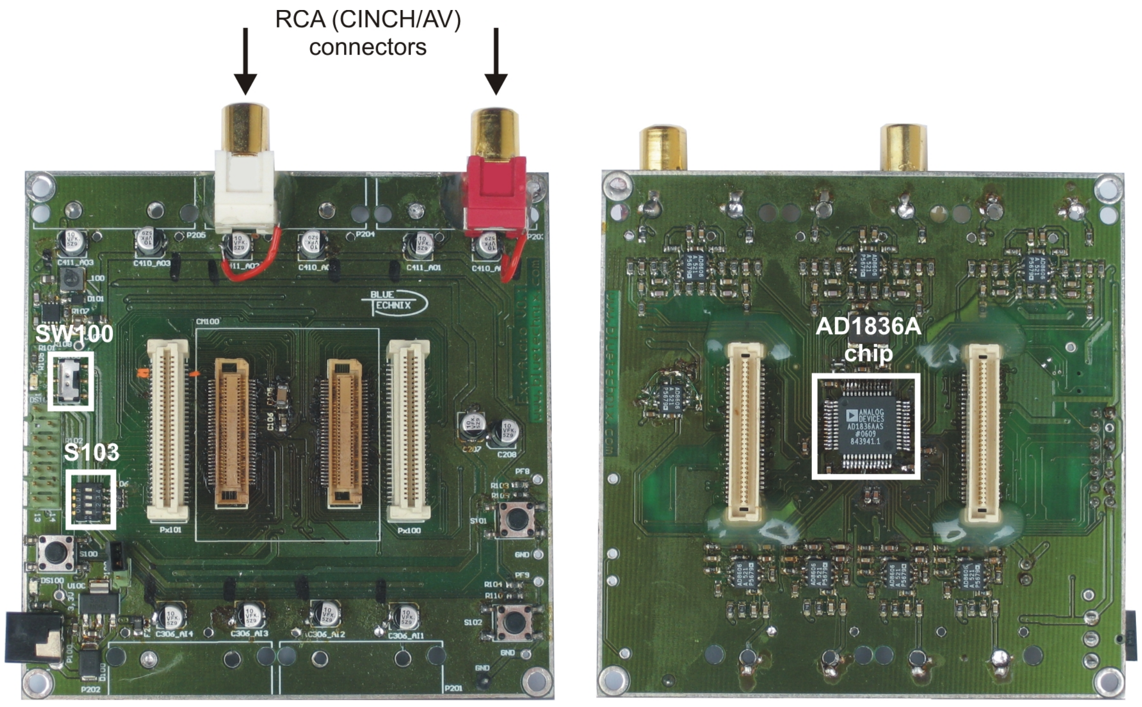


Figure 3.4: The EXT-BF5xx-Audio extension board (only two RCA connectors are mounted)

These are the main features of the AD1836A:

- Specifically designed for audio applications
- Supports up to 24-bit data samples

- Supported sample rate of up to 96 kHz
- 3 stereo DACs
- 2 stereo ADCs
- On-chip volume control
- Mute function

The exchange of PCM audio data with the Blackfin processor takes place over an SPORT. Configuration of the AD1836A is done over the SPI interface. More information can be found in the datasheet [Ana03].

The audio extender board is chosen for this project due to several reasons: First, it is compatible with the chosen core module and the other extender boards. Second, the AD1836A is a high-performance codec chip and offers good audio quality. Third, the chip's multichannel functionality allows for later enhancements of audio functions, e.g. audio recording or AC-3 codec support (usually known as "Dolby Digital 5.1").

3.1.6 Input buttons

The 4 required buttons must be connected to GPIO pins. They are mounted on an EXT-BF5xx-Experimental Extender Board, which is compatible to other Blackfin extender boards and makes all pins of the extension connectors available as solderable pads. Because only 3 GPIO pins of the Blackfin processor are unused by now, three buttons are connected to one GPIO pin respectively and the fourth one is connected to all three. Figure 3.5 shows the buttons' schematic and a picture of the experimental board with the mounted buttons.

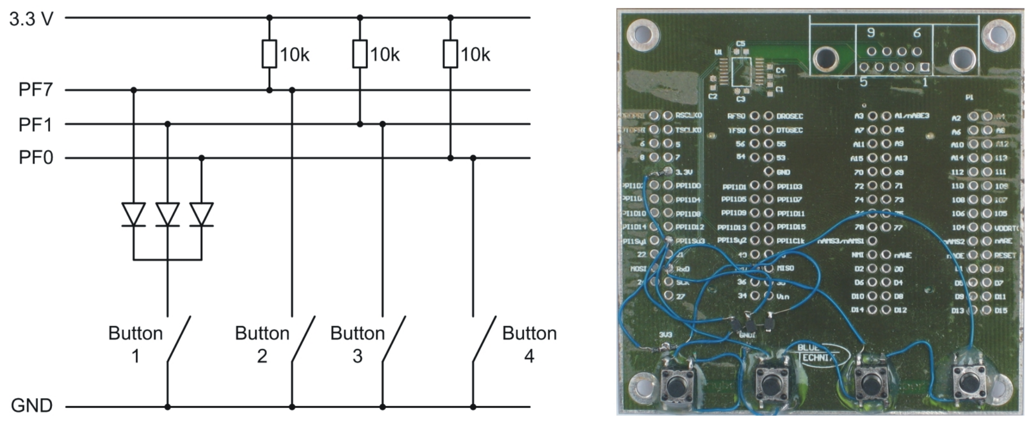


Figure 3.5: Buttons schematic and picture

3.1.7 Composition

The listed hardware components are plugged together by means of compatible expansion connectors. Table 3.1 informs about the proposed setup. The evaluation board has to be the lowermost, because it only can be expanded on the upper side. The reverse is valid for the experimental extender board with the mounted buttons.

During system development, some modifications to hardware components were made. These are exactly described in chapter 7.

Extender Board EXT-BF5xx-Experimental	
Extender Board EXT-BF5xx-Audio	
Extender Board EXT-BF5xx-Camera	Hitachi TFT-LCD
Evaluation Board EVAL-BF5xx	Core module CM-BF537E

Table 3.1: Proposed composition of hardware components

3.2 Hardware configuration

The hardware components which were introduced in the previous section do have some configuration options, since they are able to be combined with different core modules and various functions can be enabled or disabled. Configuration is done by setting single or DIP switches and mostly depends on the hardware components itself and their combination. The purpose of this section is to provide a central reference of all hardware settings. This increases traceability and renders consulting many different manuals unnecessary.

Evaluation board
Switch SW1:
This switch is in position "1", so that Rx0 and Tx0 lines of the CM-BF537E are connected to the CP2101 UART-to-USB converter. So a development workstation can be connected to the processor. This setup is only used during system development, since with the finished device the UART pins act as button inputs.
DIP switch S1:
Setting all its switches to "ON" is mandatory, because by means of these the on-board Ethernet controller of the core module is attached to the Ethernet connector.
DIP switch S4:
Via these 4 switches, the boot mode of the Blackfin processor is defined. This is purely software-dependent. For the uClinux bootloader, all switches must be set to "OFF" since it uses the "Execute from external memory" mode. If the VDK from Analog Devices is used which uses the "Boot from external flash memory" mode, switch 1 has to be "ON", others "OFF".
DIP switch S5:
The 4 switches influence the SPI chip select line of the SD card slot and the CAN interface. They have to be set to "1000", i.e. switch 1 is "ON". This connects the chip select line to SPI_CS1 pin of the core module and disconnects the CAN connector because it is not needed here.

Camera extender board
DIP switch S1:
Only switch 2 is in "ON" position, the others are off. This connects the TMR5 timer output pin of the core module to the PPI1Clk input pin, which is used for the LCD clock generation.

However, this connection is subject to a hardware workaround due to a pin conflict, which is explained in detail in section 7.1.
Switch S3:
This switch must be set to position “1” to interconnect the LCD PWM pin (for the backlight) and the TMR6 timer output pin of the core module.

Audio extender board
Switch SW100:
This must be set to position “0” (towards the S103 DIP switch) to connect the SPI chip select line of the AD1636A audio chip to the SPI_CS5 pin of the Blackfin processor.
DIP switch S103:
It is connected similar to DIP switch S4 of the evaluation board, so it is sensible to set all switches to “OFF”.

Appendix A includes a table where the utilization of each core module pin is listed with this hardware setup.

3.3 Software issues

In the previous section a fast embedded processor was selected with the Blackfin ADSP-BF537. This enables the deployment of a kernel or operating system in this project. Because the web radio firmware contains several independent software components (main application, web server, etc.), it is necessary to build upon a kernel which offers basic multitasking functionality. To come to a decision, two popular ones, namely the *BLACKSheep VDK* and the *uClinux distribution* are compared here. The crucial factor is the number of existing software components according to figure 2.3.

The *BLACKSheep VDK* was developed at the Institute of Computer Technology. It is based on the VDK (VisualDSP++ Kernel) from ADI and was enhanced by several hardware device drivers and tools. The advantage of this solution would lie in the VisualDSP++ IDE including the VDK and the CROSSCORE toolchain which provides excellent emulation debugging. [WAN] provides more information. All BLACKSheep add-ons were developed with it, and hence it is the primary development tool for the Blackfin architecture at the Institute of Computer Technology.

Another possibility is the *uClinux distribution*, which comprises the uClinux kernel and GNU software/tools. The “Embedded Linux/Microcontroller Project”, short “uClinux”, is a Linux derivative which is adapted to the needs of embedded microprocessors [WUCc]. A port to the Blackfin architecture is available, including the GCC[7] toolchain common in the Linux world. It is taken into account for this project because ADI officially supports the Blackfin uClinux distribution and contributes to it by developing applications, porting drivers and the like. The advantages of this solution are first that both the uClinux kernel and GNU software are open source software[8] [WOP], and second that Linux is a familiar computing environment whose availability on embedded systems makes it easy to build an embedded application.

Table 3.2 is the comparison result of BLACKSheep versus uClinux with regard to available software components. Based on the facts collected in this table the decision is made in favor

[7]GNU Compiler Collection
[8]For the license of specific software parts, refer to the soure code.

	uClinux	BLACKSheep
MP3 decoder/player	YES (mp3play program)	no
CGI-capable web server	YES (boa web server)	YES (GoAhead web server port)
NFS file system support (client)	YES (kernel support and mount program)	no
SMB file system support (client)	YES (kernel support and smbmount program)	no
AD1836A audio chip device driver	YES	YES
SD card device driver	yes (unstable)	YES
VFAT file system support	YES (kernel support and mount program)	YES (kernel support and mount function)
ADSP-BF537 Ethernet MAC device driver	YES	YES
DHCP protocol support	YES	YES (statically compiled in)
Hitachi TFT-LCD device driver	no	YES
Flash memory device driver	yes (generic Intel/Sharp CFI driver – only 2 Mbytes addressable)	YES

Table 3.2: Comparison of uClinux and the BLACKSheep VDK (as of Sept. 2006)

of the uClinux distribution. The most important part, an MP3 decoder/player, is available with it, together with support for DHCP, SMB and NFS. The device driver for the Hitachi TFT-LCD has to be ported to uClinux, therefore the source code of the BLACKSheep driver is available to the author. Eventually the author's familiarity with GNU/Linux contributes to this decision too. Implementing the proposed system onto an embedded Linux operating system will be an interesting challenge.

Chapter 4

Starting with uClinux

In the preceding chapter it was decided to build the web radio application upon the uClinux operating system. So, this chapter provides a general description of uClinux and the outline of the employed development environment, followed by three sections dedicated to the main software packages which are necessary to bring uClinux onto the target environment: The GNU toolchain for Blackfin will be installed which is mandatory for cross-compiling, the bootloader for uClinux called *Das U-Boot* will be compiled and stored on the target, and a basic image of the uClinux distribution itself, which concludes a Linux kernel and a wide range of applications, is created. At chapter end, uClinux will be executed on the target device.

4.1 Introduction

The name *uClinux* stands for "microcontroller Linux". It is a port of the Linux kernel specifically designed for devices without an MMU [WUCc] and is hence typically employed in embedded systems. Because of this limitation, a standard Linux kernel cannot be deployed, which results in two fundamental differences compared to the uClinux kernel: In the latter, there is no support for virtual memory addresses and there is no memory protection. A uClinux port is also available for the Blackfin processor.

Memory considerations

Actually, in section 3.1.1 it was stated that the Blackfin processor comprises an MMU. This is true, but the MMU's functionality is limited: Though it protects kernel space from user space applications, it does not protect applications from writing to another application's memory space. Neither is a virtual memory model available, which has important consequences as follow [WBLf]:

- Each process has to be loaded into the memory using a contiguous memory segment. The stack space is allocated at the end of the data segment, so if the stack grows too large it overwrites static data. To take care of this problem, a compiler option for stack checking is available.

- Swapping out data is not available. However, most embedded systems do not have storage media for swapping at all. Available memory is therefore strictly limited to the physically available RAM.
- There is no `fork()`[1] system call available in uClinux. Instead, `vfork()` is implemented, which suspends the parent's execution until the child process calls `exit()` or `execve()`.
- The `malloc()` system call to allocate memory is implemented as a wrapper to `mmap()`.
- Processes which are loaded by the kernel must be able to run independently of their position in memory. One way to accomplish this is relocation, i.e. fixing up address references in a program when loaded into RAM. This issue is related to the BFLT (Binary Flat) executable format which is by default used with uClinux. It is a simple and leightweight format and supports relocation. PIC (Position Independent Code) is also supported by uClinux.

More information about memory management, executable file formats and steps involved in building executables for uClinux can be found in [Tam05].

uClibc

Another feature which makes uClinux suitable for embedded devices is the use of the *uClibc* library. It is a C library for embedded Linux and it is much smaller than the GNU C library *glibc*, though nearly all applications written for glibc also run on uClibc. It even supports shared libraries and threading [WUCb]. uClibc was originally created to support uClinux, but nowadays, it also works fine on normal Linux systems.

uClinux distribution

When someone speaks about *uClinux*, mostly the whole uClinux distribution is meant. It includes not only the kernel itself, but a vast collection of GNU tools and programs commonly available in the Linux world. A rather complete list is available at [WBLi]. Outstanding applications are briefly listed here:

Networking: Many protocols are supported at client and/or server side: boa (web server), dropbear (small SSH2[2] server and client), ftp (FTP client), portmap (port to RPC[3] program number mapper, used also for mounting of NFSs), smbmount (mount program for SMB filesystems), telnetd (telnet deamon), ifconfig (network interface configuration), dhcpcd (DHCP client), etc.

Audio: mixer, mp3play (MP3 player), tone (simple tone generator), vrec (audio recorder)

System tools: Nearly all of typical Linux commands (for file manipulation, kernel control, user management, etc.) are also available with uClinux. This is accomplished with the BusyBox package, which combines tiny versions of many common Unix utilities into a single small executable. It has been written with optimization of binary size and run-time memory usage in mind. Furthermore, BusyBox is modular, i.e. individual commands or features can be included or excluded at compile time [WBU].

[1]The `fork()` system call clones a process to create a child process.
[2]Secure Shell
[3]Remote Procedure Call

Help resources

The first place to look for help is the wiki at `http://docs.blackfin.uclinux.org`. It provides an excellent introduction chapter to uClinux with which even users unfamiliar with Linux may put up. Topics regarding specific device drivers, kernel configuration manuals, the U-Boot bootloader, the Blackfin processor family, etc. are available there.
For help on specific problems, the forums at the main Blackfin/uClinux web site (`http://blackfin.uclinux.org`) are the right place. This site is subdivided into projects, e.g. uClinux distribution, U-Boot bootloader, GNU toolchain, and so on. The community is quite large, so one usually gets an answer within 1 day when asking for help in the forums.

4.2 Development environment

A typical environment for uClinux basically consists of the following elements [WBLc]:

- A development host: This is essentially a Linux box, where software for the target device is developed and cross-compiled. With the web radio project, a PC with the Linux operating system is used. Naturally, it is equipped with an Ethernet card and some USB ports.
- The embedded target device. Here, development was started just with the evaluation board and the plugged-in core module. Further components were added as needed.
- Connections between development host and target device: They are used to load software onto the target system and interact with programs running on the target system. With this project, a serial connection (over a USB cable) is used as well as an Ethernet connection via the local network.

Figure 4.1 gives a graphical overview of the development environment which was used for the embedded web radio.

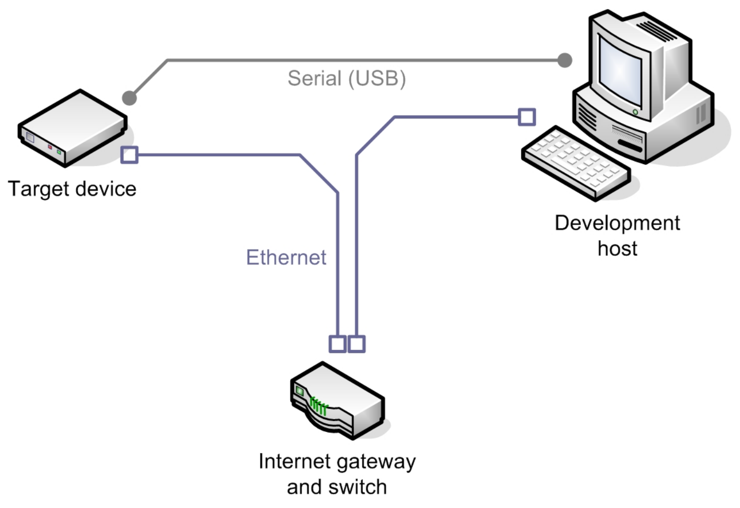


Figure 4.1: Outline of the development environment

uClinux terminal

uClinux offers a terminal on the Blackfin's first UART which is the main communication facility at beginning of development (later, e.g. *Telnet* may be used). As mentioned above, a serial connection from the workstation to the Blackfin processor is made via a USB cable. Therefore, the evaluation board comprises the USB-to-UART conversion chip *CP2101* that is connected to UART0 of the Blackfin. When the physical connection is made, a Linux workstation detects this chip[4] and creates the device file `/dev/ttyUSB0` or a similar one. This device file is used to connect to the uClinux terminal via a terminal emulator program.

Here, *Kermit* [WCO] is used for this purpose. It is available in most Linux distributions. The installation package is named *ckermit*. Kermit has to be configured to work properly with uClinux. Therefore, the file `~/.kermrc` in the home directory of the workstation must consist of the following lines:

```
line /dev/ttyUSB0
set speed 115200
set carrier-watch off
set flow-control none
set prefixing all
set parity none
set stop-bits 1
set modem none
set file type bin
set file name lit
```

Kermit is started by typing `kermit` at the Linux console. To connect to the target, the `connect` command must be entered afterwards.

Another possibility to gain access to the target is using *Telnet*. Therefore, the network on the target must be set up beforehand and uClinux must be compiled with the Telnet daemon. How to compile uClinux will be explained in section 4.5.

TFTP server

Although it is possible to transfer images of U-Boot and uClinux via the serial connection, this is not recommended because it is very slow. Instead, a TFTP[5] server is installed on the workstation and the Ethernet connection is used in exchange.

Here, the *atftp* TFTP server is used on the workstation. It is started with the command `atftpd --daemon --user nobody --group nobody`. By default, the directory `/tftpboot/` is used as server root. So, to transfer e.g. a uClinux image onto the target, it has to be copied to this location to be accessible in U-Boot via TFTP.

NFS server

To exchange files easily between target and workstation, a directory on the latter is exported via the NFS protocol which will later be mounted in uClinux on the target device.

[4]It is assumed that the device driver for the CP2101 chip is compiled into the workstation's Linux kernel.

[5]Trivial File Transfer Protocol. It is a very simple file transfer protocol with basic functionality of FTP (File Transfer Protocol). It is often used for booting small devices.

Installing an NFS server is very much dependent on the Linux distribution. However, exporting a directory via NFS is similar on nearly all Linux machines – the following line is appended to the `/etc/exports` file[6] on the workstation:

```
/home/temp *(rw,sync)
```

This denotes that the directory is exported to all computers (`*`) and both read and write access is granted (`rw`).

4.3 Toolchain

The software for the target device, i.e. U-Boot and uClinux, will be compiled on the development workstation. Due to the fact that the workstation and the target have different processor architectures, software for the Blackfin must be cross-compiled. Therefore, a dedicated toolchain is required on the workstation. The freely available GNU toolchain is chosen here because it is provided with uClinux and tightly integrated, and besides that it is the most common one in the Linux world.

The GNU toolchain is a combination of multiple projects:

GCC: The GCC (GNU compiler collection) is a set of several compilers for different programming languages. The Blackfin port currently supports C and C++ [WBLe].

Binutils: The GNU Binutils are a collection of binary tools. The most well-known parts of it are the GNU linker *ld* and the GNU assembler *as*. Moreover, the *objcopy* tool will be used later to copy and translate object files.

Debugger: The GNU debugger *gdb* is a symbolic debugger and is the most important debugging tool for any Linux system.

The Blackfin port of the GNU toolchain is available from the *GNU toolchain* subproject at [WBLa]. Two possiblities exist for installing the toolchain: First, a pre-compiled toolchain can be downloaded. This is the fastest method, since no compiling is necessary. However, those are only available once per release. Second, the source code of a toolchain can be downloaded and compiled by oneself. The source code of specific releases is available along with the pre-built toolchains, but the latest snapshot can only be checked out from the CVS repository. Latter has the advantage that one gets the most actual version of the toolchain as well as of uClinux.

This proposition is explained by the fact that toolchain and uClinux are closely related: For compilation of the toolchain, the sources of both the toolchain and the uClinux kernel are required; latter because userspace kernel headers are needed. Thereby, it is important that the dates of the kernel and toolchain sources are close to each other [WBLh].

[6]See the Linux manpage `exports(5)` for details, i.e. enter `man 5 exports` on the console.

Getting the source code

At development start it has been decided to fetch the latest sources of the uClinux distribution and the GNU toolchain from the CVS repository[7]. Two versions of the CVS toolchain have been used during the project: The first with date Sept. 27, 2006 and a later one with date Dec. 12, 2006.
Checking out abovementioned sources from the CVS repository is accomplished by the following commands:

```
cvs -z3 -d :pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/gcc3 \
  checkout toolchain
cvs -z3 -d :pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/uclinux533 \
  checkout uClinux
```

Furthermore, release 2006R2 of the U-Boot is being downloaded from [WBLa]. The availability of its source code enables building the *mkimage* tool along with the toolchain which allows creating compressed uClinux images. These are small enough to be able to be stored in the flash memory of the target. The downloaded file `u-boot_1.1.3.tar.bz2` has to be unpacked using the command

```
tar -xvjf u-boot_1.1.3.tar.bz2
```

Compiling and installing

Compiling the GNU toolchain is very simple because a build script is provided with the toolchain source code. It lives in the `buildscript/` subdirectory. From there, it is called that way

```
./BuildToolChain -s /home/harald/checkouts/toolchain \
    -k /home/harald/checkouts/uClinux/uClinux-dist/linux-2.6.x \
    -u /home/harald/sources/uboot/u-boot_1.1.3 \
    -b /home/harald/temp/ \
    -o /home/harald/toolchain/bfin
```

with numerous arguments where `-s` is the toolchain source directory, `-k` denotes the uClinux kernel source directory, `-u` is the U-Boot source directory, `-b` is the build directory (can be deleted afterwards) and `-o` is the output directory.

Building the toolchain takes awhile. First, the *bfin-elf* toolchain is built, which is used to compile stand-alone applications like the U-Boot. Second, the uClibc library is built with the bfin-elf toolchain, and third the *bfin-uclinux* toolchain with uClibc support is compiled. Latter is used to build the uClinux kernel and kernel applications in the BFLT executable format. However, the user is not concerned with selection of the right toolchain because both U-Boot and uClinux do this automatically.

After compilation of the Blackfin toolchain the *PATH* environment variable on the workstation has to be modified to include the toolchain executables so that these can be found independently of the working directory. This is best done by appending the following line to the `~/.bashrc` file in the home directory:

```
export PATH=$PATH:/home/harald/toolchain/bfin-elf/bin:\
/home/harald/toolchain/bfin-uclinux/bin
```

[7]At the time this thesis was finished, the version control system of the Blackfin/uClinux project has been changed from CVS to Subversion (SVN). Hence the checkout commands slightly changed since then.

4.4 Bootloader “Das U-Boot”

The need for a bootloader is caused by the fact that most processors can only execute code from predetermined sources at startup, e.g. from memory. To enhance boot methods, a boot loader is needed that itself lives in the ROM memory of the target and provides more sophisticated functionality. To quote an example, on the Blackfin coremodule it is only possible to store uClinux images in flash memory in a compressed format. The bootloader, in that case the U-Boot, is able to uncompress and start the image and hence uClinux.

4.4.1 Overview

The standard bootloader for the Blackfin/uClinux operating system is the *U-Boot*, which is a Free Software project. The term “U-Boot” stands for “universal bootloader”. According to [Yag03], it is the richest, most flexible, and most actively developed open source bootloader that is available. It evolved out of two projects that strived for the same goal but for different architectures: *PPCBoot* and *ARMBoot*. At the time of this writing, a great many different boards and various processor achitectures are supported by U-Boot: The release that was downloaded in section 4.3 contains 183 board-specific subdirectories and 30 CPU[8]-specific ones. The development of U-Boot is closely related to Linux, and special provision has been made to support booting of Linux images [WDE].

An overview of U-Boot’s functionality shall be given here:

Command line U-Boot provides a command line to the user, simpler but similar to uClinux. Many commands are available for booting, memory programming and examination, network configuration, etc. The command list can be retrieved by entering `help` at the command line.

Loading files Several loading commands allow for different retrieval of (image) files. With this project, `loadb` (for loading over the serial line) and `tftp` (for loading a file from a TFTP server) are most often used.

Booting Several commands support booting of different images. `bootm` is used to boot compressed uClinux images (out of RAM or ROM), whereas `bootelf` boots uncompressed ELF images which are usually stored in RAM due to their size. Furthermore, U-Boot supports autoboot which automatically boots the system after a specified period of time.

Networking U-Boot contains drivers for network devices, among others for the on-chip Ethernet MAC of the ADSP-BF537 processor. It supports common protocols like TFTP and DHCP. Configuration of the Ethernet MAC (media access control) address is also done via U-Boot.

Flash programming U-Boot is the first choice for writing application images to flash memory. Unlike uClinux, the U-Boot for the CM-BF537E core module contains a patch that supports full 4 MByte of flash memory. For programming, the command `cp.b` (copy memory) is used, `cmp.b` (compare memory) allows verifying written data.
This way, U-Boot allows even upgrading itself with a new version.

[8]Central Processing Unit

Environment variables These variables contain customizable information for the target hardware, like IP address, Ethernet MAC address, uClinux kernel boot parameters and default boot command. Variables are displayed with the command `printenv`, changed or added with `setenv`, and via `saveenv` they are written to flash memory.

4.4.2 Building U-Boot from source

Compiling the U-Boot is rather simple. Optionally, configuration of U-Boot can be customized prior by editing the file `u-boot_1.1.3/include/configs/cm-bf537e.h`. All default environment variables are defined therein and can be changed, but this is done more conveniently with the U-Boot command line. Three important options can only be changed before compiling: `CONFIG_VCO_MULT`, `CONFIG_CCLK_DIV` and `CONFIG_SCLK_DIV` which configure the core clock (for the Blackfin processor) and the system clock (for peripherals like SDRAM) according to the following formulas:

$$\textit{core clock} = 25 \cdot \frac{\texttt{CONFIG\_VCO\_MULT}}{\texttt{CONFIG\_CCLK\_DIV}}\ \text{MHz}$$

$$\textit{system clock} = 25 \cdot \frac{\texttt{CONFIG\_VCO\_MULT}}{\texttt{CONFIG\_SCLK\_DIV}}\ \text{MHz}$$

With the default values a core clock of 500 MHz (maximum: 600 MHz) and a system clock of 100 MHz (maximum for SDRAM: 133 MHz [Blu06]) are set.

To configure the U-Boot source code for the CM-BF537E core module, the command

```
make cm-bf537e_config
```

has to be entered in the `u-boot_1.1.3/` directory. Then, the simple

```
make
```

command starts compilation of the U-Boot. After completion, various files are created: `u-boot` is the original ELF binary file and used to create all others, `u-boot.bin` is the raw binary image which can be written to flash memory, `u-boot.srec` is the same in the S-record format from Motorola, and `u-boot.ldr` is for booting from serial flash. Latter two are never needed in this project.

There is also a way to create the U-Boot image in the widespread Intel HEX format. For example, the JTAG flash programmer of VisualDSP++[9] 4.0 only supports this format. Such image is created with the command:

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

(Note that entering the command `make u-boot.hex` also generates an Intel HEX file, but this one contains some offset information such that VisualDSP++ gets confused.)

[9]VisualDSP++ is an IDE for Blackfin processors made by Analog Devices. It was briefly introduced in section 3.3.

4.4.3 Bringing U-Boot onto the target

If the flash memory of the core module is empty at the beginning, as was the case with this project, the only possibility to program U-Boot is to use JTAG flash programmer hardware. Therefore, a JTAG device from Analog Devices was connected to the evaluation board and VisualDSP++ was used to program U-Boot into flash memory. Since this procedure is only necessary once, it will not be explained further. Refer to [Tam05] for short instructions including screenshots. Future upgrades of U-Boot can be conducted in U-Boot itself, which is to be explained in section 4.4.4.

As soon as U-Boot is programmed into flash memory, the evaluation board and the plugged-in core module can be connected to the development workstation as described in section 4.2. After starting Kermit and hitting the reset button (see figure 3.2 for its location) on the evaluation board, the U-Boot starts up and displays some information. Hitting any key stops the autoboot countdown and the command prompt `CM-BF537E>` is shown:

```
U-Boot-1.1.3-ADI-R06R2 (Oct  2 2006 - 19:29:27)

CPU:   ADSP BF537 Rev.: 0.2
Board: Bluetechnix CM-BF537 board
       Support: http://www.bluetechnix.at/
Clock: VCO: 500 MHz, Core: 500 MHz, System: 100 MHz
SDRAM: 32 MB
Device ID of the Flash is 890016
Memory Map for the Flash
0x20000000 - 0x20400000 Single Flash Chip
Please type command flinfo for information on Sectors
FLASH:   4 MB
In:    serial
Out:   serial
Err:   serial
Net:    ADI BF537 EMAC
Hit any key to stop autoboot:  0
CM-BF537E>
```

4.4.4 Important routines

Network configuration

To set up the network, some environment variables have to be adjusted to contain the IP address of the device, the network mask and the IP address of a TFTP server to connect to. The following commands are entered at the U-Boot command line:

```
CM-BF537E> setenv ipaddr 192.168.1.202
CM-BF537E> setenv ipaddr 192.168.1.201
CM-BF537E> setenv netmask 255.255.255.0
```

If these settings should be restored after a reset, they must be saved with the `saveenv` command.

Loading a file onto the target

For the web radio project, a TFTP server on the development host is used. Transfers are fast and simple, because only one command has to be entered. The file must exist in the `/tftpboot/` directory on the workstation.

```
CM-BF537E> tftp 1000000 uImage
Using MAC Address 02:80:AD:20:31:B8
TFTP from server 192.168.1.201; our IP address is 192.168.1.202
Filename 'uImage'.
Load address: 0x1000000
 ...
Bytes transferred = 2244216 (223e78 hex)
```

Usually, the file is stored in RAM at address 0x1000000 on the target. (Table 4.1 shows U-Boot's memory mapping for this core module.) Memory addresses are always taken as hexadecimal in U-Boot, even if they lack the *0x* prefix.

If no network or TFTP server is available, the file can also be loaded over the serial connection, although this is very slow and hence inapplicable for rapid development. The command

```
CM-BF537E> loadb 1000000
## Ready for binary (kermit) download to 0x01000000 at 115200 bps...
```

is entered and the device waits for the transfer. Now the file has to be sent with Kermit. Therefore, *Ctrl*–\ followed by *c* has to be pressed to get back to the Kermit command line. By keying in

```
(/home/harald/) C-Kermit>send /tftpboot/uImage
```

the transfer is started. To get to the U-Boot prompt again, *connect* or simply *c* is entered. The file now lives in RAM at address 0x1000000.

Memory type	Start address	End address	Size
SDRAM	0x0000 0000	0x01FF FFFF	32 Mbyte
Flash memory	0x2000 0000	0x203F FFFF	4 Mbyte

Table 4.1: Memory mapping in U-Boot with the CM-BF537E core module

Writing a uClinux image to flash memory

During development, it makes no sense to write each compilation of uClinux into flash memory. Usually, these images are transferred to the target over the network and bootet from RAM. However, this project's goal is to develop a stand-alone device, and hence the uClinux image gets programmed into flash memory eventually. Only compressed images of uClinux (usually named `uImage`) are small enough to fit into flash memory.

The flash memory chip on the CM-BF537E has a size of 4 Mbyte, but due to technical reasons only 2 Mbyte are mapped into the processor's memory space at a time[10]. Switching between lower and upper flash memory (each 2 Mbyte in size) is done via a GPIO pin.

[10]The ADSP-B537 processor provides 4 asynchronous memory banks each 1 Mbyte large. It would have been possible to map the full flash memory size into the address space but no other I/O devices would have been able to be connected then.

However, U-Boot contains a patch that cares about this GPIO pin and hence makes the full size transparently accessible.

Flash memory starts at address 0x20000000 (see figure 4.1) and is divided into 32 sectors with 128 Kbyte each. The hexadecimal sector start addresses can be listed in U-Boot with the `flinfo` command. Sector numbering starts with zero. The first sector is always occupied by U-Boot, the environment is stored in the second sector, hence a uClinux image may start in the third sector at address 0x20040000.

Before the image file can be written to the flash memory, affected sectors must be erased first:

```
CM-BF537E> erase 1:2-15
Erase Flash Sectors 2-15 in Bank # 1
 ...
```

Programming and optional memory compare is conducted by the following commands (the *filesize* variable is set automatically after a transfer):

```
CM-BF537E> cp.b 1000000 20040000 $(filesize)
Copy to Flash... Bytes for programming: 1750167
First sector: 2
Sectors needed:14
 ...
done
CM-BF537E> cmp.b 1000000 20040000 $(filesize)
Total of 1750167 bytes were the same
```

Because read accesses in U-Boot are strictly memory-mapped, commands like `cmp.b` do not work with the upper 2 MByte of flash memory. But booting an image which resides there is no problem.

Booting a uClinux image

The boot parameters for the uClinux kernel can be set in U-Boot with the *bootargs* variable, unless uClinux was built with compiled-in kernel parameters. `printenv` is used to print the current content of environment variables. To change it,

```
CM-BF537E> setenv bootargs root=/dev/mtdblock0 rw
```

(maybe with other parameters) must be entered, followed by `saveenv` if the setting should be restored after a reset.

To boot a compressed `uImage`, the `bootm` command is the right one. It takes the start address of the image as parameter, i.e. if the image was transferred to RAM,

```
CM-BF537E> bootm 1000000
```

has to be entered. For booting from flash, *1000000* must be replaced by *20040000*. This image is then uncompressed and booted, which looks like this:

```
CM-BF537E> bootm 20040000
## Booting image at 20040000 ...
   Image Name:   uClinux Kernel and ext2
   Image Type:   Blackfin Linux Kernel Image (gzip compressed)
   Data Size:    1750103 Bytes =  1.7 MB
   Load Address: 00001000
   Entry Point:  00001000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Starting Kernel at = 1000
 ...
```

An uncompressed image has the drawback that transferring it onto the target takes longer, but boot time is shorter because it does not need to be decompressed first. To boot such an image, which is usually named `linux` on the workstation, the following command has to be entered:

```
CM-BF537E> bootelf 1000000
...
## Starting application at 0x00001000 ...
```

During application development in uClinux, the process of downloading an image onto the target and booting it should be automated. This is easily accomplished in U-Boot by adjusting some environment variables. The following setup was used in this project:

```
setenv bootcmd tftp 1000000 linux\; bootelf 1000000
setenv bootdelay 1
```

Hence, after reset, the uClinux image is automatically fetched and booted after 1 second. Setting *bootdelay* to zero is a bad idea because there would be no chance to access the U-Boot command line anymore.

Upgrading U-Boot

If a new U-Boot was compiled and should be deployed, the `u-boot.bin` file is first transferred onto the target at address 0x1000000. Because writing a non-functional image would make programming the target with a JTAG device necessary, it is advisable to test the U-Boot first. This is accomplished by

```
CM-BF537E> go 1000000
```

which jumps to the given address and executes code from there. If the new U-Boot comes up, everything should be fine and it can be written to flash. After a reset the new image is again transferred onto the target and programmed with the aid of these commands:

```
CM-BF537E> protect off 1:0
CM-BF537E> erase 1:0
CM-BF537E> cp.b 1000000 20000000 $(filesize)
```

After a hardware reset the new U-Boot is booted. Environment variables are not overwritten.

4.5 uClinux distribution

In the previous sections the toolchain was installed and the U-Boot bootloader was brought onto the target board. Now it is time to attend to uClinux itself. Since an introduction was already provided in section 4.1, this one concentrates on practical aspects.

In this project, a CVS version (as of Sept. 27, 2006) of the uClinux distribution is used which contains Linux kernel version 2.6.16. It has already been checked out in section 4.3. At the beginning, a default uClinux kernel and filesystem will be compiled and booted onto the target. Usually, the kernel and all user applications are packed into one image file.

4.5.1 Building uClinux

Before uClinux can be built, it must be configured with the built-in configuration dialogs known from all Linux kernels. In the `uClinux-dist/` directory,

```
make menuconfig
```

is entered. In the first dialog, the vendor *Bluetechnix* and the product *CM-BF537E* are chosen in the *Vendor/Product Selection* menu item. Within the next item, to configure kernel and userspace programs, the respective options have to be checked:

```
Kernel/Library/Defaults Selection  --->
  [*] Customize Kernel Settings
  [*] Customize Vendor/User Settings
```

After exiting this dialog and saving the settings, another one is opened which offers all kernel configuration options. This dialog is similar to a Linux kernel configuration for a desktop PC:

```
Code maturity level options  --->
General setup  --->
Loadable module support  --->
Block layer  --->
Blackfin Processor Options  --->
Bus options (PCI, PCMCIA, EISA, MCA, ISA)  --->
Executable file formats  --->
Power management options  --->
CPU Frequency scaling  --->
Networking  --->
Device Drivers  --->
File systems  --->
Profiling support  --->
Kernel hacking  --->
Security options  --->
Cryptographic options  --->
Library routines  --->
```

The most important menu item for the web radio project is *Device Drivers*. The AD1836A audio chip, SD card and other devices are enabled and configured therein. The *File systems* menu item contains NFS and SMB support among many others.

The following option is turned off right now so that the uClinux kernel boot parameters can be set in U-Boot and are not compiled into the kernel:

```
Kernel hacking  --->
  [ ] Compiled-in Kernel Boot Parameter
```

(Specific configuration will be provided later in dedicated sections.) Exiting and saving from this dialog opens the third one for user applications configuration:

```
Core Applications  --->
Library Configuration  --->
Flash Tools  --->
Filesystem Applications  --->
Network Applications  --->
Miscellaneous Applications  --->
BusyBox  --->
Tinylogin  --->
MicroWindows  --->
Games  --->
Miscellaneous Configuration  --->
Debug Builds  --->
Blackfin build Options  --->
Blackfin test programs  --->
Blackfin app programs  --->
Blackfin canned demos  --->
```

Within the *BusyBox* menu item, the `busybox` single executable is configured which provides a great many different commands typically available on a Linux machine. It is optimized for size and hence adequate for uClinux. Applications that can not be found in Busybox or produce errors at compilation are likely available as separate executables in one of the other menu items. Audio tools are included within *Miscellaneous Applications*. Configuration of single applications will be shown as they are needed. For now, this dialog remains unchanged.

After exiting the last dialog, both kernel and applications are compiled with the `make` command. The first build process lasts several minutes. After completion, image files are saved into the `uClinux-dist/images/` directory: `linux` is the uncompressed ELF image and `uImage` is the compressed one. The next step is to copy one of these into the `/tftpboot/` directory to be accessible via TFTP. To automate these steps, the following shell script is written:

```
#!/bin/sh
rm -rf romfs/* && \
make && \
cp images/linux /tftpboot && \
beep
```

It clears the `romfs` tree from which the target file system is generated, builds uClinux and copies the `linux` image file to the TFTP server root.

Now the image can be loaded onto the target and booted with U-Boot. This happens automatically with the U-Boot setup from section 4.4.4. So, after hitting the reset button on the evaluation board, uClinux boots up and displays a lot of information in doing so. At the end the following welcome screen is shown:

```
Welcome to:
       ____ _  _
      /  __| ||_|             _  _
 _   _| |  | | _ ____  _   _ \ \/ /
| | | | |  | || |  _ \| | | | \  /
| |_| | |__| || | | | | |_| | /  \
|  ___\____|_||_|_| |_|\____|/_/\_\
|_|

For further information see:
http://www.uclinux.org/
http://blackfin.uclinux.org/
http://www.bluetechnix.at

BusyBox v1.00 (2007.03.18-09:39+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.

root:~>
```

The last line contains the uClinux command prompt. By default, there is no need for a login and all commands are executed with the rights of the almighty *root* user. Here, the Minix shell is used which is part of BusyBox. It is very similar to the Bourne-again shell (`bash`) usually available on Linux workstations.

4.5.2 Adding own drivers and applications

Device drivers

At least one device driver must be implemented from scratch for the embedded web radio device: The TFT-LCD driver. The kernel source code is contained in the directory `uClinux-dist/linux-2.6.x/`. Therein, all devices drivers live in the `drivers/` subdirectory[11], which again has lots of subdirectories for specific device classes like *block*, *input*, *usb*, *video* etc. To add a new device driver, three steps within the adequate subdirectory have to be accomplished:

1. The source code file(s) have to be created. This/These will mostly be written in the C language. (In this example, `source_code_file.c` is used.)

2. The following lines have to be added to the `Kconfig` file which makes the device appear in the kernel configuration dialog:

   ```
         config SAMPLEDRIVER
           tristate "Driver name"
           depends on OPTION_A && OPTION_B
           select DRIVER_C
           help
             HELP CONTENT
   ```

   `depends on` means that the new driver is only available if these kernel configuration options are chosen whereas `select` selects other options as soon as the new driver is enabled.

[11] Sound drivers are an exception: They are included in the `uClinux-dist/linux-2.6.x/sound/` directory.

3. One line has to be added to the `Makefile`:

   ```
   obj-$(CONFIG_SAMPLEDRIVER) += source_code_file.o
   ```

   It is responsible for building the device driver.

The following files may be important in conjunction with device drivers:

- `uClinux-dist/vendors/Bluetechnix/CM-BF537E/device_table.txt`: Therein all device files which will exist in the `/dev` directory in uClinux are enlisted. It is used by the `bfin-uclinux-genext2fs` command that generates the uClinux filesystem at the end of the build process.
- `uClinux-dist/linux-2.6.x/arch/blackfin/mach-bf537/boards/cm_bf537.c`: This is the board-specific file for the CM-BF537E core module. Configuration of kernel modules and other stuff that is highly dependent on the particular target hardware is done therein.

Applications

Adding a user application involves the following steps:

1. A new subdirectory in `uClinux-dist/user/` is created to store the application's source code files. (In this example, `newapp` is used.)
2. One line has to added to the file `uClinux-dist/user/Makefile` that adds the new application to the build process:

   ```
   dir_$(CONFIG_USER_NEWAPP_NEWAPP) += newapp
   ```

3. To add the new application to the application configuration dialog, a line similar to

   ```
   bool 'Application name' CONFIG_USER_NEWAPP_NEWAPP
   ```

   has to be added to `uClinux-dist/config/config.in`. The file is subdivided into sections, e.g. the web radio application will be added to the *Miscellaneous applications* submenu.
4. Some help content should be provided with an application, which is displayed during configuration when selecting the *Help* button. It is added to the `uClinux-dist/config/Configure.help` file, e.g.:

   ```
   CONFIG_USER_NEWAPP_NEWAPP
     This is the help content for the new application.
   ```

5. A `Makefile` has to be created in the application's directory. This is best done by copying it from another one's directory and adapting it: The `EXEC` variable must be set to the name of the executable (e.g. `newapp`) and `OBJS` contains all object files used to create the executable (e.g. `newapp.o` if the source file is `newapp.c`).

   The makefile contains the two standard targets *all* and *clean*. The third target, *romfs*, is called at the end of the uClinux build process and installs the executable file into `uClinux-dist/romfs/bin/`. A sample makefile is shown at [WBLb].

If an application should be started automatically when uClinux boots up, it is added to the `uClinux-dist/vendors/Bluetechnix/CM-BF537E/rc` file. As an example, the network can be set up with the `ifconfig` command this way.

4.5.3 Debugging

There are several methods for debugging in uClinux. The simplest method is to produce console output. Indeed, this was used most frequently in this project. With device drivers and the kernel in general, `printk()` is used to generate output. Different *loglevels* are available which define a message's severity. For debugging purposes, those two with the least severity are sufficient: *KERN_DEBUG* is for debugging messages. They are not printed instantly onto the console, but can only be viewed with the `dmesg` command or by `cat /var/log/messages`. Messages with loglevel *KERN_INFO* are printed instantly onto the console of uClinux. A typical code line looks like this:

```
printk(KERN_INFO "Write access to address %lx\n", adr);
```

Further information on `printk()` is provided in [CRKH05]. For debugging applications in the user space, `printf()` must be used instead which works almost alike, but there are no loglevels.

A more sophisticated debugging method is to use *GDB*, the *GNU debugger*. As stated in section 4.3, the GDB is built together with the Blackfin toolchain. It runs on the workstation and connects to the target via a serial line or over the local network. The counterpart running on the target is *gdbserver*, which comes with uClinux. It is called with the executable's name that shall be debugged. [WBLd] has a good step-by-step documentation for it. However, with the uClinux source code that was used in the beginning, building the GDB server failed. Hence as soon as it was clear that debugging with `printk()/printf()` would be sufficient for this project, GDB was left aside.

With KGDB (Kernel GDB), it is even possible to debug the uClinux kernel, but some patches have to be applied before. This method was never tested, fully in terms of Linus Torvalds, the inventor of the Linux kernel, who does not believe in interactive debuggers because they would lead to poor fixes and would not address the real cause of problems [CRKH05].

Chapter 5

Software components configuration

This chapter is dedicated to software components that are already available in uClinux and will be used with the web radio device. Their setup is mainly related to the configuration of the kernel and userspace programs as stated in section 4.5. With the latter ones, it is also important to determine necessary command line parameters a program must be called with.

5.1 Audio

The uClinux kernel contains a device driver for the AD1836A chip. Its configuration is done in the kernel configuration dialog (submenus are denoted by indentation):

```
Device Drivers  --->
  Sound  --->
    [*] Sound card support
    Advanced Linux Sound Architecture  --->
      [*] Advanced Linux Sound Architecture
      [*]   OSS Mixer API
      [*]   OSS PCM (digital audio) API
      ALSA Blackfin devices  --->
        [*] AD1836 Audio support for BF53x
        Interface between Blackfin and AD1836: TDM interface
        5.1 Channels or 3 Separate Stereos: 3 Stereos
        (0) Blackfin Audio SPORT port
        (5) Blackfin Audio SPI channel selection bit
```

The interface between Blackfin processor and AD1836A chip can be set to TDM or I2S[1]. TDM mode works properly whereas with I2S, some hang-ups of the audio driver have been noticed during initial test. Hence, TDM is selected.
Because the web radio device needs a stereo channel for audio output, *3 Stereos* is chosen in the next option.

Last two options are specifically set for the audio extender board. Because the ADSP-BF537 processor has 2 SPORT ports, a selection is provided – SPORT0 must be chosen here. The audio chip's SPI chip select line is connected to the *SPI_CS5* pin of the Blackfin with the

[1]TDM mode supports more channels than I2S. However, according to the needs of the web radio device both are suitable. Detailed information can be retrieved from the datasheet [Ana03].

proposed hardware setup made in section 3.2. The Blackfin SPI kernel driver is automatically selected together with the audio driver, so one does not need to take care of it.

The driver is compiled into the kernel (in contrast to compiling it as a kernel module) and hence will initialize the hardware at boot. After kernel compilation and deployment onto the target, the device file `/dev/dsp` is connected to the audio chip. It is typically opened by userspace programs that output audio data.

Audio applications

Two audio applications of uClinux are used here: *mp3play* and *mixer*. To compile and pack them into the uClinux image, they must be selected in the uClinux configuration:

```
Miscellaneous Applications  --->
  [*] mp3play
  [*] mixer
```

mp3play already contains an MP3 decoder library that is automatically built with the player. To play a file in the local directory structure, the call looks like this: `mp3play /path/to/audiofile.mp3`. The player is also capable of playing audio streams from a remote server. In this case, the stream's URL has to be provided instead of the filename. However, some attention has to be paid to the format of the URL: If it comprises only a server name or IP address and a port number, a trailing slash "/" is mandatory, whereas the slash must not be given if the URL also contains subdirectories. An example shall be given for clarification:

```
mp3play http://some.server.com:1234/
mp3play http://some.server.com:1234/path/to/stream
```

The *mp3play* player will be subject to some changes in section 6.3.2 to allow better cooperation with the webradio application.

mixer is a program that allows volume control of the audio chip. It is called this way: `mixer vol 100`. Values can reach from 0 to 100.

5.2 Ethernet

The device driver for the on-chip MAC of the ADSP-BF537 processor is enabled by default in the kernel configuration. Nevertheless, the necessary options are shown here:

```
Device Drivers  --->
  Network device support  --->
    [*] Network device support
    Ethernet (10 or 100Mbit)  --->
      [*] Ethernet (10 or 100Mbit)
      [*]   Generic Media Independent Interface device support
      [*] Blackfin 536/537 on-chip mac support
```

For configuration of Ethernet, some userspace tools are necessary:

- The *dhcpcd* (*DHCP client daemon*) is used for configuring the network automatically if a DHCP server is available. It gets IP address and subnet mask, sets up the default route for accessing the Internet and makes a name server entry in the `/etc/resolv.conf` file. Usually, it is called without any command line arguments. Applying a timeout value with the `-t` parameter produces run-time errors with the current version and hence is not used.

- *ifconfig* is the standard Linux tool for manual configuration of network devices. It is called this way: `ifconfig eth0 up 192.168.1.202 netmask 255.255.255.0` wherein *eth0* is the first Ethernet device and the netmask is preceded by the keyword *netmask*.

- The *route* program sets up kernel network routes. When called without arguments, current routes are printed to the console. The web radio device needs it to set the default gateway for accessing the Internet. The respective program call is `route add default gw 192.168.1.1 eth0`.

The following uClinux configuration options enable these tools:

```
Network Applications  --->
  [*] dhcpcd-new (2.0/2.4)
  [*] ifconfig
  [*] route
```

5.3 Flash memory

The flash memory chip that is used onto the CM-BF537E core module is compatible with CFI (Common Flash Interface), for which a uClinux device driver exists. With the CM-BF537E core module, only half of the flash memory (2 Mbyte) can be mapped into the processor's address space at a time. Switching between the banks is done via the GPIO pin *PF4*, but this is not integrated into the kernel driver. Nevertheless, the CFI auto-detection makes the device 4 Mbyte large for the kernel, but writing to the upper bank is generally prohibited because this may corrupt other devices.

Due to this limitation, the following strategy is applied to the web radio device: The last sector of the flash memory is large enough (128 Kbyte) for the intended configuration files. It is mapped as a Linux MTD (Memory Technology Device) by the driver. The *PF4* pin is configured to high output which enables the upper bank. This can be easily achieved with the *Programmable Flags driver* which the next section is devoted to. The kernel configuration is as follows:

```
Device Drivers  --->
  Memory Technology Devices (MTD)  --->
    [*] Memory Technology Device (MTD) support
    [*]   MTD partitioning support
    [*]   Direct char device access to MTD devices
    [*]     Readonly block device access to MTD devices
      RAM/ROM/Flash chip drivers  --->
        [*] Detect flash chips by Common Flash Interface (CFI) probe
        [*] Support for Intel/Sharp flash chips
        [*] Support for RAM chips in bus mapping
        [*] Support for ROM chips in bus mapping
      Mapping drivers for chip access  --->
        [*] CFI Flash device in physical memory map
        (0x201E0000) Physical start address of flash mapping
        (0x20000) Physical length of flash mapping
        (2)   Bank width in octets
        [*] Generic uClinux RAM/ROM filesystem support
```

The length of 0x20000 corresponds to a size of 128 Kbyte. Flash memory starts at physical address 0x20000000, so 0x201E0000 points to the last sector within 2 Mbyte. As a result, the block device file `/dev/mtdblock0` in uClinux points to that sector. There is also a character device file `/dev/mtd0` which is used e.g. for erasing.
As soon as the flash memory device driver is enabled in the kernel, the *bootargs* parameter in U-Boot has to be changed because the root file system now lives on `/dev/mtdblock1`. Otherwise, a kernel panic happens at uClinux startup.

To be able to store and load several configuration files without using a file system, two shell scripts are written that will be called by the webradio application. The files are packed into a *gzip*[2]-compressed tar[3] archive which is sent to the memory device:

flash-save

```
#!/bin/sh
MTD="/dev/mtd0"
SAVE_LIST="/etc/webradio_conffile ..."
flash_erase "$MTD" || exit 1
tar c $SAVE_LIST | gzip > "$MTD" || exit 1
```

flash-restore

```
#!/bin/sh
MTDBLOCK="/dev/mtdblock0"
cd /
cat ${MTDBLOCK} | gzip -d | tar x
```

gzip detects the end of the archive while unpacking. This is important because the end of the last configuration file is not known without a file system.

[2] *gzip* is short for “GNU zip”, a freely available file compression program.
[3] *tar* is an abbrevation of *tape archive* and is an archive file format. Initially used for sequential backups on tapes, it is now commonly used to collate files into a larger file. The Unix/Linux command `tar` outputs this file format.

The following uClinux configuration options must be chosen to make the scripts work:

```
Flash Tools  --->
  [*] mtd-utils
  [*]   erase
BusyBox --->
  [*] gunzip
  [*] gzip
  [*] tar
  [*] tar: allow creation
```

5.4 Network applications

For mounting NFS and SMB file systems, support must be first enabled in the kernel configuration:

```
Device Drivers  --->
  Network File Systems  --->
    [*] NFS file system support
    [*]   Provide NFSv3 client support
    [*] SMB file system support (to mount Windows shares etc.)
  Native Language Support  --->
    (iso8859-1) Default NLS Option
    [*]   Codepage 437 (United States, Canada)
    [*]   NLS ISO 8859-1  (Latin 1; Western European Languages)
```

For SMB file systems a special mount application is necessary, whereas NFS requires the *portmap* application to be running. The corresponding uClinux configuration is as follows:

```
Filesystem Applications  --->
  [*]   smbmount
Network Applications  --->
  [*] portmap
BusyBox --->
  [*] mount
  [*] mount: support NFS mounts
```

To mount an SMB shared directory, the command `smbmount //server/share /mnt -o username=guest,guest,ro` will be used. The *guest* option makes the need for a password superfluous, *username* must not be empty hereby. Access is restricted to read-only according to the principle of least privilege[4].

Mounting NFS file systems is easy too. However, NFS was designed with reliability of file operations in mind. Consequences are that file operations are not allowed to be interrupted by default and very long timeouts are used [WUN]. Hence, programs like *mp3play* that access NFS files are blocked if for example, the NFS server is shut down. This behavior is not well suited for the web radio device, so a great effort was made to find reasonable mount parameters which was impeded by the facts that the BusyBox *mount* does not support several options and the self-contained mount tool did not compile correctly.

As a result, the command `mount -t nfs -o ro,soft,nolock server:/path/to/share /mnt` will be used for NFS mounting, whereby the *soft* option cancels a file operation after a timeout and *nolock* disables locking of files which does not matter because read-only

[4]In computer science, the principle of least privilege requires that every process is only able to access resources which are necessary for its legitimate purpose.

access is used. With these options, file operations are canceled 10 seconds after the occurence of an error. This is acceptable for the embedded web radio.

Web server

The web server *boa* is used for the proposed configuration web service. It was designed with speed and security in mind [WBO] which makes is perfectly suitable for this embedded system. To build *boa*, the following uClinux configuration option has to be enabled:

```
Network Applications  --->
  [*] boa-new(ver 0.94.14)
```

Furthermore, 2 files have to be edited:
1. In `uClinux-dist/user/boa-new/src/defines.h`, the *SERVER_ROOT* define is changed to `/etc` because boa's configuration file lives there.
2. To enable CGI support, the line `AddType application/x-httpd-cgi cgi` has to be appended to file `uClinux-dist/vendors/Bluetechnix/CM-BF537E/boa.conf`. The *UserDir* and *ServerName* variables are commented out because they are not needed.

5.5 Programmable Flags driver

The *Programmable Flags driver* provides an interface to the Blackfin's GPIO pins for userspace programs by means of the device files `/dev/pf0` to `/dev/pf15`. Its purposes within this project are reading button inputs and switching between the lower and upper bank of the flash memory chip. It is enabled in the kernel configuration dialog by a single option:

```
Device Drivers  --->
  Character devices  --->
    [*] Blackfin BF53x Programmable Flags Driver
```

Chapter 6

Software development

This is the main chapter of the document. It corresponds to that phase of the embedded web radio project which involves the most time. The following software components must be developed: The device driver for the LCD device has to be implemented from scratch, whereas the SD card driver is already available and must be improved to provide stable read functionality. The user space application *mp3play* is then subject to modifications to allow co-operation with the web radio application. The latter is the “heart” of this embedded device and controls all other components. At last, the web configuration service is developed. As usual with Linux development, the C programming language is used exclusively.

6.1 Hitachi TFT-LCD device driver

6.1.1 Frame buffer and memory mapping

In Linux, graphics hardware is generally supported via the *frame buffer* device class. Such a device provides an abstraction for the underlying hardware and offers a well-defined interface to user space applications so that they do not need to know anything about the real hardware. A frame buffer device is a character device and associated with a device file, e.g. `/dev/fb0` for the first display available in a system. The device file has major number 29 which represents the class. The minor number starts with 0 and differentiates between several frame buffer devices. A detailed illustration of the device file concept of Linux can be found in [CRKH05].

Frame buffer devices are similar to memory devices, both offer a memory area that can be read from and written to. In the former case, the memory area corresponds to the content of the video device. Normally, if some software wants to display content, it does not simply write to the frame buffer device file, but make a *memory mapping* which means the graphics memory is mapped into the process' address space. The Linux kernel then translates every access within this region to an operation on the corresponding frame buffer byte [BC05]. There are two kinds of memory mapping:

1. With *shared mapping*, each write operation changes the original memory buffer. If several processes map the same memory region, write operations are immediately visible to all of them.

2. A *private mapping* is made for read-only access. Writing to the memory map does not change the original memory content.

Clearly, shared mapping is required for frame buffers. The memory mapping is initiated with the `mmap()` system call which will be treated in detail in section 6.4.

Documentation of the Linux frame buffer is very poor[1]. A good starting point for programmers is the header file `uClinux-dist/linux-2.6.x/include/linux/fb.h` for frame buffer devices. It contains functions to register and deregister such devices and a great many structures that are filled with information regarding the memory mapping, hardware acceleration capabilities, screen resolution, color depth, etc. The device driver for the Hitachi display will naturally only provide basic functionality compared to a workstation's graphics card and hence will only use some of those structures. One important aspect that should be mentioned is that the Linux frame buffer is based on what is known as the RGB color model.

6.1.2 Implementation

First, the source file `uClinux-dist/linux-2.6.x/drivers/video/hitachi-tx09.c` is created in the directory suitable for all video devices. A configuration option has to be added to the `Kconfig` file within the same directory:

```
config FB_HITACHI_TX09
tristate "Hitachi TX09D70VM1CDA TFT LCD"
depends on FB && BF537
help
  This is the frame buffer device driver for a HITACHI TX09D70VM1CDA TFT LCD
  attached to an ADSP-BF537.
```

After adding a target to the `Makefile` as explained in section 4.5.2, the Hitachi display driver can be selected in the kernel configuration dialog. For the web radio device, it is compiled into the kernel and not as kernel module. Anyway, the display is not started at uClinux boot but when the `/dev/fb0` file is opened.

As a starting point, the driver source code for the SHARP LQ035 TFT-LCD in the file `bf537-lq035.c` included in uClinux is used as a template regarding frame buffer structures and the like. It is similar to the Hitachi LCD in size and intended employment. Furthermore, a device driver for the Hitachi display was already written for the BLACKSheep VDK at the Institute of Computer Technology. Its source code is also available to the author and mostly covers configuration of the Blackfin processor.

Controlling the display is quite a complex task for the Blackfin processor. Figure 6.1 shows a schematic of the display hardware interface and components of the processor that are involved.

The role of each part is discussed now, code excerpts are inserted where they are helpful. The datasheet of the LCD is available at [Kao06]. A detailed assignment of processor and LCD pins is provided in appendix A.

[1]A quick overview is given in the kernel documentation directory which is also included in uClinux: `uClinux-dist/linux-2.6.x/Documentation/fb/`

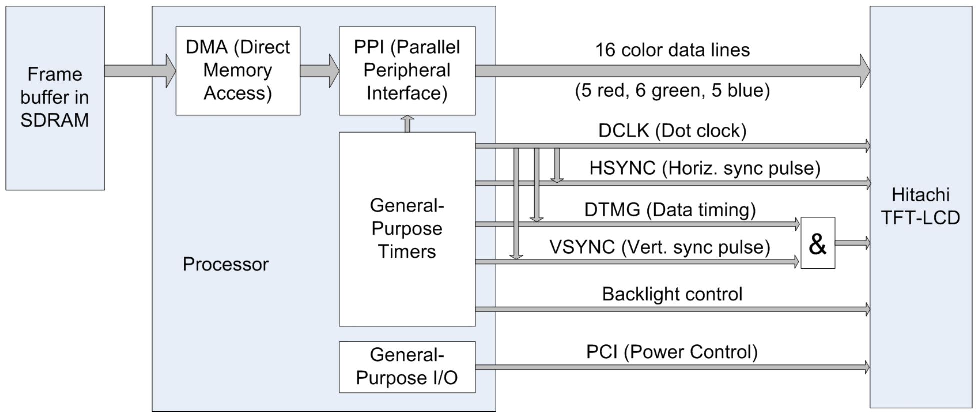


Figure 6.1: Display control and involved components

Frame buffer structures and functions

At initialization the function `tx09_fb_init(void)` is called. It first allocates a coherent memory region for the frame buffer, then sets up the structure `tx09_fb` of type *fb_info* with which the frame buffer device is registered in the kernel. Furthermore, a backlight device is registered which can be used to adjust the LCD's brightness:

```
fb_buffer = dma_alloc_coherent(NULL, 240*320*2, &dma_handle, GFP_KERNEL);
...
if (register_framebuffer(&tx09_fb) < 0)   // register frame buffer
...
backlight_device_register("hitachi-backlight", NULL, &tx09fb_bl);
```

The display has a size of 240 x 320 pixels each allocating 2 bytes of color data, hence 150 Kbyte of memory are needed. `tx09_fb` points to 3 other structures of types *fbops*, *fb_fix_screeninfo* and *fb_var_screeninfo*. The former includes function pointers for opening/releasing the frame buffer device and for making a memory mapping, which is called if a user space program issues `mmap()`. Out of several fields of the structure *vm_area_struct* which are listed in [BC05], it is sufficient to set start and end addresses and a flag:

```
static int direct_mmap(struct fb_info *info, struct vm_area_struct * vma)
  {
    vma->vm_start = (unsigned long)fb_buffer;
    vma->vm_end = vma->vm_start + 320*240*2;
    vma->vm_flags |=  VM_MAYSHARE;
    return 0;
  }
```

The `VM_MAYSHARE` flag indicates that pages can be shared by several processes. On architectures without an MMU, this flag must be set.

Structure `tx09_fb_fix` contains the following unchangeable information about the frame buffer which can be retrieved with the `ioctl()` system call by userspace programs: The driver name (`hitachi-tx09`), the memory length (153600 bytes), the memory length of a display line (480 bytes), the frame buffer type (*packed pixels* which means that there is a direct mapping from video memory to a pixel value), the visual type (true color) and the hardware acceleration (none).

The `tx09_fb_defined` structure is of type *fb_var_screeninfo* and provides information that can usually be changed to access different modes of graphics hardware. However, the Hitachi LCD driver will only provide a single mode and hence these values will never be changed[2]. Variable screen information includes the resolution (240 x 320 pixels because the display is orientied in portrait format), bits per pixel (16) and bitfield descriptions for each basic color. The organization of color bits/lines on the EXT-BF5xx-Camera board is shown in table 6.1. Note that the color lines are aligned bit-reversed!

B0	B1	B2	B3	B4	G0	G1	G2	G3	G4	G5	R0	R1	R2	R3	R4
blue (5 bits)					green (6 bits)						red (5 bits)				
MSB[3]															LSB[4]

Table 6.1: Alignment of RGB color lines

Processor configuration and LCD initialization which are described from now on are not done until the device file is opened, i.e. the function `tx09_fb_open()` is called.

General-purpose timers

To generate the different clock signals for the LCD, 5 general-purpose timers are necessary. All work in PWM mode and must be activated in the *PORTF_FER* and *PORT_MUX* registers.

Timer 7 is used to generate the *DCLK* (dot clock) signal which determines the speed of transferring single pixels. Data must be valid on falling edge [Kao06]. The Blackfin processor has the capability of synchronizing timer modules to an external clock input [Ana05]. The output pin of timer 7 is hardwired to the TMRCLK input pin of the processor and so the DCLK can be used as clock source for the remaining timers and even for the PPI (which itself controls the DMA). Hence it is possible to adjust the refresh rate at a single point – the DCLK timer. A maximum frequency of 6 MHz is allowed by the display which yields to a maximum refresh rate of 67 Hz. The following code lines configure TMRCLK as input and set up the DCLK PWM output by writing to the corresponding processor registers:

```
bfin_write_PORTFIO_DIR(bfin_read_PORTFIO_DIR() & ~0x8000);   /* set TMRCLK as input */
bfin_write_PORTFIO_INEN(bfin_read_PORTFIO_INEN() | 0x8000);

BFIN_WRITE(TIMER_DCLK,CONFIG)(PERIOD_CNT|PULSE_HI|PWM_OUT);  /* configure DCLK timer */
timer_period = get_sclk() / (REFRESH_RATE * 89271);          /* calculate period */
BFIN_WRITE(TIMER_DCLK,PERIOD)(timer_period);                 /* set length of cycle */
BFIN_WRITE(TIMER_DCLK,WIDTH)(timer_period/2);                /* set length of high pulse */
```

The *PERIOD_CNT* bit must be set for continuous operation, *PULSE_HI* starts with high output and generates a falling edge, at which time the color data will be stable. The period of the DCLK timer is clearly dependent on the processor's system clock. The unit of `REFRESH_RATE` is Hertz. The number of DCLK cycles for refreshing the LCD is 89271 because after each line 33 blanking pixel follow (horizontal blanking interval) and 7 blanking lines (vertical blanking interval) must be inserted each time after 320 lines:

$$(240 + 33)\ pixels \cdot (320 + 7)\ lines = 89271\ cycles$$

[2]To let an application know of this limitation, the `fbops` structure actually points to a fourth function for checking the variable screen information. This functions always returns the -EINVAL value.

Timer 0 creates the *HSYNC* (horizontal synchronization pulse) signal which is a low pulse during the horizontal blanking interval. It will be used for frame synchronization of the PPI. This is possible because the timer's output pin is identical with the PPI1Sy1 pin. Timer 0 is clocked by the DCLK PWM signal, so its configuration is slightly different compared to timer 7 (The *CLK_SEL* and *TIN_SEL* bits make the timer be clocked by TMRCLK):

```
BFIN_WRITE(TIMER_HSYNC,CONFIG)(CLK_SEL|TIN_SEL|PERIOD_CNT|PWM_OUT);  /* clocked by TMRCLK */
BFIN_WRITE(TIMER_HSYNC,PERIOD)(273);                                 /* 240 + 33 blanking */
BFIN_WRITE(TIMER_HSYNC,WIDTH)(5);
```

The *DTMG* signal is the data timing signal. It is high until horizontal display end is reached and hence low during horizontal blanking. In the vertical blanking interval it must be low, too. 2 timers are necessary for generation of this signal: Timer 1 is used for the DTMG waveform, and timer 3 outputs a *VSYNC* signal that is low during vertical blanking. Configuration is similar to timer 0. Both signals are connected to the LCD's DTMG pin via an AND gate.

Timer 6 is used for backlight control. By default, it generates a PWM signal with a 100% duty cycle for the highest available brightness. This timer is independent from the others and configured like timer 7.

PPI (Parallel peripheral interface) configuration

The PPI is a half-duplex, bidirectional port. Up to 16 data bits are supported, which are all used for the LCD. It is clocked by the TMRCLK input pin which is connected to the DCLK PWM signal. Configuration is rather simple: First, the data pins PPI1D0 to PPI1D15 have to be activated in the *PORTG_FER* and *PORT_MUX* registers according to [Ana05]. Thereafter, 3 PPI registers have to be set:

```
bfin_write_PPI_DELAY(27);
bfin_write_PPI_COUNT(240-1);
bfin_write_PPI_CONTROL(0x380e);
```

The PPI_CONTROL value chooses the following options: The PPI works in output mode and drives data on the falling edge of TMRCLK. The data length is set to 16 bits, and 1 frame synchronization signal (HSYNC) is used. The PPI_COUNT register must be set to the number of pixels per line minus one[5]. Because the PPI uses the HSYNC pulse (length 5 DCLK cycles) for frame synchronization and this pulse is generated at the beginning of the horizontal blanking interval, PPI_DELAY has to be set to 27 which sums up to 33 cycles. This equals the amount of horizontal blanking pixels and hence data transfer is started exactly at the beginning of a new line.

DMA (Direct memory access) configuration

By means of DMA, frame buffer data is fetched and transferred to the PPI data pins. According to [Ana05], the PPI must be used with the Blackfin's DMA engine. An own DMA channel is available that has to be configured prior using the PPI interface. By default, this channel has the highest priority.

[5]During the horizontal blanking interval, no data is written out.

Because the LCD requires a vertical blanking interval of 7 lines after each screen, actually 327 lines must be transferred via the PPI, although the 7 blanking lines are nowhere visible on the LCD. To avoid enlarging the frame buffer, DMA is used in the descriptor list mode where it is possible that the invisible 7 lines contain data from anywhere in the frame buffer memory.

Thereby, a chained list of descriptors is set up. Each descriptor is 8 bytes large, whereas the lower 4 bytes contain a pointer to the next descriptor and the upper 4 bytes point to the frame buffer data for the actual display line. To ensure fast access to the list, it is placed into the processor's L1 memory:

```
static unsigned long dma_desc_table[2*(320+7)] __attribute((section(".l1.data")));
...
for (i=0; i<7; i++)        // blanking lines point to first line in FB
  {
    dma_desc_table[2*i] = (unsigned long)&dma_desc_table[2*i+2]; //pointer to next desc.
    dma_desc_table[2*i+1] = (unsigned long)fb_buffer;
  }
for (i=7; i<327; i++)      // visible lines
  {
    dma_desc_table[2*i] = (unsigned long)&dma_desc_table[2*i+2];
    dma_desc_table[2*i+1] = (unsigned long)fb_buffer + 2*240*(i-7);
  }
dma_desc_table[2*326] = (unsigned long)&dma_desc_table[0]; // last desc. points to first
```

Further configuration involves requesting the PPI DMA channel (with `request_dma()`) and setting the following DMA registers: X_COUNT is 240 (the length of one line), X_MODIFY is 2 (the data size of a pixel), NEXT_DESC_ADDR points to `dma_desc_table`, and CONFIG is 0x7404. The latter selects the large descriptor list model with 32-bit addresses, sets the descriptor size to 4 bytes and configures DMA for 16-bit read transfers.

Now that everything is configured properly, the following steps have to be accomplished to start the LCD. Once the mechanism is kicked off, the Blackfin's ALU is not concerned with it anymore and hence LCD control does not contribute to processor utilization.

- The DMA channel is enabled by setting bit 0 of its control register.
- The PPI is enabled similarly.
- All necessary timers are started by writing to the TIMER_ENABLE register. This effectively starts the data transfer.
- After 50 ms, the PCI line is pulled high and the backlight timer is started.

If the `/dev/fb0` device file is closed, these components are disabled in reverse order.

6.1.3 Troubles

With the first version of the device driver, the screen content was displaced upwards. This was realized by displaying a test picture. Studying the hardware reference manual, it was eventually found out that the PPI was enabled prior its DMA channel which was clearly a fault and the cause of the distortion.

During long periods of LCD operation it was noticed that screen content moved rightwards by 1 pixel from time to time. Though no profiling techniques were applied, it was assumed that at this moments the DMA access to SDRAM is deferred by whatever cause. So two steps were accomplished which indeed made an impact:

1. First, the system clock was increased from 100 MHz to 131 MHz because SDRAM is clocked by it. In the U-Boot configuration file (see section 4.4.2) these new values were used: `CONFIG_VCO_MULT` = 21, `CONFIG_CCLK_DIV` = 1, `CONFIG_SCLK_DIV` = 4. Thereby, the core clock is also slightly increased to 525 MHz.
2. Second, the refresh rate (adjustable via the `REFRESH_RATE` macro) was decreased from initially 60 Hz to 25 Hz which is sufficient for the web radio device.

The camera extender board eventually used was subject to 2 hardware workaround which are described in section 7.1.

6.2 SD card device driver

The device driver for Secure digital memory cards has not been implemented from scratch. The SD/MMC[6] driver in the uClinux kernel has been a good starting point and saved a lot of time because writing new block device drivers is rather complex. The available driver has been marked *experimental* and suffered from major problems at the initial test: Often there were read errors, some cards did not get recognized at all, and sometimes the driver crashed and a kernel bug message was displayed: `kernel BUG at block/elevator.c:534! Kernel panic - not syncing: BUG!`

6.2.1 Configuration

At the start of development, the latest driver source code[7] was checked out from the Blackfin/uClinux CVS repository. It is located in the directory `uClinux-dist/linux-2.6.x/drivers/mmc/spi_mmc/`. Also the Blackfin SPI framework driver `uClinux-dist/linux-2.6.x/drivers/spi/spi_bfin5xx.c` was updated. According to [WVP], an SPI dummy device was added to the board-specific file[8] prior configuring and compiling the uClinux kernel with SD card support. The web radio device will support the VFAT file system on SD cards, so it must be enabled too:

```
Device Drivers  --->
  MMC/SD Card support  --->
    [*] MMC/SD for SPI support (EXPERIMENTAL)
    (1)   SPI chip select signal for MMC/SD card
    [*]   Print debug text on errors
File systems  --->
  DOS/FAT/NT Filesystems  --->
    [*] VFAT (Windows-95) fs support
    (437) Default codepage for FAT
    (iso8859-1) Default iocharset for FAT
```

[6]The MMC (Multimedia card) is the ancestor of the SD card. They have identical physical dimensions.
[7]as of Oct 20, 2006
[8]`uClinux-dist/arch/blackfin/mach-bf537/boards/cm-bf537e.h`

The block device file `/dev/mmc` corresponds to an inserted SD card from now on. For mounting the first partition, the special file `/dev/mmc1` has to be used.

The SD/MMC driver source code is divided into 2 parts:

1. The `spi_mmc_core.[ch]` files contain code related to the Linux block device layer. So there are functions for opening and releasing the block device, issuing an `ioctl()` system call etc. Dealing with the I/O scheduler and the block request queue is also done here. Furthermore, low-level functions that communicate with the SPI framework driver are included as well as initialization and registration of the SPI driver with the kernel. An explanation of Linux block device drivers and I/O scheduling would go beyond the scope of this document. [BC05] provides such an explanation.
2. The files `mmc_spi_mode.[ch]` contain code relevant to the SPI communication protocol for SD cards[9]. It provides functions for card initialization, sending commands, reading and writing blocks, reading the CSD (card specific data), CID (card identification) and status registers and error handling. The protocol is specified in [San03].

6.2.2 Debugging and driver improvement

The kernel bug

The first step to accomplish is to track down the cause of the kernel bugs. The affected code file `elevator.c` contains I/O scheduling functions, and the bug is initiated by the code line `BUG_ON(list_empty(&rq->queuelist));` in the `elv_dequeue_request()` function. This means that a request shall be removed from the request queue whereas the queue is empty at this point of time. Hence the driver code dealing with I/O scheduling has been analyzed first.

The driver offers two block request modes, by default the full-blown version with request queue and strategy function typical for Linux block devices is used. The queue is prepared by the `blk_init_queue()` function. The second mode is simpler because it does not use a request queue but requests are passed directly to the block driver. It uses `blk_queue_make_request()` for initialization. Switching to the simple mode makes the kernel bugs go away immediately, but a new problem is created thereby: While playing MP3 audio files with *mp3play*, playback is cut off each time new data is transferred from the SD card. This behavior is not acceptable for use with the web radio device, so the first mode must be used.

The real cause of the kernel bug was eventually found after a long phase of debugging and trying various kernel sources: The wrong I/O scheduler was used in the uClinux kernel configuration. *CFQ* (Complete fairness queueing) is the default one when configuring a kernel for the CM-BF537E core module. It was replaced by the *Anticipatory* scheduler which is the most sophisticated one offered by Linux and usually the default one. Hence the following kernel configuration options are of prime importance:

```
Block layer  --->
  IO Schedulers  --->
    [*] Anticipatory I/O scheduler
    Default I/O scheduler: Anticipatory
```

[9]The protocol is similar for MMCs.

Changes to `spi_mmc_core.c`

By default, media is searched in the SD card slot when the device driver is loaded. Because the web radio device does not need to access an SD card at boot, this function is disabled by commenting out the scheduling of the `card_work` work queue in the driver's probe function. Hence an SD card is initialized first at the time when it is mounted.

Furthermore, the speed of the SPI is adjusted to a maximum of 7.3 MHz, more precisely to the same value that is used by the AD1836A audio chip. The reason is that both devices use the SPI and in the worst case, the SD card gets disturbed even if this should not happen in theory because of different chip select lines. Decreasing speed does not matter for the web radio device – it is still fast enough for audio playback. This adjustment was accomplished by setting `SPEED_HZ` to a value of 9. The resulting speed is then calculated $\frac{system\ clock}{2 \cdot SPEED\_HZ}$. It is set for each SPI transfer that is handed over to the framework driver.

Changes to `mmc_spi_mode.c`

Significant changes were made to this file because some divergencies were found in this code with respect to the protocol described in the SD card product manual [San03]. Special attention was paid to making read access to SD cards free of errors, since this is mandatory for the web radio project. It is important to mention that the driver is only enhanced for SD cards. It is not necessary that MMCs work with the driver, although this was the original driver's intention. Below a list of functions is made that are altered:

`mmc_spi_init_card()`: The function is called if an SD card is accessed the first time. It initializes the card with the commands GO_IDLE_STATE and SEND_OP_COND.
Initialization is made simpler compared to the original code where MMCs are taken into concern. The GO_IDLE_STATE command takes the SD card into idle mode. With the old driver, it is only sent once and this was insufficient for some SD cards. So the command is issued up to 100 times until the cards sends the R1_IDLE_STATE response token. In between, some clock cycles are sent on the SPICLK line which are produced by the SPI dummy device (a fictional device with the non-existent SPI chip select line 0).

```
while(send_cmd_and_wait(pdev, GO_IDLE_STATE, 0, R1_IDLE_STATE, MMC_INIT_TIMEOUT) &&
      (++timeout_cnt < SD_IDLE_TIMEOUT))
  {
    mmc_spi_dummy_clocks(pdev, SD_CLK_CNTRL);
  }
mmc_spi_dummy_clocks(pdev, SD_CLK_CNTRL);
```

Thereafter, the command SEND_OP_COND is sent which activates the initialization process of the SD card. It is sent repeatedly until the card returns the R1_OK response. Then the function returns.

`send_cmd_and_wait()`: As suggested by its name, this function sends a command to the SD card and then waits for a response.
With the old driver, some zero bytes were sent prior the command. However, it was not checked if the card is actually ready for retrieving a command. This fault is corrected by polling the card at begin. If anything different from 0xFF (SPI MISO line is high) is read, the card is somehow busy and polling continues till a timeout is reached:

```
while(++timeout_cnt < SD_SPI_TIMEOUT)
  {
    if(pdev->read(&resp, 1, pdev->priv_data) < 0)
      {
        rval = ERR_SPI_TIMEOUT;
        goto out;
      }
    if (resp == 0xff)    // card is not busy
      break;
  }
```

In the following, the command is sent and `mmc_wait_response()` is called to wait for a valid response byte. Zero bytes are not needed anymore.

`mmc_wait_response()`: This function is used to wait for a response after a command was sent.
In the new driver, it is only called by `send_cmd_and_wait()`, all other functions have their own waiting procedure. A valid response byte is denoted by the most significant bit set to zero. If an SD card is not busy and no command was sent, it always returns 0xFF when reading from it. By checking the validity of the response byte more exactly with the query `if ((resp & 0x80) == 0)`, a little change was made within this function.

`mmc_spi_read_mmc_block()`: The purpose of this function is to read a single data block, i.e. 512 bytes, from the SD card.
Initially, the READ_SINGLE_BLOCK command is sent whereon the card should answer with the R1_OK response token. Next, the card has to be polled for the start block token SBT_S_BLOCK_READ. The `mmc_wait_response()` function is not used any more for it. After the token, 512 bytes of data are transmitted. The SD card specification requires that 2 bytes CRC data is read afterwards though no CRC checking is done in SPI mode [San03]. Reading CRC bytes is skipped by the original driver and may be a pitfall for some SD cards. Also, the `mmc_spi_error_handler()` function is not called after each read access.

`mmc_spi_write_mmc_block()`: This function writes a single data block to the SD card. It was also enhanced, but it will be skipped here because there is no relevance for the embedded web radio system.

`mmc_spi_read_status()`: The function reads out the SD card's status which consists of 2 bytes containing several status bits.
Instead of sending some zero bytes before sending the command, the card is polled to assure it is not busy. This code is already used in `send_cmd_and_wait()`. The old driver calls `mmc_wait_response()` for each of the 2 bytes. The start of the status bytes is indicated by the most significant bit of the first byte being low, but there are no such restrictions for the second byte. So the procedure was changed to poll for the first byte and then immediately read in the second one:

```
while(++timeout_cnt < SD_RESPONSE_TIMEOUT)
  {
    if(pdev->read(&b1, 1, pdev->priv_data) < 0) // read first byte
      {
        rval = ERR_SPI_TIMEOUT;
        goto out;
      }
    if ((b1 & 0x80) == 0x0)                     // bit 7 is always 0
      break;
  }
...
if (pdev->read(&b2, 1, pdev->priv_data) < 0)   // read second byte
...
```

Furthermore, the output of 8 dummy clock cycles is added after read as claimed by the SD card specification.

`read_mmc_reg()`: The function is called at card initialization to read out the CSD (card specific data) and CID (card identification) registers.
It is only slightly changed by adding the output of 8 dummy clock cycles after register read. Furthermore, the call of the error handler function is considered obsolete for SD cards and is removed.

Through these enhancements of the driver it can be deployed for the embedded web radio. One problem has to be mentioned that is related to hardware: A conflict of LCD and SD card slot has been detected regarding the unused SD card detect pin. The inevitable hardware manipulation is described in section 7.1.

6.3 Modifications to available software

6.3.1 Kernel page allocation patch

To complete all development steps for the uClinux kernel before moving to user space applications, this section appears first.

During tests of the SD card device driver, page allocation failures were noticed each time several audio files were played consecutively (with *mp3play*). By looking at the system's memory information in `/proc/meminfo` it was noticed that the cache seems to occupy all available RAM if the kernel outputs that allocation failure message. A full cache is not abnormal for Linux systems since files which are read in are automatically cached. However, if memory is needed (for caching more recent files or starting an application), the kernel must free at least a part of the cache. This obviously did not happen.

A solution to this problem was found at [WUCa], where a patch that is related to the fact that no MMU is available is included. It must be applied to the file `uClinux-dist/linux-2.6.x/mm/page_alloc.c`, which contains code for managing the list of free memory pages and allocating them. After applying the patch these errors were gone.

6.3.2 MP3 audio player

The source code of the MP3 audio player together with the decoder library is located in the directory `uClinux-dist/user/mp3play/`. The name of the executable file is `mp3play`. This application is designed for being called from the Linux command line, which means that informational output is provided to the user by writing it to the terminal. Error messages, e.g. if a file has the wrong format or an audio stream cannot be opened, is provided the same way.

With the web radio device, the player is called by the main application and its outputs are not visible to the user. This is no problem if no errors occur while playing. However, if there are errors, the web radio application must be notified out of two reasons: First, a description of the error must be provided to the user (via the LCD), and second, continuation of the main application depends on the kind of error. The proposed feedback from player to main control application will be realized by means of different exit codes. Because `mp3play` mostly returns zero value (for success) independent from the occurrence of an error, the first task to accomplish is the integration of different exit codes corresponding to various errors than can happen.

The other customization task is to build in a kind of watchdog timer into the player. This is important for the embedded web radio because it shall provide reasonable response times to actions from the outside world. Through initial tests it is observed that there exist error scenarios where `mp3play` does not terminate, e.g. if the network cable is disconnected while playing an audio stream from the Internet. The problem then is that playback stops but the main application is not noticed because these are two different processes running in uClinux. This is where the timer comes into play: As soon as decoding is stopped, the player is forced to terminate (and to return a proper exit value).

Integration of exit codes

The first step is to make a list of error scenarios and assign exit codes to them which is shown in table 6.2.

Error scenario	Exit code
No error occured	0
Opening MP3 stream failed	1
File or stream URL is not found	2
Stall of data stream	3
Any other error	4

Table 6.2: Audio player exit codes to be integrated

Exit code 1:
To determine whether opening an MP3 stream (for both regular files and streams) failed, the return value of the function `MPEGDEC_open()` has to be observed in the player source code file `mp3play.c`. A variable `error_audiostream` is created which acts like a flag and is set to 1 if this error occurs. It is later checked at the end of `main()`:

```
  mps = MPEGDEC_open(mp3_filename, &mpa_ctrl);
  if (!mps)
    {
      ...
      error_audiostream = 1;
      goto badfile;
    }
  ...
  if (error_audiostream)
    exit(1);
```

Exit code 2:
To record a “not found” error, the `error_open` variable is used. The mechanism is similar to the previous step, but there are two functions to check: The `open()` system call is used for opening regular files, whereas the built-in `openhttp()` function is used for Internet audio streams. Both return an error value if opening fails, which causes the flag to get set.

```
    ...
    mp3_fd = openhttp(mp3_filename);
  }
else
  {
    mp3_fd = open(mp3_filename, O_RDONLY);
  }
if (mp3_fd < 0)
  {
    http_streaming = 0;
    error_open = 1;
    goto badfile;
  }
```

At program termination it is checked whether exit code 2 must be returned.

Exit code 3:
The program shall terminate with exit code 3 when a stall in the data stream occurs. A typical reason for this error is the removal of the SD card while audio files get played from there. Knowing about the occurrence of such an error is important for the web radio application because further actions have to be taken accordingly.

To track down those errors, the source code of player and library is analyzed to find a function that is able to detect the error. The `MPEGDEC_decode_frame()` function which is located in the `mpegdec_lib/mp3onlydec.c` source file is suitable. By default, it returns the value -1 at both the end of a regular file and if the data stream stalls. These two cases have to be distinguished.

The function has several code lines that contain return statements which are possibly executed before decoding any MP3 frames at all. The following functions are called prior decoding is done:

```
   if( mps->need_sync ) {
      err = synchronize( mps );
      if( err ) return err;
      mps->need_sync = FALSE;
   }
   else {
      err = MPEGDEC_read_header( mps, FALSE );
      if( err ) return err;
   }
```

For example, the end of a regular file is detected there, whereupon the function immediately returns with a value that can range from -1 to -5 according to `mpegdec_lib/mpegdec.h`.

If these functions are successfully executed, the next step is to decode the next MP3 audio frame. This is done in `MPEG3_decode_frame()` which returns the number of decoded samples. If a stall in the data stream occurs, this decoding function will get noticed and return a negative value, usually -1. This value which is returned to the player application is changed to -6 to be able to distinguish it from other errors that may occur earlier.

```
    ...
    count = MPEG3_decode_frame( mps );
    break;
    default:
    break;
  }
mps->frame++;
if (count < 0) // error occured in MPEG3_decode_frame
  return(-6);  // return -6 to distinguish from other return values (defined in mpegdec.h)
return count;
```

Now that these preparations are made, it is easy to create the right exit value for these error scenario in `mp3play.c`. After the loop for decoding audio frames ends, the negative return value is checked for the special value -6 which leads to setting another flag named `error_resource`:

```
while ((pcmcount = MPEGDEC_decode_frame(mps, pcm)) >= 0) {
    writepcm(dspfd, pcm, pcmcount);
    ...
  }
if (pcmcount == -6)   // problems with resource (sd-card, nfs/samba-mount, etc.)
  error_resource = 1;
```

Exit code 4:
Finally, all remaining `exit()` calls in the player's source code are modified to return exit value 4. Hence there is only one way for `mp3play` to terminate with a success exit value.

Watchdog timer

The proposed watchdog timer is easily realized by means of the `alarm()` system call together with a signal handler for the SIGALRM signal which is generated when the timer expires.

This is the code for the signal handler:

```
void alarm_sighandler(int sig)
{
  if (init)       // error while initializing
    exit(2);
  else            // error while playing
    exit(3);
}
```

The `init` variable is set at program start and cleared before actual decoding begins. If a timeout occurs during initialization, this is treated as an error opening the MP3 stream. Some streaming servers were observed that are able to be connected to but do not provide any data. This keeps the player running without playback and is hence deprecated for the embedded web radio.

There are error scenarios that deal with stalls of data streams which cause the `mp3play` process to be blocked. Shutting down an NFS server while audio files are played from it is such a case due to reasons illustrated in section 5.4. Because the application is blocked, it can not detect this error and terminate with exit code 3. Instead, it is tried to do that with the watchdog mechanism.

At the beginning of the `main()` function, the signal handler must be installed. The timer is set to 5 seconds to detect initialization errors discussed above.

```
signal(SIGALRM, alarm_sighandler);
alarm(5);
```

The timer is reset to 5 seconds in the decoding loop after each call of `MPEGDEC_decode_frame()`:

```
while ((pcmcount = MPEGDEC_decode_frame(mps, pcm)) >= 0) {
    writepcm(dspfd, pcm, pcmcount);
    ...
    alarm(5);
}
```

Furthermore, minor changes are made in the file `http.c` which is responsible for opening Internet audio streams. With the original code, it is possible that the `openhttp()` function locks up while connecting to broken streaming servers. This problem is resolved by counting up a variable and quitting the loop at a value of 100.

6.3.3 SMB mount tool

A minor code modification is applied to the SMB mount tool regarding an error exit code. The web radio application must know if a mount attempt of an SMB share fails. However, the *smbmount* program does return the exit value 0, meaning “success”, even if mounting fails. Hence the responsible code was traced and corrected to return an exit value of 1 in this case. This is done by slightly altering the `daemonize()` function in `uClinux-dist/user/samba/source/client/smbmount.c`:

```
  /* If we get here - the child exited with some error status */
  // exit(status);
  exit(1);   // always indicate error (HK)
```

6.4 Web radio application

At this development stage, each hardware component that is necessary for the embedded web radio is supported in the uClinux kernel and ready for operation. Furthermore, among others the most important software component – the MP3 audio player – is prepared for the system’s needs. All these parts can be comfortably tested on the target system with the aid of the uClinux console.

Development of the missing software component, i.e. the web radio application, is the most time-consuming step in this phase. Besides steering the embedded system by means of the available tools and programs, creating a well arranged user interface and preparing information to be displayed on the LCD is a very important aspect of the main application.

6.4.1 Configuration

The steps for adding a new application may be gleaned in section 4.5.2. The web radio application is added at the beginning of the *Miscellaneous applications* list in the configuration file `uClinux-dist/config/config.in` and a short help text is placed in `Configure.help` in the same directory. Also, an entry for the web radio must be added to the make file in `uClinux-dist/user/` which simply includes the `webradio/` subdirectory where all code written from now on will be placed in.

The first file that is included is a make file to build the web radio executable:

```
EXEC = webradio
OBJS = webradio.o graphics.o buttons.o
all: $(EXEC)
$(EXEC): $(OBJS)
$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
romfs:
$(ROMFSINST) -e CONFIG_USER_WEBRADIO           /bin/flash-save
$(ROMFSINST) -e CONFIG_USER_WEBRADIO           /bin/flash-restore
$(ROMFSINST) -e CONFIG_USER_WEBRADIO /bin/webradio
# these files are normally written into flash and loaded from there!!!
# $(ROMFSINST) -e CONFIG_USER_WEBRADIO /etc/webradio_stations
        ...
clean:
rm -f $(EXEC) *.gdb *.elf *.o
```

Hence, graphics related code will reside in `graphics.c`, button input is treated in `buttons.c` and the web radio application itself is built in `webradio.c`. These files correspond to the 3 parts the web radio application is subdivided into. As a consequence, the following sections conform to this scheme. All source code is compiled into one executable which will simply be called `webradio`. Furthermore, the shell scripts for flash memory loading/storing from section 5.3 are included. Files added to the *romfs* target are automatically added to the ROM file system of the uClinux image. During development, configuration files are also included, eventually they will be excluded and loaded from flash memory.

The web radio application must be started automatically after uClinux has booted. This is accomplished by adding the line `webradio&` to the `uClinux-dist/vendor/Bluetechnix/CM-BF537E/rc` file to start the application in the background.

6.4.2 Graphics library

To be able to create a user interface on the LCD, basic graphics functions must be available. The first idea has been to use an existing library. The uClinux distribution for Blackfin contains the *Nano-X window system*, former known as *MicroWindows*. It is designed to provide features of a graphical windowing system to embedded systems with limited disk and RAM space [WMI].

While testing it with respect to suitability for the web radio system, some drawbacks become obvious: A lot of configuration is necessary for correct compilation, and the ordering of the source code is very confusing. The biggest drawback is that collaboration with the implemented LCD device driver seems to be very poor. Furthermore, though Nano-X is designed for embedded systems, it is still too powerful for the web radio, since it provides mouse and keyboard drivers, a client/server model similar to the *X window system* for desktop

PCs, an event model etc. As a result, it was decided to implement the graphics functions needed for the web radio device by oneself.

Basic graphics routines

As a starting point, some basic graphics functions are borrowed from a tiny test application included in the uClinux distribution (`uClinux-dist/user/lissa/lissa.c`) which by default draws Lissajous curves on a screen. However, the following functions are included and used with the web radio graphics library[10]: `draw_line()` draws any line on the screen by means of `draw_xish_line()` and `draw_yish_line()`, a rectangle can be drawn with `draw_rectangle()` or `draw_filled_rectangle()`.

The inline function `draw_pixel()` is completely rewritten because it must be adjusted exactly to the Hitachi LCD driver. For the ease of use, all graphics routines take 24 bit RGB values as input. These must be cut down to 16 bit with bitreversed color fields as seen in table 6.1. Additionally, the screen is rotated clockwise to display the content in landscape format. Rotating and bitreversed colors are able to be selected in the header file. A lookup table is used to reverse each color value:

```
inline void draw_pixel(short x, short y, int color)
{
  unsigned short red=0,green=0,blue=0;
  if ((x < 0) || (x >= 320) || (y < 0) || (y >= 240))
    return;
  red = (color & 0xFF0000) >> 16;
  green = (color & 0xFF00) >> 8;
  blue = (color & 0xFF);

  red /= 8;      /* 5 bits for red */
  if (red > 31)
    red = 31;
  green /= 4;     /* 6 bits for green */
  if (green > 63)
    green = 63;
  ...
#ifdef BITREVERSED /* colors are connected bitreversed */
#ifdef LANDSCAPE_R
  *(unsigned short *)(buffer + x*240*2 + (239-y)*2) =
    bitreversed_blue[blue] | bitreversed_green[green] | bitreversed_red[red];
  ...
```

To deal with the frame buffer device, 3 functions are implemented: In `init_framebuffer()`, the device file is opened and a memory mapping is made with this statement:

```
screen_ptr = mmap(0, 240*320*2, PROT_WRITE, MAP_PRIVATE, screen_fd, 0);
```

MAP_PRIVATE (unlike *MAP_SHARED*) specifies that the mapping is not shared among several processes. Both flags refer to the *VM_MAYSHARE* flag used in the LCD device driver [BC05]. The memory map is shown in the `/proc/maps` file in uClinux.

Closing the frame buffer device is done in the `close_framebuffer()` function by calling `munmap()` and `close()`. The third function `update_fb()` simply updates the frame buffer

[10]Actually, the web radio graphics functions will not be compiled as a Linux library, but compiled into the webradio executable.

from the internal display buffer. This feature prevents flickering of the LCD which occurs if single elements are drawn directly.

At this stage, an important feature the graphics library must have is missing yet: Font support. Therefore, 2 freely available fonts are borrowed from the Nano-X package: *Sans Serif* with a character size of 11 x 13 pixels and *System* with size 14 x 16. These are available as a C code file including a large array with ASCII[11] characters from 0x20 (space) to 0xFF ("ÿ").

The `draw_character()` function gets the character, position, color and font as arguments and returns the number of horizontal pixels the imaginary "cursor" is moved rightwards. A single pixel is placed if the corresponding font data bit is set:

```
short draw_character(short x, short y, char character, int color, short size, short font)
{
  ...
  if (font == SANSSERIF)
    {
      for (i=0; i<13; i++)     // 13 entries per character
        for (j=0; j<sansserif_width[character-0x20]; j++)
          if (((sansserif[(character-0x20)*13 + i] >> (15-j)) & 0x1) == 0x1)
            draw_pixel(x+j,y+i,color);

      return(sansserif_width[character-0x20]);
      ...
```

Building upon character drawing, a `draw_string()` function is implemented that gets a `char*` pointer as argument. Writing centered text to the screen is supported. In this case, the string is analyzed at the beginning to sum up all character widths. Then the upper left coordinates of the text are calculated. Between characters, a spacing of 1 pixel is inserted.

Now the necessary set of basic drawing functions is complete.

User interface specific functions

Graphics functions that operate at a higher level are advantageous for the web radio application. These correspond to specific elements of the screen and provide an abstraction from the coordinates-based level. To start illustration with, a screenshot is shown in figure 6.2 with the planned screen layout.

The top of the screen is filled by the respective title. It is drawn by `draw_title()`. The title string is contained within the argument of type `struct menustruct`. (The structure is defined in the header file `webradio.h`.)

The bottom area of the screen tells about the current assignments of the buttons which are located underneath the LCD. Drawing this area is done by the `draw_buttons()` function. The same structure is used as argument because the strings reside there.

For the main area of the screen, 5 functions are implemented to show different kinds of information:

[11] American Standard Code for Information Interchange

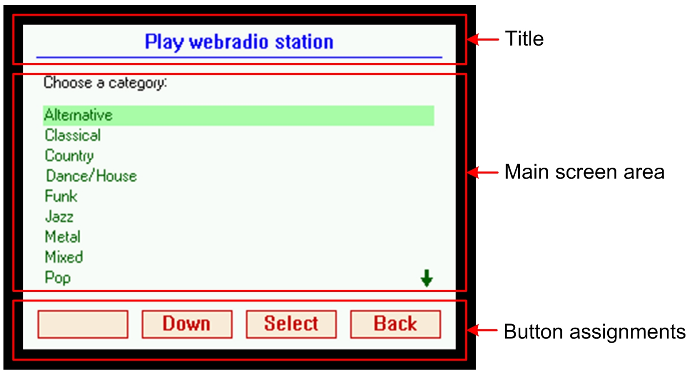


Figure 6.2: Layout of the web radio screen

Listings contain a headline for user instructions and a list of up to 9 menu items. At any time, one entry is selected, denoted by a colored rectangle. (All colors that are used for the user interface are adjusted centrally in the `graphics.h` header file.) To signal the existence of additional items, arrows can be placed upwards and/or downwards. The corresponding source code is as follows:

```
void draw_listmenu(struct menustruct *ms)
{
  short i;
  draw_filled_rectangle(0,36,319,194,COL_BG); // delete old list menu
  draw_string(15,36,ms->menutext,COL_TEXT,1,LEFT,SANSSERIF);// draw instructions
  if (ms->marked != 0)
    draw_filled_rectangle(15,60+(ms->marked-1)*15,303,74+(ms->marked-1)*15, \
                          COL_MENU_MARKED);   // draw background for marked item
  for (i=0;i<9;i++)
    draw_string(16,60+i*15,ms->item[i],COL_MENU_ITEM,1,LEFT,SANSSERIF);

  if ((ms->more & 0x1) == 0x1) // more entries downwards -> draw arrow
    {
      ...
    }
  ...
  draw_filled_rectangle(304,36,319,194,COL_BG); // delete right border
}
```

This layout will be used most often. It is taken for the screenshot in figure 6.2.

Text The function `draw_text()` simply draws 10 lines of centered text onto the screen. (Of course, there may be empty lines.) It will be used to give information to the user, for example the current network configuration parameters. The text strings are also contained in the *menustruct* structure provided as an argument.

Network address Network configuration must be done directly onto the web radio device. Hence it is necessary to have network addresses displayed in a way that makes editing by means of the buttons easy. This environment draws an IP address in the commonly used dotted quad[12] format. The digit which is actually being edited is highlighted.

[12] "192.168.1.2" is an example for dotted quad format.

This is the corresponding code:

```
void draw_netaddress(struct menustruct *ms)
{
  draw_filled_rectangle(15,36,303,194,COL_BG);    // delete previous content

  draw_string(159,50,ms->text[0],COL_TEXT,1,CENTER,SANSSERIF);   // instructions
  // highlight a digit
  draw_filled_rectangle(119+ms->netdigit*6+(ms->netdigit/3)*3, 90, \
                        119+ms->netdigit*6+(ms->netdigit/3)*3+5, 102, \
                        COL_MENU_MARKED);
  draw_string(119,90,ms->cur_addr,COL_MENU_ITEM,1,LEFT,SANSSERIF);
}
```

`ms->cur_addr` is a pointer to the actual IP address (device address, subnet mask, etc.). The *menustruct* structure is also used here.

Messages Two different functions display messages: `draw_pleasewait()` clears the main screen area and outputs the message "Please wait...". To tell the user about errors, the function `draw_error()` is implemented. Thereby, the error message itself and further instructions for the user are provided as arguments. An error is always displayed for 3 seconds and afterwards the function returns. One difference of these functions exists compared to the ones above: They update the frame buffer itself by calling `update_fb()` which is sensible because messages are always displayed immediately.

6.4.3 Button inputs

The source code files `button.[ch]` contain functions to configure the buttons' GPIO pins initially and check button states. Debouncing is supported as well as simulating repeated button presses when a button is held down.

As already stated in section 5.5, the *Programmable flags* driver is used to read inputs from the buttons. According to the schematic in figure 3.5, the character device files `/dev/pf0`, `/dev/pf1` and `/dev/pf7` are used. In the initialization function they are opened like regular files. Before button states can be read in, the GPIO pins on the Blackfin processor must be configured for it. This is done by means of the `ioctl()` system call which tells the kernel driver to set the corresponding processor registers. The necessary configuration code is shown for the `/dev/pf0` instance (simplified):

```
short init_buttons(void)
{
  pfzero = open("/dev/pf0", O_RDWR); /* open pflags device */
  ioctl(pfzero, SET_FIO_DIR, 0);     /* set as input */
  ioctl(pfzero, SET_FIO_POLAR, 1);   /* set polarity to active low/falling edge */
  ioctl(pfzero, SET_FIO_INEN, 1);    /* enable input buffer */
  return 0;
}
```

The names of the I/O control requests correspond to Blackfin's register names, e.g. SET_FIO_DIR manipulates the PORTFIO_DIR register according to [Ana05].

Description of button checking

`check_buttons()` will be called regularly in the web radio code. It returns an unsigned char which contains information about button presses, whereas the lower 4 bits are only set once to the number of the pressed button and the upper 4 bits deliver repeated presses at an interval of 200 ms if a button is held down. So for each button it is possible to choose if holding it down shall generate repeated presses[13].

The operating sequence of this function is as follows:

1. Data from the mentioned 3 device files is read in with the statement `read(pfzero,read_pfs[0],2);`. If a line is pulled to low by a button, a "1" is read, otherwise "0".
2. By means of nested *if* queries, the received data is analyzed and assigned to buttons which leads to inherent button priorities. Button 1 gets the highest priority, button 4 the lowest, e.g. if buttons 2 and 3 are pressed concurrently, only button 2 will get noticed. Furthermore, in this step the debouncing is done: The `butstat[0]` value is not set to the button number until 2 identical samples are received in consecutive function calls.
3. The third step depends on a change in button presses. `butstat[0]` contains the actual button number (or zero for no button), `butstat[1]` stores the last value. If there is no change it is checked if a repeated button press must be signaled by measuring the time since the last one:

   ```
   ...
   if (gettimeofday(&curtime,NULL) != 0)    // get current time
     perror("error in gettimeofday()\n");
   if (timeval_interval(&buttime,&curtime) >= REPEAT_INTERVAL)
     {
       butpress |= (butstat[0] << 4);    // indicate "button pressed"
       buttime = curtime;
     }
   ```

   The `timeval_interval()` function is also contained in the current source file. It calculates the absolute difference of 2 `struct timeval` variables.

   If there is a change of values, i.e. a button has been pressed, that one is stored and returned by the function.

As soon as button input is integrated into the web radio application, the serial console cannot be used anymore because the Blackfin's UART0 pins coincide with GPIOs *PF0* and *PF1* which are used for the buttons [Blu06]. In U-Boot, the parameter `console=` has to be added to the `bootargs` variable for disabling the serial console. Accessing the uClinux console is possible all the same via *Telnet*.

6.4.4 Main application

According to the proposed menu hierarchy of figure 2.2, a state machine is best suited for implementation. It builds the main mechanism of the program. With the C language, it is realized this way:

[13]It will be used for e.g. scrolling through directory listings where single button presses are unacceptable for the user.

```
while(1)                          /* infinite loop */
  {
    butpress = check_buttons();   /* poll buttons */
    switch(ms.sm)                 /* ms.sm is the state machine variable */
      {
      case some_state:            /* specific state, e.g. "main menu" */
        switch(BUT_REPEAT)        /* repeated button presses when held down, e.g. for */
          {                       /* scrolling up                                     */
          case 1:
            /* action if button 1 was pressed */
            break;
          ...
          default:
            break;
          }
        switch(BUT_ONCE)          /* single button press when held down, e.g. for */
          {                       /* stopping                                     */
          case 3:
            /* action if button 3 was pressed */
            break;
          ...
          }
        some_statement;           /* will always be executed in this state */
        break;
      case another_state:         /* specific state */
        ...
      }
    usleep(10000);                /* polling interval of buttons is 10 ms */
  }
```

At the beginning of each loop, user input via buttons is read in. Depending on the current state and the button input, different actions have to be implemented. At the end of the loop, the application sleeps for 10 ms. This corresponds to the buttons' polling interval that makes it impossible to miss a button press. The choice of employing polling instead of interrupts was taken in favor of the whole application's stability. It ensures that temporal control always remains within the main process [Kop97]. In the case of a failure of the main application, the embedded system would need to be reset manually. The only interrupt that is granted is caused by the termination of the MP3 player, but the interrupt handler function only contains 1 line of code. Another aspect of stability is concerned with memory allocating: During development, precise attention was paid to avoid memory leaks in the code.

Besides the state machine, the source code contains several functions that take smaller tasks. The accompanying header file `webradio.h` is the place to change error messages, maximum values (for URLs, filenames, etc.) and names of configuration files. Furthermore, it contains an enumeration *statemachine* for the states, a structure *listentry* which is used to build doubly-linked lists and a structure *menustruct*. The latter one is used as argument to several functions. It comprises the state machine variable, GUI specific data (marked item number, pointer to title, button names, etc.) and system data (volume, random flag, current directory, etc.).

Configuration files

The following configuration files are used to store different kinds of data essential for the application's functionality:

`/etc/webradio_stations` contains the list of web radio stations, divided into categories. The format is specified as follows: The start of a category is indicated by a line that starts with the keyword `CATEGORY` followed by a space character and the name, which must not exceed 30 characters. All stations that belong to this category come next. Each station is built up of two lines, where the first contains the name and the second the URL. Both may contain up to 100 characters.

`/etc/webradio_shares` builds the list of network shares. Each entry consists of two lines – the name is first, the URL second.

`/etc/webradio_network` contains configuration data for Ethernet. In the case DHCP is chosen, only the keyword `dhcp` is stored. With manual configuration, the data consists of 4 IP addresses in the dotted quad format: `[Device address] [Subnet mask] [Gateway] [Name server]`.

`/etc/webradio_settings`: Its only content is the volume. It may range from 0 to 100.

Basic functions

Function `configure_pf4()` configures the PF4 pin to enable the upper bank of the flash memory.

There are 4 functions that deal with listings of directories, categories and stations, and network shares: `fill_items_from_list()` fills the menu items with strings from the doubly-linked list. `clear_linked_list()` destroys such a list and frees the memory. `list_go_up()` and its counterpart `list_go_down()` are used for navigating through menu items on the screen. Page wrapping is supported for the case that a linked list contains more than 9 entries. Button assignments are also updated.

To manage the MP3 player application, 4 functions are implemented: `player_fork()` forks the *webradio* process, whereby the child executes *mp3play*:

```
switch(chldpid = vfork())       // fork the process, child becomes player
  {
  case 0:                       // child process
    execlp("mp3play","mp3play",url,NULL); // executes player with url as argument
    _exit(EXIT_FAILURE);                  // failure: exit without flushing buffered streams
    ...
```

In the parent process, `player_sighandler()` is installed as signal handler for the *SIGCHLD* signal. It only sets a flag. The main application must know if the player quits. To actively terminate *mp3play*, the `quit_player()` function is used: It sends the *SIGTERM* signal with the statement `kill(chldpid,SIGTERM);` and then waits up to 1 second. If the player does not react within this time, a "forced mode" is available. These 2 phases serve the purpose of being able to display a message to the user in between. Very important is to know the exit code of the player process. This is accomplished by `get_player_exitcode()` which calls `waitpid()`.

Volume control is done by `set_volume()` and `mute_volume()`. They use the `system()` call[14] to execute the *mixer* application.

[14]This function executes the given string as a shell command.

Advanced functions

`goto_mainmenu()`: It draws the main menu onto the screen by calling the respective graphics functions and resets the state machine variable.

There are several parsing functions that create the doubly-linked lists. The first is `make_category_list()`. It scans the stations file buffer for the `CATEGORY` keyword and collects the category names. An error code is returned if no categories can be found in the file. A code excerpt is shown with the two key C functions where `ms->actual` points to the current list element:

```
if ((pointer = strstr(pointer,"CATEGORY ")) == NULL)     // find next category-line
  ...
sscanf(pointer,"CATEGORY %30[^\n]",ms->actual->textptr); // store category string
```

`make_stations_list()` draws a list of web radio station names on the screen that belong to the chosen category. Parsing is not done here, but in the code of the state machine. The reason is that the linked list which contains the stations is not overwritten or destroyed until the category is left, so there is a single program path where the parsing is done. The application is designed so that there never exist two doubly-linked lists concurrently. An error code is returned if the linked list is empty and hence the category does not contain any stations.

`make_shares_list()` fills a linked list with the names of registered network shares. It is similar to the category parsing function. Title, instructions and buttons are updated accordingly on the screen. If no network shares were entered before, an error code is returned.

`make_dir_listing()` is a quite involved function. It is used to make a directory listing of both a network share and the SD card. The content of a directory is read in from a directory stream. The header file `<dirent.h>` must be included. The following code shows how these functions are operated within the web radio code:

```
DIR *dp;                /* directory stream var */
struct dirent *ep;      /* information about a file/directory */

dp = opendir(ms->cur_dir);   /* open directory stream */
if (dp != NULL)
  {
    while ((ep = readdir(dp)))
      {
        if (S_ISREG(fileattr->st_mode) != 0)     /* is a regular file */
          ...
        elseif (S_ISDIR(fileattr->st_mode) != 0) /* is a directory */
          ...
      }
    closedir (dp); // close directory stream
  }
else
  /* error */
```

The *dirent* structure is explained in the Linux manual, also available at [WDIa]. Within this code, the names of directories and MP3 audio files (end with *.mp3* or *.MP3*) are copied into a linked list. Because they are not sorted alphabetically, this must be done manually afterwards: The sorting is done by the `qsort` function in the C standard library. For this step, the names are temporarily copied to an array `sortarray`. As comparing function `strcasecmp()` is used which ignores the case of characters.

```
qsort(sortarray, ms->count_files, MAX_FILENAME, strcasecmp);
                              /* size of elements */
               /* count of elements */
```

The last step involves concatenating the two doubly-linked lists and inserting a "`..`" entry at the beginning. Then the list menu is drawn to the screen and the button assignments are updated.

The next function, `make_play_screen()`, only deals with screen elements. It is used for both playing web radio stations and regular files. In the former case, the title is drawn in the larger font and the URL is also displayed. Otherwise, it is only the file name. Furthermore, the current volume is displayed and buttons are updated accordingly.

`next_song()` is implemented to choose the next song to play and manage starting the audio player. It is typically called if the user skips the song currently played. Clearly, it is only available for regular files, not for streams. First, it checks the *random* flag of the *menustruct* structure. If random mode is disabled, simply the next song out of the doubly-linked list gets played. Otherwise, a pseudorandom value that is uniformly distributed is calculated with the formula

$$\texttt{randomvar} = \frac{\frac{\texttt{rand()}}{\texttt{MAX\_FILES}} \cdot \texttt{ms->count\_files}}{\frac{\texttt{RAND\_MAX}}{\texttt{MAX\_FILES}} + 1};$$

where `RAND_MAX` is the highest value `rand()` may return and `MAX_FILES` is a macro in the web radio header file which is set to 10000. Starting at the first file entry, `randomvar` steps through the doubly-linked list are made forward and this file gets played next. Then the function for starting the player is called. If this fails, an error message is shown and a directory listing is made. If this step also fails (e.g. due to the removal of the SD card), an error value is returned.

The last function is `display_netconf()`. It contains code to display information about the current network configuration. First, the file `/etc/webradio_network` is read in. In the case of manual configuration, the values in the file are used. If the IP address has been received with the DHCP protocol, it must be acquired by means of a socket:

```
int s = socket (PF_INET, SOCK_STREAM, 0);  /* creates a socket for internet protocol */
struct ifreq ifr;                          /* set up structures */
struct sockaddr_in *sin = (struct sockaddr_in *) &ifr.ifr_addr;
ifr.ifr_ifindex = 2;
ioctl (s, SIOCGIFADDR, &ifr);              /* get ip address of interface eth0 */
strncat(ms->text[5], inet_ntoa(sin->sin_addr), 15);  /* convert */
```

The structure *ifreq* for configuring Linux network devices is explained in the *netdevice* manual page [WDIb]. Members of *sockaddr_in* for handling Internet addresses may be viewed at [WREa].

The main function and the state machine

Before the loop of the state machine starts, some initialization steps have to be made:

- The PF4 output pin is configured with the function `configure_pf4()`.

- Configuration files must be restored from flash memory. This is carried out by a shell script which is simply called by `system("flash-restore");`. As long as this script executes, the web radio program is paused.
- Initialization of frame buffer and buttons is next. So, `init_framebuffer()` and `init_buttons()` are called.
- The network is set up according to the configuration file. Therefore, 2 possibilities exist: Configuration by DHCP is done by the statement `system("dhcpcd&");`. Note that this application is started in background, so the call returns immediately. Timeout occurs roughly after 1 minute (if no address was received), hence it is unreasonable for the user to wait.

  For manual configuration, the *ifconfig* command is executed in the shell according to section 5.2. The standard gateway is used to set up the default route in the system. The following code shows how the command string is prepared and finally executed:

  ```
  memset(command,'\0',65);                     /* clear memory */
  strncat(command,"route add default gw ",21); /* add command string */
  strncat(command,strtok(NULL," "),15);        /* strtok adds address */
  strncat(command," eth0 &",7);                /* finish command string */
  system(command);                             /* execute */
  ```

  Additionally, the *portmap* program is launched in the background which is needed for mounting NFSs.
- By drawing the main menu to the screen, the initialization is complete.

In the following, each state is considered. The states in the source code can easily be mapped to the ones proposed in figure 2.2. Directory listing states of SD card and network shares are combined to one state – only a flag distinguishes between them.

`m_main` – Main menu: Button 1 and 2 have functions “up” and “down” to navigate through the main options, and Button 3 selects one of them. For web radio stations, the configuration file is read in and `make_category_listing()` lists all categories. The next state is `m_wrs`. For SD memory card, it is first mounted into the local filesystem

```
system("mount -r -t vfat /dev/mmc1 /mnt");
```

and then a directory listing is made by the function `make_dir_listing()`. If no errors occur, the next state is `m_browsedir`. For network devices, the steps are similar to web radio stations: The configuration file is read in, `make_shares_list()` draws the list of available network shares onto the screen and the state variable is set to `m_share`. For network configuration, just the `display_netconf()` function is called. Next state is `m_net`.
Button 4 puts the device into off mode, after the current volume setting is written to the configuration file and the *flash-save* shell script was executed. `close_framebuffer()` switches off the LCD. Next state is `m_off`. With the current implementation, the device stays powered up completely except of the LCD. After entering off mode, it is save for the user to disconnect the power line.

`m_wrs` – Category listing of web radio stations: Buttons 1 and 2 navigate through the categories, button 3 selects one and button 4 goes back to main menu (state `m_main`). The course after selecting a station is as follows: The line where the category starts in the file buffer is searched first with `strstr()`. Then a new linked list is set up that contains all names of web radio stations within the category. Parsing is done with the `sscanf()` C library function. The following code shows the corresponding statement including the concatenation of the search string:

```
searchstring = malloc(20);
CHECK_MALLOC(searchstring);    /* quit program if malloc() fails */
memset(searchstring,'\0',20);
strcat(searchstring,"%");
sprintf(searchstring+1,"%d",MAX_STATION);
strcat(searchstring,"[^\n]");
...
sscanf(pointer,searchstring,ms.actual->textptr); /* read in MAX_STATION characters at most*/
```

Afterwards, `make_stations_list()` puts the list to screen. Next state is `m_wrs2`.

`m_wrs2` – Listing of web radio stations: Buttons 1, 2 and 4 act similar to the state described previously with the exception that button 4 calls `make_category_list()` and returns to the `m_wrs` state. Button 3 is pressed to start playback of the selected web radio station. Again, the name is searched by means of `strstr()` and `sscanf()` is then used to parse the URL which may contain 100 characters at maximum. Afterwards, the player is started with `player_fork()` and the appropriate screen is drawn with the statement `make_play_screen(&ms,url,WEBRADIO);` where `ms` is the *menustruct* structure and `WEBRADIO` is a flag. A state transition is made to `m_wrsplay`.

`m_wrsplay` – Web radio station playback: Button 1 and 2 are used to decrease and increase the volume. Pressing these leads to calls of `set_volume()` and `make_play_screen()` to update the screen. Button 3 has no assigned function in this state. Button 4 stops playback with `quit_player()`, calls `make_stations_list()` and sets the state variable to `m_wrs2`. Besides the button actions, in each loop it is checked whether *mp3play* has quit unexpectedly:

```
if (player_quit)  /* this flag is set in the SIGCHLD interrupt handler */
  {
    player_quit = 0;
    switch(get_player_exitcode())   /* analyze exit code */
      {
      case 0:  /* no error - restart playback (interruption may occur) */
        if (player_fork(&ms,url) == -1)
          ...
      case 1:  /* error with audio stream */
        draw_error(ERR_WEB_STREAM);
        ...
```

On playback errors, an appropriate error message is displayed and next state is set to `m_wrs2`.

`m_share` – Listing of network shares: Buttons 1, 2 and 4 offer the usual functions: Up, down and back. Button 3 selects a network share to which a connection shall be made. In the first step, the selected name is searched in the configuration file and its URL is determined. Then the URL is analyzed to differentiate between SMB and NFS shares

```
if (strncmp(url,"//",2) == 0)   /* check whether URL starts with slashes */
  /* it's a SMB share */
else
  /* it's an NFS share */
```

and the appropriate mount command is formed (according to section 5.4) which is executed by means of the `system()` C library function. If an error occurs, a message is displayed and the state remains unchanged. Normally, `make_dir_listing()` is called afterwards to output directory contents to the screen and the next state is `m_browsedir`.

`m_browsedir` – (SD card) directory listing: Assignment of buttons 1 and 2 is as usual. If button 3 is pressed, there are 3 different actions depending on the kind of item that is selected: If the ".." entry is chosen, a directory listing of the parent directory shall be made. So the actual directory path, contained in `ms.cur_dir`, is modified this way

```
temp = strlen(ms.cur_dir)-1;     /* get last position in string */
while (ms.cur_dir[temp] != '/')  /* last slash not yet found */
  temp--;                        /* go back one position */
ms.cur_dir[temp] = '\0';         /* overwrite slash with string end */
```

and handed over to `make_dir_listing()`. If the current directory is already the top one, the network share respectively the SD card is unmounted. In the latter case, it is returned to the main state `m_main`, otherwise `make_shares_list()` is called and the next state is `m_share`. If a directory entry is chosen (characterized by a trailing "/"), the new directory path is formed and handed over to `make_dir_listing()` like above.
In the third case a file is selected which shall be played. So, the file name is appended to the current directory path and the result is used as argument for the player. `player_fork()` and `make_play_screen()` are called and the state variable is set to `m_play`.

`m_play` – Playback: Button 1 and 2 may be used for adjusting the playback volume. Button 3 is used to skip the current audio file: First, the audio player must be terminated with `quit_player(0);`, then the next song (in alphabetic or random order) is started with `next_song()`. If there is a problem with a network share or the SD card, the player may not quit immediately. Thereby `quit_player(1);` is issued whereupon it is returned to main menu or network shares list.
Button 4 is used to stop playback. It employs the `quit_player()` function exactly as was just described. If no error happens, the current directory is listed with `make_dir_listing()` and next state is set to `m_browsedir`.
Furthermore, it is always checked if *mp3play* has quit similar to state `m_wrsplay`. If the exit code is 0, playback of the next file is started or it is gone back to the directory listing if no more files are available. In the case the exit code differs from 0, some error happened within the player which is traced back to the unavailability of audio data. Hence SD card or network share is unmounted and the next state is `m_main` respectively `m_share`.

`m_net` – Network configuration: By pressing button 1, this menu is left by calling `goto_mainmenu()`. Button 2 is pressed if Ethernet configuration has to be changed. The question "Use DHCP?" is drawn on display and the buttons are updated. The next state is `m_netconf`. Buttons 3 and 4 have no assigned functions.

`m_netconf` – DHCP choice: By pressing button 1, DHCP is chosen for network configuration. This is written into the configuration file and afterwards `system("dhcpcd");` is executed. Here, the application is not started in background, hence the user immediately finds out if the request was unsuccessful. The next state is `m_net`. If button 2 is pressed, the manual configuration is launched, i.e. instructions are shown on the screen and the state variable is set to `m_netenter`.

`m_netenter` – Entering various IP addresses: Button 1 and button 2 are used to increase and decrease the current digit. Overflows are thereby allowed and treated properly. As an example, the maximum value of the second digit in a quad may be 5 or 9, because the maximum value is 255. To refresh the IP address on the LCD, `draw_netaddress()` is used. Button 3 switches to the next digit or IP address. If a complete address was entered, a `switch()` statement is used to determined what to do next. The following code belongs to the first address, the device's own address:

```
switch (ms.netaddr)  /* one of n_ip, n_subnet, n_gateway, n_nameserver */
{
case n_ip:
  ms.text[0] = "Enter subnet mask:";   /* instruction displayed next */
  ms.netaddr = n_subnet;               /* address which comes next */
  ms.netdigit = 0;                     /* which digit to edit */
  draw_buttons(&ms);                   /* refresh buttons on screen */
  ms.cur_addr = ms.subnetmask;         /* cur_addr points to current address */
  draw_netaddress(&ms);                /* draws the address */
  break;
case n_subnet:
  ...
```

After the nameserver was entered as last address, the network is set up similarly to the initialization of the web radio application. The configuration is saved to the appropriate file.

`m_off`: In off mode, the only thing that needs to be done is to wait for any button press that wakes up the device. If one is detected, the frame buffer is initialized to start the LCD, and the `m_main` state is entered.

6.4.5 Troubles

Sometimes, the web radio application was terminated with a "bus error" that points to the use of an illegal address. These problems could never be traced back to a specific statement, and they were all gone by just deleting or adding e.g. a `printf()` statement. So it was suspected to blame the toolchain for this. As is confirmed by automated tests [WBLg], no toolchain is free of errors.

If a subdirectory has to be displayed, the string contained in the `ms.cur_dir` variable gets longer and hence more memory has to be reserved for it. This was first done with `realloc()`, but later it was found out that this function does not work correctly somehow. The solution was simply to avoid it and use `free()` and `malloc()` instead.

6.5 Web service

The source code for the web configuration service is contained in the directory `uClinux-dist/user/webradio/webapp/`. A line was added to the user applications make file so that it is automatically included if the web radio application is enabled in the uClinux configuration. Furthermore, a *Makefile* for the web application itself is generated which automatically builds the executables and installs all required files (including static components like `index.html`) in the `/home/httpd` directory of the target file system:

```
EXECS = stations password shares pwchange
OBJS = stations.o password.o shares.o pwchange.o
all: $(EXECS)
$(EXECS): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $@.o $(LDLIBS)
romfs:
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/index.html
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/tux-wink.gif
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/cgi-bin/stations.cgi
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/cgi-bin/password.cgi
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/cgi-bin/shares.cgi
	$(ROMFSINST) -e CONFIG_USER_WEBRADIO /home/httpd/cgi-bin/pwchange.cgi
	...
```

The embedded web server *boa* was already configured in section 5.4. Like the main application, it must be loaded automatically after the uClinux boot process. So, the line `boa&` is added at the end of the file `uClinux-dist/vendors/Bluetechnix/CM-BF537E/rc`.

6.5.1 The common gateway interface

For the web application, creation of dynamic web pages is mandatory. The CGI (Common Gateway Interface) is a widespread standard to facilitate this need. It is an interface specification which defines the way a web server calls CGI applications, forwards input data from the browser to these applications and returns their output to the web browser [Lou00].

A CGI program that is executed by a web server usually generates an HTML file, but may also output other data, e.g. pictures. It is important that the data is preceded by a valid header. For HTML, it consists of the string `Content-type: text/html`, followed by two new-line characters. If a CGI program needs to receive data from the browser, two possiblities exist:
With the GET method, data is appended to the URL of the CGI program which shall be called. The web server then copies it into an environment variable the program is able to read. Typically, the data length is hereby limited to 1 Kbyte.
With the POST method, the web server calls the CGI program and provides the data at its standard input port. The environment variable `CONTENT_LENGTH` is thereby set to the data length. The method has to be specified in the `<form>` tag of an HTML file:

```
<form name="someform" method="post" action="CGIprogram.cgi">
```

The latter method is selected for the web radio configuration service.

The programming language for CGI applications is not predetermined. Perl is often used for web programming, but also shell scripts as well as the C language may be used. With this project, the latter one is used. Aside from the rich set of standard C language functions, another advantage is the fast execution speed. To be able to use CGI applications with *boa*, the executables must have the file extension "`.cgi`".

Programming CGI applications is different to the web radio program: Because their execution time is very short, less attention has to be paid to memory management, because memory is automatically freed after termination. On the other hand, security is highly important, because malign user input may corrupt the system's functionality.

6.5.2 Managing web radio stations

The source file `stations.c` contains the CGI program for the management of web radio stations. It is called if the option "Manage web radio stations" is chosen from the welcome page `index.html`. Figure B.1 in the appendix shows a screenshot of the generated web page.

When the web page is first generated, a temporal copy of the `webradio_stations` configuration file is made, that is written back not until the user wishes to save the settings and provides the correct password. This file is then parsed from top to bottom and all stations are displayed in a table on the web page. Starting with the category, all entries that belong to it come afterwards. The name and URL are listed for each station.

HTML form elements

The web interface shall provide 4 functions: Delete a single station, delete a whole category, add a station (to a category), add a new category. Such inputs are easily realized with HTML forms. Buttons (HTML input type *submit*) and text boxes (input type *text*) are used here. These actions are contained within one form that calls the *stations.cgi* program again on submit.

Deleting a category or station is done by clicking a button. To be able to detect which button was pressed when the form input is processed, the names must be unique. Hence, the name of a "Delete category" button contains the corresponding configuration file line number preceded by the string "delcat":

```
printf("<input type=\"submit\" name=\"delcat%d\" value=\"Delete category\">\n",linenumber);
```

Likewise, a "Delete station" button also contains the line number and is preceded by the string "del". Hence, the decoded input will contain the expression `delcat45=Delete category` respectively `del45=Delete` for line number 45. The strings after the equal sign are the visible button contents.

On the bottom of the web page, an input field and a button is placed with which a new category can be added:

```
printf("<input name=\"catname\" type=\"text\" maxlength=\"%d\" size=\"30\">\n",MAX_CATEGORY);
printf("<input type=\"submit\" name=\"addcat\" value=\"Add category\">\n");
```

A button click produces the input expression `addcat=Add category`. As seen in this code excerpt, maximum allowed lengths for names and URLs are forced by the *maxlength* form attribute.

To add a new station, on the end of each category the necessary form elements are placed: A "name" text field, a "url" text field and a button named "add". The line number of the configuration file where the new station, more exactly its name, would be placed is appended to each of these 3 elements to make them unique within the web page. For example, if the expression `add45=Add` is contained in the input, the corresponding data can be found in `name45=...` and `url45=...` fields. (Button values are added to the input only if they are clicked, but text inputs always are present, even if they are empty.)

The last form element is a button for saving the changes. Before this is done, the user has to provide a password which is queried by the *password.cgi* CGI program treated in section 6.5.4. Hence this button belongs to another form within the web page.

Input processing

The first action of this CGI program is to read the form input from *stdin*. This data is encoded in the URL format described in [Ber94]. Hence, a simple function `decode()` was written for converting. It returns a string where single form tokens are separted by a new-line character.

```
env = getenv("CONTENT_LENGTH");    /* get POST data length */
if (env == NULL || sscanf(env,"%ld",&post_length) != 1)
  post_data = NULL; // there is no formular data   /* no POST data */
else
  { ...
    fgets(post_input, post_length+1, stdin);  /* receive data */
    decode(post_input, post_input+post_length, post_data);  /* decode data */
  }
```

The next step is to determine the function which has to be accomplished, whereupon the variable `func` is set accordingly. In the case of deletion of a category, the following queries evaluate to *true*:

```
if ((pointer = strstr(post_data, "\ndelcat")) != NULL)  /* "delcat" found */
    {                                                   /* read in the line number */
      if (sscanf(pointer, "%*[\n]delcat%d=Delete%*[ ]category%*[\n]", &number) > 0)
        {
          func = DEL_CAT;               /* set function */
          ...
```

Next, depending on the determined function, the temporal configuration file is altered. If a category has to be deleted, the file is read into a memory buffer and then rewritten: Every line of the frame buffer is copied to the file until the affected line is found. The following lines are neglected and the start of a new category is searched for. From there the file buffer is copied again till the end.

Deleting a single station is even simpler because the number of lines that have to be deleted is known before. The number of the first line is contained in the POST data. While copying, this and the following one are missed out.

If a category has to be added, the new name is first parsed from the input data:

```
pointer = strstr(post_data, "catname="); /* search form element in post data */
sscanf(pointer,"catname=%[^\n]",token);  /* read name of category */
```

Afterwards, the configuration file is opened with the `O_APPEND` flag and a line starting with the *CATEGORY* keyword is appended. An empty name or one containing quotation marks is treated as an error. An appropriate message is stored in the `errmsg_cat` variable which is printed beneath the text input field on the web page.

Adding a web radio station starts by reading its name out of the input data. Empty names or quotation marks therein are treated as errors. The variable containing the error message is `errmsg_name`. Likewise, the URL is read. It must start with "http://". Since it is passed to *mp3play* and hence is a part of the command, it is very important for security to filter out all characters that may be used for manipulation because of their special meaning in the Linux shell: Semicolons (";"), pipes ("|"), "&" characters, quotation marks (", ', ‘), redirectors ("<", ">"), braces ("()", "[]", "{}") and the dollar sign "$". Furthermore, spaces are not allowed. `errmsg_url` is the variable for error messages regarding the URL:

```
if (strstr(url,";") != NULL)
  errmsg_url = "Error: URLs must not contain a semicolon \";\"!";
```

If no error occurred before, the configuration file is read into a memory buffer and then truncated. All lines are copied until the affected line number, known from the name of the *add* input field, is reached. There, the 2 lines for the new station are inserted and then copying resumes:

```
do
  {
    linenumber++;
    if (number == linenumber) /* number contains line number of new station */
      {
        write(filedesc,name,strlen(name));   /* write new name */
        write(filedesc,"\n",1);
        write(filedesc,url,strlen(url));     /* write new URL */
        write(filedesc,"\n",1);
      }
    write(filedesc,token,strlen(token));     /* write old line */
    write(filedesc,"\n",1);
  }
while((token = strtok(NULL,"\n")) != NULL);  /* get next line */
```

After an action was successful, a message is displayed on the web page beneath the title. If an error occurred, the error message is shown beneath the according input field. After input processing and altering the configuration file is finished, the web page is created as explained above.

The web radio stations configuration file may contain up to 1100 lines. If this size is reached, no more elements can be added.

6.5.3 Managing network shares

The source code for management of network shares is contained in *shares.c*. It is called if the option “Manage network shares” is chosen on the web service welcome page. The structure is similar to the stations CGI program, but simpler because no division into categories is necessary. To decode the form input, the same function is used. Saving the settings also redirects to the *password.cgi* CGI program explained in section 6.5.4.

To build the basic web page, all network shares contained in the configuration file are presented within an HTML table. Every entry has a button for deletion. Afterwards, two text input fields and a button are provided to add a new network share. The following code creates the corresponding HTML code:

```
printf("<tr>\n");    /* table row */
...                                                 /* text inputs for name and URL */
printf("<input name=\"name\" type=\"text\" maxlength=\"%d\" size=\"50\"",MAX_SHARENAME);
printf("<input name=\"url\" type=\"text\" maxlength=\"%d\" size=\"50\"",MAX_SHAREPATH);
...
printf("<input type=\"submit\" name=\"add\" value=\"Add\">\n");  /* button */
```

The input which is sent when submitting the form may contain the following expressions: `del36=Delete` is generated by a “Delete” button. The number corresponds to the line number of the share's name in the configuration file. If the "Add" button is clicked, `add=Add` is generated. `name=...` and `url=...` contain the data thereby.

Deleting a network share is done exactly the same way as deleting a web radio station in the *stations.c* CGI program. If adding a new share, the same stringent URL filter rules have to be applied as with stations' URLs. The lines are simply appended to the existing configuration file.

The `webradio_shares` configuration file may contain up to 100 lines. These restrictions are made because in section 5.3 it was specified that configuration files are stored into the last sector of flash memory which has a size of 128 Kbyte. Due to the limitations of the flash memory driver, this size must be exceeded under no circumstances. A simple calculation proves that this is ensured by those restrictions (compressing via *gzip* is neglected because it is not predictable):

$$\underbrace{64\ Byte}_{network\ conf.} + \underbrace{\underbrace{1100}_{max.\ lines} \cdot \underbrace{101\ Byte}_{max.\ length}}_{stations\ conf.} + \underbrace{\underbrace{100}_{max.\ lines} \cdot \underbrace{101\ Byte}_{max.\ length}}_{shares\ conf.} + \underbrace{4\ Byte}_{settings} + \underbrace{11\ Byte}_{password} = 118\ Kbyte$$

6.5.4 Password protection

Password query

If the user wants to save any changes to web radio stations or network shares, the "Save settings" button redirects to the *password.cgi* CGI program. The source code is contained in `password.c`. To store the password, another configuration file is needed: `/etc/webradio_password`. It is added to the *flash-save* shell script.

The code creates a web page that only contains 2 HTML form elements, a password input field and a submit button:

```
printf("    <input name=\"pwfield\" type=\"password\" maxlength=\"10\" size=\"11\">\n");
...
printf("    <input type=\"submit\" name=\"submit\" value=\"Submit\">\n");
```

Pressing the "Submit" button sends the entered password to the CGI program. Afterwards, the password file is opened and its content is compared with the input. If the correct password was supplied, the old configuration files get overwritten by the temporal ones created by *stations.cgi* and *shares.cgi* and the shell script to save them to flash memory is called:

```
sscanf(pointer,"%*[\n]pwfield=%[^\n]",password);  /* read password from input */
filebuf = get_filedata("/etc/webradio_password"); /* read password from file */
if (strcmp(password,filebuf) == 0)                /* compare */
  {                                               /* make changes persistent */
    system("mv /etc/webradio_stations_temp /etc/webradio_stations; mv /etc/webradio_share...
    system("flash-save");                         /* save to flash memory */
    ...
```

Afterwards, the web browser is redirected to the HTML file `index_success.html` which differs from `index.html` only by the additional message "Settings were successfully saved.".

To make brute-force password attacks impossible, only 5 attempts are granted to provide the correct password before all changes are discarded. The remaining attempts are stored in the temporal file `/etc/webradio_pw_attempts`. This file gets reset each time one of the previous CGI programs is called. After discarding data, the web browser is redirected to the static HTML file `index_failure.html` which contains an error message.

Changing the password

The web configuration service is protected from unauthorized access by a password. Hence it is necessary to provide a facility to change this password. This is the purpose of `pwchange.c`. The CGI program can be launched from the welcome page by clicking the link “Change password”.

It creates a web page that contains 4 form elements, 3 of type *password* and one of type *submit*. The first password field is to fill in the old password, hence the name `oldpw`. The new password has to be supplied 2 times to the fields `newpw1` and `newpw2`. The button has the simple name `submit`. Their order within the input data string is fixed, hence the entered strings are read in by one function call of `sscanf()`:

```
sscanf(post_data,"%*[\n]oldpw=%[^\n]%*[\n]newpw1=%[^\n]%*[\n]newpw2=%[^\n]%*[\n]submit=Chang
e%*[ ]password%*[\n]",oldpw,pw1,pw2);
```

If the correct password is entered and the fields with the new password do not differ, it is stored into the password file and the web browser is redirected to the static HTML file `index_pw.html` which is identical to the welcome page except of the message “Password was successfully changed”.

Providing erroneous input leads to the same web page for password changing, including an error message.

Chapter 7

Modifications to the hardware

The hardware components that build the web radio device can not be used with the setup shown in table 3.1 without modifications. Some of these arose from improvements of hardware components, others became necessary during the software development phase because run-time conflicts between peripheral devices appeared.

Appendix A contains a table denoting the assignments of the Blackfin's hardware pins to various peripheral components.

7.1 Display

For improvement of hardware, an AND gate is soldered onto the EXT-BF5xx-Camera board. The goal is to be able to create the *DTMG* (Data timing) signal for the LCD in a more easy way. As stated in section 6.1.2, this signal must pause during the horizontal blanking time. If it would be connected to a single Blackfin timer output pin, the timer would have to be stopped and started precisely all the time the display is on. By adding the AND gate, two timers are used instead which need only be configured and started once.

The direct connection of the LCD's *DTMG* pin to the Blackfin's *PPI1Sy2* must be cut before. The AND gate is then soldered to have pin number 54 (TMR3) and pin number 42 (TMR1/PPI1Sy2) of the ADSP-BF537 as inputs and pin number 8 (DTMG) of the LCD as output. Figure 7.1 contains a schematic including the timer signals on each pin. The schematic of the affected board is available at [WBLj].

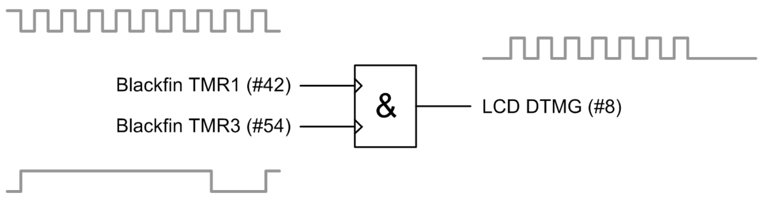


Figure 7.1: The AND gate soldered onto the EXT-BF5xx-Camera board

The second hardware modification is concerned with the dot clock pin of the LCD. By default, it is connected to the Blackfin's pin number 7 which may act as GPIO pin *PF4* or as timer output pin *TMR5*. This causes a problem because *PF4* is used internally on the CM-BF537E core module to switch between the lower and upper flash memory banks. Hence, it cannot be used to generate timing signals. To solve this issue, the pins with numbers 7 (PF4) and 40 (TMR7) are interconnected by soldering a wire onto the EXT-BF5xx-Camera board. This does not affect the core module, because that pin is not even leaded out to the expansion connector. The dot clock signal for the LCD can now be generated by timer 7.

Another hardware line has to be cut due to a conflict between the SD card connector and the LCD. The timer 7 output pin used for dot clock generation (pin number 40) may be used to detect the insertion of an SD card onto the EVAL-BF5xx board and is hence connected to pin number 9 of the SD slot. The chip detect feature is not used with the web radio device. However, as long as a memory card is inserted, this line is pulled to low and disables the correct output of the dot clock signal. To make both peripherals work simultaneously, the line was cut on the evaluation board. Its schematic is available at [WBLj].

7.2 Audio chip

The EXT-BF5xx-Audio board containing the AD1836A audio chip is located on top of the camera extender board with the setup shown in figure 3.1. After development of the LCD device driver was completed, it was noticed that audio playback stops a few seconds after enabling the display. With the aid of an oscilloscope, this strange behavior was traced back to an obvious disturbance of the audio chip by the signals generated for the LCD: As soon as the display device driver was started, the audio chip stopped to generate the *TSCLK0* (Transmit Serial Clock) signal which is needed by the Blackfin's SPORT0 to transmit digital audio data.

The exact cause of the problem was not known, but it was guessed that the signals on the LCD hardware lines somehow influence the audio chip. A drastic measure was taken to solve this problem: All pins that are not used on the audio board or on the experimental board containing the buttons were cut from the expansion connectors on the underside of the audio board. This way, no hardware lines connected to the LCD are leaded onto the audio board. Table 7.1 lists all pin numbers that were left connected.

Furthermore, the pull-down resistors on the 3 SPI lines MISO (Master In, Slave out), MOSI (Master Out, Slave In) and SPICLK (SPI Clock) were removed from the audio board. This was specially important for the MISO line, since it has a pull-up resistor on the evaluation board which would result in a voltage divider holding the line at a voltage level between 0 and 3.3 V. The pull-downs on the other 2 lines are not necessary either because SPI lines are usually treated as active low. The Blackfin processor takes care to output a high voltage level on these lines if there is no data transfer.

The schematic of the audio board is available at [WBLj].

7.3 Buttons

Pin number 19 (PF7) of the Blackfin processor is used for button input. All lines connected to buttons need pull-up resistors because pushing a button connects them to GND and hence

Pin no.	*Function*	*Pin no.*	*Function*
1	SPORT0	38	button
2	SPORT0	51	ground
3	SPORT0	52	ground
4	SPORT0	53	SPI
9	power	57	SPORT0
10	power	58	SPORT0
19	button	59	SPORT0
23	button	60	SPORT0
24	SPI	73	ground
25	SPI	79	ground
27	ground	99	reset
37	SPI		

Table 7.1: Hardware pins connected to the EXT-BF5xx-Audio board

"0" is read. However, on the camera board the LCD is connected to, the pull-down resistor *R5* is mounted for the affected line. For proper function of the buttons, this resistor has to be removed.

When the 4 boards used for the web radio device were first used altogether, the U-Boot bootloader did not start up anymore. Since any combination of 3 boards worked, it was guessed that for some hardware connections – probably the data and address bus lines – the capacity due to the increased length of signal lines became a serious problem. To solve this issue, the same drastic measure was taken as with the audio board in the previous chapter. (Indeed, the problem just described appeared first, so the experimental board was modified before the audio board.) All pins were disconnected at the expansion connector of the EXT-BF5xx-Experimental board, except of the 3 lines connected to the buttons, a power line and a ground line.

Chapter 8

Testing and final work

This chapter describes the last phase of the development process of the embedded web radio device. To ensure proper operation, several tests are conducted: Starting with software testing, possible error scenarios that may happen in the environment and influence the device are run through next. An example scenario is removing the network cable during playback of a web radio station. The device must not fail in these cases. Afterwards, security of the device within the network is checked.

The last step involves creating the final uClinux configuration and deploying the image on the target.

8.1 Software testing

The LCD device driver was not subject of any software tests. Since the code of this driver only deals with setting up the needed components of the Blackfin processor, actually no code is executed while the display is switched on. Correct functionality has already been tested during development of the web radio application. No malfunction or degradation of service regarding the LCD was observed and hence the device driver is proved to be technically mature.

8.1.1 SD card device driver

The device driver for secure digital cards has to be tested for stability to ensure playback without interruption. Each test is conducted with several SD cards with different memory sizes and by different vendors.

First, the correctness of read data is proven: Therefore, an SD card is formatted with the VFAT file system and filled with one or more files that contain random data and allocate the whole available disk space. The files are then copied onto the workstation that is connected to the target to be able to compare the read data later. On a standard Linux workstation that is capable of writing SD cards, these tasks are easily accomplished by the commands

```
mount -t vfat /dev/sda1 /mnt/sdcard
cat /dev/urandom > /mnt/sdcard/testfile1
cp /mnt/sdcard/testfile1 /home/nfs_share/
```

On the target platform, the SD card is afterwards inserted and mounted with the same command used in the web radio application printed in section 6.4. Furthermore, the shared NFS directory `/home/nfs_share/` of the workstation is mounted in uClinux according to section 5.4. Now, the content of the SD card is copied to that directory and is afterwards compared to the original file (on the workstation):

```
diff /home/nfs_share/testfile1 /home/nfs_share/testfile_sd
```

Since the `diff` command does not output anything, the files are identical and hence the read test has been successful.

The second test that is conducted involves audio playback from SD card with *mp3play* while permanently changing the volume of the AD1836A audio chip. Because both functions need transfers over SPI, this is a critical scenario for the SD card device driver. It must not get disturbed by other SPI transfers.

For this test, an SD card is filled with MP3 audio files with a Linux workstation. The memory card is then inserted into the web radio device and mounted in uClinux. The audio player is started as a background process:

```
mp3play *.mp3 &
```

With the following shell command, the volume of the audio chip is changed every second by means of the *mixer* command until it is interrupted by *Ctrl–C*:

```
while true; do mixer vol 100; sleep 1; mixer vol 50; sleep 1; done
```

No failure has been observed during this test. The SD card device driver prints errors to the kernel log which can be viewed by the command `dmesg`. To be able to enter this command during the test, a second *Telnet* connection is made to the target. Some errors were contained within the kernel log which were obviously related to the concurrent use of the SPI. However, the device driver was able to recover from all these errors by repeating the requests to the memory card so that a reading error was never reported by *mp3play*.

Due to the successful conduction of these tests the enhancements of the SD card device driver are proven to be effective to provide stable read access to a memory card.

8.1.2 Web radio application

To eliminate programming errors within the web radio application, *white-box testing* is conducted. The goal of this technique is to verify all possible execution paths in the program [Mac05]. Thereby *condition coverage*, also known as C1 coverage, shall be achieved. Test data is specially contrived to exercise the code.

Most execution paths of the main application are executed during normal operation via the buttons, i.e. navigating through the various functions. No failures of the application were detected thereby. The focus of this test lies in executing those program paths that are executed only if a specific error occurs, e.g. the audio player cannot be started. These cases are very rare and have not been tested up to now. In the following, code excerpts of the state machine are shown where deviation of execution flow starts in the case of an error. A test case is created for each. Via these tests two programming errors could be corrected within the web radio application.

Creating the list of categories is done with `make_category_list()`. If it cannot be generated, the function returns “1” indicating an error:

```
if (make_category_list(&ms,filebuf))   /* display list of categories */
  {
    draw_error(ERR_CATEGORY);          /* error +/
    ...
```

The corresponding test case was to empty the configuration file `webradio_stations`, because no categories could be found then. Correctly, an error message was shown and the system returned to the main menu.

Mounting an SD memory card is accomplished by a `system()` call. The *mount* command returns an error code if mounting fails:

```
if (system("mount -r -t vfat /dev/mmc1 /mnt")) /* mount sd-card */
  {
    draw_error(ERR_SD);                        /* error */
    ...
```

The test case was not to insert a memory card into the SD card slot, whereupon an error message was displayed and the main menu was shown.
To test the correctness of code that treats errors while mounting SMB or NFS network shares, test cases were created by adding non-existent URLs to the configuration file `webradio_shares`.

The function `make_dir_listing()` for generating a directory listing returns “1” if the directory stream could not be opened. In the state machine, the error code is checked:

```
if (make_dir_listing(&ms))   /* lists directory on screen */
  {
    draw_error(ERR_SD_LIST); /* error */
    ...
```

Since the described error case did never happen, the code of `make_dir_listing()` was expanded to generate random errors to be able to simulate errors and execute the corresponding code:

```
int randomvar = rand();                    /* get random number */
if (randomvar < (RAND_MAX / 2)) return 1;   /* simulate error rate 50% */
```

In the case of a (simulated) error, a corresponding error message was shown on the LCD and the system returned to the previous state correctly.

To generate the list of network shares, `make_shares_list()` is used. This functions returns “1” if it fails. In this case, the following code is executed:

```
if (make_shares_list(&ms,filebuf))    /* make list of shares */
  {
    draw_error(ERR_NOSHARES);         /* error */
    ...
```

The corresponding test case is to empty the configuration file `webradio_shares`, whereupon no shares can be found. The web radio application displayed an appropriate error message and returned to the main menu. Hence this test was successful.

Getting the network configuration with DHCP may fail due to several reasons and hence the *dhcpcd* return value is checked:

```
if (system("dhcpcd"))      /* execute dhcpcd */
  {
    draw_error(ERR_DHCP);  /* error - got no address */
    ...
```

If an error occurs, a value different from zero is returned. To test the corresponding code, the network cable was disconnected before selecting DHCP configuration, so that the request must fail. An appropriate error message was shown as correct response.

The code for parsing web radio stations within a specific category contains 3 paths that deviate from normal execution flow: In the first case, after finding the category start in the configuration file, no newline can be found which means that the category is empty:

```
cat_start = strstr(cat_start,"\n");   /* search for next newline */
if (cat_start == NULL)
  {
    err=1;       /* no newline found -> set error flag */
    ...
```

The test case is generated by adding the line `CATEGORY testcategory` without a newline character at the end of configuration file `webradio_stations`.
In the second test case, the newline character is present but the file ends thereafter, which leads to the execution of the following code:

```
if (!err) cat_start++;          /* this is the start for web radio station entries */
if (!err && (cat_start >= filebuf+filebuflength))
{
  err=1;          /* beyond end of filebuf - set error flag */
  ...
```

In the third case, the program searches for web radio stations within a category. It must stop as soon as the next line containing the `CATEGORY` keyword is reached. An error occurs if at this point the doubly-linked list containing the station names is empty and hence the category is empty:

```
if (strstr(ms.actual->textptr,"CATEGORY") != NULL) /* oops! that's already next category */
  {
    if (ms.actual->prev == NULL)
      {                          /* this category has no entries */
        free(ms.actual);
        ...
```

The error is not detected in the code listed above, but by the `make_stations_list()` function which returns “1” if the linked list is empty:

```
if (make_stations_list(&ms))
  {
    draw_error(ERR_STATIONS); /* error happened - no stations */
```

Successfully, each of the three error cases leaded to an error message and the system returned to the list of web radio categories.

The audio player *mp3play* is started in the function `player_fork()` that may fail together with the `vfork()` system call. In this case the error value “-1” is returned which leads to an alternative execution flow:

```
if (player_fork(&ms,url) == -1)   /* starts player */
  {
    draw_error(ERR_PLAYER);       /* error happened */
    ...
```

Because generating a corresponding test case would have been too complicated, the function was extended to generate random errors for testing purposes similar to `make_dir_listing()`. The test was successful since the system displayed an error message and returned to the previous submenu.

The function `next_song()` tries to start the audio player with the selected song. If this fails, it tries to create a directory listing. Not until this fails too, the error value “1” is returned. Because both functions for starting the audio player and listing a directory randomly return simulated errors, all execution paths were executed once to prove their correctness. Furthermore, the alternative execution flow after `next_song()` is called within the state machine could be tested successfully:

```
if (next_song(&ms,url))
  { ...                       /* error occured */
    if (ms.sdcard == 0)
      {                       /* back from network share */
        make_shares_list(&ms,filebuf);
        ms.sm = m_share;      /* next state */
      }
    else                      /* back from sd card */
      goto_mainmenu(&ms);
    ...
```

8.2 Environmental error scenarios

The next step within the testing phase is to analyze the system’s behavior if an error occurs in the environment of the embedded web radio that directly influences its operation. It is important that the system detects these errors and reacts to them by displaying an appropriate error message and bringing itself in a consistent state. In the following, the error scenarios are explained and for each one the system’s behavior is described.

Loss of Internet connection

In the case the web radio device plays an audio stream from the Internet and the connection becomes unavailable, the audio player immediately terminates and the main application displays an error message on the LCD. This was tested by simply disconnecting the network cable from the system.

Removal of SD memory card

Another error scenario is triggered by the removal of the SD card from the slot while the device plays audio files from it. The detection of this fault lasts for up to 8 seconds. Afterwards an appropriate error message is displayed and the system returns to the main menu, ready for operation.

Loss of an NFS connection

If the web radio device currently plays audio files from an NFS network share, a loss of this connection causes the audio player to be blocked for some time due to reasons explained in section 5.4. Hence, the user has to wait until the audio player has terminated before normal operation can continue.

If the device providing the NFS network share respectively the NFS service running on it is shut down, playback stops after a few seconds due to the lack of audio data. While it is tried to fetch more data via NFS, it lasts up to 70 seconds until an NFS timeout is reached, the player is terminated and the web radio device is ready for use again. The same response is observed when the network cable is disconnected during playback.

In the case the user presses the "Stop" button after playback has stopped, it only lasts 10 seconds from this point of time until the device is ready for use again. A message is shown asking the user to wait in the meanwhile.

Loss of an SMB connection

Behavior of the SMB protocol differs from NFS regarding error scenarios. If the device providing the SMB network share respectively the SMB service running on it is shut down during playback, the web radio device immediately returns to normal operation after displaying an appropriate error message to the user.

If the network cable is disconnected, the user has to wait up to 35 seconds until the audio player is terminated and the device is ready for normal operation again. An error message followed by a "Please wait" message is shown on the display.

Since in no case the web radio device becomes unstable or crashes, this test is completed successfully.

8.3 Network security

Nessus is a professional open source security scanner that is able to check devices connected to a network regarding security vulnerabilities. It is used to scan routers, firewalls, servers and gateways from outside for known vulnerabilities and possible points of attack [Uma04].

The web radio device is checked by the Nessus security scanner to ensure that uClinux and running server components do not suffer from known security vulnerabilities. Before this test, the *inetd* daemon was removed from the file `uClinux-dist/vendors/Bluetechnix/CM-BF537E/rc`. It launches *telnetd* as soon as a Telnet connection is made to the target, but finally this access must be removed anyway.

Nessus is launched onto the Linux workstation. After entering the target's IP address, the scan is started. Two open ports were detected onto the web radio device: The *sunrpc* port (111/TCP) is opened by the *portmap* program which is needed to mount NFS network shares. No security-related issues regarding this port were displayed by Nessus.

The second open port is *http* (80/TCP) which is opened by *boa*, the embedded web server providing the configuration web service. Regarding this port, the following informations were given by Nessus:

```
Problems regarding: http (80/tcp)

Security note:
A web server is running on this port

The remote web server type is:
Boa/0.94.14rc21

The following directories were discovered:
/cgi-bin

While this is not, in and of itself, a bug, you should manually
inspect these directories to ensure that they are in compliance
with company security standards
```

Hence, no security vulnerability was found by Nessus, though "CGI scanning" was enabled during the scan. The CGI programs contained in the mentioned directory were developed with security in mind. Measures that were taken to avoid abuse were already described in section 6.5.

8.4 Final work

Before the web radio device can be used as a stand-alone embedded system, some steps have to be accomplished:

First, the uClinux configuration is cleaned up, i.e. all applications that are not needed for proper operation are removed. This makes the web radio image smaller that will be written to flash memory and thereby allows faster decompressing at boot and hence faster boot time.

Table 8.1 contains a list of uClinux tools that are necessary for the web radio device.

All other commands are removed from the corresponding submenu in the uClinux configuration dialog, including *inetd* and *telnetd* which were very useful during development. However, these are not required for operation of the web radio device and would even impose a large security risk because everyone connected to the LAN may login to the target. The complete configuration of the uClinux kernel and the user space applications is listed in appendix C.

By means of this tight configuration it is possible to decrease the size of the uClinux kernel and filesystem image from 2 MByte to 1.63 MByte.

Finally, this image has to be deployed onto the target. As soon as the image resides in flash memory, the U-Boot bootloader has to be configured to automatically boot from there. Both

cat	mixer
cp	mount
cut	mp3play
dhcpcd	mv
flash_erase	portmap
grep	ps
gzip	route
ifconfig	smbmount
kill	tar
mkdir	umount

Table 8.1: Required uClinux tools

steps are described in section 4.4.4. To make the boot process as fast as possible, the *bootdelay* environment variable in U-Boot is set to zero so that uClinux is started immediately after U-Boot. However, this way the U-Boot console is disabled and cannot be reached anymore with *kermit* because the boot process of uClinux is unable to be interrupted with *Ctrl–C*. To accomplish this again, JTAG flash programmer hardware would be necessary to delete that flash memory sector containing the U-Boot environment.

Chapter 9

Conclusion

As the final part of this document, a summary of the work is presented here. The aim of the work was to develop an embedded web radio. Its features can be summarized by 3 points: Playback of web radio stations, of audio files from the local network, and from an SD memory card. Combining this functionality into a single stand-alone device is new and not offered by any commercial product at the time of this writing. Furthermore, the device provides a concise graphical user interface and a web service for easy configuration with a web browser.

Basic decisions were made in the requirements analysis phase: The MP3 codec is supported by the device, because it is the most widespread one. Wired Ethernet is used for the Internet connection. A target hardware platform based on the Blackfin processor ADSP-BF537 was chosen to be suited for the web radio device, besides the AD1836A audio chip and a color TFT-LCD – both high quality hardware parts that make the device prepared for future enhancements.

By comparing the Linux derivative *uClinux*, which is also available for the Blackfin architecture, to the *BLACKSheep* VDK developed at the Institute of Computer Technology the former was found more suited for the embedded web radio, because several software components were already included that were inevitable for the project.

Software development was the main phase of this project. The start was made by implementing the device driver for the display which provides a Linux frame buffer device in the system. The Blackfin processor itself is capable of interfacing the color display, hence no additional hardware controller is necessary. The LCD device driver will be integrated into the official Blackfin/uClinux distribution. The next step was to enhance the device driver for the SD memory card. It now provides stable read access.

Development continued with user space applications: The available MP3 audio player of uClinux was slightly modified to provide meaningful exit codes. The main control application was implemented from scratch, which is built as a state machine. The states correspond to different functions the embedded web radio is capable of. Furthermore, a lightweight graphics library was implemented as add-on for the web radio application. The web application for configuration of the device was the last one implemented. It consists of CGI programs and is offered on the network by an embedded web server included in the uClinux distribution. The application allows configuration of web radio stations and network shares and offers password protection.

The aims of the work as stated in the introductory chapter were clearly fulfilled. The proof is the full-functional embedded web radio device that conforms to the requirements worked out in section 2.2. A picture of the device during operation is shown in figure 9.1. Furthermore, customized hardware was developed at the Institute of Computer Technology that comprises all hardware necessary for the embedded web radio onto a single board. Figure 9.2 shows a picture of it. Both devices were actually shown at the *Embedded World 2007* exhibition in Nuremberg (Germany) in February 2007.

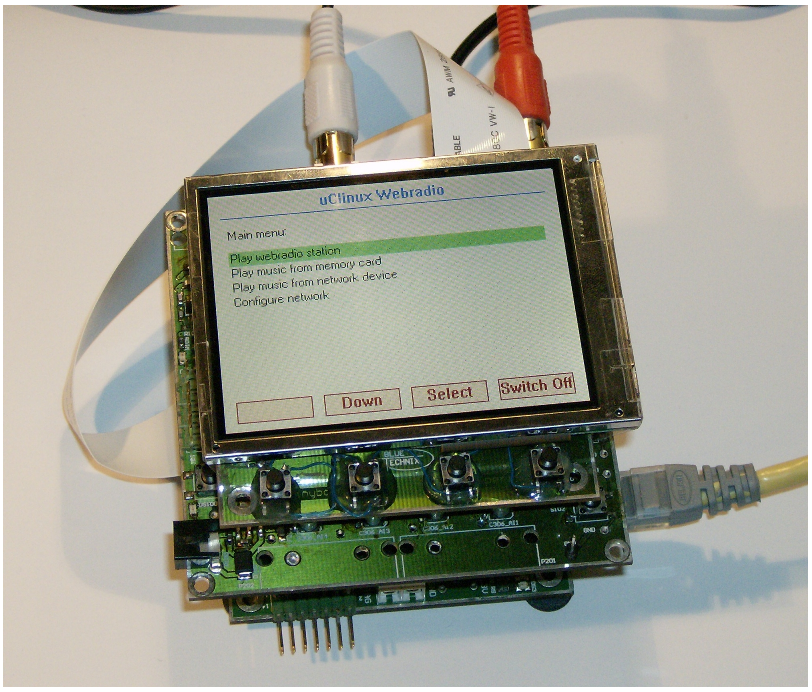


Figure 9.1: Picture of the embedded web radio (development hardware)

The following conclusions of the work are drawn:

- The advantage of Linux in general is that it is free of royalties because it is open source software. Everyone has access to the complete source code which is of high importance when developing embedded systems.
- uClinux is a fully functional operating system which makes software development for embedded systems much easier.
- A wide range of drivers, protocols and user space applications are available with the uClinux distribution.
- uClinux is very modular like known from standard Linux. It is very easy to slim a uClinux image by removing device drivers, tools and applications that are not needed.
- The version of uClinux ported to the Blackfin processor that was used for the embedded web radio was very stable, crashes of the kernel did never happen. However, a few bugs were detected in important applications or tools. For example, the *dhcpcd* tool for network configuration with DHCP suffered from a programming error that leaded to a failure if a custom timeout value was applied. The conclusion thereof is that

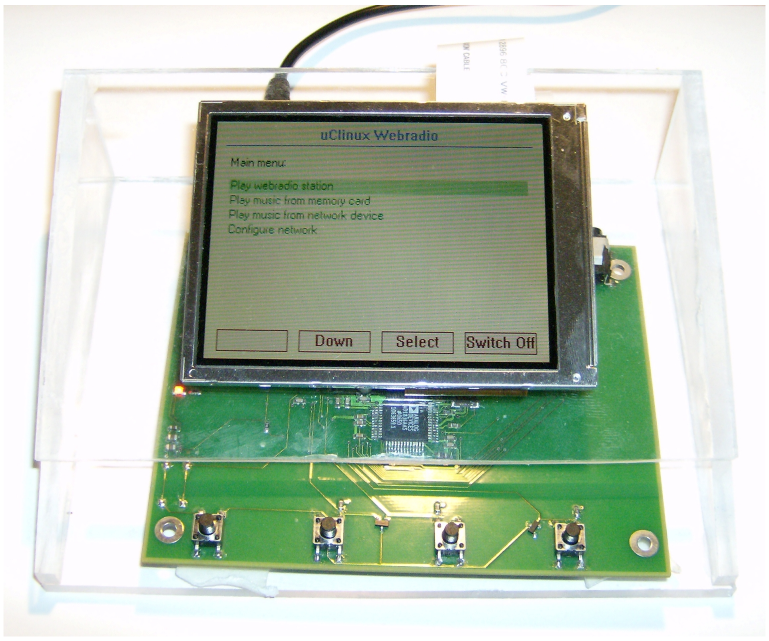


Figure 9.2: Picture of the embedded web radio (customized hardware)

thoroughly testing of each application or kernel module is a must before it is employed in an embedded system.
For the future, uClinux for the Blackfin processor including the applications is expected to become more and more stable. Development activity in the community is very high. At the end of this writing in April 2007, the 2007R1-RC3 release of the Blackfin uClinux distribution was released which is said of by the main developers that "it is stable enough to develop, evaluate and use in products".

- Documentation of Blackfin/uClinux is good and is steadily growing.
- Developing an embedded system always involves considering hardware and software together. This is even more valid as soon as some failures occur. With the embedded web radio, both hardware conflicts were first thought of software programming errors and hardware was blamed for failures caused by software. Getting out of such difficulties was arduous sometimes but very educational.

Finally, some ideas for future enhancements of the embedded web radio are presented:

- An alarm clock function and a sleep mode would be great features for the device. They are included in many traditional radio receivers and would make sense here, too.
- The random playback mode could be enhanced to support selecting files from a directory including its subdirectories. Also, a repeat function is common with audio playback devices.
- Support for more audio codecs could be added.
- The web service could be enhanced to allow moving and sorting of web radio stations and categories.

- The support of playlists would be a great extension for the web service.
- A large additional feature would be the parsing of web radio stations out of publicly accessible web sites like `www.shoutcast.com`.
- Resetting the password of the web service on the device itself should be possible.
- Support for picture compression codec like JPEG could be added, so that photos can be displayed on the LCD during playback.
- The graphical user interface presented on the display is artless up to now. Enhancements regarding graphical design are possible.

Appendix A

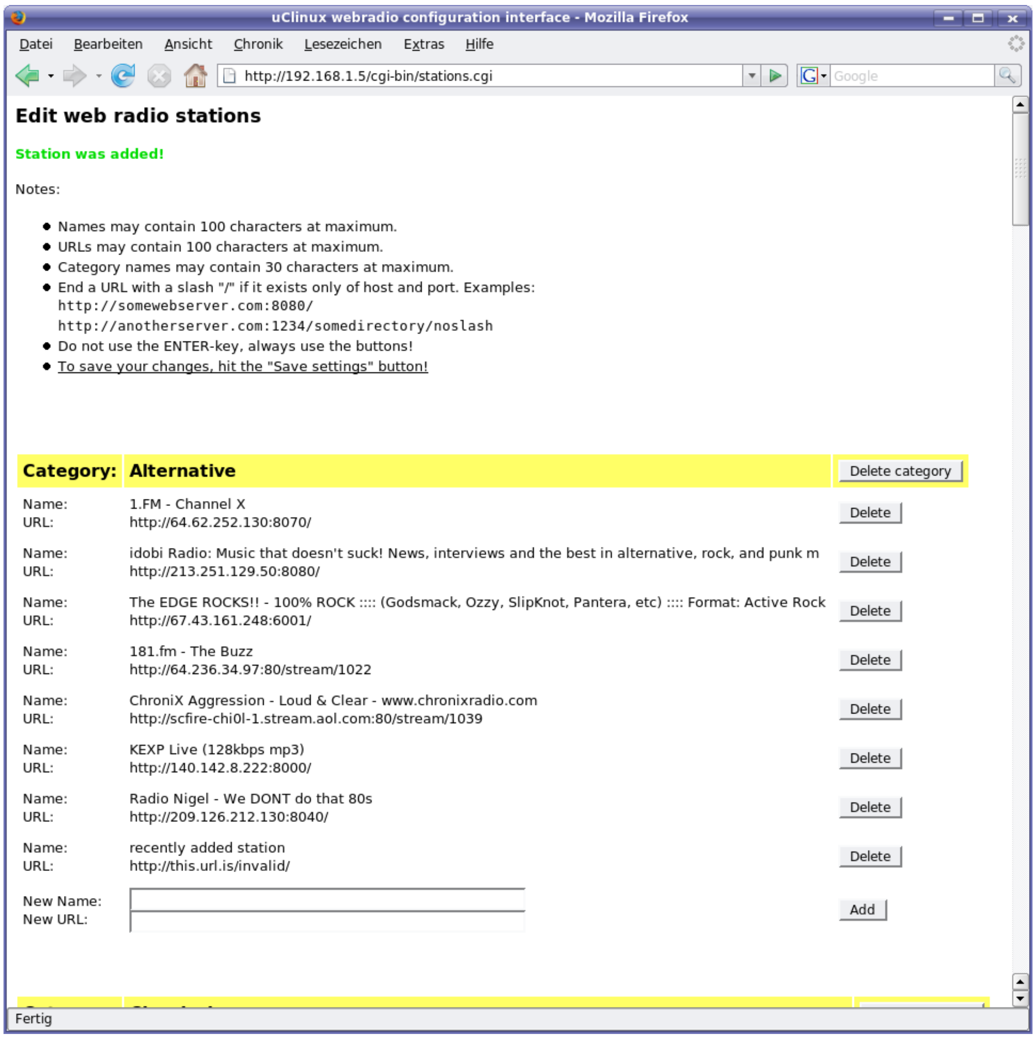
Hardware pins assignment

CM-BF537E	AD1836A	Ethernet	SD card slot	TFT-LCD	Buttons
1 (RSCLK0)	43 (ABCLK)				
2 (DR0PRI)	47 (ASDATA1)				
3 (TSCLK0)	37 (DBCLK)				
4 (DT0PRI)	38 (DSDATA1)				
11 (PPI1D0)				12 (R5) 18 (R0)	
12 (PPI1D2)				14 (R3)	
13 (PPI1D4)				17 (R1)	
14 (PPI1D6)				21 (G4)	
15 (PPI1D8)				24 (G2)	
16 (PPI1D10)				26 (G0)	
17 (PPI1D12)				29 (B4)	
18 (PPI1D14)				32 (B2)	
19 (PF7)					1, 2
20 (TMR0)				6 (HSYNC)	
22 (TMR6)				36 (PWM)	
23 (PF1)					1, 3
24 (MOSI)	2 (CData)		2 (CMD)		
25 (SPICLK)	51 (CCLK)		5 (CLK)		
37 (MISO)	49 (COUT)		7 (DAT0)		
38 (PF0)					1, 4
39 (PF14)				35 (PCI)	
40 (TMR7)				4 (DCLK)	
41 (PPI1Clk)				4 (DCLK)	
42 (TMR1)				*8 (DTMG) via AND gate*	
43 (PPI1D15)				33 (B1)	
44 (PPI1D13)				30 (B3)	
45 (PPI1D11)				28 (B5) 34 (B0)	
46 (PPI1D9)				25 (G1)	

CM-BF537E	AD1836A	Ethernet	SD card slot	TFT-LCD	Buttons
47 (PPI1D7)				22 (G3)	
48 (PPI1D5)				20 (G5)	
49 (PPI1D3)				16 (R2)	
50 (PPI1D1)				13 (R4)	
53 (SPI_CS5)	50 (CLATCH)				
54 (TMR3)				*8 (DTMG) via AND gate*	
55 (SPI_CS1)			1 (CD/DAT3)		
57 (DT0SEC)	41 (DSDATA2)				
58 (TFS0)	36 (DLRCLK)				
59 (DR0SEC)	48 (ASDATA2)				
60 (RFS0)	44 (ALRCLK)				
74 (RX+)		3			
75 (RX-)		6			
99 (RESET)	3 (PD/RST)				
107 (TX-)		2			
108 (TX+)		1			

Appendix B

Screenshot of the web configuration interface


uClinux webradio configuration interface - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://192.168.1.5/cgi-bin/stations.cgi
Google
Fertig

Edit web radio stations

Station was added!

Notes:

- Names may contain 100 characters at maximum.
- URLs may contain 100 characters at maximum.
- Category names may contain 30 characters at maximum.
- End a URL with a slash "/" if it exists only of host and port. Examples:
  `http://somewebserver.com:8080/`
  `http://anotherserver.com:1234/somedirectory/noslash`
- Do not use the ENTER-key, always use the buttons!
- To save your changes, hit the "Save settings" button!

Category:	**Alternative**	Delete category
Name: URL:	1.FM - Channel X http://64.62.252.130:8070/	Delete
Name: URL:	idobi Radio: Music that doesn't suck! News, interviews and the best in alternative, rock, and punk m http://213.251.129.50:8080/	Delete
Name: URL:	The EDGE ROCKS!! - 100% ROCK :::: (Godsmack, Ozzy, SlipKnot, Pantera, etc) :::: Format: Active Rock http://67.43.161.248:6001/	Delete
Name: URL:	181.fm - The Buzz http://64.236.34.97:80/stream/1022	Delete
Name: URL:	ChroniX Aggression - Loud & Clear - www.chronixradio.com http://scfire-chi0l-1.stream.aol.com:80/stream/1039	Delete
Name: URL:	KEXP Live (128kbps mp3) http://140.142.8.222:8000/	Delete
Name: URL:	Radio Nigel - We DONT do that 80s http://209.126.212.130:8040/	Delete
Name: URL:	recently added station http://this.url.is/invalid/	Delete
New Name: New URL:		Add

Figure B.1: Screenshot of the web configuration interface

Appendix C

Complete uClinux configuration

uClinux kernel configuration

```
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.16.27
# Thu Apr 12 15:01:57 2007
#
CONFIG_RWSEM_GENERIC_SPINLOCK=y
CONFIG_BFIN=y
CONFIG_SEMAPHORE_SLEEPERS=y
CONFIG_UCLINUX=y
CONFIG_FORCE_MAX_ZONEORDER=14
CONFIG_GENERIC_CALIBRATE_DELAY=y

#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y
CONFIG_BROKEN_ON_SMP=y
CONFIG_INIT_ENV_ARG_LIMIT=32

#
# General setup
#
CONFIG_LOCALVERSION=""
CONFIG_INITRAMFS_SOURCE=""
CONFIG_UID16=y
CONFIG_EMBEDDED=y
CONFIG_PRINTK=y
CONFIG_BUG=y
CONFIG_BASE_FULL=y
CONFIG_FUTEX=y
CONFIG_EPOLL=y
CONFIG_CC_ALIGN_FUNCTIONS=0
CONFIG_CC_ALIGN_LABELS=0
CONFIG_CC_ALIGN_LOOPS=0
CONFIG_CC_ALIGN_JUMPS=0
CONFIG_SLAB=y
CONFIG_TINY_SHMEM=y
CONFIG_BASE_SMALL=0
CONFIG_OBSOLETE_INTERMODULE=y
```

```
#
# IO Schedulers
#
CONFIG_IOSCHED_NOOP=y
CONFIG_IOSCHED_AS=y
CONFIG_DEFAULT_AS=y
CONFIG_DEFAULT_IOSCHED="anticipatory"

#
# Processor and Board Settings
#
CONFIG_BF537=y
CONFIG_BF_REV_0_2=y
CONFIG_BLACKFIN=y
CONFIG_BFIN_SINGLE_CORE=y
CONFIG_BFIN537_BLUETECHNIX_CM=y
CONFIG_MEM_MT48LC16M16A2TG_75=y
CONFIG_BFIN_HAVE_RTC=y
CONFIG_IRQ_PLL_WAKEUP=7

#
# PORT F/G Selection
#
CONFIG_BF537_PORT_F=y

#
# Priority
#
CONFIG_IRQ_DMA_ERROR=7
CONFIG_IRQ_ERROR=7
CONFIG_IRQ_RTC=8
CONFIG_IRQ_PPI=8
CONFIG_IRQ_SPORT0_RX=9
CONFIG_IRQ_SPORT0_TX=9
CONFIG_IRQ_SPORT1_RX=9
CONFIG_IRQ_SPORT1_TX=9
CONFIG_IRQ_TWI=10
CONFIG_IRQ_SPI=10
CONFIG_IRQ_UART0_RX=10
CONFIG_IRQ_UART0_TX=10
CONFIG_IRQ_UART1_RX=10
CONFIG_IRQ_UART1_TX=10
CONFIG_IRQ_CAN_RX=11
CONFIG_IRQ_CAN_TX=11
CONFIG_IRQ_MAC_RX=11
CONFIG_IRQ_MAC_TX=11
CONFIG_IRQ_TMR0=12
CONFIG_IRQ_TMR1=12
CONFIG_IRQ_TMR2=12
CONFIG_IRQ_TMR3=12
CONFIG_IRQ_TMR4=12
CONFIG_IRQ_TMR5=12
CONFIG_IRQ_TMR6=12
CONFIG_IRQ_TMR7=12
CONFIG_IRQ_PROG_INTA=12
CONFIG_IRQ_PORTG_INTB=12
CONFIG_IRQ_MEM_DMA0=13
CONFIG_IRQ_MEM_DMA1=13
```

```
CONFIG_IRQ_WATCH=13

#
# Board Setup
#
CONFIG_CLKIN_HZ=25000000
CONFIG_MEM_SIZE=32
CONFIG_MEM_ADD_WIDTH=9
CONFIG_BOOT_LOAD=0x1000

#
# Console UART Setup
#
CONFIG_BAUD_115200=y
CONFIG_BAUD_NO_PARITY=y
CONFIG_BAUD_1_STOPBIT=y

#
# Timer Tick
#
CONFIG_HZ_250=y
CONFIG_HZ=250

#
# Memory Optimizations
#
CONFIG_RAMKERNEL=y
CONFIG_SELECT_MEMORY_MODEL=y
CONFIG_FLATMEM_MANUAL=y
CONFIG_FLATMEM=y
CONFIG_FLAT_NODE_MEM_MAP=y
CONFIG_SPLIT_PTLOCK_CPUS=4
CONFIG_LARGE_ALLOCS=y
CONFIG_BFIN_DMA_5XX=y
CONFIG_DMA_UNCACHED_1M=y

#
# Cache Support
#
CONFIG_BLKFIN_CACHE=y
CONFIG_BLKFIN_DCACHE=y
CONFIG_BLKFIN_WT=y
CONFIG_L1_MAX_PIECE=16

#
# EBIU_AMBCTL Global Control
#
CONFIG_C_AMCKEN=y
CONFIG_C_CDPRIO=y
CONFIG_C_AMBEN_ALL=y

#
# EBIU_AMBCTL Control
#
CONFIG_BANK_0=0x7BB0
CONFIG_BANK_1=0x7BB0
CONFIG_BANK_2=0x7BB0
CONFIG_BANK_3=0xFFC3
```

```
#
# Executable file formats
#
CONFIG_BINFMT_FLAT=y

#
# Networking
#
CONFIG_NET=y

#
# Networking options
#
CONFIG_PACKET=y
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_IP_FIB_HASH=y
CONFIG_SYN_COOKIES=y
CONFIG_INET_DIAG=y
CONFIG_INET_TCP_DIAG=y
CONFIG_TCP_CONG_BIC=y

#
# Generic Driver Options
#
CONFIG_STANDALONE=y
CONFIG_PREVENT_FIRMWARE_BUILD=y

#
# Memory Technology Devices (MTD)
#
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y

#
# User Modules And Translation Layers
#
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK_RO=y

#
# RAM/ROM/Flash chip drivers
#
CONFIG_MTD_CFI=y
CONFIG_MTD_GEN_PROBE=y
CONFIG_MTD_MAP_BANK_WIDTH_1=y
CONFIG_MTD_MAP_BANK_WIDTH_2=y
CONFIG_MTD_MAP_BANK_WIDTH_4=y
CONFIG_MTD_CFI_I1=y
CONFIG_MTD_CFI_I2=y
CONFIG_MTD_CFI_INTELEXT=y
CONFIG_MTD_CFI_UTIL=y
CONFIG_MTD_RAM=y
CONFIG_MTD_ROM=y

#
# Mapping drivers for chip access
#
CONFIG_MTD_PHYSMAP=y
```

```
CONFIG_MTD_PHYSMAP_START=0x201E0000
CONFIG_MTD_PHYSMAP_LEN=0x20000
CONFIG_MTD_PHYSMAP_BANKWIDTH=2
CONFIG_MTD_UCLINUX=y

#
# Block devices
#
CONFIG_BLK_DEV_RAM_COUNT=16

#
# Network device support
#
CONFIG_NETDEVICES=y

#
# Ethernet (10 or 100Mbit)
#
CONFIG_NET_ETHERNET=y
CONFIG_MII=y
CONFIG_BFIN_MAC=y
CONFIG_BFIN_MAC_USE_L1=y
CONFIG_BFIN_TX_DESC_NUM=10
CONFIG_BFIN_RX_DESC_NUM=20

#
# Character devices
#
CONFIG_BF533_PFLAGS=y

#
# Non-8250 serial port support
#
CONFIG_SERIAL_BLACKFIN=y
CONFIG_SERIAL_BLACKFIN_DMA=y
CONFIG_UNIX98_PTYS=y
CONFIG_LEGACY_PTYS=y
CONFIG_LEGACY_PTY_COUNT=256

#
# SPI support
#
CONFIG_SPI=y
CONFIG_SPI_MASTER=y

#
# SPI Master Controller Drivers
#
# CONFIG_SPI_BITBANG is not set
CONFIG_SPI_BFIN=y

#
# Graphics support
#
CONFIG_FB=y
CONFIG_FB_HITACHI_TX09=y

#
# Logo configuration
```

```
#
CONFIG_BACKLIGHT_LCD_SUPPORT=y
CONFIG_BACKLIGHT_CLASS_DEVICE=y
CONFIG_BACKLIGHT_DEVICE=y
CONFIG_LCD_CLASS_DEVICE=y
CONFIG_LCD_DEVICE=y

#
# Sound
#
CONFIG_SOUND=y

#
# Advanced Linux Sound Architecture
#
CONFIG_SND=y
CONFIG_SND_TIMER=y
CONFIG_SND_PCM=y
CONFIG_SND_OSSEMUL=y
CONFIG_SND_MIXER_OSS=y
CONFIG_SND_PCM_OSS=y

#
# ALSA Blackfin devices
#
CONFIG_SND_BLACKFIN_AD1836=y
CONFIG_SND_BLACKFIN_AD1836_TDM=y
CONFIG_SND_BLACKFIN_AD1836_MULSUB=y
CONFIG_SND_BLACKFIN_SPORT=0
CONFIG_SND_BLACKFIN_SPI_PFBIT=5

#
# USB support
#
CONFIG_USB_ARCH_HAS_HCD=y

#
# MMC/SD Card support
#
CONFIG_SPI_MMC=y
CONFIG_SPI_MMC_CS_CHAN=1
CONFIG_SPI_MMC_DEBUG_MODE=y

#
# File systems
#
CONFIG_EXT2_FS=y
CONFIG_EXT2_FS_XATTR=y
CONFIG_FS_MBCACHE=y
CONFIG_INOTIFY=y
CONFIG_DNOTIFY=y

#
# DOS/FAT/NT Filesystems
#
CONFIG_FAT_FS=y
CONFIG_VFAT_FS=y
CONFIG_FAT_DEFAULT_CODEPAGE=437
CONFIG_FAT_DEFAULT_IOCHARSET="iso8859-1"
```

```
#
# Pseudo filesystems
#
CONFIG_PROC_FS=y
CONFIG_SYSFS=y
CONFIG_RAMFS=y

#
# Network File Systems
#
CONFIG_NFS_FS=y
CONFIG_NFS_V3=y
CONFIG_LOCKD=y
CONFIG_LOCKD_V4=y
CONFIG_NFS_COMMON=y
CONFIG_SUNRPC=y
CONFIG_SMB_FS=y

#
# Partition Types
#
CONFIG_MSDOS_PARTITION=y

#
# Native Language Support
#
CONFIG_NLS=y
CONFIG_NLS_DEFAULT="iso8859-1"
CONFIG_NLS_CODEPAGE_437=y
CONFIG_NLS_ISO8859_1=y

#
# Kernel hacking
#
CONFIG_LOG_BUF_SHIFT=14

#
# Library routines
#
CONFIG_CRC32=y
```

uClinux configuration

```
#
# Automatically generated by make menuconfig: don't edit
#

#
# Core Applications
#
CONFIG_USER_INIT_INIT=y
CONFIG_USER_INIT_CONSOLE_SH=y
CONFIG_USER_OTHER_SH=y

#
# Flash Tools
#
CONFIG_USER_MTDUTILS=y
CONFIG_USER_MTDUTILS_ERASE=y
CONFIG_LIB_ZLIB=y

#
# Filesystem Applications
#
CONFIG_USER_SAMBA_SMBMOUNT=y

#
# Network Applications
#
CONFIG_USER_BOA_SRC_BOA_NEW=y
CONFIG_USER_DHCPCD_NEW_DHCPCD=y
CONFIG_USER_PORTMAP_PORTMAP=y
CONFIG_USER_NET_TOOLS_IFCONFIG=y
CONFIG_USER_NET_TOOLS_ROUTE=y

#
# Miscellaneous Applications
#
CONFIG_USER_WEBRADIO=y
CONFIG_USER_MP3PLAY_MP3PLAY=y
CONFIG_USER_VPLAY_MIXER=y

#
# BusyBox
#
CONFIG_USER_BUSYBOX_BUSYBOX=y
CONFIG_USER_BUSYBOX_CAT=y
CONFIG_USER_BUSYBOX_CP=y
CONFIG_USER_BUSYBOX_CUT=y
CONFIG_USER_BUSYBOX_GREP=y
CONFIG_USER_BUSYBOX_GUNZIP=y
CONFIG_USER_BUSYBOX_GZIP=y
CONFIG_USER_BUSYBOX_HOSTNAME=y
CONFIG_USER_BUSYBOX_KILL=y
CONFIG_USER_BUSYBOX_KLOGD=y
CONFIG_USER_BUSYBOX_MKDIR=y
CONFIG_USER_BUSYBOX_MOUNT=y
CONFIG_USER_BUSYBOX_NFSMOUNT=y
CONFIG_USER_BUSYBOX_MV=y
CONFIG_USER_BUSYBOX_PS=y
```

```
CONFIG_USER_BUSYBOX_SHELL=y
CONFIG_USER_BUSYBOX_MSH=y
CONFIG_USER_BUSYBOX_SH_IS_MSH=y
CONFIG_USER_BUSYBOX_SYSLOGD=y
CONFIG_USER_BUSYBOX_TAR=y
CONFIG_USER_BUSYBOX_TAR_CREATE=y
CONFIG_USER_BUSYBOX_TEST=y
CONFIG_USER_BUSYBOX_UMOUNT=y
CONFIG_USER_BUSYBOX_MOUNT_FORCE=y
CONFIG_USER_BUSYBOX_BUFFERS_GO_ON_STACK=y
CONFIG_USER_BUSYBOX_VERBOSE_USAGE=y

#
# Miscellaneous Configuration
#
CONFIG_USER_RAMIMAGE_RAMFS256=y

#
# Blackfin build Options
#
CONFIG_BLACKFIN_USE_FLAT=y
```

Bibliography

[Ana03] Analog Devices, Inc.: *AD1836A data sheet [Online]*. 2003. – Available at: http://www.analog.com/UploadedFiles/Data_Sheets/AD1836A.pdf [retrieved on Sept. 11, 2006] 27, 48

[Ana05] Analog Devices, Inc.: *ADSP-BF537 Blackfin Processor Hardware Reference, Rev. 2.0 [Online]*. 2005. – Available at: http://www.analog.com/UploadedFiles/Associated_Docs/4206716165649BF537_HRM_whole_book_o.pdf [retrieved on Feb. 27, 2007] 23, 57, 58, 73

[Ana06] Analog Devices, Inc.: *Blackfin Embedded Processor ADSP-BF537 Datasheet [Online]*. 2006. – Available at: http://www.analog.com/UploadedFiles/Data_Sheets/ADSP-BF534_BF536_BF537.pdf [retrieved on March 2, 2007] 23

[BC05] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux Kernel.* O'Reilly Media, 2005. – ISBN 0–596–00565–2 54, 56, 61, 70

[Ber94] Tim Berners-Lee: *RFC 1630: Universal Resource Identifiers in WWW [Online]*. 1994. – Available at: http://tools.ietf.org/html/rfc1630 [retrieved on April 6, 2007] 1, 85

[Blu06] Bluetechnix: *Hardware User Manual CM-BF537E V1.1 [Online]*. 2006. – Available at: http://www.bluetechnix.at/rainbow2006/site/tinyboards/__core_modules/__cm-bf537e/313/cm-bf537e.aspx [retrieved on Feb. 15, 2007] 38, 74

[CPS95] Callaghan, B.; Pawlowski, B.; Staubach, P.: *RFC 1813: NFS Version 3 Protocol Specification [Online]*. 1995. – Available at: http://tools.ietf.org/html/rfc1813 [retrieved on March 15, 2007] 8

[CRKH05] Corbet, Jonathan ; Rubini, Alessandro ; Kroah-Hartman, Greg: *Linux Device Drivers, 3rd Edition.* O'Reilly Media, 2005. – ISBN 0–596–00590–3 47, 54

[Fre03] Freescale Semiconductor, Inc.: *SPI Block Guide V03.06 [Online]*. 2003. – Available at: http://www.freescale.com/files/microcontrollers/doc/ref_manual/S12SPIV3.pdf [retrieved on March 2, 2007] 24

[Her03] Hertel, Christopher: *Implementing CIFS: The Common Internet File System.* Prentice Hall, 2003. – ISBN 0–13–047116–X 8

[Kao06] Kaohsiung Hitachi Electronics Co., Ltd.: *Customer's acceptance specifications TX09D70VM1CDA [Online].* 2006. – Available at: http://www.hitachi-displays-eu.com/faqs/pinfo3.asp?pno=TX09D70VM1CDA [retrieved on Jan. 11, 2007] 26, 55, 57

[Kop97] Kopetz, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer Netherlands, 1997. – ISBN 0–7923–9894–7 75

[Lou00] Louis, Dirk: *Jetzt lerne ich Perl.* Markt+Technik, 2000. – ISBN 3–8272–5841–3 83

[Mac05] Maciaszek, Leszek A.: *Requirements Analysis and System Design.* Addison Wesley Publishing Company, 2005. – ISBN 0–321–20464–6 5, 12, 15, 93

[San03] SanDisk Corporation: *SanDisk SD Card Product Manual Version 1.9 [Online].* 2003. – Available at: http://www.cs.ucr.edu/˜amitra/sdcard/ProdManualSDCardv1.9.pdf [retrieved on Feb. 27, 2007] 14, 61, 62, 63

[Sun89] Sun Microsystems, Inc.: *RFC 1094: NFS: Network File System Protocol specification [Online].* 1989. – Available at: http://tools.ietf.org/html/rfc1094 [retrieved on March 15, 2007] 8

[Tam05] Tamandl, Thomas. *Embedded Linux for Signal Processing Platform.* 2005 32, 39

[Uma04] Uman, Herbert: *Nessus 2.x kompakt.* Bomots Verlag, 2004. – ISBN 2–915925–00–3 97

[Yag03] Yaghmour, Karim: *Building Embedded Linux Systems.* O'Reilly Media, 2003. – ISBN 0–596–00222–X 37

Web references

[WAN] http://www.analog.com [retrieved on Oct 3, 2006]

[WBLa] http://blackfin.uclinux.org [retrieved on Feb 20, 2007]

[WBLb] http://docs.blackfin.uclinux.org/doku.php?id=adding_user_applications [retrieved on March 18, 2007]

[WBLc] http://docs.blackfin.uclinux.org/doku.php ?id=components_of_a_development_system [retrieved on March 16, 2007]

[WBLd] http://docs.blackfin.uclinux.org/doku.php?id=debuggers [retrieved on March 18, 2007]

[WBLe] http://docs.blackfin.uclinux.org/doku.php?id=gcc_and_gas [retrieved on March 16, 2007]

[WBLf] http://docs.blackfin.uclinux.org/doku.php?id=operating_systems #introduction_to_uclinux [retrieved on March 10, 2007]

[WBLg] http://docs.blackfin.uclinux.org/doku.php?id=testing_the_toolchain &s=testing%20toolchain#automated_testing [retrieved on April 5, 2007]

[WBLh] http://docs.blackfin.uclinux.org/doku.php?id=toolchain_build_script [retrieved on March 16, 2007]

[WBLi] http://docs.blackfin.uclinux.org/doku.php?id=uclinux_features [retrieved at March 11, 2007]

[WBLj] http://www.bluetechnix.at [retrieved on Feb 22, 2007]

[WBO] http://www.boa.org/documentation/boa-1.html#ss1.1 [retrieved on March 25, 2007]

[WBU] http://www.busybox.net/downloads/README [retrieved at March 11, 2007]

[WCO] http://www.columbia.edu/kermit/ [retrieved on March 16, 2007]

[WDE] http://www.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=blob_plain; f=README;hb=HEAD [retrieved on March 17, 2007]

[WDIa] http://www.die.net/doc/linux/man/man3/readdir.3.html [retrieved on April 5, 2007]

[WDIb] http://www.die.net/doc/linux/man/man7/netdevice.7.html [retrieved on April 5, 2007]

[WMI] http://www.microwindows.org/microwindows_architecture.html [retrieved on April 1, 2007]

[WOP] http://www.opensource.org/docs/definition.php [retrieved on March 4, 2007]

[WREa] http://retran.com/beej/sockaddr_inman.html [retrieved on April 5, 2007]

[WREb] https://www.reciva.com/aewir-ts.html [retrieved on April 13, 2007]

[WSL] http://www.slimdevices.com/pi_squeezebox.html [retrieved on April 13, 2007]

[WST] http://www.streamit.eu/?p=product&id=39&l=en [retrieved on April 13, 2007]

[WUCa] http://blackfin.uclinux.org/gf/project/uclinux-dist/forum/
?action=ForumBrowse&forum_id=39&thread_id=3900
&_forum_action=ForumMessageBrowse [retrieved on Jan 8, 2007]

[WUCb] http://www.uclibc.org [retrieved on March 9, 2007]

[WUCc] http://www.uclinux.org [retrieved on Sept 12, 2006]

[WUN] http://unixhelp.ed.ac.uk/CGI/man-cgi?nfs+5 [retrieved on March 25, 2007]

[WVP] http://www.vpx.nu/dokuwiki/doku.php?id=mmc_sd_configuration [retrieved on Oct 27, 2006]

[WWIa] http://en.wikipedia.org/wiki/8P8C [retrieved at March 15, 2007]

[WWIb] http://en.wikipedia.org/wiki/Internet_radio [retrieved on April 13, 2007]