MAIN PROJECT:

TITTEL

The DaisyPlayer Project

AUTHORS:

André Lindhjem
Kjetil Holien
Terje Risa
Øyvind Nerbråten

DATE:

22.05.2006

SUMMARY OF THE MAIN PROJECT

Title:	The DaisyPlayer Project		Nr. :
			Date : 22.05.06
Participants:	André Lindhjem Kjetil Holien Terje Risa Øyvind Nerbråten		
Supervisor:	Øyvind Kolås		
Customer:	Skolelinux		
Contact person:	Herman Robak		
Catch words	Daisy, Digital Talking Book, GNU/Linux, Open Source		
Pages: 85	Appendixes: 9	Availability (open/confidential): open	

Short summary of the main project:

During the progress of this Open Source project, we have created a library for parsing and playing a Daisy Digital Talking Book. We have also developed two front-ends which utilizes our library, one with a graphical user interface and one with a console based user interface.

The Daisy standard gives blind, visually impaired, and otherwise print challenged individuals access to information using multimedia representation of a print publication.

To make our library useful for other developers, we have made an application programming interface. The library is meant to hide the complexity of the Daisy standards, enabling developers to create their own Daisy software, without extensive knowledge about the Daisy format.

This projects software have been developed using C, C++, Qt and numerous other libraries.

We have worked using an Open Source development model with elements from evolutionary development. The development process have been divided into release cycles, each spanning two weeks.

Preface

This project has been carried out by four students at Gjøvik University College in Norway working on this project as our bachelor assignment. The DaisyPlayer Project is the final task in our last semester of our bachelor degrees in Computer Science. We chose this project because we wanted to create something useful and because the Linux operating system lacks a good, free Daisy player.

We found this bachelor assignment to be a great opportunity for us to expand our horizon by learning new things and gain experience in working on a relatively large project. Development for- and on the Linux platform was completely new to us and posed as an exiting way to gain new topical knowledge. We also liked the idea of starting an Open Source project and cooperate with the Open Source community. The Open Source community is a vast resource, which we can turn to for tools, help and guidance.

We would like to thank the following people for their help and contributions to this project:

- Øyvind Kolås, our supervisor, for giving us tips and help along the way.
- Herman Robak, our principal, for giving us this assignment and introducing us to the Open Source community.
- Petter Reinholdtsen for all the help and support he has given us. We really appreciate all the time and effort Petter Reinholdtsen have put in the project. Petter has been a person we could turn to for help and guidance.
- Bjørn Erik Nilsen from the last years Stopmotion project, for helping out with the GUI part and always answering our questions.
- The Skolelinux community for hosting the project, and giving us feedback, resources, pointers and suggestions.

Gjøvik, 22th of May, 2006

Øyvind Nerbråten   André Lindhjem

Kjetil Holien   Terje Risa

Contents

1 Introduction **1**

1.1 About Daisy . . . 2
1.2 Task background . . . 2
1.3 Limitations . . . 3
1.3.1 Product . . . 3
1.3.2 Project constraints . . . 3
1.4 Defining the assignment . . . 4
1.5 Target group . . . 4
1.6 Objective and why we chose this project . . . 5
1.7 The groups academical background and expertise . . . 6
1.8 Method of work . . . 6
1.9 The organization of the report . . . 7
1.9.1 Terminology . . . 8
1.9.2 The layout of the report . . . 11

2 Analysis **12**

2.1 Amis . . . 13
2.2 Emacspeak . . . 13
2.3 IDAIR . . . 13
2.4 Listen-up . . . 14

3 Requirements specification **15**

3.1 Use case model . . . 16
3.2 Supplementary specifications . . . 17
3.3 Requirement ranking . . . 17

4 Design **19**

4.1 Front-ends . . . 20

4.2 Threads 20
4.3 Audio engine 20
4.3.1 Audio interface 21
4.3.2 Audio decoder / converter 22
4.3.3 Audio output module 22
4.4 Parser 22
4.4.1 NCC & NCX parser 24
4.4.2 SMIL 1.0 & SMIL 2.0 parser 25
4.5 Data structure 26
4.5.1 Audio engine 27
4.5.2 Parser 27
4.6 Libdaisy interface 28
4.7 Internationalization 29
4.7.1 GUI front-end 29
4.7.2 Console front-end 30

5 Implementation **32**
5.1 Choices of programming language and libraries 33
5.1.1 Programming languages 33
5.1.1.1 Ruby 33
5.1.1.2 C 33
5.1.1.3 C++ 34
5.1.2 Libraries 34
5.1.2.1 GStreamer 34
5.1.2.2 REXML 35
5.1.2.3 Libxml2 35
5.1.2.4 GUI 36
5.1.2.5 Audio design considerations 37
5.2 Threads 39

6 Management tools **40**
6.1 Documentation and choices of aiding- and developing tools 41
6.1.1 Doxygen 41
6.1.2 LaTeX and LyX 42
6.1.3 Subversion and TortoiseSVN 42
6.1.4 Planner 43
6.1.5 Eclipse 43
6.2 Code quality tools 43
6.2.1 Splint 43

6.2.2 Electric fence 43
6.2.3 Valgrind 44
6.2.4 GNU Debugger 44
6.2.5 GNU Binary Utilities package (*nm*) 44
6.2.6 Lintian 44
6.3 Packing and release tools 45
6.3.1 Make 45
6.3.2 QMake 45
6.3.3 Debian Package building tools 46
6.4 Scripts 47

7 Testing **48**
7.1 User testing 49
7.2 Product quality 50
7.3 Product quality tools used 50

8 Public relations **53**
8.1 Web page 54
8.2 Plan for advertising 56

9 Discussion of results **58**
9.1 Evaluation of the result 59
9.2 Evaluation of choices and technologies 60
9.2.1 Development model 60
9.2.2 Choice of programming language 61
9.2.3 Choice of GUI 62
9.2.4 Choice of libraries 63
9.2.4.1 POSIX threads 63
9.2.4.2 MAD - MPEG Audio Decoder 63
9.2.4.3 Libxml2 64
9.2.4.4 Libao 64
9.2.5 Choice of documentation tools 65
9.2.5.1 Doxygen 65
9.2.5.2 LaTeX and LyX 65
9.2.6 Evaluation of development environment 66
9.2.6.1 Eclipse 66
9.2.6.2 Scripts 66
9.2.7 SVN 66
9.2.8 Problems encountered 67

9.2.9 Evaluation of the PR work . . . 68
9.3 Evaluation of the groups work . . . 69
9.4 Further work on the project . . . 70

10 Conclusion 71

11 Bibliography 72

Index 74

A Pre-project report for the DaisyPlayer Project (without appendixes) 76
A.1 Goals and constraints . . . 76
A.1.1 Background . . . 76
A.1.2 Effect goal . . . 76
A.1.3 Result goal . . . 77
A.1.4 Target group . . . 77
A.1.5 Constraints . . . 77
A.2 Extent of task . . . 77
A.2.1 Task description . . . 77
A.2.2 The platform . . . 78
A.2.3 Libraries, frameworks and standards . . . 78
A.2.4 Programming languages . . . 79
A.2.4.1 Programming languages . . . 79
A.2.4.2 Python . . . 80
A.2.4.3 Ruby . . . 80
A.2.4.4 C++ . . . 80
A.2.4.5 C . . . 80
A.2.4.6 Availability of compilers and interpreters 80
A.2.5 Constraints . . . 81
A.3 Project organization . . . 81
A.3.1 Principal . . . 81
A.3.2 Group . . . 81
A.3.3 Roles and responsibilities . . . 81
A.4 Planning and reporting . . . 82
A.4.1 Choice of methodology . . . 82
A.4.2 Plan for status meeting and decision dates . . . 82
A.4.3 Code convention . . . 82
A.5 Organization of quality assurance . . . 83

A.5.1 Revision management 83
A.5.2 Quality handling 83
A.5.3 Risk analysis 83
A.6 Development plan 84

B Use cases **85**
B.1 daisy_init 85
B.2 daisy_term 86
B.3 daisy_load 86
B.4 daisy_play 87
B.5 daisy_seek 88
B.6 daisy_get_position 89
B.7 daisy_goto_position 89
B.8 daisy_stop 90
B.9 daisy_pause 91
B.10 daisy_get_info 92
B.11 daisy_get_chapter_count 92
B.12 daisy_get_chapter_info 93

C Code conventions **94**

D Gantt chart **97**

E Status reports **100**

F Work log **106**

G Libdaisy API **107**

H Manuals **120**
H.1 What is daisyconsole? Where can I get it? 122
H.2 Getting started 122
H.3 Open a Daisy DTB 123
H.4 Playback 123
H.4.1 Play 123
H.4.2 Pause 123
H.4.3 Stop 123
H.4.4 Seek 124
H.5 Positioning and bookmarking 124
H.6 Book indexing 124

H.7 Shortcut keys 125
H.8 What is daisygui? Where can I get it? 127
H.9 Getting started 127
H.10 Open a Daisy DTB 128
H.11 Playback 129
H.11.1 Play 129
H.11.2 Pause 130
H.11.3 Stop 130
H.11.4 Seek 130
H.12 Bookmarking 130
H.13 Book indexing 130
H.14 Shortcut keys 131

I CD contents **132**

List of Figures

3.1 Use case diagram. 16

4.1 Audio module outline 21
4.2 The logical structure of a Daisy book. 23
4.3 An extract of an NCC file. 25
4.4 An extract of a SMIL file. 26
4.5 An extract of the resulting lists in the data structure . 28
4.6 An example how internationalization work with Designer. 29
4.7 Adding a new translation to the project file. 29
4.8 A screenshot from Linguist. 30

5.1 Showing the KDE architecture 36

6.1 Doxygen source example 41
6.2 Doxygen output example 41
6.3 A simple example on a project file. 46
6.4 Part of the Makefile used to generate this report. . . . 47

7.1 The evolutionary system development model. 49

8.1 A screenshot of the DaisyPlayer Project web page. . . . 55

D.1 Gantt chart - part 1. 98
D.2 Gantt chart - part 2 99

H.1 Daisyconsole screenshot. 122
H.2 Daisygui screenshot. 127
H.3 Daisygui screenshot. 129

Chapter 1

Introduction

In today's world the boundaries between technology and education have melted away and are now very close integrated. More and more of the teaching in the school are being done with the help of computers. Not only is it, in many cases, more interesting, fun, and motivating for the students, but the opportunities that lies with this technology is almost infinite. This is where Skolelinux comes into focus.

Skolelinux[1] is a Custom Debian Distribution that is customized for schools with focus on being easy to install and maintain. Skolelinux tries to give schools and students a good and free alternative to proprietary software. This is used in many schools already and the goal is to cover as many schools as possible.

In order to achieve this goal, Skolelinux need software to cover the different needs the student have in their education. One such program, after a request of a Norwegian teacher, is an application able to play Daisy Digital Talking Books (DTB). Daisy DTB is a multimedia representation of a print publication. Linux lacks a good and easy to use Daisy player at this point.

[1] Skolelinux http://www.skolelinux.org/portal/

1.1 About Daisy

Common audio books are an audio representation of printed publications, and is something most people are familiar with. Daisy Digital Talking Books[2] incorporates open standards and multimedia to present books in a better way to people who have problems using traditional printed media.

Both traditional audio books and Daisy books use a human voice to present the contents of a book to the reader. But, as Daisy books utilizes multimedia technology, it has some features that normal audio books lacks. This includes features like synchronized audio and text, as well as navigation features.

The main advantages of Daisy books over traditional audio books are illustrated by an example; consider an encyclopedia or a phone catalogue and how they should be made accessible for print-disabled persons. Traditional printed books are not an option, and audio books will not be useful because it lacks when it comes to navigation (listening to a whole encyclopedia in sequential order are not very useful). Daisy books, however provides support for audio read contents as well as extensive navigational features.

1.2 Task background

Audio books are often used by people for many reasons. The users might be blind or visually impaired, or they might have reading disorders or other troubles which makes it difficult for them to use conventional printed media. Daisy books are usually audio books with synchronized text that provides navigating capabilities which are superior to that of normal audio books.

This project task was requested by Ole-Anders Andreassen because there was no suitable player for Daisy books he could use on Linux. He had used to play the audio as normal audio books, but this means that a lot of the Daisy capabilities (such as navigating the books) are lost.

Also, the Skolelinux project are interested in providing a Daisy-Player with their product.

There were few good software Daisy players altogether. The ones that exists are usually not actively maintained, depends on old tech-

[2] What is a DTB? `http://www.daisy.org/about_us/dtbooks.asp`

nology and are not user friendly enough to be used in schools. The long time effect goal of this project have been to make it possible to develop one or more user friendly Daisy players that can help students, in an easy way, to play Daisy books on the Linux platform.

1.3 Limitations

1.3.1 Product

The primary goal of this project is not to create a complete Daisy player with GUI which is ready to be used. This will probably be a too large task for our time schedule, and it makes more sense to focus on implementing the Daisy functionality and leave the GUI out of it, at least until the Daisy functionality is implemented.

The UI that are to be produced should primarily demonstrate the functionality of the DaisyPlayer.

In this project we have to be conscious about which libraries and dependencies we use. It is a goal to keep the number of dependencies to a minimum. At the same time, we should use existing libraries and as much as possible. We need to be conscious about what libraries we use and make sure that they are good choices when it comes to maintainability, supported platforms and licenses.

1.3.2 Project constraints

The project must be Open Source. It must be able to be licensed and distributed according to The Debian Free Software Guidelines (DFSG)[3]. The group also decided on some guidelines to follow, beside being an Open Source project, we felt it important to write all our documentation in English. This to make it easier for other to adapt our project after we are finished with it. Hopefully will other take interest in this project, so that the project does not become another abandoned project. We also felt it natural to have an open repository on our web page, giving anybody the chance to look into our work. This giving the users the ability to really follow the project closely. Another decision made by the group where to avoid any proprietary programs like

[3] The Debian Free Software Guidelines http://www.debian.org/social_contract

Microsoft Project or Microsoft Word, for instance to make our Gantt-charts and writing the report.

1.4 Defining the assignment

The primary goal is to create a Daisy player ”engine” that can easy be incorporated in other programs and provide the Daisy functionality. Future Daisy players could be implemented by writing a GUI which works against the DaisyPlayer engine and a clearly defined API. In-depth knowledge about the different Daisy books and standards will not be required. This should make the development of new Daisy-aware applications much simpler which we hope will benefit the users of Daisy books.

It is preferred by the principal, if the project can incorporate or build upon other Open Source Daisy projects, as the project probably has a better chance of surviving and involving other developers. Because of this, a part of the assignment involves evaluating existing free Daisy players and either learn from or build upon them.

As a part of this task, the group have to make themselves familiar with several new areas such as Open Source development, software licenses, Daisy standards and the libraries that will be used.

The project should release regularly so that the progress should be easy to follow for people who are interested in testing the Daisy-Player. External input, bug reports and experiences should be considered throughout the whole project.

1.5 Target group

The target group for this project report are mainly those who shall evaluate the project. Parts of this report is written in a technical manner, so the reader is expected to have some knowledge related to software development. This report is also meant as supplementary documentation for software developers who wish to use or extend the DaisyPlayer engine.

This project as a whole, addresses multiple target groups through the different products we have created.

The main product of this project, the DaisyPlayer engine (libdaisy), targets software developers. This project will also include two simple

UI to demonstrate the functionality of the engine, meant for software developers and users who want to test our product.

The two UI also targets regular Linux users and should motivate regular users to test and get involved with the DaisyPlayer Project. They also, to some degree, motivate regular Daisy users to test their Daisy books with our software.

We target end users by providing easy-to-install Debian packages and a user manual that explains how to use our example players. This way, we hope to enable regular users to be able to use our software with a minimum of trouble.

Our main portal towards the Internet, users and the Open Source community are our web page[4]. The web page targets all of the above mentioned user groups. It contains a presentation, basic information, FAQ and install information targeted at regular users. It also contains project information and documentation, open to anyone who wishes to read about the project. Developers can find the libdaisy API documentation, and even browse the current libdaisy up-to-date source code. The web page design is friendly, informative, and should encourage interested visitors to try our software, contact us, or involve themselves in the project.

1.6 Objective and why we chose this project

Skolelinux is more and more used in the schools at lower level, not only in Norway but also in the rest of the world. At this point Skolelinux lacks a good program for playing Daisy Digital Talking Books (DTB). One objective of this project is to start an Open Source development of a Daisy DTB player.

We chose this project because we thought it would be an interesting and challenging project. The group had minimal experience with programming for Linux, so we considered this project to be a very good way to learn something new. Some of us also took this project as a good opportunity to learn more about Linux as an operative system. We also looked at this project, as it would be interesting to develop something that other people could use and maintain when we are finished. Since the existing free Daisy players for Linux are so unfinished

[4]The DaisyPlayer Project `http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/`

and no longer under maintenance, we really saw the need for a new project which could get other people interested. We really thought and hoped that since we started this project it would eventually help a lot of students with reading disabilities. Further we thought it could be of interest to get to know an Open Source community and learn how projects are developed.

1.7 The groups academical background and expertise

The group consists of four graduating students in bachelor of engineering, computer science at HIG[5]. One specialized in system administration and the other three in software development.

We had all experience in using development languages such as C++ and Java, so adapting to C was no problem. As none of us had done any development under GNU/Linux before and three of the group members was almost completely new to the GNU/Linux operation system, we had to start from scratch and learn the ways of software development on this platform. We also needed to learn all the libraries included in the project, as we had no experience what so ever in use of these.

A part of this project has been to investigate, learn and use lots of standards, libraries, software and technologies. These include, but are not limited to: XHTML, CSS, XML, libxml2, libao, libfaad2, MAD, Make, LaTeX, Daisy, SMIL and Subversion. Some had to be rejected in the process as they did not fit our needs.

1.8 Method of work

In our pre-project phase we decided on a development model, a mixture of evolutionary system development and Open Source development, to use throughout the project, see section A.4.1 on page 82 for more information. This model will hopefully give us the freedom to carrying out our project in a way that suits the group and the project. As an Open Source developing project, we feel it natural that our working methods are adaptable, so that the project is not entirely controlled by decisions made early in the project, but are open for changes as

[5]Gjøvik University College http://www.hig.no/eway/default0.asp?pid=248

we go along. This way of working, requires good communication and excellent collaboration, but since the group are used to work together from earlier project and also live side by side in a hall of residence, we did not feel this as a threat to the project.

We also decided to have an open way of organizing the project, and giving each other tasks based on individual knowledge and wishes. Another thing we felt natural where to be able to always helping each other out when we stumble upon difficulties. On the decision of the project leader position, we discussed if we should pick one leader, or maybe rotate the position so that everybody got the chance to be leader. We decided to go with one leader throughout the project, and Øyvind Nerbråten were picked.

Our Gantt-chart, see appendix D, was submitted to the web page so that anyone could follow our progress plan.

1.9 The organization of the report

The report is based on HIG's final project reports template.

1. Introduction
Contains a brief description of our project, tasks and report.

2. Analysis
An analysis of the existing Daisy players available.

3. Requirements specification
The requirements specification to the project.

4. Design
A solution to the applications design.

5. Implementation
How the modules has been implemented and what libraries we used and rejected.

6. Testing
How we decided to test our code and what we did to ensure the quality of the source code.

7. Discussion of results
Reflections about the result.

8. Conclusion
The conclusion of the project.

9. Bibliography
Books and web pages which has been referenced in the report.

Appendixes
Pre-project, code conventions, use cases, Gantt-charts, status reports, libdaisy API, manuals and CD contents.

1.9.1 Terminology

AAC — Advanced Audio Coding. AAC is a lossy audio encoding and compression format.

ALSA — Advanced Linux Sound Architecture is a Linux kernel component intended to replace the Open Sound System (OSS).

API — Application Programming Interface describe how to access a set of functions.

ARTS — Analog RealTime Synthesizer is an application that simulates an analog synthesizer. The aRts soundserver mixes several sound streams and is used as the default sound server for KDE.

DAISY — Digital Accessible Information SYstem. A digital talking book standard.

DFSG — Debian Free Software Guidelines. A set of commitments that the Debian GNU/Linux system has agree to abide by.

DOM — Document Object Model. Is a description of how a HTML or XML document is represented in an object-oriented fashion.

DTB — Digital Talking Book. A multimedia representation of a print publication.

ESD — Enlightened Sound Daemon is the sound server for Enlightenment window manages and the GNOME desktop.

FAQ — Frequently Asked Question. A collection of common questions and the accompanying answers.

FPU — Floating Point Unit is a part of a computer system specially designed to carry out operations on floating point numbers.

GNU/GPL — GNU General Public License. The GNU/GPL is a common license that grants the users certain rights.

GUI — Graphical User Interface. Also see UI.

HIG — Høgskolen i Gjøvik/Gjøvik University College (GUC).

IRC — Internet Relay Chat. IRC is a form of instant communication over the Internet.

LaTeX — LaTeX. A High-quality typesetting system, with features designed for the production of technical and scientific documentation.

LyX — LyX. A document processor following the "what you see is what you mean" (WYSIWYM) paradigm. This means that the user only have to care about the structure and content of the text, while formatting is done by LaTeX.

MP3 — MPEG-1 Audio Layer 3 is an audio encoding and lossy compression format.

NCC — The Navigation Control Center document contains an index of navigable entry points into the DTB, used by the Daisy 2.02 standard. The NCC also implicitly represents the continuous playback order of all the media objects that make up the DTB. This is sometimes referred to as "the flow" of narration and/or text.

NCX — The Navigation Control file for XML applications exposes the hierarchical structure of a DTB, used by the Daisy Z39.86-2005 standard, to allow the user to navigate through it. The NCX is similar to a table of contents.

NUUG — Norwegian Unix User Group. A non-commercial organization for Unix users in Norway.

Open Source — Open Source describes practices in production and development that promote access to the end product's sources.

OSS — Open Sound System is a portable sound interface, available on many Unix/Linux systems.

PCM — Pulse-Code Modulation is a digital representation of an analog signal. The analog signal are sampled at fixed intervals, and the magnitude of each sample are stored as a digital value.

PDF — Portable Document Format is a file format developed by Adobe Systems for representing documents.

POSIX — POSIX is the name of a series of standards specified by the IEEE to define the API Unix compatible software.

RIFF — Resource Interchange File Format is the formal name of the file format used by the common WAV files.

SAX — Simple API for XML. An event based interface for processing XML documents.

SMIL — The Synchronized Multimedia Integration Language is a standard for definition and playback of multimedia presentations over the Internet. SMIL defines the sequence of playback for one or more media objects. In the case of DTB's, the primary media objects are audio and textual content files; SMIL provides for their parallel and synchronized presentation.

UI — User Interface. The graphical, textual and auditory information the program presents to the user, and the control sequences (such as keystrokes with the computer keyboard,

movements of the computer mouse, and selections with the touchscreen) the user employs to control the program.

WAV — A Microsoft and IBM audio file format. A common format for raw audio. A variant of the RIFF file format.

XPath — XML Path Language uses a non-XML syntax for addressing portions of an XML document. XPath is often used by developers as a small query language.

1.9.2 The layout of the report

The whole report, except the libdaisy API, has been made using LaTeX[6]. The libdaisy API is an auto generated PDF generated from Doxygen[7]. We decided to have a terminology list in the introduction chapter so that the reader knows where to look if he stumbles upon an abbreviation. We have also used footnotes to give better explanations and link to external web pages. In the end of the main report we have an index containing references to many keywords.

This report are also available in PDF format with many clickable links, an index and cross references. The amount of external links and cross references makes the PDF version easier to navigate and better suited for most readers.

[6] LaTeX http://www.latex-project.org/
[7] Doxygen http://www.stack.nl/~dimitri/doxygen/

Chapter 2

Analysis

In the early stages of the project we mapped other Open Source Daisy project available to us. In this process we found one project for Windows and three for Linux. We had to make a choice if it was possible for us to make use of some of this code. None of these existing projects seemed to fit our image on how we would like to implement a Daisy player, so we decided not to base our work on any of these projects, but rather use them as possible ideas and hints on how things can be done. By reading the problems others have stumbled on, we can learn and maybe avoid coming across them ourselves.

2.1 Amis

The Amis[1] project is a fully functional Daisy player written for the Windows operating system. The project is written in C++, which did not really fit our plans and some of the libraries used would make to many dependencies. Because of this it would be difficult to port the code to a Linux environment. Read more about our choice of programming language in chapter 5 section 5.1.1. The Amis project consists of a very large code, so it would take us much time to get to know the code properly and the fact it is written for Windows, makes it difficult to use. We decided not to base our player on this code, but rather "steal" some ideas along the way.

2.2 Emacspeak

Emacspeak[2] is a subsystem of Emacs that produces speech output and can be used with the Daisy format. We had problems getting this to work and did not really see why this was listed as a Daisy player on some sites. Emacspeak is more like a screen reader than a Daisy player. The fact that it just works inside Emacs made it useless for our purpose.

2.3 IDAIR

Iduna Daisy Reader[3]. This is a Linux Daisy player project written in C. The code looks incomplete and the project has not been updated since December 2003. The project could be used to get some ideas, but we decided not to reuse this code as it was incomplete and a bit messy. The project had implemented a own XML parser with lack of almost everything, which is a bad idea when there are so many free Open Source XML parsers available.

[1] Amis `http://amis.sourceforge.net/`
[2] Emacspeak `http://emacspeak.sourceforge.net/`
[3] IDAIR `http://idair.sourceforge.net/`

2.4 Listen-up

Listen-up[4] is a Linux Daisy player project written in C++. This project has been inactive since august 2003. The audio library used by this player is an outdated and inactive project. The code is in C++, which is not our first choice of language.

[4] Listen-up http://www.linux-speakup.org/listenup.html

Chapter 3

Requirements specification

A requirement specification is an important part of all software development projects. This part of the document is meant to aid the programmers in the implementation phase. To identify the requirements in our project we decided to create use cases of the functionality of our library, we also had some supplementary specifications. From the beginning we had the main functionality figured out, and as we learned more about the standard we discovered new functionality which we needed to add to the library.

3.1 Use case model

The use cases we have figured out are shown in figure 3.1. As you can see from the picture, the user has access to all of the functions in the library.

In appendix B there are expanded use case descriptions for the cases.

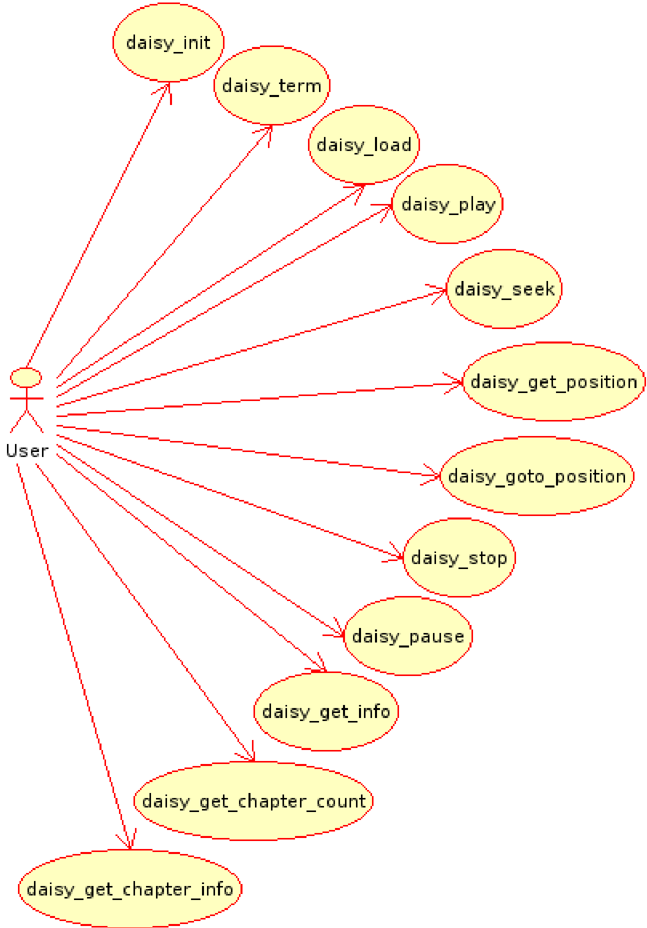

Figure 3.1: Use case diagram.

3.2 Supplementary specifications

Debugging should be available during development. This means that states and key points in the library should be reported during run time (using one common method) while debugging is enabled.

Error handling that the library cannot automatically resolve should be presented to the front-end program so that the programmer (or user) who use libdaisy can investigate the error and decide the proper action to take.

3.3 Requirement ranking

There are almost endless features we could implement in such a project, but due to the time limitation, we have to prioritize what we should implement.

AAC
: AAC support will not be prioritized. We have not found a single Daisy DTB which uses AAC.

Front-ends
: We will try to make a small and intuitive front-end that shows how the engine can be used.

MP3
: MP3 are by far the most common audio format in Daisy books. The support for MP3 audio playback is vital to the project.

Navigation
: Basic navigating functionality should be supported. The user should be able to forward / rewind both on a passage and chapter level. Methods for stopping and pausing the book should also be provided.

Portability
: We will try to use libraries which easily could be ported to both Windows and Mac OS. If we got time this will be done, but it is not a priority.

Speech synthesis
: We will try to implement text to speech to our front-end using Festival[1] if we get the time, but it is not a priority.

[1] Festival http://www.cstr.ed.ac.uk/projects/festival/

Standards — Daisy standard 2.02 will be prioritized since this is the most used standard today. But we will have support for the main functionality in the new standard, Z39.86-2005. The 2.0 standard will not be supported.

WAV — WAV support will not be prioritized. We have not found a single Daisy DTB which uses WAV.

Chapter 4

Design

As this is an Open Source project we wanted to be very clear on the design. This because we wanted to help other interested users to fairly easy understand the code. One of the first decisions we made were to separate the engine from the UI. This was done because we wanted to have the main focus on the engine and Daisy functionality, and not mix it together with how this were presented to the end user. As a result of this separation, it is relatively easy to make different UI to the engine. In our case we created both a commando based UI (console front-end) and a GUI to show how the engine can be used.

4.1 Front-ends

By creating both a graphical UI and a console based UI, we aim to demonstrate two quite different target groups which this project aims to help. Users who require large fonts, large buttons etc. will probably need to rely on a graphical UI. New users might want a simple interface with just the basic functions and as few buttons as possible. On the other hand, blind users have no need for graphical UI's, and might require highly configurable functionality. Because the Daisy target group includes many people with very different needs, we mean our approach with a library and multiple specialized front-ends are a good solution. Our examples aim to demonstrate this. By providing a console-based player, we can benefit from existing screen readers, and navigation by pressing keyboard keys. The GUI based alternative have configurable font-size, use traditional mouse navigation, and it has a listing of the book chapters.

4.2 Threads

Libdaisy requires a few events to happen simultaneously. Handling and acting on input will need to happen at the same time as we are decoding and playing audio. For example, actions like forwarding, stopping or pausing will need to be acted upon even though libdaisy are playing audio.

To achieve this, we decided that we needed to base our library on threads. POSIX threads are the most common, and libraries that uses the POSIX threads interface exists on almost all modern operating systems.

4.3 Audio engine

The audio engine is the part of libdaisy which parses audio files, decoding audio and playing the sound for the user.

The audio engine is separated into multiple components as shown in figure 4.1:

- an audio interface towards the rest of libdaisy.

- a decoder or converter which creates PCM audio from an encoded audio file.
- an audio output component that plays the PCM data to the desired audio output source.

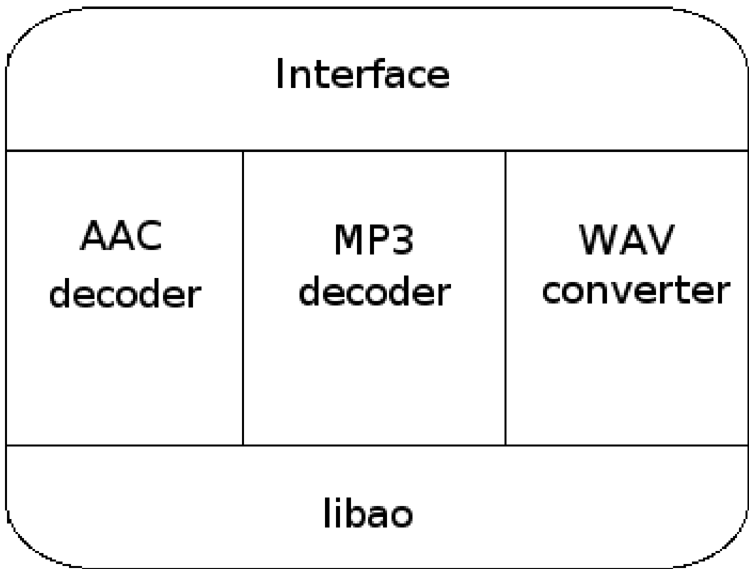

Figure 4.1: Audio module outline

These parts works together to create sound from the Daisy audio source files. The MP3 decoder has the highest priority because MP3 encoded audio are by far the most common format. But the audio interface should enable easy integration of both an AAC decoder and a WAV converter as well.

4.3.1 Audio interface

The audio interface is basically a set of functions that provides libdaisy with the basic audio functions it requires. The main idea is that the complexity of decoding and playing audio should be well hidden from the code that parses Daisy books.

This audio interface allows for extending the audio module at a later point, and support for decoding other formats could be implemented without affecting the rest of libdaisy.

4.3.2 Audio decoder / converter

The decoder, controlled from the audio interface, converts encoded audio to PCM audio which is sent to the output module.

MP3 and AAC will have to be decoded, while the WAV format has PCM encoded audio as a component, so a converter should probably be suffice.

4.3.3 Audio output module

In libdaisy, this is basically a very simple wrapper to the equivalent libao functions. Once the decoder has decoded a buffer, a function in the audio output module is called, and audio will be played to the user, using the preferred audio module or server.

4.4 Parser

Before we explain how the parser in libdaisy works, we must explain the logical structure of a Daisy book. An image is shown in figure 4.2.

Figure 4.2: The logical structure of a Daisy book.

The NCC/NCX file is the main file, you can look at it as a table of contents, from this file we get information about where the chapters are to be found. Each chapter links further to a SMIL file and it also has an identifier, in most cases this identifier is the first id tag in the SMIL file, but not necessarily. The SMIL file consist of several passages, in each passage you can find an audio passage and/or a text passage (in the new standard it is also possible to link to an image). If there is a text passage, the passage is linked to a HTML/XML file with an identifier.

A Daisy DTB consists of XML markup text combined with sound.

This makes parsing one of the main tasks of libdaisy. The parser is divided into four parts, a NCC parser for parsing a Navigation Control Center file for the 2.02 standard, a NCX parser for parsing a Navigation Control File for the Z39.86-2005 standard, a SMIL 1.0 and a SMIL 2.0 parser. As the structure of the files to parse follows strict rules, we decided to make the parsers recursive.

The parser consists of many small functions which all parses one kind of tag each. For each tag the parser encounter, the function for parsing this tag is called. The different functions call each other recursively, so when the process returns to the caller function, the parsing is done. All functions return a value to its caller to tell if the job was successful or not. If a critical function fails, the error return value is returned all the way up to the top calling function and the parser returns an error.

4.4.1 NCC & NCX parser

The NCC/NCX file is, as mentioned earlier, like the table of contents in a book. It exposes the hierarchical structure of a DTB to allow the user to navigate through it. When a book is loaded by libdaisy the NCC/NCX file is parsed.

The top of the NCC/NCX file consists of the books meta information, title, authors, total time and such. This data is stored in a data structure and made available to the user at anytime after a book is loaded. In addition to meta data, the NCC/NCX contains information about the hierarchical structure of the DTB as references to SMIL files for each chapter or section. The parser stores the names and positions of the SMIL files and some meta data like title, id, audio reference and such. A two way linked list is made by structs containing data from one SMIL reference each. This linked list represents the hierarchical structure of the DTB and can be traversed both ways. This way we do not have to parse the whole DTB into memory at once, but are able to parse one SMIL file at a time. A pointer variable keeps track of what SMIL file is loaded, and the linked list makes it easy to get the next or last SMIL file and send it to the SMIL parser.

The NCC and NCX parsers works the same way and uses the same data structure to store data. The reason we have made a separate parser for both formats is because a NCC and a NCX file has different structure and tags. The NCX format is a new and improved version of

the NCC format. Because of the major difference between the structure of these two formats, we found it best to make two separate parsers to prevent messy code.

In figure 4.3 below, you can see an example of the structure of a NCC file.

```
<body>
    <h1 class="title" id="oeil0001"><a href="oeil0001.smil#hsnc_0002">Introduction</a></h1>
    <h1 id="yunb0002"><a href="yunb0001.smil#ewxm_0002">Windows Classic View and Other Assumptions</a></h1>
    <h2 id="yunb0003"><a href="yunb0002.smil#ewxm_0002">Windows Classic View</a></h2>
    <h2 id="yunb0004"><a href="yunb0003.smil#ewxm_0002">JAWS Tutor Mode</a></h2>
    <h2 id="yunb0005"><a href="yunb0004.smil#ewxm_0002">Access Keys</a></h2>
    <h1 id="ppkw0006"><a href="ppkw0001.smil#worh_0005">Starting JAWS and Setting Up Hot Keys</a></h1>
    <h2 id="ppkw0007"><a href="ppkw0002.smil#worh_0072">Setting Up a Hot Key to Start JAWS</a></h2>
    <h2 id="ppkw0008"><a href="ppkw0003.smil#worh_0023">Starting JAWS from the Start Menu</a></h2>
    <h1 id="ityg0009"><a href="ityg0001.smil#pkdf_0001">An Introduction to the Windows Desktop</a></h1>
```

Figure 4.3: An extract of an NCC file.

To learn more about the structure of a NCC or a NCX file, see the Daisy standards[1].

4.4.2 SMIL 1.0 & SMIL 2.0 parser

SMIL is a standard for definition and playback of multimedia presentations over the Internet. SMIL defines the sequence of playback for one or more media objects. In the case of DTB's, the primary media objects are audio and textual content files; SMIL provides for their parallel and synchronized presentation.

The SMIL parser can parse one SMIL file into memory at a time after the NCC/NCX parser is done and the linked list of references to SMIL files is made. When a DTB is loaded and the NCC/NCX file has been successfully parsed, the first SMIL reference in the linked list is parsed. The reference to a SMIL file is sent to the SMIL parser, which builds another linked list where each link contains the data from a single passage in the book, including text and reference to an audio segment. A passage can contain synchronized text and audio, just text or just audio. The SMIL file contains references to synchronized passages of text and audio. For each passage the parser opens a referenced XML file and retrieves the text matching a passage identifier from the SMIL file and stores the text in the data structure. For the corresponding audio segment it stores a reference to the audio file with start and stop offsets.

[1] Daisy technical specifications http://www.daisy.org/publications/specifications.asp

In figure 4.4 below, you can see an example of the structure of a SMIL file.

```
<body>
  <seq dur="85.542s">
    <par endsync="last" id="hsnc_0002">
      <text src="01_introduction.htm#hsnc_0001" id="nrxo_0001"/>
        <seq>
           <audio src="01_introduction.mp3" clip-begin="npt=0.000s" clip-end="npt=0.865s" id="audio_0001"/>
        </seq>
    </par>
    <par endsync="last" id="hsnc_0003">
      <text src="01_introduction.htm#hsnc_0002" id="nrxo_0002"/>
        <seq>
          <audio src="01_introduction.mp3" clip-begin="npt=0.865s" clip-end="npt=4.367s" id="audio_0002"/>
        </seq>
    </par>
    <par endsync="last" id="hsnc_0004">
      <text src="01_introduction.htm#hsnc_0003" id="nrxo_0003"/>
        <seq>
          <audio src="01_introduction.mp3" clip-begin="npt=4.367s" clip-end="npt=6.742s" id="audio_0003"/>
        </seq>
    </par>
```

Figure 4.4: An extract of a SMIL file.

To learn more about the structure of a SMIL file, see the Daisy standards.

4.5 Data structure

In order for libdaisy to run as a shared library, we chose to gather all the data in one struct and let the library functions be independent of global variables. When the library is initialized, the user gets a pointer reference which he will need to pass to all of libdaisy's functions. The user should never try to modify any of the data in this reference, and does not need to know anything about the data it contains.

Internally in libdaisy, all data (apart from local variables) is contained in one single struct. This struct contains data directly, or contains references to other data or structs which is used by the individual modules. The main parts of this data structure consists of pointers to the users callback functions, a pointer to the data structure for the audio engine and a pointer to the data structure for the parser.

Libdaisy is based on callback functions. When the data structure is being initialized with *daisy_init*, the user must supply pointers to a set of callback functions. These functions will be called by the engine to report to the user during playback. The main data structure also give the user the opportunity to give the *daisy_init* a void data pointer to any object or data structure as an argument. This data pointer will be available in all callback functions. The user can e.g. pass along a

GUI object in C++ so that you can output the text from the callback functions in the GUI.

4.5.1 Audio engine

The audio data structure contains data related to audio files and audio decoding. Notable data are file information, a buffer, start- and stop-times, a mutex lock, and callback function pointers.

Some information about and related to the the audio file are stored. This helps us in keeping track of what file we already have read into memory and lets us prevent loading the same file multiple times.

The buffer is filled with data that the decoder will process. The start- and stop-times are used to specify start and stop offsets in an audio segment.

The data structure also contains function pointers to the callback functions that needs to be available, and a mutex lock to provide a way to serialize access to the data.

4.5.2 Parser

The parsers data structure is created to hold the data parsed from the loaded Daisy DTB. It has a two way linked list for storing the data parsed from the NCC/NCX file and a pointer to the current playback position. This linked list represents the hierarchical structure of the DTB and is used to control chapter or section navigation.

The data structure also has a two ways linked list for storing the data parsed from a SMIL file. One SMIL file is parsed and stored in the data structure at a time, according to the position pointer in the linked list containing all SMIL information. This second linked list contains data from each synchronized passage in the DTB, like the text parsed from the XML files with reference to the corresponding audio segment, and a pointer to the current playback position. This linked list is used for passage navigation.

Figure 4.5 below shows an example of the two linked lists.

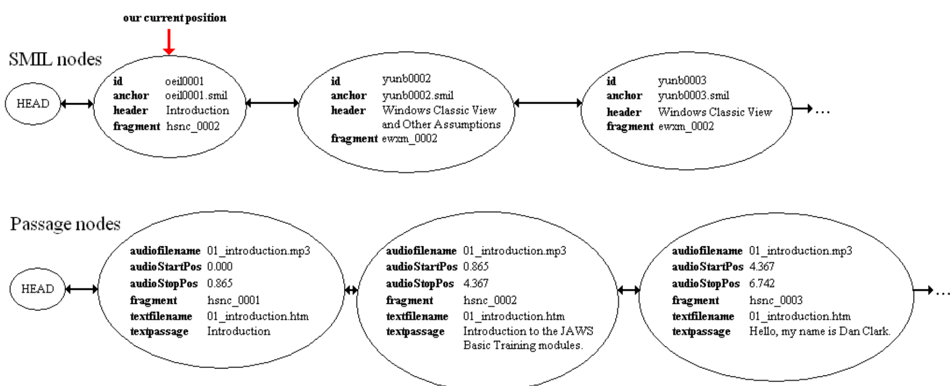

Figure 4.5: An extract of the resulting lists in the data structure

The parsers data structure also contain a struct for storing the loaded DTB's meta data, such as title and total playtime, a linked list for all the authors and a string representation of the path from where the DTB was loaded.

4.6 Libdaisy interface

When developing a library it is very important to make a good interface. The interface makes the functionality available to others, separating the implementation from the external communication with the user. This way the user does not have to worry about changes in the implementation. The interface should be as consistent and easy to use as possible, but without loosing it is user applicability.

We have tried to make a small, easy to use interface which require minimal knowledge about Daisy DTB's for the developer to make a Daisy player. We have also provided a number of callback functions that gives the front-end designer continuous information about the playback, errors, and a large degree of control over the Daisy engine. The callback functions informs the front-end about key-events in the Daisy engine, such as player progress, errors, finished passages, text or passage ID's.

4.7 Internationalization

Internationalization is a very important matter for many programs, and perhaps specially if the software is an Open Source project. We have a hope that people from different countries will be interested in this project and perhaps maintain our front-ends rather than creating a new front-end from scratch. In order to have a realistic hope for that to happen, we must use internationalization. Every single word and sentence we present to the end user must be able to be replaced by a localized alternative.

4.7.1 GUI front-end

Qt Designer[2] made internationalization very simple for us when we created the GUI front-end. Designer has very good support for internationalization, all we needed was to add a few line of source code to install a translator, and everything we need to translate will be automatically caught up. Sentences we hard coded in the source code, e.g the status bar message on figure 4.6 below, had to be placed inside a tr(...) in order to be processed by the translator.

```
statusBar()->message (tr ("File succesfully opened"), 2000);
```

Figure 4.6: An example how internationalization work with Designer.

When every sentence was placed inside tr(), we had to edit our project file and add translations files shown on figure 4.7.

```
31 TRANSLATIONS += translations/daisygui_no_nb.ts \
32                 translations/daisygui_en.ts
33
34 target.path = /usr/bin/
35 translations.path = /usr/share/doc/daisygui/translations
36 translations.files = translations/*.qm
```

Figure 4.7: Adding a new translation to the project file.

When that was in place, all we had to do was to run Qt's tool *lupdate <file name>.pro* and our ts files were generated. To edit the

[2]Qt Designer http://www.trolltech.com/products/qt/features/designer/

translations files we used another software from Qt called Linguist[3], a screenshot of the program is show in figure 4.8 below. All of the words and sentences we needed to translate was surveyable organized. When the translation was done we had to run *lrelease <file name>.ts* to generate the binary file used by the application. As we see in the project file in figure 4.7 the path to the binary files (*.qm) had to be put in its own folder.

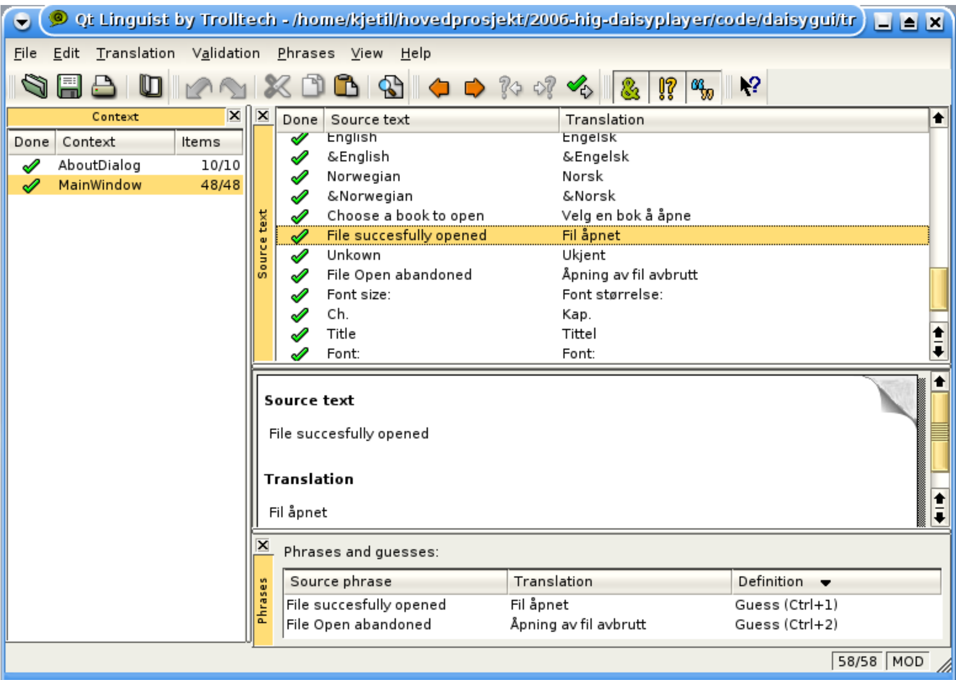

Figure 4.8: A screenshot from Linguist.

4.7.2 Console front-end

For the console front-end we used the GNU gettext[4] utilities for internationalization. Gettext is fairly easy to use and has a simple API. It gives the program access to message catalogues through the system localization variables. When the program starts, the gettext functionality searches for a message catalogue matching the system locale with

[3] Qt Linguist http://www.trolltech.com/products/qt/features/internationalization/

[4] GNU gettext http://www.gnu.org/software/gettext/

a translation file matching the programs domain. Gettext will fetch the translated strings from this file or the default strings if no translation file matching the specific language is found. A template of the translation file is public available so the program can be translated by anyone to any language. A translation has to be converted to a binary format with 'msgfmt' before it can be used. The program is set to search both the standard system path and a local directory for message catalogues.

Chapter 5

Implementation

As this project is an Open Source project we have tried to reuse much existing code and not “reinvent the wheel” more than absolutely necessary. Therefore, we have used existing libraries and standards to a large degree.

In this chapter we will look at which libraries we used and briefly explain why we believe they were good choices for this project. We will also explain how these libraries have been used in the project.

5.1 Choices of programming language and libraries

5.1.1 Programming languages

5.1.1.1 Ruby

Ruby[1] is a relatively new programming language, only about 10 years old, and has lately become more and more popular. It is a free interpreted scripting language and it is a very object-oriented programming language, every bit of data is an object. Further has Ruby good support to process text files and it is very simple and understandable. But due to slow parsing and difficult audio handling, which will be discussed in chapter 9 section 9.2.2, we dropped Ruby.

5.1.1.2 C

Since Ruby did not satisfy our needs, we had to choose another language. C and C++ became our main candidates. By searching the web we found endless discussions on why one was better than the other and vice versa. To get a glimpse of the discussions, read the article "Why don't C++ and free software mix?"[2] by Dan Egnor. For our library we decided to go with C because of it is wide use and because of C++ portability and binary linkage problem. We decided to use C as our programming language for libdaisy and daisyconsole, because it is a very widely used and powerful language, especially in the Unix world. We knew that this language would give us the speed we were looking for in our XML parsing. After some consideration, we went with the first standardized C known as ANSI C89 (American National Standards Institute (ANSI) adopted it in 1989). This is an old and widely adopted and used standard. Compilers for this language exists on almost every hardware platform. Writing our source code in ANSI C, gave us some strict rules to follow in our code convention. See appendix C on page 94. This to make our source code as easy to read and maintain as possible, giving the project a better chance of being adopted, than if our code was a big mess.

[1] Ruby Language http://www.ruby-lang.org/en/

[2] Why don't C++ and free software mix? http://www.advogato.org/article/207.html

5.1.1.3 C++

With our GUI front-end on the other hand, we decided to use Qt, see section 5.1.2.4, which made C++ a natural choice. C++ is like C a very powerful language, it was originally named "C with classes" because it was an enchantment to C with support for object orientation. The only difference we encountered in the little GUI code we made was the use of classes and how new pointers are allocated (*new* instead of *malloc*). We also had to define a standard wrapper around the libdaisy header file shown below:

```
#ifndef LIBDAISY_H_
#define LIBDAISY_H_
#lnclude <pthread.h>
#ifdef __cplusplus
extern "C"
{
#endif

...

#ifdef __cplusplus
}
#endif
#endif /*LIBDAISY_H_*/
```

5.1.2 Libraries

5.1.2.1 GStreamer

To begin with we chose GStreamer[3] to be our sound library. GStreamer is used by many programs, and look just like the kind of multimedia framework we needed to make our sound engine. It supported all the different audio file formats our Daisy player needed to support. So in the beginning GStreamer looked very promising, and we started using it right away. We soon discovered that GStreamer was not so easy to use. Because of Skolelinux using an older version of the library than

[3] GStreamer http://gstreamer.freedesktop.org/

we liked, and the poor API existing on the Internet. We used the first release cycle on getting GStreamer to work, without success. It was with big relief we decided to drop GStreamer.

5.1.2.2 REXML

In our Ruby experimentation period we needed a XML processor. The decision landed on REXML[4], which was well documented and had a good and understandable API. It had features like DOM and SAX parsing, full XPath support, good support for different character set and was written entirely in Ruby. By using REXML our code became easy and understandable, which is important for us since we are developing an Open Source project. So things looked really promising, and after a few hours of work we got a working parser which was able to parse the 2.02 standard. We tried different books and never got satisfied with the time it took to parse the meta data. It took unacceptable long time to parse a medium sized book. Ruby may not be the best language to use when we need fast parsing. Since we already had problems with GStreamer, we decided to drop REXML and Ruby and move to a lower level programming language.

5.1.2.3 Libxml2

When we decided to use C as our programming language we needed a new XML parser. After some searching we went for libxml2 [5]. Libxml2 is, as REXML, a library which is very well documented with a very thorough API. Libxml2 is written in C and is know to be very portable. Further libxml2 implements a lot of existing standards related to markup languages. The code needed to be written in order to parse was also fairly understandable and intuitive. We noticed a large improvement is parsing speed. What took several seconds with Ruby and REXML was now done in an instant. We had some minor problems in the beginning concerning that Debian uses an older version of libxml2, but after some examination we found good, working solutions. We can with big confidence say that we made the right choice using libxml2.

[4] REXML `http://www.germane-software.com/software/rexml/`

[5] Libxml2 `http://xmlsoft.org/`

5.1.2.4 GUI

We had some discussion on how we were going to implement a GUI front-end for our engine. In the beginning, we tried to make a web browser solution, but we were unable to find an easy way to interact with our engine. Since our main goal in this project was to make an engine, we decided to make a simple GUI front-end instead. Our decision therefore landed on Qt[6] as our GUI library. Qt is a class library that are written in C++, and since we used C in our code, it would not be any problem using the engine and front-end together. Another reason why we used Qt is that Skolelinux uses KDE[7] as its desktop environment. As shown in figure 5.1 KDE applications are written with Qt and KDE libraries on the top of two low level programming API's included in X Window System[8].

We first started to look at the new Qt 4 application, but soon realized that Debian Sarge only had Qt 3.3 by default. To make our front-end compatible with Debian Sarge, we decided to use Qt 3.3 instead. To create the GUI layout with its buttons and menus we used Qt Designer[9]. Designer also had a built in source editor to create functions and make the application do just as we wanted it to.

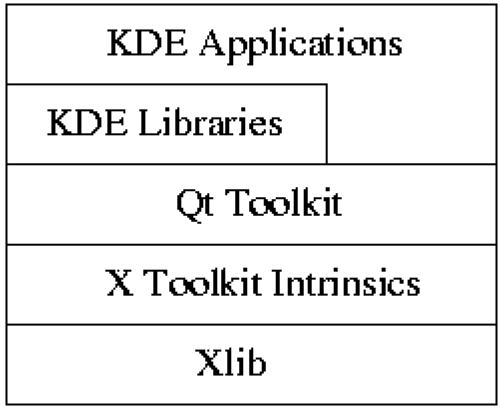

Figure 5.1: Showing the KDE architecture

[6]Qt 3 http://www.trolltech.com/products/qt/qt3/
[7]KDE http://www.kde.org/
[8]X Window System http://www.xfree86.org/current/X.7.html
[9]Qt Designer http://www.trolltech.com/products/qt/features/designer/

5.1.2.5 Audio design considerations

When planning the audio engine it is, of course, important that it follows the project guidelines with respect to licensing and platform requirements. This meant that decoder libraries and output libraries licensing should be compatible with DFSG[10], and that it should run on Linux platforms and be easy for users to acquire.

According to the Daisy ANSI/NISO Z39.86-2005, a player should be able to decode one or more of the audio formats

- MPEG-4 AAC
- MPEG-1/2 Layer 3 (MP3)
- Linear PCM - RIFF WAVE format.

During the initial examination, we found that the MP3 format were by far the most common audio format. We chose to give support for the MP3 format the highest priority. Later in the project, we had not come across one single MPEG-4 (AAC) or PCM-encoded DTB, so we decided to prioritize the development on other components. The audio module is designed with the AAC and WAV formats in mind, so implementing these once the need arises should go smooth.

Audio output library Libao is a cross platform audio library. It provides a very simple API, supports a large number of different platforms and is standard in most Linux distributions. It supports all the output devices and sound servers we hoped to be able to support - and more.

From the projects early stages, we had attempted to use GStreamer which proved itself to be a dead end. By examining other stable Linux programs which we have had good experience with, we concluded that libao was very commonly used. The facts that it was quite widely used, and that it had a relatively simple well-documented API, made us decide to use libao.

MP3 decoder MAD[11] is a high-quality MPEG audio decoder which supports MPEG-1 and MPEG-2, layers I, II, and III[12]. MAD provides

[10] Debian Free Software Guidelines http://www.debian.org/social_contract

[11] MPEG Audio Decoder http://www.underbit.com/products/mad/

[12] Commonly referred to as "MP3".

features such as 24bit PCM output and fixed point computation. The fact that MAD does not rely on a floating point unit were considered an advantage in the event of libdaisy being run on platforms that does not provide a FPU. By using MAD, we got a widely used MPEG audio decoder which followed the project principle of not bringing in unneeded dependencies.

Other MP3 decoders were briefly evaluated, but were quickly discarded. Some due to incompatible non-free licenses, and some for not providing the same degree of documentation and example code as MAD did.

AAC decoder FAAD was chosen as an AAC decoder, but AAC support was unfortunately not completed.

The status at this moment is that the AAC decoder can play audio output, but it does not yet fully support the seek, pause and stop functions that is required to play Daisy books. AAC support can be implemented in a relatively short time span if the need arises, but this were not prioritized during this project as we could not find one single Daisy book which actually used AAC encoded audio. As the Daisy standard only specifies that one of the audio decoders needs to be supported, we decided to use a pragmatic approach and that we would not prioritize AAC audio until an actual need arose (i.e. someone encountered a book which actually used AAC), or until we had the spare time-resources to spend.

There have also been some controversy regarding whether or not libfaad2 are compatible with the DFSG[13]. The AAC-related code in our project are optional and can be enabled or disabled during build time using a single "#define". The person who builds our software can make the decision himself based on the laws in his country and where the software will be distributed and used.

WAV support The WAV format is very well-documented and consists of a number of different "chunks", and audio data in linear PCM format. Stripping away the chunks and outputting the PCM data to libao should be a relatively easy task. WAV support in Linux programs have already been implemented numerous times, it is well-documented,

[13] A mail on debian-legal mailing list containing some information. `http://lists.debian.org/debian-legal/2006/04/msg00286.html`

and the group decided that it was safe to assume that this would be easily implemented if the need for WAV support arose.

This has, however, been given a low priority in this project because we have not found one single Daisy book which actually uses WAV audio. As with the AAC decoder, implementing support for this in the future should be a relatively simple task, but it is not prioritized until the need arises, or we have the spare time.

5.2 Threads

Threads are necessary in order to be able to communicate with the front-end and play audio at the same time. Threads potentially exposes the software to a whole lot of problems single-threaded programs does not suffer from, so we wanted to use as few threads as possible. Threads makes debugging more complicated, much harder and error symptoms are not necessarily consistent.

Libdaisy uses POSIX threads to do things like playing audio and handling input without blocking the engine or the front-end. Libraries that supports the POSIX threads interface exists on all modern operating systems. The facts that POSIX threads have been widely used through a number of years and is well documented with API, tutorials and example code, convinced us to choose the standard pthreads for our project.

Chapter 6

Management tools

Since this is an Open Source project, we decided early to avoid any proprietary programs like Microsoft Word or Microsoft Project. We felt it natural to follow the Open Source and free software philosophy related to open standards and free software. We stated that all the tools and formats we were going to use had to be Open Source and available to anyone. It is also important that everyone who are interested in this project is able to involve themselves.

6.1 Documentation and choices of aiding- and developing tools

6.1.1 Doxygen

One of our main goals was to have a good documentation to the API of our software. We decided that Doxygen[1] was a natural choice to help us with this task. Doxygen is a very common and widely used API documentation system for C and C++ among others. Below is an example on how Doxygen works, in figure 6.1 we see the source code, while in figure 6.2 we can see the generated API documentation.

```
408 /**
409  * Function to pause playback.
410  * @param daisy - the daisy data stuct which must be
411  *                passed along with all the API functions.
412  * @returns 1 on success, -1 on fail.
413  */
414 int daisy_pause (daisyplayer_t daisy)
415 {
416     struct_daisyplayer_t *_daisy = daisy;
417     if (DEBUG_DAISYPLAYER) report ("daisy_pause\n", REP_DEBUG);
418     return audio_pause (_daisy->audio_data);
419 }
```

Figure 6.1: Doxygen source example

int daisy_pause (daisyplayer_t *daisy*)

Function to pause playback.

Parameters:

daisy - the daisy data stuct which must be passed along with all the API functions.

Returns:

1 on success, -1 on fail.

Definition at line **413** of file **libdaisy.c**.

References **struct_daisyplayer_t::audio_data**, **audio_pause**(), **daisyplayer_t**, **REP_DEBUG**, and **report**().

Figure 6.2: Doxygen output example

[1]Doxygen http://www.stack.nl/~dimitri/doxygen/

6.1.2 LaTeX and LyX

We decided that all our documentation, not only the reports, should be available in the LaTeX format[2], so that we could focus on content and spend less time on manual formatting and document appearance. Two of the group members had some former experience with LaTeX, through the LyX[3] front-end, and the decision was relatively easy to make. We felt that LyX would give us the right output that we wanted without too much trouble. While LyX gives the user an easy way of generating LaTeX code, it can implement more advanced LaTeX code through use of ERT (Evil Red Text) boxes. Therefore we went with the LyX front-end, instead of other word- or document processors such as MS Word (who was not really an alternative since we wanted to use non-proprietary software) or OpenOffice Writer.

The LyX/LaTeX formats also has exciting advantages when it comes to our development environment. We use SVN for version handling, and because the LyX/LaTeX files are text-based, they have advantages over binary formats when it comes to version handling. Also, by using tools available on the developer machine [4], we could automatically generate up to date versions of the documents in several formats. By automating the task of generating updated documents, we saved some time on administrative work. We could focus on other parts of the project, knowing that the documentation on the web page would always be up to date.

Since the group did not have that much experience with LaTeX, we took the opportunity to use this project to really learn how to use it.

6.1.3 Subversion and TortoiseSVN

All developers for Skolelinux is using Subversion[5] as its version control system, so we did not had to worry about version handling and backup. For Windows XP we found the TortoiseSVN[6] client to be a really good alternative to the terminal based Subversion. TortoiseSVN is more intuitive when it comes to moving files, folders etc. - at least for people who are new to Subversion.

[2] LaTeX http://www.latex-project.org/
[3] LyX http://www.lyx.org/
[4] Devloper Skolelinux http://developer.skolelinux.no/
[5] Subversion http://subversion.tigris.org/
[6] TortoiseSVN http://tortoisesvn.tigris.org/

6.1.4 Planner

To make our Gantt-charts, we used an Open Source project management tool called Planner[7]. It is really easy to use, and supports all the features we needed.

6.1.5 Eclipse

To aid us in our coding, we chose Eclipse[8] as our development tool. Eclipse is an Open Source community which creates i.a. application frameworks for building software. It has good support for different programming languages by installing different plugins and extensions. To get Eclipse to support C we needed the Eclipse CDT package[9], this package has an editor, debugger and launcher, as well as a makefile generator. Eclipse is very intuitive and has a clear design.

6.2 Code quality tools

With our second release (the first release written in Ruby), we experienced some common problems related to wrong allocating and freeing of memory. To find bugs and improve the quality of our code, we used the tools mentioned below.

6.2.1 Splint

Secure Programming Lint[10] is a tool for statically checking the program source code and has provided good information on parts of our code that can be written better.

6.2.2 Electric fence

Efence[11] is a memory debugger written by Bruce Perens which we have used to some degree to debug our memory management. However, we have found Valgrind slightly simpler for our use.

[7] Planner `http://developer.imendio.com/wiki/Planner`
[8] Eclipse `http://www.eclipse.org/`
[9] Eclipse CDT package `http://packages.debian.org/unstable/devel/eclipse-cdt`
[10] Splint `http://www.splint.org/`
[11] Electric fence `http://perens.com/FreeSoftware/ElectricFence/`

6.2.3 Valgrind

Valgrind[12] is a GNU/GPL system for debugging and profiling Linux programs. It can automatically detect many memory management and threading bugs, making our program more stable. It pinpoints where the memory management bugs happens.

6.2.4 GNU Debugger

GNU Debugger[13] is the standard debugger for GNU software systems. It gives us a lot of good options when it comes to debugging programs. The features we found to be really useful were backtraces on individual threads. This has really simplified the work of debugging thread executions and have helped us locate some critical bugs.

6.2.5 GNU Binary Utilities package (*nm*)

nm[14] examines binary files (libraries, compiled object modules, shared-object files, and standalone executables) and displays the contents of those files, or meta information stored in them. *nm* is used as an aid for debugging and resolving conflicts. *nm* is a part of the GNU Binary Utilities package.

In our project, *nm* was used (together with Doxygen) mainly for locating leftover global variables in libdaisy.

6.2.6 Lintian

"Lintian dissects Debian packages and reports bugs and policy violations. It contains automated checks for many aspects of Debian policy as well as some checks for common errors."[15]

[12] Valgrind http://valgrind.org/

[13] GNU Debugger http://www.gnu.org/software/gdb/

[14] GNU Binary Utilities package (nm) http://www.gnu.org/software/binutils/manual/html_chapter/binutils.html

[15] Lintian http://lintian.debian.org/

6.3 Packing and release tools

6.3.1 Make

In order to create an executable file from our source code, we use Make[16]. To build a program we needed a makefile, where all the build rules are set. With Make we can also install the program to the users binary folder and it is also possible to uninstall a package. Some of the great advantages of Make is that the end user does not have to know anything about the details of what actually happens in the background. Further it figures out automatically which files that needs to be updates based on the changes in the source code. Make is also not limited to any particular language, we also used Make to generate the project report in PDF format from the LyX files.

6.3.2 QMake

Qmake[17] is a tool created by Trolltech which helps the user create a makefile. Qmake is very powerful and can create makefiles for different compilers and platforms. It can also be used if the source is not made with Qt[18]. Having such a tool saves us a lot of hours with hard and difficult work, and let us have full focus on the code. Qmake was used to help us create makefiles for the GUI front-end, all we needed was a simple project file, see figure 6.3 for an example, and run *qmake <file name>.pro*. Most of our project file was auto generated by Qt Designer[19], but we had to modify it to fit some of our special needs, like for instance to include translation files and where to put them. This is discussed in section 4.7 on page 29.

[16] Make http://www.gnu.org/software/make/
[17] QMake http://doc.trolltech.com/4.0/qmake-manual.html
[18] Qt 3 http://www.trolltech.com/products/qt/qt3/
[19] Qt Designer http://www.trolltech.com/products/qt/features/designer/

```
 1  TEMPLATE        = app
 2  LANGUAGE        = C++
 3
 4  HEADERS += src/foo.h
 5
 6  SOURCES += src/main.cpp \
 7          src/foo.cpp
 8
 9  unix {
10    SOURCES += src/foo_unix.cpp
11  }
```

Figure 6.3: A simple example on a project file.

6.3.3 Debian Package building tools

NUUG hosted a lecture in Oslo on how to make Debian packages, this lecture was recorded and put on the NUUG web page. This intro on how to make a Debian package, and the information we read on the Internet, were the only experience we had with creating Debian packages. There are many different tools you can use to aid you in creating a Debian package. Since we were fresh to the whole Debian package building scene, we started out doing very much of it manually because we felt we had best control this way.

After much help from Petter Reinholdtsen in setting up an environment for using debhelper, we finally went over to a more automatically way of making our Debian packages. "Debhelper is a collection of programs that can be used in a debian/rules file to automate common tasks related to building Debian packages. Programs are included to install various files into your package, compress files, fix file permissions, etc. Most Debian packages use debhelper as part of their build process."[20] This automatize our package build phase, and made it easier for us to make our Debian packages accordingly to the Debian Policy Manual[21].

The environment Petter Reinholdtsen helped us put together also made the tarball for us, making the whole Debian packaging process automated. One of the things we had to take care of regarding Debian packages, where version numbering. We had not any experience on how to version numbering our release, therefore we decided after some examination to use the numeric versioning scheme[22]. Here we use three different numbers separated by periods to give the software a unique numerical identifier, where the first number is the major number, the

[20] Debhelper http://packages.debian.org/stable/devel/debhelper

[21] Debian Policy Manual http://www.debian.org/doc/debian-policy/

[22] Software Version numbering http://en.wikipedia.org/wiki/Version

second is the minor number and the third is the revision number. This making our first release being named “daisyplayer_0.0.1_i386.deb” where “0.0.1” meaning that it was our first release.

6.4 Scripts

We have had shell access to the developer server[23] and opportunities to run scripts and automate tasks. We have used this on some areas, and it has helped us during our project.

```
project: project.tex
        pdflatex project.tex
        makeindex project.idx
        pdflatex project.tex

        latex_count=5 ; \
        while egrep -s 'Rerun (LaTeX|to get cross-references right)' project
            do \
              echo "Rerunning latex...." ;\
              pdflatex project.tex ;\
              latex_count=`expr $$latex_count - 1` ;\
            done
```

Figure 6.4: Part of the Makefile used to generate this report.

The two main areas where we have used scripts have mainly been to produce up-to-date documentation and to generate source code documentation using Doxygen.

Our scripts generally consists of bash scripts and makefiles. Makefiles are very helpful and helps us only generate new documentation if any of the source files have changed.

The main idea behind the use of scripts are that we should spend more time working on the project and not spend it on repetitive tasks that can be automated.

[23]Skolelinux developer server. http://developer.skolelinux.no/

Chapter 7

Testing

Testing is a vital part of every software project. Many bugs and errors are encountered and can be fixed during extensive testing. It is easy to not see or comprehend different bugs in the source code if the projects testing methods is not efficient enough.

7.1 User testing

As a mixture of evolutionary system development and Open Source development, testing was a major part of our project. Like figure 7.1 indicates, testing and validating the product is an important task. When working after this specific system development model, we had to test our product continuously during the project.

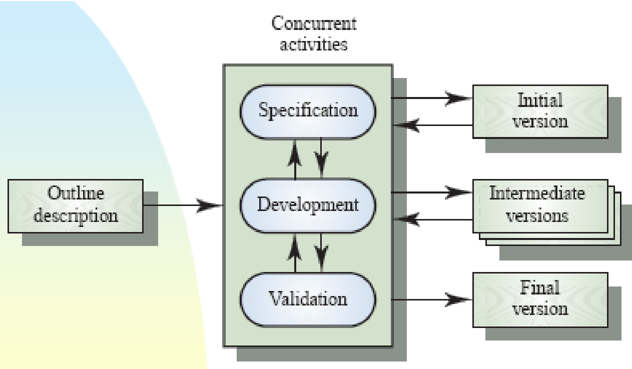

Figure 7.1: The evolutionary system development model.

One method we used a lot, was testing each others code and ideas. As a four person group, this method worked very good. Many bugs and errors where encountered and fixed together. This method also gave us a good understanding on what the different members of the group where working on and made us learn much from each other on various subjects.

Early in the project we found some different Daisy books, giving us the ability to test our DaisyPlayer against different books. Some of the Daisy books followed the 2.02 standard and some followed the ANSI/NISO Z39.86-2005 standard, some with audio and others with only text. From our experience, many of the Daisy books that are available follows the Daisy standards to a varying degree. Testing with several real books has allowed us to make libdaisy more tolerant and able to parse many books that does not adhere completely to the respective standards. This variety of Daisy books has been important during the tests of our DaisyPlayer.

Because we did not want to have a lot of users reporting and commenting on issues we already worked on, we did not include as many

external test users in the early phase of the project for a number of reasons.

We were aware of many bugs and spent much time fixing them. The extra load of managing bug reports, suggestions and repeatedly talk to test users about bugs that we already had filed, would not benefit the project - at least not in the early phases. We felt it would be best to test each others code out internally in the group, and rather ask for help if we needed it.

The technical nature of the project would have required the testers to be quite familiar with program development on Linux and, even then, would require a large effort to test our software.

Once we had front-ends that showed libdaisy functionality, we received some helpful feedback and tips from persons within the Skolelinux community who had tested it. We would like to have included more end users, if we have had an user friendly front-end earlier. The nature of this project made this hard for "regular" Linux users to really test our software until perhaps in the last phases, where we had created user-friendly front-ends.

7.2 Product quality

We really put a big effort in making the product as good as possible. A lot of time went into making the engine stable. Since we used C as programming language we had to be extremely careful with our pointers when it comes to allocating and deallocating memory. Every time we used a pointer we had to check whether it contained valid information, or just points to NULL, to avoid segmentation faults.

To help us in the debugging process we added debug information in every function that got called. This debug information can easily be turned on and off, and it is also possible to choose which C file to debug.

We followed the ANSI C standard in the libdaisy code, to ensure the quality and portability of the source code.

7.3 Product quality tools used

Another method we used to test our product, was using a number of different tools for sorting out common mistakes. See section 6.2 about

code quality tools on page 43, for some information on the tools we used to test our product.

Valgrind We used Valgrind[1] for debugging our program, and found and fixed many memory management and threading issues. Valgrind is very good at reporting when memory are either not freed, or freed multiple times.

Electric fence (Efence) Efence[2] aims to debug two very common programming bugs. Efence will monitor both read- and write access to areas that have been malloc()'ed and report when the software touches areas that are outside the malloc()'ed area or that have already been free()'ed. Efence has, in combination with Valgrind, been very helpful in detecting problems where memory have not been handled properly.

Splint Splint[3] statically checks the C code for security vulnerabilities and coding mistakes. The error-messages from splint have been very detailed and extremely helpful in increasing our code quality.

GNU Binary Utilities package (nm) The GNU Binary Utilities[4] provides "nm" which we used to detect and remove global variables in our library.

GNU Debugger (gdb) GNU Debugger[5] has been great help when it comes to locating the exact places of critical errors, or to inspect states while programs and libdaisy are running. The bugs that caused segfaults were usually easy to pinpoint thanks to gdb.

Lintian We used Lintian[6] to validate and check our Debian packages for common mistakes. This allowed us to make our Debian packages accordingly to the Debian Policy, and we discovered a lot of bugs with our packages. We encountered many bugs, with the help of Lintian,

[1] Valgrind `http://valgrind.org/`
[2] Efence `http://perens.com/FreeSoftware/ElectricFence/`
[3] Splint `http://www.splint.org/`
[4] GNU Binary Utilities package `http://www.gnu.org/software/binutils/manual/`
[5] GNU Debugger `http://www.gnu.org/software/gdb/`
[6] Lintian `http://lintian.debian.org/`

when we created Debian packages. Lintian was an important help to ensure product quality of our software packages.

These different ways of testing our product, made it possible for us to find and improve a lot of different bugs. This ensured product quality throughout the project.

Chapter 8

Public relations

As this is an Open Source project, public relations is actually very important. To make useful software, we must get people to use it as well as generate interest in our target groups. As an attempt to make the DaisyPlayer Project known among the Open Source community, and people who use Daisy products, we have focused on reaching potential interested people. We have put effort into creating a good web page, and we have been present on several IRC channels almost at any time of the day. We posted messages on mailing lists and forums when we had something that might be of interest.

8.1 Web page

In an attempt to make the project known, we decided to create a web page for the DaisyPlayer Project as early as possible. We also wanted to keep the web page as easy as possible, but also informative as you can see in figure 8.1 on the following page. The reason we wanted to make a good web page is that this is where many new users will get their first impression of the project. If the page is a real mess most people will loose interest in the project. During these months the page has gotten almost 900 hits from users all around the world and if you search after 'Daisyplayer' with Google[1] our project appear as one of the first hits. [2]

[1] Google http://www.google.com/
[2] As of May 2006

Figure 8.1: A screenshot of the DaisyPlayer Project web page.

- On the main page you will get some brief information about the project and the latest news. There is also a link to a Frequently Asked Questions (FAQ) page.
- In the news section you will get all the news and can read about the progress of the project.

- On the progress page you will see our Gantt-chart, showing our project schedule.
- The download page contains Debian packages and tarballs of all our releases, you will also find information on how to download our repository. We have added some screenshots of our front-ends.
- In the documentation section you can download PDF's of our reports and you can find the documentation on our source code and API to the library.
- The developer page contains information about the developers of the DaisyPlayer Project.
- On the contributor page you will find information about the contributors of this projects.
- On the links page you will find links to all libraries, software and technologies relevant to the project.

8.2 Plan for advertising

One of the best ways of advertising and presenting the project was to have a nice and updated web page. We put a lot of effort into the web page, both with regards to content and design. One issue related to our advertising plan, was that we did not want to advertise to much before we had a decent product. Because of this, in the beginning of the project, the only advertisement we had were our web page and through the #skolelinux IRC channel.

When we had a product worthy of advertising and the need for user feedback emerged, we notified the following Linux forums and mailing lists:

- Debian[3] - under “General Discussion”
- Ubuntu[4] - under “Ubuntu Cafe”
- Gentoo[5] - under “Unsupported Software”

[3] Debian forum http://forums.debian.net/
[4] Ubuntu forum http://www.ubuntuforums.org/
[5] Gentoo forum http://forums.gentoo.org/

- Linuxforum[6] - under “Linux Software”
- Linuxlinks[7] - under “Software”
- Skolelinux and DebianEdu mailing lists.

In addition we registered an account on Freshmeat[8], where we made our project easier to find.

We also spent some time updating Wikipedia[9] with some of the information and knowledge about Daisy that we had gathered during this project. The Wikipedia entry for the Daisy format was minimal and in quite a bad state. By writing about what we have learned during our project, we could give knowledge back to the community through Wikipedia, and we could make available our information about the DaisyPlayer Project. Naturally, it is extremely important to us that our contributions benefits Wikipedia, and that it should not be interpreted as an “advertisement” or be irrelevant to Wikipedia.

[6]Linux forum http://www.linuxforum.com/forums/
[7]Linuxlinks forum http://www.linuxlinks.com/portal/phpBB2/index.php
[8]Freshmeat http://freshmeat.net/
[9]Wikipedia http://www.wikipedia.org/

Chapter 9

Discussion of results

On the following pages we are going to discuss the resulting product we ended up with, our development environment and project management tools. Further we are also going to discuss the experiences with the libraries and technologies we have used, evaluate how we have been working during this project and what we have learned.

9.1 Evaluation of the result

We are very pleased with the work we have done on this project. We have learned a lot and gotten valuable experience in a number of different areas. It has been very rewarding to work on a big project for such a long period of time and we are proud of what we have achieved.

This was supposed to be a start on an Open Source project and we feel that the project have all the possibilities to be an active project also after our project period is over. As we already have mentioned, we put a big effort into make the web page[1] look nice and tidy. We are very happy with how the page looks, with its nice design and informative text. The web page is written in strict XHTML and CSS, and gives this project the serious web page it needs to attract users and developers.

We are also very pleased with our documentation, created with the help of Doxygen[2]. Everything we have written is in English, so we all got the opportunity to freshen up our English. As you can see in our source code documentation and the API, we have commented the code very well, so outsiders easily can understand what we have done and why.

Since we decided to use C as our programming language, it brought in its train a big challenge. We soon discovered that we had to be very disciplined when allocating and deallocating memory. We always had to check our pointers, and we got a big portion of our problems in the world of threads. This time consuming labour made us learn very much about programming techniques and how to program more complex solutions.

When it comes to the visible result we are very pleased, even though we know lots of improvements and extensions which could have been done. Since this is our first project of such size we are proud of what we have created. The library with both its parser- and audio-part are working well. We only wish we could have had some more time on this project so that we could support more of the complexity in the Daisy standards and get more people involved, now that we have front-ends to show to. The front-ends we have created to test the functionality of the library are very plain and simple. As mentioned earlier in the report, our main focus was on the library. We also feel that we have

[1]The DaisyPlayer Project `http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/`

[2]Doxygen `http://www.stack.nl/~dimitri/doxygen/`

kept the software dependencies to a absolute minimum. When we look at the goals we decided in the pre-project, see section A.1 on page 76, we can with good confident say that we have successfully met our goals.

The guidelines we sat for this project in the project outline document found on our web page[3], have also been followed. We have documented all our sources with the utmost respect to the authors and we have used the DFSG approved license GNU/GPL[4] on all our work. We have also made the front-ends support internationalization, the GUI front-end is translated to English and Norwegian, and created manuals for them, see Appendix H.

9.2 Evaluation of choices and technologies

9.2.1 Development model

As explained in the pre-project report on page 82 and in section 1.8 on page 6 , we chose a mixture of evolutionary and Open Source development model. This model fitted the project and the group well. It required very good communication and cooperation. Fortunately, the group could fulfill those requirements. We knew from the very beginning of the project that communication would not be a problem, as all the group members were neighbours. We have also worked together a lot in the past and use other means of communicating such as IRC and e-mail.

The model we decided to use was very open for changes during development. This meant that we could develop more freely than, for instance, if we had decided on a pure incremental development model. To give the project some time frames and goals to work against, we created two weeks long release cycles. This mean that we would make a release of our product every two weeks, forcing us to set achievable goals and structure our work according to the time boundaries. The model worked well for this project, and provided structure to our work. One issue with the release cycles were that we spent a lot of time to prepare and make Debian packages. A consequence was late evenings

[3]Project Outline http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/project_management/reports/project_outline/project_outline.pdf

[4]The Debian Free Software Guidelines http://www.debian.org/social_contract

before each release. A number of times, we were forced to work on the release until close to midnight.

One feature we included in our development model that we did not plan, was the use of eXtreme Programmings pair programming. On the difficult parts we tend to ask each other for help, which resulted in much use of pair programming. This way we sorted out many difficult issues. While most programming was done individually, pair programming proved itself to be incredibly (and surprisingly) helpful when programming and debugging hard parts of our code.

The Gantt-chart, on page 97, which we followed through the whole project fitted our needs good, but in retrospect some changes could have been done to it. The first release cycle could have been longer, maybe the two first release cycles could have been merged into one. The project did not have any set dates where we had to complete one specific part of the engine, so having release cycles lasting two weeks fitted the project nicely.

9.2.2 Choice of programming language

In our choice of programming language, we decided early on in the project that we did not have to pick one and stand by it throughout the project, but keep our options open. We discussed and examined different languages. Together with our supervisor we decided that Ruby would be a good language to learn and use.

We tried using Ruby early on, but concluded that it was not the right language for this project. It did not fulfill our requirements, neither when it came to audio decoding and output, nor XML parsing. One problem the Ruby language presented us was that the XML parsing went really slow, even though we did use SAX parsing, and not the slower DOM parsing method. This was a major issue, and we could not settle for this XML parsing alternative. When trying to handle the audio output, Ruby did not prove itself to fulfill our needs either.

We decided to change to C which we have had some previous experience with. We knew that with C we would have good support for audio decoding since it is a very widely used language, and most audio libraries are already implemented in C. Our experience from using C with libraries like MAD and FAAD are that the programmer has better control over what happens and how it happens. Decoding and playing audio with MAD and libao were implemented without any of

the strange undocumented issues we had experienced with GStreamer. This project have benefited greatly from using these well-used well-documented libraries.

We knew that the parsing would go quicker with C since it is a compiled language instead of Ruby which is interpreted, and quickly registered that parsing speed was no longer an issue. In most cases, the parsing of an entire chapter was done in the blink of an eye. The only problem is when the chapter is very big or, in a worst case scenario, the whole book consist of only one single chapter. We discussed whether we should parse smaller segments in cases of very large chapters, but new issues with a higher priority needed to be sorted out, so we have not had time to tackle this potential issue yet.

With implementing the library in C, came a large responsibility of creating robust code. We needed to be more conscious of everything we did, because C lacks the safety features many younger programming languages have. We quickly concluded that we had to be very strict with our pointers and always check for NULL-pointers to make sure we worked on existing data and to prevent segmentation faults. Also, allocated memory needed to be manually deallocated when it is no longer in use. C++ would probably be an easier language to use as it is object-oriented, but using C have taught us to be a whole lot more disciplined and aware of what we do when we write code.

The advantages we achieved from using C were fast code and a very large degree of control over what the computer are instructed to do. Finesses such as the "register" keyword which instructs the compiler that a variable will be used frequently, and that the programmer wants it (if the compiler allows) to exist in the CPU register rather than in memory makes C a rewarding language to use.

Using C++ for our GUI front-end did not pose as a problem because it can easily implement C libraries. A minor wrapper around the libdaisy header file was all we needed in order to link against our library.

9.2.3 Choice of GUI

Since none of us had any wide experience creating GUI with languages like C or C++, we wanted a tool for helping us with this task. We read a lot about different possibilities, and tried e.g. an embedded web

browser created by a tool called Glade [5]. Glade is a free user interface builder for GTK+ and Gnome. Trolltechs Qt[6] which KDE is based upon, could give us what we wanted much easier and quicker, so the decision landed on Qt. After some weeks experience with developing with Qt, we feel that we chose correct. We are very happy with the result, and will most likely chose Qt again later. It is very well documented and the library is easy to use. It was also a real big plus to use QMake to create our makefiles, that saved us from lots of hours with hard work. Another plus is how easy Qt solves internationalization.

9.2.4 Choice of libraries

9.2.4.1 POSIX threads

We decided to use POSIX threads for our library due to the fact that it is widely used, well-tested and exists on almost all platforms. As Linux are the primary platform, we chose to link against the libpthread shared library.

We expected that programming with threads would cause some problems and we were correct. We have had quite a lot of problems caused by issues related to how we used threads.

One error we made was to first write code that handles audio playback without bringing in threads. Once we had successful audio playback, we tried to change the existing code to use threads. The result of this was quite messy code and bugs which were hard to track down.

For future development, a redesign of the audio engine with focus on threads rather than just audio playback will benefit this project enormously. This has been added to our future plans (see section 9.4 on page 70), and have high priority.

9.2.4.2 MAD - MPEG Audio Decoder

We chose MAD for a number of reasons. MAD is:

- Well documented.
- Provides a good, clean API.

[5] Glade http://glade.gnome.org/

[6] Qt 3 http://www.trolltech.com/products/qt/qt3/

- Widely used by a number of good applications. (AlsaPlayer, MPlayer, mpg321, ScummVM)
- Does not depend on a FPU.

MAD is also known to deliver very high-quality audio compared to many other decoders, though this were not considered to be important for our project.

In our experience, MAD have been very good to work with. The decoder was very simple to use. Many of the alternatives were to take code from other MP3 players and try to make it work in our project, which would probably not have given us the same result. MAD provides example code, and it is widely used in other Open Source projects. This helped us greatly as we had much code to look at which provides helpful hints about how to best use MAD in our project.

The choice of using MAD for our decoder was a good choice which we have had no reason to regret.

9.2.4.3 Libxml2

The main reason we chose libxml2 is that we knew it was used in numerous other projects, so it is well-tested and used. Also, we quickly noticed that libxml2 was well documented and had a good API. There was also some simple code examples which helped us in the beginning.

Another reason why we chose libxml2 is that the library is very portable and could easily be ported to e.g. Windows or Mac OS. A minor issue in the beginning was that not all the functions in the API was supported in the version of libxml2 which is supplied with Debian Sarge. But after some reading and testing, we managed to solve them nicely. So we are very happy with our choice choosing libxml2 as our XML parser.

9.2.4.4 Libao

We used libao to output decoded audio from the decoder. Libao is a cross-platform audio output library with a very simple, well-documented API and provides good example code.

Libao supports all the project requirements such as low number of dependencies, portability, and it supports all the required audio sys-

tems (ARTS, OSS, ALSA, ESD). Libao will automatically use the users default audio driver.

Libao served us well in this project, and saved us from creating our own interfaces towards ARTS, ESD and ALSA. All audio output functionality is hidden behind a very intuitive API.

9.2.5 Choice of documentation tools

9.2.5.1 Doxygen

We all had little or no experience with Doxygen[7] before we started this project, but we knew that Doxygen could help us a lot in order to create good documentation from our source code. In order for Doxygen to work for us we had to comment our source code in a special way, but that was no problem for us since we were used to doing this from previous projects. We are very happy with our choice of Doxygen, it created a good documentation of our library[8] and a very nice API, see appendix G. Another great feature is that it auto-generated up-to-date documentation throughout our development phase.

9.2.5.2 LaTeX and LyX

Two of the members of the group have had some minor experience with LaTeX and LyX from previous projects, but none of us could call ourselves experts with this technology. We knew that it would be a great documentation tool, when we learned to use it. It took some hours to learn different aspects of LyX, but we feel that it was the right tool to use. It eventually gave a very good result and we are very pleased with the layout of the report. It is also really great that with a simple little script all the PDF's are automatically generated when needed from the LyX files.

For this project, the combination of LaTeX and LyX have really paid off. We have trouble seeing how the same result could be achieved using traditional word-processors with binary file formats like for example Microsoft Word. The combination of SVN and text-based file formats enabled the four of us to work simultaneously on a relatively large

[7] Doxygen `http://www.stack.nl/~dimitri/doxygen/`

[8] Libdaisy Documentation `http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/documentation/doxygen/html/index.html`

document like this report and still keep conflicts to a minimum. The creating of contents, index, list of figures etc. are handled automatically and saves the group for keeping track of things like this manually.

The only problem we had with LyX was that the version on Skolelinux's server are quite old. That gave us some complication with the auto generation of the PDF's, but we found ways to solve this. For future reports, LaTeX and LyX will be a natural choice.

9.2.6 Evaluation of development environment

9.2.6.1 Eclipse

Since we already were familiar with Eclipses powerful development tool, we knew when choosing Eclipse[9] we would not regret it. It is a great free development tool, and it has saved us a lot of unnecessary work. Eclipse does not only create makefiles for us, but it has a very great debugging tool. Further, it is very well arranged and it helps us in saving time with minor details like auto-complete variable and function-names. Eclipse also showed us the location of warnings and errors in the code. The only negative side with Eclipse is that it sometimes used a lot of memory, and since some of us used a virtual machine with limited memory, Eclipse sometimes crashed.

9.2.6.2 Scripts

The use of scripts have been of great assistance to us during this project. As an example, one of the scripts helped us constantly updating our code documentation as changes was submitted. We have been able to spend time on other tasks, knowing that we would not need to update documentation manually whenever we made a change. By spending time making the scripts, we saved a lot of time we normally would have used on tasks that are now automated.

9.2.7 SVN

We were all excited about using a repository and version control system, as we had never used one before. Since we were developing for Skolelinux we did not have any choice on what version control we should

[9] Eclipse http://www.eclipse.org/

use. We were all amazed how incredibly easy SVN[10] was and how great it is to work with version control. It only took a few minutes to set up the framework and we soon came into a good routine where we updated and committed changes often. We also tried to always commit source code that worked so that another could work on a different part of the engine without worrying about compilations errors.

The only real problem we had with conflicts came when we worked with LyX, these conflicts came because we used both LyX on Windows and Linux[11]. We solved this problem by only using LyX under Linux. Early in the project we learned the habit of checking difference with our working copy with what lies on the server. The reason we did this is to reduce the chance of conflicts, so we were very thorough when we submitted something. During this project we have a total of 2000 commits, and a total of 9400 file changes. For us SVN has been a great tool that is extremely useful, and we all have had great use of this system.

9.2.8 Problems encountered

When we started this project we knew we had to read and follow the specification on the structure of a Daisy DTB. We downloaded the ANSI/NISO Z39.86-2005 standard[12] and started reading. The standard was quite extensive, so it took some time to get to know it.

When we started implementing the first simple version of the parser, we started looking for Daisy DTB's for testing. We encountered some free books on the Internet that we could download, and later we got hold of retail CD versions of several books which we could borrow. None of this books followed the standard we had. It was then we realized it was several older versions of the standard, and the newer standards was not backward compatible.

At first we thought it had to be only minor changes and just additional tags that would differ between the standards, but after looking closer on the other standards we found them to differ quite a bit. Making a parser that would support all the standards would take way to

[10] SVN `http://subversion.tigris.org/`

[11] Windows- and Linux newlines differ, so whole files were submitted rather than just a patch.

[12] ANSI/NISO Z39.86-2005 `http://www.niso.org/standards/resources/Z39-86-2005.html`

much time. Both the implementation of the parser and us getting to know all these standards would take a lot of time.

We decided to do some examination on which standard today's books were normally using. It did not take to much time to find out that the second newest standard, called Daisy 2.02[13], was the most used. In fact, we had a hard time finding books using any of the other standards.

Based on our examination, we decided to concentrate on the 2.02 standard, but also implement as much as possible of the newest standard, Z39.86-2005. Looking back now, it may have been better to concentrate on just one standard, the 2.02 standard, because this is most widely used. Implementing these standards were very time consuming, and it would be better to fully support one, rather than partially support two. But since we have concentrated on the most important functionality, we have come out of all of this with a parser and library which gives the user enough functionality to enjoy a Daisy DTB.

After implementing a parser which supports enough of the standard to parse a Daisy DTB, we still got a lot of errors when we tested it on the books we had. After checking the markup of the DTB's, it seems like they did not follow the standard all the way. We have yet to discover a book that validates a 100% according to the standard. Because of this we had to use more time on the parser, to make it more tolerant to errors in the markup.

9.2.9 Evaluation of the PR work

In the final stages of this project, PR became a very important task. We put a lot of effort into advertising our project, and stuck to our advertising plan. The plan made it easier for us to advertise the project, and saved us a lot of time. The effect of our PR work was reflected in the amount of hits on our web page. After we executed our advertisement plan, more people become aware of the project, resulting in more hits on our web page.

We decided that it was very important for us to quickly give good replies on enquiries related to the project. Anyone who were interested or had questions should get a good response as soon as possible. A step to ensure this was to register an e-mail address that forwards everything

[13]Daisy 2.02 standard http://www.daisy.org/publications/specifications/daisy_202.html

to all project members. A new enquiry will notify all four developers, and we should be able to quickly provide an answer.

Contributing to Wikipedia[14] was not done primarily to promote the project, but to share information we had acquired during the project, and to help increase the general awareness of the Daisy standard. The result is an increase in the Daisy-related information available on the English Wikipedia, as well as a small article about Daisy on the Norwegian Wikipedia. The DaisyPlayer Project are only mentioned in the English version of Wikipedia.

Public relations and the projects image are very important to Open Source projects like this which relies on reaching interested people. With our clean, tidy web page and good focus on documentation, we believe we have started a good project which can attract interested developers and users. Even though we have not got as much interest and feedback from our advertisement as hoped, we feel pretty certain that in the future this can change. When people start to use the Daisy standards more, people will look for players, and then our project will become very useful.

9.3 Evaluation of the groups work

The group cooperation has been very good, and we have had at every point good communication within the group. One reason for this is that we all knew each other very well before the project started, and have worked together on previous school related projects. It also help a lot that we all live in the same hallway, so we could easily go to each other and ask if there was something we wondered about. Another great advantage with living so close is that we could take meetings whenever we felt for it without making appointments and such. Already from the beginning we divided the work and responsibility within the group. When it was time to develop a GUI we divided the group in two. We all feel that the we have worked very well. All the time we have put into this project is documented in "work log"[15], which we have been very thorough with updating.

We are very satisfied with how the group have worked together, ev-

[14] Wikipedia `http://wikipedia.org/`

[15] Work log `http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/project_management/project_files/work_log.txt`

eryone have put a lot of effort into this project. The pair programming mentioned in section 9.2.1 on page 61 could not have been done without the groups cooperation being as flawless as it was. From the "work log" you can see that it has been put a total of 1800 hours effective working time into this project.

9.4 Further work on the project

There are some features that this project could have implemented, but either we have not got the time to implement or we have not felt that we needed to implement it. These are all things that can be implemented in the future. Much of the future work should be to test and stabilize the product. Especially should the library get more testing then that we got time to.

Features to implement in the future are as following:

- Support for the Daisy 2.0 standard.
- Redesign the audio engine. Keep a clear focus on threads and engine states. Create a cleaner interface towards the rest of libdaisy and separate the audio-engines data from libdaisy's data.
- Utilizing more of the information that comes with the Daisy standards.
- Parse only part of the chapter if the chapter is huge.
- Port both the library and the front-ends to Windows and Mac.
- Use automake and autoconf.
- Adding the front-ends to the KDE menus.
- Get more languages supported for the front-ends.
- Add support for Braille.
- Test/modify or develop a front-end specially fitted for people who are blind or visually impaired.
- Front-ends that remember progress, configuration settings etc.
- Speech-synthesis for books which lacks audio.

Chapter 10

Conclusion

We have spent this last half a year on this project as our bachelor assignment in our last semester at HIG[1]. This project has been a great experience for us and has taught us much about software development, especially Open Source development on the GNU/Linux platform.

We have come a long way since the beginning of this project. We started out with something completely new to us and have had to work our way from there. We have investigated and learned to use many libraries, standards and techniques. Our contact with the Open Source community has given us a lot of experience in cooperating with users and other developers. It has also given us much insight and thoughts on how most projects needs to be advertised and marketed in order to get users and developers interested.

Teamwork has been a key factor in this project, as we have been working as a group of four students. Working as a group has taught us a great deal on dividing work and responsibility. It has been very important for us to learn how to work good as a team, as almost all development projects is done by more than one person.

The experience and knowledge we have acquired throughout this project have given us valuable skills and competence which we will bring along into future projects after we graduate at HIG.

[1]Gjøvik University College http://www.hig.no/eway/default0.asp?pid=248

Chapter 11

Bibliography

- Brian W. Kernighan, Bob Pike. *The Practice of Programming.* Addison-Wesley, Inc., 1999.
- Dimitri van Heesch. Doxygen Manual, 2006. http://www.stack.nl/~dimitri/doxygen/manual.html.
- Eric S. Raymond. *The Cathedral and the Bazaar.* Eric S. Raymond*, 1997. http://www.firstmonday.org/issues/issue3_3/raymond/index.html.*
- Gnome. Reference Manual for libxml2, 2006. http://www.xmlsoft.org/html/index.html.
- Josip Rodin. Debian new maintainers guide, 2005. http://www.debian.org/doc/maint-guide/ch-start.en.html.
- Sean Russell. REXML - an XML toolkit for Ruby, in Ruby, 2006. http://www.germane-software.com/software/XML/rexml/doc/.
- Stan Seibert. Libao Documentation, 2003. http://www.xiph.org/ao/doc/.
- The DAISY Consortium. Specifications for the Digital Talking Book, ANSI/NISO Z39.86-2005, 2005. http://www.daisy.org/z3986/2005/z3986-2005.html.
- The DAISY Consortium. DAISY 2.02 Specification, 2001. http://www.daisy.org/publications/specifications/daisy_202.html.

- The Open Group. Pthread API, The Single UNIX Specification, Version 2, 1997. http://www.opengroup.org/pubs/online/7908799/xsh/pthread.h.html.
- The Ruby community. Ruby Standard Library Documentation, 2006. http://www.ruby-doc.org/stdlib/.
- Trolltech. Qt Reference Documentation, 2005. http://doc.trolltech.com/3.3/.

Index

AAC, 38
Audio, 20
  Decoder, 22
  Engine, 20
  Interface, 21
  Output module, 22

Daisy, 2
Data structure, 26
  Audio, 27
  Parser, 27
Debhelper, 46
Debian packages, 46
Development model, 6
Doxygen, 41, 65
DTB, 2

Eclipse, 43, 66

Front-end, 20, 36

GNU Gettext, 30

Internationalization, 29
  Console front-end, 30
  GUI front-end, 29

KDE, 36

LaTeX, 42, 65
Libdaisy interface, 28
Libraries, 34
  AAC decoder, 38
  Audio design, 37
  Audio output, 37
  GStreamer, 34
  GUI, 36, 62
  Libao, 37, 64
  Libxml2, 35, 64
  MAD, 37, 63
  MP3 decoder, 37
  REXML, 35
  WAV, 38
LyX, 42, 65

Make, 45
Makefile, 47
MP3, 37
MPEG, 37, 63

Parser, 22
  NCC, 24
  NCX, 24
  SMIL, 25
PCM, 37, 38
Planner, 43
POSIX, 20, 39, 63
Product quality, 50
Product quality tools
  Efence, 43, 51
  GNU Binary Utilities, 44, 51
  GNU Debugger, 44, 51

Lintian, 44, 51
Splint, 43, 51
Valgrind, 44, 51
Programming languages
C, 33
C++, 34
Ruby, 33

QMake, 45
Qt 3, 36
Qt Designer, 29
Qt Linguist, 30

Scripts, 47, 66
Skolelinux, 1
Subversion, 42
SVN, 66

Threads, 20, 39

User testing, 49

WAV, 38
Web page, 54

Appendix A

Pre-project report for the DaisyPlayer Project (without appendixes)

A.1 Goals and constraints

A.1.1 Background

Skolelinux is a Custom Debian Distribution that is customized for schools with focus on being easy to install and maintain. Skolelinux tries to give schools and students a good and free alternative to proprietary software. This is used in many schools already and the goal is to cover as many schools as possible.

In order to achieve this goal Skolelinux need software to cover the different needs the student have in their education. One such program, after a request of a Norwegian teacher, is an application able to play Daisy Digital Talking Books (DTB). Daisy DTB is a multimedia representation of a print publication. Linux lacks a good and free Daisy player at this point.

A.1.2 Effect goal

The effect goal is to start the development on a free Daisy player which, in an easy way, helps students play Daisy-books using Skolelinux or other Linux-distributions.

A.1.3 Result goal

Result goals are to start an Open Source project and to create an engine for playing Daisy DTB's with an API and a simple user interface. It must be able to play at least one audio format, and it should be easy for others to program against the API.

A.1.4 Target group

The main target group for this project is the students and other users which need a free Daisy player for Linux. But we must also have in mind that the project are going to be evaluated by the school.

A.1.5 Constraints

This project must be Open Source and licensed according to The Debian Free Software Guidelines (DFSG). [1] Software created must run on "Skolelinux" and it is a goal to keep the software dependencies to a minimum.

A.2 Extent of task

A.2.1 Task description

The task is to create a software Daisy player for Linux. This software must follow the Daisy\NISO standard [2]. The program should be able to play the audio and display synchronized text. Navigating and searching through the books will also be supported.

A simple UI to the daisy engine will be made. A more advanced GUI have been requested, but has a low priority. The creation of a GUI depends upon the DaisyPlayer engine and API and are limited by the projects time resources.

Main functionality:

- Make an engine able for audio playback.
- Display the synchronized text.

[1] DFSG http://www.debian.org/social_contract.html
[2] Daisy/NISO Standard http://www.daisy.org/z3986/

- Make a simple user interface which makes it possible for user interaction.
- Make navigation- and searching functionality.
- The project should be easily extendible.

It is very important that we try to separate the engine from the user interface. This is done because it should be a simple job to change the user interface at a later time if wanted.

The principal may come with suggestions and more functionality on the way, but at this point the above points are our goals.

A.2.2 The platform

The Skolelinux system is based on Debian GNU/Linux and the KDE Desktop and is the target platform for this application. We will try to make it portable to other platforms, but this is not a priority.

Audio output to aRts is a minimum requirement, but a goal is to support the other common methods such as ALSA, OSS, ESD too.

A.2.3 Libraries, frameworks and standards

This far we have identified the following libraries, frameworks and standards which we may need in the development of this application.

Audio

- Audio formats (MP3, MP4-AAC, WAV)
- Audio back-ends (ARTS, ESD, OSS, ALSA)
- MAD (MPEG Audio Decoder)
- Initial research suggests that the libao library is a good choice for communicating with audio back-ends. Libao claims to support the back-ends mentioned above.

Graphics

- JPEG
- PNG
- SVG

GUI

- GTK+
- QT

XML

- SMIL
- XHTML

IPC

- DBUS
- Sockets
- Pipes

Other

- Unicode
- Festival

A.2.4 Programming languages

A.2.4.1 Programming languages

We have evaluated a few programming languages to come up with the best choice in relation to the project. This project will use a lot of existing technologies, so the availability of libraries are important (we do not want to re-invent audio- or XML libraries). The project will also be split up into multiple strongly separated modules. As long as the modules supports the same IPC, there are possible to use different programming languages for the individual modules.

A.2.4.2 Python

Python is an object-oriented language with good support for technologies we are likely to use. Support exists for XML, Unicode, wrappers to audio output-libraries, widget toolkits etc.

A.2.4.3 Ruby

Ruby lacks proper support for Unicode, but has partial support for UTF-8. We must look into this in the event that we use Ruby for components that needs to deal with Unicode.

A.2.4.4 C++

C++ is a powerful language, but has some issues as we see it. The size of variables are not fixed, but depends upon the HW-platform and compiler. There have also been expressed worries that we might not get the compatibility and backwards-compatibility we need because of variation between compilers, HW-platforms and libraries.

A.2.4.5 C

C is a powerful language, but it does not fit for this particular project as we see it. It is not object-oriented and, even though the technologies we are going to use are supported through external libraries, we think there are better and easier alternatives which will allow us to implement our program faster.

There is a slight possibility we will have to review or modify existing code in libraries etc. These are likely to be written in C, so we must be ready to face the C programming language at some point through our project. But, as a primary programming language, C is not what this project requires.

A.2.4.6 Availability of compilers and interpreters

Compilers for C/C++, Java and more are shipped with Skolelinux.

An interpreter for Python are shipped with Skolelinux as default, while the Ruby interpreter needs to be installed as an extra. [3]

[3] DebianEdu/FAQ/General/Software http://www.skolelinux.org/portal/faq/general/software/

A.2.5 Constraints

A goal is to keep the software dependencies to a minimum, so it can be easily installed on a typical workstation. Because of time limitations we focus on the player engine and at least one UI. A future extension is to implement a GUI, but this is not a priority until the engine is complete.

A.3 Project organization

A.3.1 Principal

Skolelinux. Contact: Herman Robak.

A.3.2 Group

The project group consists of:
André Lindhjem, 03HBINDP HiG
Kjetil Holien, 03HBINDP HiG
Terje Risa, 03HBINDD HiG
Øyvind Nerbråten, 03HBINDP HiG

A.3.3 Roles and responsibilities

The four members on this group will have different roles and responsibilities. Most of this will be distributed as the need occur during the project, but some roles have already been assigned.

- Our project leader will be Øyvind Nerbråten, he will be responsible to make sure that all of us works in the terms we have agreed.
- Each one of the members are responsible to keep a log about the work they do and we will try to work about 25 to 30 hours a week. When we will work will mostly depend on the others lectures we got to attend to, but try to work at the same time of the day since we then can discuss problems and solutions that may occur.

Skolelinux package list http://developer.skolelinux.no/cgi-bin/viewcvs.cgi/~checkout~/skolelinux/src/task-skolelinux/pkgdeblist.txt

- If a member do not work sufficient the group leader will talk to him and give him a warning, if that does not help, the teaching supervisor will be contacted.

A.4 Planning and reporting

A.4.1 Choice of methodology

The group have decided to use an mixture of evolutionary system development and Open Source development. Since the project is an Open Source project we felt it natural to follow the Open Source development standard, with a few minor changes. We will try to release a new version every two weeks, giving the project some time boundaries and certain goals to work against. All our work will be on the Internet in a revision management system, giving users the option of downloading the work as it is develops, meaning that they do not have to wait for the new version to be released to be updated. We will also try to implement user suggestions and establish an user group that hopefully will find this project interesting, in true Open Source style.

The only other development choices we truly consider where incremental development or a more pure evolutionary development. Since we did not feel that either one of the development methods mention before suited our needs, we decided to use an adaptation of the evolutionary model.

A.4.2 Plan for status meeting and decision dates

The formal status meetings will be held on these dates: 20/02-06, 20/03-06 and 24/04-06. But we will have more informal meeting with the supervisor more often. The status meetings will be a natural time to decide large decision which regards the projects outcome.

A.4.3 Code convention

We will try to keep the source code as readable and structured as possible. This is done to make it easier for both us, and other readers, to understand and maintain the code. We will agree on code conventions for the languages we use. Since we are planning to take advantage of

existing code, we must be flexible on programming languages and code conventions.

A.5 Organization of quality assurance

A.5.1 Revision management

In this project we will be using Subversion to handle our version control of all our work. This includes both the source code and the project documentation. Because all Skolelinux projects are stored in a shared repository, the choice of manual version handling was eliminated, not that manual version handling was an eligible alternative either way. Since Skolelinux just started using Subversion instead of CVS, the choice was already made for us.

A.5.2 Quality handling

We must be dynamic and prepared to receive and act on user input to ensure quality. As most Open Source projects we are dependant upon user feedback to do testing maintain code quality.

The creation of an API for developers and a good users manual are a priority. Presentable versions of important documents created throughout the project are automatically generated within one hour and linked to from the project website.

A.5.3 Risk analysis

We have identified the following risks:

User feedback: Lack of feedback from the user community will make it difficult to develop software for people with special needs. To avoid this we must reach out to the Open Source community and advertise for the project in the right forums and mailing lists.

Future maintenance: It is important for this project survival to get users and other developers involved. It is imperative for the project to be active to maintain the user-mass. To do this, we must advertise the project.

Sickness: This is a common risk with software development projects. In our case this is not a very big risk, since we are four members with almost equally knowledge. To prevent any serious problems that sickness may cause, we will keep each other updated with how things work and what we do.

Principal or target group: The risk of the principal or target group looses interest are considered to be minimal. There is a genuine need for a DaisyPlayer for the Linux platform, and (as we are an Open Source project) we are not dependant upon the principal to the same degree as "traditional" projects are.

Violations of patents and trademarks: This could be a serious risk if we are not careful and thorough enough in our search for code we can use.
In appendix A we have ranked the risks.

A.6 Development plan

As already mentioned under the choice of methodology we are working in a mixture of evolutionary system development and Open Source development. That means that we are not able to plan the whole project very specific, like we would have done with e.g. a waterfall model. Because of the fact that we must be prepared to accept and act on code-contributions, bug-reports and fixes, we must be flexible during the development. Since we are planning to release a version every other week, it means that we work in cycles. In each cycle, we add more functionality. Each cycle consist of planning, coding and testing. A coarse grain sketch of our Gantt chart can be found in appendix B.

Appendix B

Use cases

B.1 daisy_init

Goal Initiate the Daisy library with its data structure and callback functions.

Precondition The callback functions must be declared.

Postcondition The library has been initialized and the data structures has been allocated.

Description In order to use the libdaisy library the *daisy_init* functions must be called to initialize the data structure and callback functions. The user must pass along pointers to the functions he wishes to use to get callback messages from the engine. The user can also pass along a pointer to an object or a data structure which will be available in all the callback functions.

Main success scenario

1. The user request to initialize the library.
2. The system allocates the data structure and stores the pointers to the users callback functions.
3. The system return a pointer to the data structure, this must be passed along to all other functions in the library.

B.2 daisy_term

Goal Terminate the Daisy library and free all memory held by it.

Precondition The data structure must have been initialized by *daisy_init*.

Postcondition The supplied data structure is destroyed and all memory allocated by it is released.

Description The user must pass along a data structure initialized with *daisy_init* in order to free memory occupied by it.

Main success scenario

1. The user calls the *daisy_term* function and supplies a initialized data structure.
2. The system destroys the supplied data structure and deallocates the memory occupied by it.

B.3 daisy_load

Goal Parse a Daisy DTB and store the data in the data structure.

Precondition The data structure must have been initialized by *daisy_init*.

Postcondition The playback data from the Daisy DTB has been stored in the data structure.

Description The user must pass along a data structure initialized with *daisy_init*, and a valid path to a Daisy DTB in order to load the DTB.

Unresolved issues

- Support more of the possibilities in the Daisy standards.
- What if the book does not follow the standard?

Technical implication

- Library support for XML parsing.

Main success scenario

1. The user calls the *daisy_load* function and supplies an initialized data structure and a full path to a Daisy DTB.
2. The data structure now contains the playback data of the loaded DTB.

B.4 daisy_play

Goal Start playback of the loaded Daisy DTB.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition The engine starts playing the loaded Daisy DTB.

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. If a book is loaded, the data structure will contain the playback data.

Technical implication

- Need libraries for audio playback with various audio formats.

Main success scenario

1. The user calls the *daisy_play* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The engine enters playing state.

B.5 daisy_seek

Goal Changing the current playback position in the data structure.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition Continues playback from the new position in the data structure.

Description Perform various user specified seek operations. The valid options are next/previous chapter and passage, and seek to the beginning. The seek option must be passed along with the function call.

Main success scenario

1. The user calls the *daisy_seek* function and supplies an initialized data structure with a loaded Daisy DTB, and a seek option.
2. The system changes its playback position pointers in the data structure and continues playback from there.

Variations

1a The user tries to seek to the previous chapter/passage when we are at the first chapter/passage.

1. The system changes its playback position to the first passage and continues playback from there.

1b The user tries to seek the next chapter/passage when we are at the last chapter/passage.

1. The system returns “end of book” and the audio playback stops.

B.6 daisy_get_position

Goal Retrieve the current playback position in the data structure.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. To retrieve the playback position, the data structure must contain a loaded Daisy DTB. This function returns the current position in the data structure as a struct of two integers. The first integer tells which chapter we are at and the second which passage.

Main success scenario

1. The user calls the *daisy_get_position* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The user gets a struct with the current playback position.

B.7 daisy_goto_position

Goal Change the current playback position in the data structure.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition Continues the playback from the new position in the data structure.

Description To use this function the user must pass along a data structure initialized with *daisy_init*. In order change playback position, the data structure must contain a loaded Daisy DTB. The user must also pass along a struct containing the new playback position.

Main success scenario

1. The user calls the *daisy_goto_position* function and supplies an initialized data structure with a loaded Daisy DTB and a struct with the new position.
2. The system changes playback position in the data structure and continues from there.

B.8 daisy_stop

Goal Stop playback of the loaded Daisy DTB.

Precondition The data structure must have been initialized by *daisy_init*, the DTB must have been loaded by *daisy_load* and playback started by *daisy_play*.

Postcondition The engine stops playing the loaded Daisy DTB.

Description In order to use this function the user must pass along a data structure initialized with *daisy_init* and the engine must be in playback mode. The engine will be put in stopped state at the current playback position.

Main success scenario

1. The user calls the *daisy_stop* function and supplies an initialized data structure with a loaded Daisy DTB.

2. The engine enters stopped state at current playback position.

B.9 daisy_pause

Goal Pause or resume the playback of the loaded Daisy DTB.

Precondition The data structure must have been initialized by *daisy_init*, the DTB must have been loaded by *daisy_load* and the engine must be in playing or paused state.

Postcondition The engine pauses the current playback if the state was playing, continue playback it the state was paused.

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. If the engine is in playing state, the engine will be put in a paused state, if the engine already is in a paused state, the playback will continue from the position it was paused.

Main success scenario

1. The user calls the *daisy_pause* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The engine enters paused state at the current playback position or continues playing from the last position.

Variations

1a The user calls the *daisy_pause* function while the current playback state is playing.

1. The engine enters a paused state at the current playback position.

1b The user calls the *daisy_pause* function while the current playback state is paused.

1. The engine continues playback from the where it was paused.

B.10 daisy_get_info

Goal Get meta info about the loaded Daisy DTB.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. If a book is loaded, the data structure will contain the meta data about the book, if available in the loaded Daisy DTB. The user must also pass along what kind of information he wants to retrieve.

Main success scenario

1. The user calls the *daisy_get_info* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The user gets the information from the data structure.

B.11 daisy_get_chapter_count

Goal Return the number of chapters in the loaded Daisy DTB.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. This function returns the number of chapters in the loaded Daisy DTB.

Main success scenario

1. The user calls the *daisy_get_chapter_count* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The user gets the number of chapters in the loaded Daisy DTB.

B.12 daisy_get_chapter_info

Goal Return information about the chapter to the user.

Precondition The data structure must have been initialized by *daisy_init*, and the DTB must have been loaded by *daisy_load*.

Postcondition

Description In order to use this function the user must pass along a data structure initialized with *daisy_init*. The user must also pass along which chapter and what kind of information he wants.

Main success scenario

1. The user calls the *daisy_get_chapter_info* function and supplies an initialized data structure with a loaded Daisy DTB.
2. The user gets the information about the chapter from the data structure.

Appendix C

Code conventions

All C code follows the ANSI C standard.

```
/**
 * Comment..
 * @param x ...
 * @param y ...
 * @returns ..
 */
int function (x, y)
{
   ...
}
```

All functions must be commented with its parameters and return values (if any). The source will be indented one indent for each level.

```
if (value == NULL)
{
   ...
}
else
{
   ...
}

while (i < 5) foo (i++);
```

With one line statement we do allow to not use curly brackets around it.

```
switch (currentToken)
{
   case 1:
   {
      ...
   break;
   }
   ...
}
```

```
struct Foo
{
   ...
};


struct Foo *foo = (struct Foo *) malloc (sizeof (struct Foo));
```

- We try to avoid using global variables.

```
/* control.c */
#include <stdio.h>
#include <stdlib.h>
...

#include "control.h"
...
```

Other notes:

- Loop and short lived test variables have names consisting of one character such as i, j or n.
- Other long lived variables will have more explaining names such as authorNode.
- Constants are named with capital letters with underscores between each word: DEBUG_NCCPARSER

Appendix D

Gantt chart

Below is our Gantt-chart from the pre-project which shows how we planned the time.

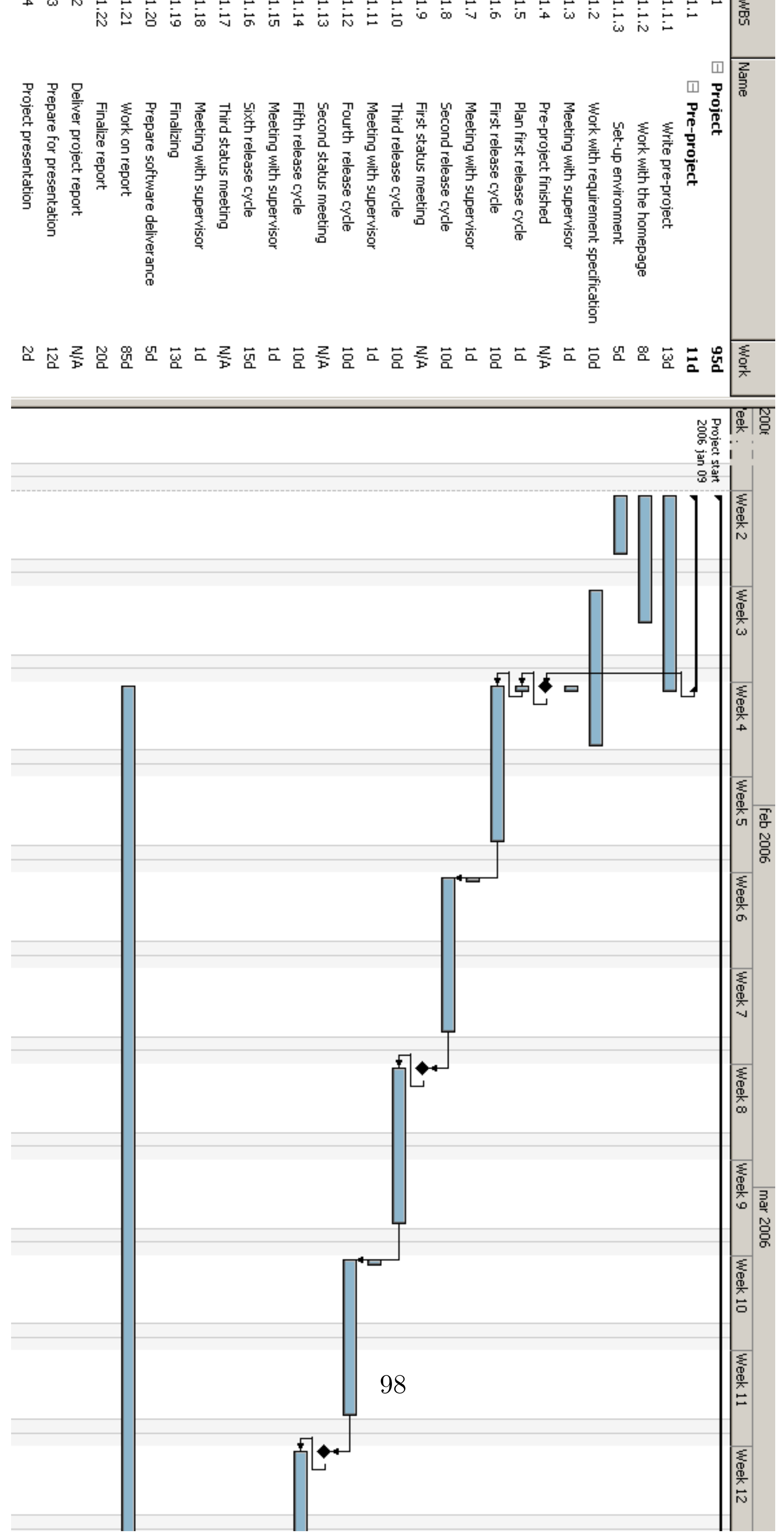

Figure D.1: Gantt chart - part 1.

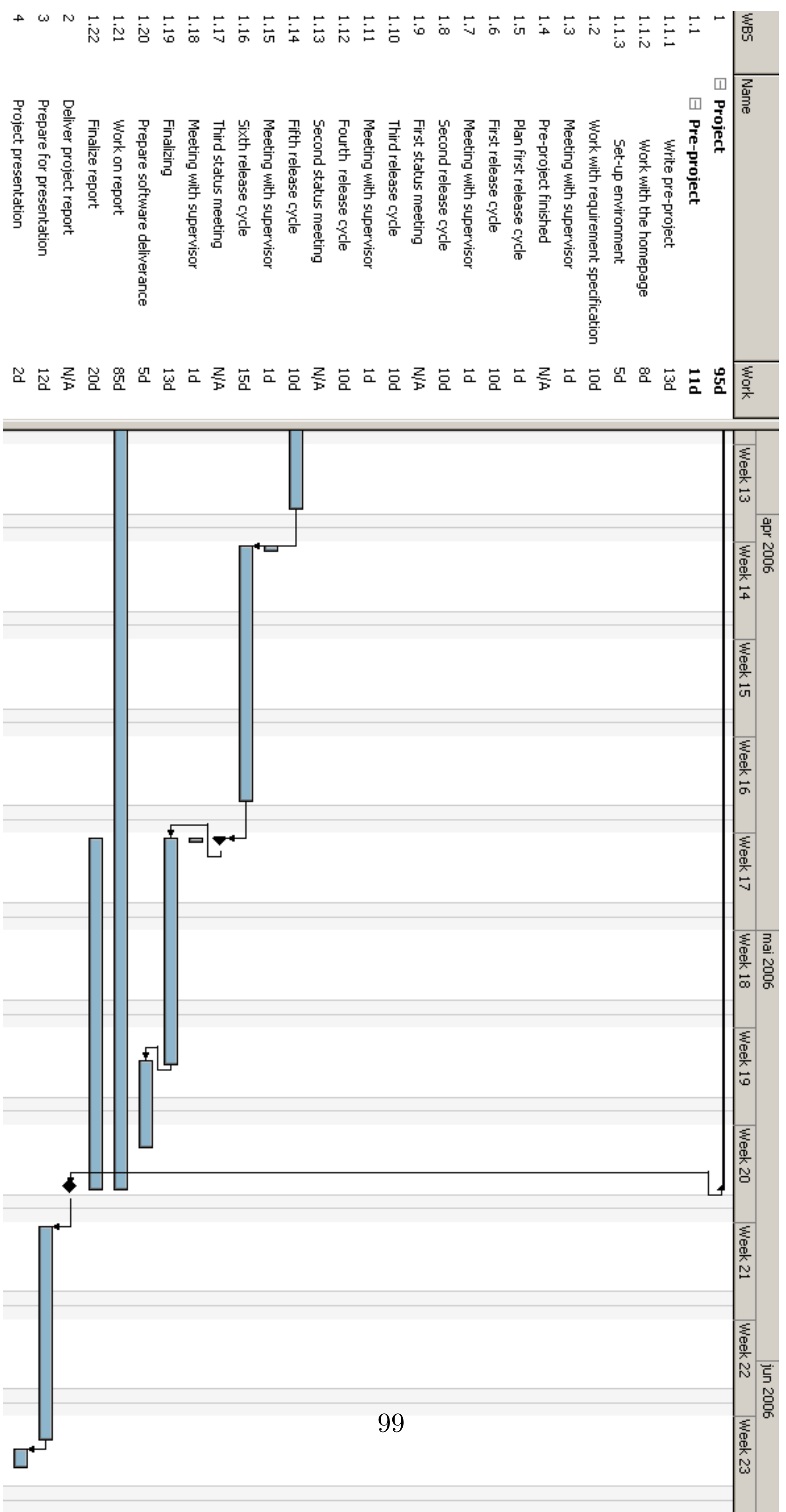

Figure D.2: Gantt chart - part 2

Appendix E

Status reports

We decided to have three status meetings every fourth week. After those meetings we wrote a report where we discussed the current progress and problems that have occurred.

Status report 20.02.06

Progress report

We have just started our third release cycle and we have implemented a simple audio engine able to play MP3 and XML parsing (SAX parsing) of the 2.02 standard. In our first release cycle we used GStreamer and REXML with Ruby, but were not happy with the result. In our second release cycle we reimplemented the functionality we made in the first release cycle, using C instead of Ruby as programming language. We are using mad, libao and libxml2 as our libraries, which seems to work nicely.

Our current work is to improve and expand our code, and add support for the new Daisy standard (ANSI/NISO Z39.86-2005).

Group cooperation

We have good communication and cooperation within the group, and we are pretty confident that the good collaboration will continue.

Report writing

We have not had so much focus on report writing up to now. Our focus in the early phase has been to explore good solutions and implement them. We feel that this method has given us the flexibility needed to quickly evaluate, implement, and possibly discard alternatives. In the time ahead, documenting our work will have a higher priority as we now to a larger degree have decided on specific solutions and need to create documentation for those solutions.

Problems

We have experienced a few problems this far.

- We had difficulty getting gstreamer to provide sufficient audio control. Also, in our experience, GStreamer is not well enough documented. The ruby gstreamer wrapper we worked against lacked both in documentation and basically did not give us enough control over the output to support our needs.
- The speed in our first XML parser caused us concern. While it was not critical, the Ruby implemented XML parsing was really slow.

To solve our problems, we decided to discard GStreamer and REXML for the Daisy engine and use well known C libraries like libmad, libao and libxml2. As we now would implement both audio and XML parsing functionality using C libraries, we decided not to use Ruby and implement everything in C. When we switched over from Ruby to C, the speed problem with the XML parsing solved itself.

Motivation

With this second release we feel a lot more confident in the project. It is a relief to discard REXML and GStreamer and our current implementation holds a lot more promise when it comes to maintainability, speed and control. In the future we hopefully will not experience the same problems with our choices as in our first release cycle. Now that these frustrations are behind us, our motivation is restored.

______________________	______________________
Øyvind Nerbråten	André Lindhjem
______________________	______________________
Kjetil Holien	Terje Risa

Status report 21.03.06

Progress report

We have just started our fifth release cycle and it is time to finalize the engine API. We have started to make a front-end for the engine. It is still a lot to be done with the engine, but we have to focus on the most important functionality because we are running out of time. Since last status report we have made the engine run as threads and implemented playback functionality. We have also partially implemented the new standard.

Group cooperation

We have good communication and cooperation within the group, and we are pretty confident that the good collaboration will continue.

Report writing

We have started to focus more on the report writing, but still got much work to do. It is hard to balance the report writing and coding when we got so much to do.

Problems

Since last report we have encountered some new problems:

- We have had some setback due to sickness.
- We have used a lot of time on a potential problem with parsing according to the standard. We are still a bit uncertain how to solve this problem, but we will try to make solution that works.
- We have had some problems with threads, deadlocks and race conditions.
- Implementing other audio formats seems to be to time consuming and have not seen any books using other formats than MP3. So implementing these will have a low priority.

Motivation

We can finally see some results and are able to play a Daisy DTB with some playback functionality. We need to focus defining the API and wrapping up the engine rather than implementing new features. We do not have that much time left, so we have to focus on the important matters.

Øyvind Nerbråten	André Lindhjem
Kjetil Holien	Terje Risa

Status report 24.04.06

Progress report

We have just finished our last release cycle and it is time to finalize the report. We have created two front-ends for our engine, one console and one GUI. They work as examples on how to use the engine. We have removed some bugs and the global variables in the library.

Group cooperation

We have good communication and cooperation within the group, and we are pretty confident that the good collaboration will continue.

Report writing

We have started to focus more on the report writing, but still got much work to do. It is hard to balance the report writing and coding when we got so much to do.

Problems

Since last report we have encountered some problems:

- Because of limited time resources we could not do all we wanted to do.

Motivation

Continue writing a good project report that reflects the work we have done.

____________________	____________________
Øyvind Nerbråten	André Lindhjem
Kjetil Holien	Terje Risa

Appendix F

Work log

All work hours we have put into this project has been documented in a work log. This log has become too big to enclose in this document. You can look at our log in the progress section of the project web page[1]. We have also created a plan for each iteration, which you can find on the same section of the web page.

[1] The DaisyPlayer Project - progress http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/www-files/p-progress/

libdaisy Reference Manual

0.2.2

Generated by Doxygen 1.4.2

Thu May 18 12:46:33 2006

Contents

1 Libdaisy **1**

2 libdaisy Data Structure Index **2**

3 libdaisy File Index **2**

4 libdaisy Data Structure Documentation **3**

5 libdaisy File Documentation **3**

1 Libdaisy

1.1 Introduction

Libdaisy is a toolkit for parsing and playing Daisy Digital Talking Books (DTB). Libdaisy is developed for the Linux operation systems under the `GNU General Public License`.

Libdaisy does not offer the complete functionality according to the Daisy standards and is not a finished library. Even though the library is not complete, it offers the most important functionality for playing a Daisy DTB. Libdaisy partially supports two of the Daisy standards, `DAISY 2.02` and the new standard `ANSI/NISO Z39.86`.

Please visit the `DaisyPlayer` project web page for more information.

1.2 Requirements

- libxml2 >=2.6.16-7
- libao2 >=0.8.6-1
- libmad0 >=0.15.1b-1.1

1.3 Install

From package:

Install the Debian package, using 'dpkg -i libdaisy_0.2.0_i386.deb'.

From source:

'tar -zxvf libdaisy_0.2.0.tar.gz'

'cd libdaisy_0.2.0'

'make'

'make install' or 'make install DESTDIR=/tmp/foo' #'make install' needs root privilege

'make install-dev" #install the development header, needs root privilege

1.4 License

Copyright (C) 2006 by AndrÃ169 Lindhjem <belgarat@sdf.lonestar.org>, Kjetil Holien <kjetil.holien@gmail.com>, Terje Risa <terje.risa@gmail.com> & Ã152yvind NerbrÃ165ten <oyvind@nerbraten.com>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

1.5 Bug Reports

Any kind of bug reports are welcome. If you find bugs, please email us (See the Author section for contact information).

Author:

AndrÃ169 Lindhjem <belgarat@sdf.lonestar.org>
Kjetil Holien <kjetil.holien@gmail.com>
Terje Risa <terje.risa@gmail.com>
Ã152yvind NerbrÃ165ten <oyvind@nerbraten.com>

Date:

26.04.2006

Version:

0.2.2

2 libdaisy Data Structure Index

2.1 libdaisy Data Structures

Here are the data structures with brief descriptions:

daisy_position (Struct for holding a Daisy DTB playback position) 3

3 libdaisy File Index

3.1 libdaisy File List

Here is a list of all files with brief descriptions:

libdaisy.h (Header file for the libdaisy development package) 3

libdaisy_mainpage.doxygen 10

4 libdaisy Data Structure Documentation

4.1 daisy_position Struct Reference

4.1.1 Detailed Description

Struct for holding a Daisy DTB playback position.

Meant for storing chapter (smilops) and passage (nodepos) jump positions. Works as a bookmark. The integer values is equal to the chapter number and passage numer in the playback sequence (smilpos = 1 & nodepos = 1 is the position for passage 1 in chapter 1).

See also:
daisy_get_position
daisy_goto_position

Data Fields

- int smilpos
- int nodepos

4.1.2 Field Documentation

4.1.2.1 int daisy_position::nodepos

4.1.2.2 int daisy_position::smilpos

The documentation for this struct was generated from the following file:

- libdaisy.h

5 libdaisy File Documentation

5.1 libdaisy.h File Reference

5.1.1 Detailed Description

Header file for the libdaisy development package.

Libdaisy is a toolkit for parsing and playing Daisy Digital Talking Books (DTB).

Date:
21.03.2006

Typedefs

- typedef void * daisyplayer_t
  This will need to be initialized and passed along to the daisy functions.

Enumerations

- enum daisy_status {

  DAISY_ERROR_UNKNOWN = 0x0000, DAISY_ERROR_AUDIO_NOT_INITIALIZED = 0x0200, DAISY_ERROR_AUDIO_CREATE_MMAP = 0x0201, DAISY_ERROR_AUDIO_FSTAT = 0x0202,

  DAISY_ERROR_AUDIO_OPEN = 0x0203, DAISY_ERROR_AUDIO_FREE_MMAP = 0x0204, DAISY_ERROR_AUDIO_INITIATE_DATA = 0x0205, DAISY_ERROR_AUDIO_NOT_PLAYING = 0x0206,

  DAISY_ERROR_AUDIO_NOT_STOPPED = 0x0207, DAISY_ERROR_AUDIO_DATA_IS_NULL = 0x0208, DAISY_ERROR_AUDIO_MALLOC = 0x0209, DAISY_ERROR_AUDIO_PAUSED_WHILE_NOT_PLAYING = 0x0210,

  DAISY_ERROR_AUDIO_STOPPED_WHILE_NOT_PLAYING = 0x0211, DAISY_ERROR_PLAYBACK_NO_TEXT_IN_SEGMENT = 0x0301, DAISY_ERROR_PLAYBACK_NO_AUDIO_IN_SEGMENT = 0x0302, DAISY_ERROR_PLAYBACK_NO_DTB_LOADED = 0x0303,

  DAISY_ERROR_PLAYBACK_SEEK_FAILED = 0x0304, DAISY_ERROR_MISC_INIT_MUTEX = 0x0400, DAISY_END_OF_BOOK = 0x1100 }

  The different sort of status messages libdaisy might return.

- enum daisy_seek_option {

  DAISY_SEEK_PREV_CHAPTER = 1, DAISY_SEEK_PREV_PASSAGE = 2, DAISY_SEEK_NEXT_CHAPTER = 3, DAISY_SEEK_NEXT_PASSAGE = 4,

  DAISY_SEEK_TO_BEGINNING = 5 }

  The different seek operations supported by the engine.

- enum daisy_bookinfo_option { DAISY_BOOKINFO_TITLETEXT = 1, DAISY_BOOKINFO_TITLEIMAGE = 2, DAISY_BOOKINFO_TOTALTIME = 3 }

  The different types of book information.

- enum daisy_chapter_info { DAISY_CHAPTER_TITLE = 1, DAISY_CHAPTER_WEIGHT = 2 }

  The different types of chapter information.

Functions

- daisyplayer_t daisy_init (void ∗data, void(∗l_cb_daisy_audio_done)(void ∗), void(∗l_cb_daisy_audio_next)(void ∗, unsigned long int), void(∗l_cb_daisy_text)(void ∗, void ∗), void(∗l_cb_daisy_id)(void ∗, void ∗), void(∗l_cb_daisy_error)(void ∗, enum daisy_status, const char ∗daisy_status_msg), void(∗l_cb_daisy_progress)(void ∗, long int))

  Initializes the daisy library.

- void daisy_term (daisyplayer_t daisy)

  Terminates the daisy library and frees memory used by it.

- int daisy_load (daisyplayer_t daisy, char ∗path)

  Loads a new daisy book.

- int daisy_play (daisyplayer_t daisy)

Starts playback if a book is loaded.

- int daisy_seek (daisyplayer_t daisy, int seek_option)

  Seek operations which can be performed when a book is loaded.

- daisy_position ∗ daisy_get_position (daisyplayer_t daisy)

  Retrieves the current playback position.

- int daisy_goto_position (daisyplayer_t daisy, daisy_position ∗position)

  Seeks to a playback position (bookmark) and continue playback from there.

- int daisy_stop (daisyplayer_t daisy)

  Stops playback.

- int daisy_pause (daisyplayer_t daisy)

  Toggle pause.

- char ∗ daisy_get_info (daisyplayer_t daisy, int value)

  Retrieves book meta information.

- int daisy_get_chapter_count (daisyplayer_t daisy)

  Retrieves the number of chapters in in the loaded Daisy DTB.

- char ∗ daisy_get_chapter_info (daisyplayer_t daisy, int num, int option)

  Retrieves information about a given chapter.

5.1.2 Typedef Documentation

5.1.2.1 typedef void∗ daisyplayer_t

This will need to be initialized and passed along to the daisy functions.

5.1.3 Enumeration Type Documentation

5.1.3.1 enum daisy_bookinfo_option

The different types of book information.

See also:

daisy_get_info

Enumeration values:

DAISY_BOOKINFO_TITLETEXT

DAISY_BOOKINFO_TITLEIMAGE

DAISY_BOOKINFO_TOTALTIME

5.1.3.2 enum daisy_chapter_info

The different types of chapter information.

See also:
daisy_get_chapter_info

Enumeration values:

DAISY_CHAPTER_TITLE

DAISY_CHAPTER_WEIGHT

5.1.3.3 enum daisy_seek_option

The different seek operations supported by the engine.

See also:
daisy_seek

Enumeration values:

DAISY_SEEK_PREV_CHAPTER

DAISY_SEEK_PREV_PASSAGE

DAISY_SEEK_NEXT_CHAPTER

DAISY_SEEK_NEXT_PASSAGE

DAISY_SEEK_TO_BEGINNING

5.1.3.4 enum daisy_status

The different sort of status messages libdaisy might return.

Enumeration values:

DAISY_ERROR_UNKNOWN

DAISY_ERROR_AUDIO_NOT_INITIALIZED

DAISY_ERROR_AUDIO_CREATE_MMAP

DAISY_ERROR_AUDIO_FSTAT

DAISY_ERROR_AUDIO_OPEN

DAISY_ERROR_AUDIO_FREE_MMAP

DAISY_ERROR_AUDIO_INITIATE_DATA

DAISY_ERROR_AUDIO_NOT_PLAYING

DAISY_ERROR_AUDIO_NOT_STOPPED

DAISY_ERROR_AUDIO_DATA_IS_NULL

DAISY_ERROR_AUDIO_MALLOC

DAISY_ERROR_AUDIO_PAUSED_WHILE_NOT_PLAYING

DAISY_ERROR_AUDIO_STOPPED_WHILE_NOT_PLAYING

DAISY_ERROR_PLAYBACK_NO_TEXT_IN_SEGMENT

DAISY_ERROR_PLAYBACK_NO_AUDIO_IN_SEGMENT

DAISY_ERROR_PLAYBACK_NO_DTB_LOADED

DAISY_ERROR_PLAYBACK_SEEK_FAILED

DAISY_ERROR_MISC_INIT_MUTEX

DAISY_END_OF_BOOK

5.1.4 Function Documentation

5.1.4.1 int daisy_get_chapter_count (daisyplayer_t *daisy*)

Retrieves the number of chapters in in the loaded Daisy DTB.

Parameters:

daisy - the daisy struct which must be passed along with all the API functions.

Returns:

the number of chapters, or -1 in case of error.

5.1.4.2 char* daisy_get_chapter_info (daisyplayer_t *daisy*, int *num*, int *option*)

Retrieves information about a given chapter.

Parameters:

daisy - the daisy struct which must be passed along with all the API functions.

num - the chapter number to retrive information from (use daisy_get_chapter_count to get the number of chapters available).

option - a daisy_chapter_info which states what information to retrieve.

Returns:

a pointer to a string containing the information, or NULL in case of error.

See also:

daisy_chapter_info
daisy_get_chapter_count

5.1.4.3 char* daisy_get_info (daisyplayer_t *daisy*, int *value*)

Retrieves book meta information.

Parameters:

daisy - the daisy struct which must be passed along with all the API functions.

value - a daisy_bookinfo_option value which states what information to retrieve.

Returns:

the string if found, otherwise NULL. The string must be deallocated by the caller.

See also:

daisy_bookinfo_option

5.1.4.4 daisy_position* daisy_get_position (daisyplayer_t *daisy*)

Retrieves the current playback position.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

Returns:

a pointer to a daisy_position struct containing the chapter and passage positions, or NULL in case of error. The struct must be deallocated by the caller.

See also:

daisy_position
daisy_goto_position

5.1.4.5 int daisy_goto_position (daisyplayer_t *daisy*, daisy_position ∗ *position*)

Seeks to a playback position (bookmark) and continue playback from there.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

position - a pointer to a daisy_position struct with the chapter and passage position.

Returns:

1 in case of success and -1 in case of error.

See also:

daisy_position
daisy_get_position

5.1.4.6 daisyplayer_t daisy_init (void ∗ *data*, void(∗)(void ∗) *l_cb_daisy_audio_done*, void(∗)(void ∗, unsigned long int) *l_cb_daisy_audio_next*, void(∗)(void ∗, void ∗) *l_cb_daisy_text*, void(∗)(void ∗, void ∗) *l_cb_daisy_id*, void(∗)(void ∗, enum daisy_status, const char ∗daisy_status_msg) *l_cb_daisy_error*, void(∗)(void ∗, long int) *l_cb_daisy_progress*)

Initializes the daisy library.

It should be called before any attempt to use the daisy functionality.

Parameters:

data - a void pointer to any object or datastructure you may need in the callback functions. This data pointer will be available in all callback functions. You can e.g. pass along a GUI object in c++ so that you can output the text from the callbackfunctions in the GUI. Set this parameter to NULL if you don't need it.

l_cb_daisy_audio_done - a pointer to the function which will be called when an audio segment is done playing.

l_cb_daisy_audio_next - a pointer to the function which will be called when an new audio segment starts playing, supplying the duration of the segment in ms.

l_cb_daisy_text - a pointer to the function which will be called when an new audio segment starts playing, supplying the text corresponding to the audio.

l_cb_daisy_id - a pointer to the function which will be called when an new audio segment starts playing, supplying the id of the text passage in the xml file.

l_cb_daisy_error - a pointer to the function which will be called when an engine error occures.

l_cb_daisy_progress - a pointer to the function which will be called during playback, supplying the progress in ms.

Returns:

daisyplayer_t - the daisy data struct which must be passed along with all the API functions.

5.1.4.7 int daisy_load (daisyplayer_t *daisy*, char * *path*)

Loads a new daisy book.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

path - a full path to the daisy dtb (ncc.* | *.ncx) to open.

Returns:

1 in case of success and -1 in case of error.

5.1.4.8 int daisy_pause (daisyplayer_t *daisy*)

Toggle pause.

Pauses playback if state is playing and continues playing of state is paused.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

Returns:

1 in case of success and -1 in case of error.

5.1.4.9 int daisy_play (daisyplayer_t *daisy*)

Starts playback if a book is loaded.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

Returns:

1 in case of success and -1 in case of error.

5.1.4.10 int daisy_seek (daisyplayer_t *daisy*, int *seek_option*)

Seek operations which can be performed when a book is loaded.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

seek_option - a daisy_seek_option.

Returns:

0 if end_of_book, 1 in case of success and -1 in case of error.

See also:

daisy_seek_option

5.1.4.11 int daisy_stop (daisyplayer_t *daisy*)

Stops playback.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

Returns:

1 in case of success. Has no other return values at this point.

5.1.4.12 void daisy_term (daisyplayer_t *daisy*)

Terminates the daisy library and frees memory used by it.

It should be called when daisy are no longer needed.

Parameters:

daisy - the daisy data struct which must be passed along with all the API functions.

5.2 libdaisy_mainpage.doxygen File Reference

Index

daisy_bookinfo_option
libdaisy.h, 5
DAISY_BOOKINFO_TITLEIMAGE
libdaisy.h, 5
DAISY_BOOKINFO_TITLETEXT
libdaisy.h, 5
DAISY_BOOKINFO_TOTALTIME
libdaisy.h, 5
daisy_chapter_info
libdaisy.h, 5
DAISY_CHAPTER_TITLE
libdaisy.h, 5
DAISY_CHAPTER_WEIGHT
libdaisy.h, 5
DAISY_END_OF_BOOK
libdaisy.h, 6
DAISY_ERROR_AUDIO_CREATE_MMAP
libdaisy.h, 6
DAISY_ERROR_AUDIO_DATA_IS_NULL
libdaisy.h, 6
DAISY_ERROR_AUDIO_FREE_MMAP
libdaisy.h, 6
DAISY_ERROR_AUDIO_FSTAT
libdaisy.h, 6
DAISY_ERROR_AUDIO_INITIATE_DATA
libdaisy.h, 6
DAISY_ERROR_AUDIO_MALLOC
libdaisy.h, 6
DAISY_ERROR_AUDIO_NOT_INITIALIZED
libdaisy.h, 6
DAISY_ERROR_AUDIO_NOT_PLAYING
libdaisy.h, 6
DAISY_ERROR_AUDIO_NOT_STOPPED
libdaisy.h, 6
DAISY_ERROR_AUDIO_OPEN
libdaisy.h, 6
DAISY_ERROR_AUDIO_PAUSED_WHILE_-
NOT_PLAYING
libdaisy.h, 6
DAISY_ERROR_AUDIO_STOPPED_-
WHILE_NOT_PLAYING
libdaisy.h, 6
DAISY_ERROR_MISC_INIT_MUTEX
libdaisy.h, 6
DAISY_ERROR_PLAYBACK_NO_AUDIO_-
IN_SEGMENT
libdaisy.h, 6
DAISY_ERROR_PLAYBACK_NO_DTB_-
LOADED
libdaisy.h, 6
DAISY_ERROR_PLAYBACK_NO_TEXT_-
IN_SEGMENT
libdaisy.h, 6
DAISY_ERROR_PLAYBACK_SEEK_-
FAILED
libdaisy.h, 6
DAISY_ERROR_UNKNOWN
libdaisy.h, 6
daisy_get_chapter_count
libdaisy.h, 6
daisy_get_chapter_info
libdaisy.h, 6
daisy_get_info
libdaisy.h, 7
daisy_get_position
libdaisy.h, 7
daisy_goto_position
libdaisy.h, 7
daisy_init
libdaisy.h, 8
daisy_load
libdaisy.h, 8
daisy_pause
libdaisy.h, 8
daisy_play
libdaisy.h, 9
daisy_position, 2
nodepos, 3
smilpos, 3
daisy_seek
libdaisy.h, 9
DAISY_SEEK_NEXT_CHAPTER
libdaisy.h, 6
DAISY_SEEK_NEXT_PASSAGE
libdaisy.h, 6
daisy_seek_option
libdaisy.h, 5
DAISY_SEEK_PREV_CHAPTER
libdaisy.h, 6
DAISY_SEEK_PREV_PASSAGE
libdaisy.h, 6
DAISY_SEEK_TO_BEGINNING
libdaisy.h, 6
daisy_status
libdaisy.h, 6
daisy_stop
libdaisy.h, 9
daisy_term
libdaisy.h, 9
daisyplayer_t

libdaisy.h, 5

libdaisy.h, 3
daisy_bookinfo_option, 5
DAISY_BOOKINFO_TITLEIMAGE, 5
DAISY_BOOKINFO_TITLETEXT, 5
DAISY_BOOKINFO_TOTALTIME, 5
daisy_chapter_info, 5
DAISY_CHAPTER_TITLE, 5
DAISY_CHAPTER_WEIGHT, 5
DAISY_END_OF_BOOK, 6
DAISY_ERROR_AUDIO_CREATE_-MMAP, 6
DAISY_ERROR_AUDIO_DATA_IS_-NULL, 6
DAISY_ERROR_AUDIO_FREE_MMAP, 6
DAISY_ERROR_AUDIO_FSTAT, 6
DAISY_ERROR_AUDIO_INITIATE_-DATA, 6
DAISY_ERROR_AUDIO_MALLOC, 6
DAISY_ERROR_AUDIO_NOT_-INITIALIZED, 6
DAISY_ERROR_AUDIO_NOT_-PLAYING, 6
DAISY_ERROR_AUDIO_NOT_-STOPPED, 6
DAISY_ERROR_AUDIO_OPEN, 6
DAISY_ERROR_AUDIO_PAUSED_-WHILE_NOT_PLAYING, 6
DAISY_ERROR_AUDIO_STOPPED_-WHILE_NOT_PLAYING, 6
DAISY_ERROR_MISC_INIT_MUTEX, 6
DAISY_ERROR_PLAYBACK_NO_-AUDIO_IN_SEGMENT, 6
DAISY_ERROR_PLAYBACK_NO_-DTB_LOADED, 6
DAISY_ERROR_PLAYBACK_NO_-TEXT_IN_SEGMENT, 6
DAISY_ERROR_PLAYBACK_SEEK_-FAILED, 6
DAISY_ERROR_UNKNOWN, 6
daisy_get_chapter_count, 6
daisy_get_chapter_info, 6
daisy_get_info, 7
daisy_get_position, 7
daisy_goto_position, 7
daisy_init, 8
daisy_load, 8
daisy_pause, 8
daisy_play, 9
daisy_seek, 9
DAISY_SEEK_NEXT_CHAPTER, 6
DAISY_SEEK_NEXT_PASSAGE, 6
daisy_seek_option, 5
DAISY_SEEK_PREV_CHAPTER, 6
DAISY_SEEK_PREV_PASSAGE, 6
DAISY_SEEK_TO_BEGINNING, 6
daisy_status, 6
daisy_stop, 9
daisy_term, 9
daisyplayer_t, 5
libdaisy_mainpage.doxygen, 10

nodepos
daisy_position, 3

smilpos
daisy_position, 3

Appendix H

Manuals

This is the user manuals for our two front-ends for libdaisy, Daisyconsole and Daisygui.
The latest versions of the manuals can be retrieved from the Documentation section of the DaisyPlayer Project web page[1].

[1] The DaisyPlayer Project http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/

Daisyconsole user manual

H.1 What is daisyconsole? Where can I get it?

Daisyconsole is a free Linux program for playing Daisy Digital Talking Books. Daisyconsole is a front-end for the libdaisy library, created for those who wants a simple console user interface.
You can download the latest version of daisyconsole and libdaisy from the DaisyPlayer Project's web-page (http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/).

H.2 Getting started

Once you have installed daisyconsole you can start it by typing *daisyconsole* in a console.
When the program is started you will see something like figure H.1 below.

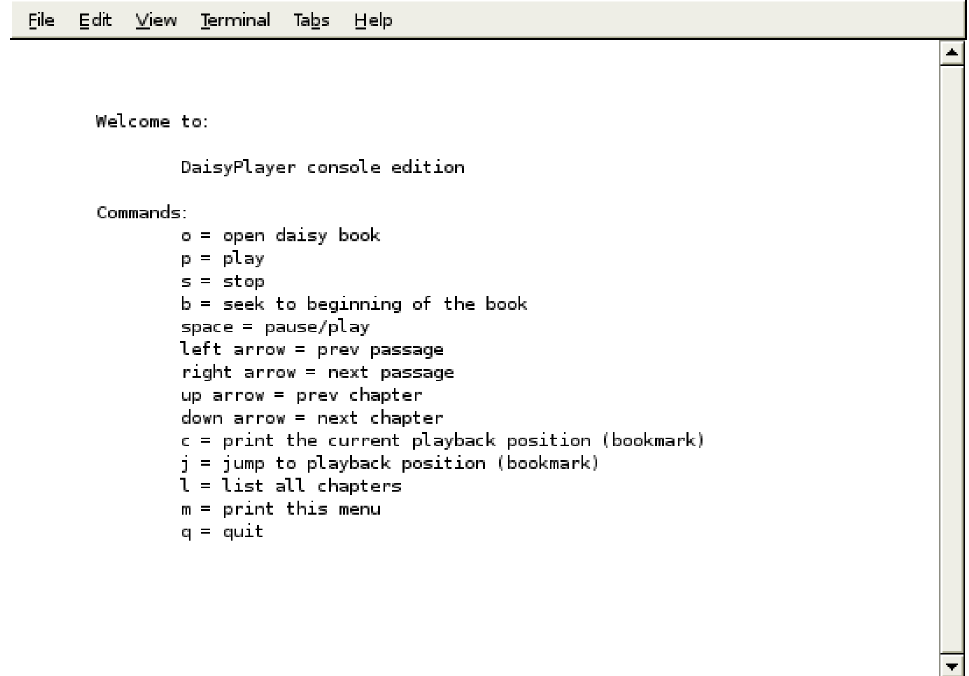

Figure H.1: Daisyconsole screenshot.

The menu shows the keys to access the various functionality of

daisyconsole. If you like the menu to reappear anytime during playback of the Daisy DTB, simply press the letter *m.*

H.3 Open a Daisy DTB

The first step for playing a Daisy DTB is to load the book. You can load a book by pressing the letter *o* and supply the full path to the DTB's ncc.* file (Daisy 2.02) or *.ncx file (Daisy Z39.86-2005) when prompted for it and press *enter*. A message will appear on the screen telling you if the book was successfully loaded or not.

H.4 Playback

The playback functionality will be available after a book is successfully loaded.

H.4.1 Play

You can start the playback by pressing the letter *p*. The program will start playing the passages in chronological order outputting the corresponding text to the screen as a passage is played. If the book does not contain any text, the audio will play without any text being outputted. If there is only text in the book, one passage at the time is displayed at screen. The text will normally be outputted synchronized with the audio, so when there is no audio present, the user must navigate to the next passage when he is done reading the last output.

H.4.2 Pause

When a book is playing, you can toggle pause by pressing the *space bar*. By pressing the *space bar* once more, the playback will continue.

H.4.3 Stop

When a book is playing, you can stop playback by pressing the letter *s.* To restart the playback from the last passage after a stop, simply press *p.*

H.4.4 Seek

Seeking is possible as soon as a book is loaded. You can seek while the book is playing, stopped or paused.
You can seek to the next passage by pressing *right arrow*, previous passage by pressing *left arrow*, next chapter by pressing *down arrow*, previous chapter by pressing *up arrow* and to the beginning of the book by pressing the letter *b*. If you try seeking forward beyond the last passage or chapter you will get a *end of book* message and if you try seeking backwards beyond the start of the book, playback will jump to the beginning.

H.5 Positioning and bookmarking

Daisyconsole has the ability to jump to any chapter and passage in the loaded Daisy DTB. By pressing the letter *c*, the current playback position is displayed at the screen as two integers representing the chapter and passage position. This can be used as a bookmark to continue playback later at the same playback position. To jump to a playback position, press the letter *j* and enter the chapter and passage integers when prompt. The playback position will not change if you enter invalid integers.
The jump functionality can be used to jump to a specific chapter by entering the chapter number for the chapter integer and *1* for the passage integer. To get the list of chapter with corresponding chapter numbers see section H.6.

H.6 Book indexing

You will get a list of chapters, with its chapter numbers, in a loaded Daisy DTB by pressing the letter *l*. Because there can be a lot of chapters, only 10 chapters is outputted at a time. If there are more chapters left to output, a "—*more*—" line will appear at the end of the list. To get the next 10 chapters simply press any key. Press the letter *q* to exit the chapter listing before all chapters has been listed.

H.7 Shortcut keys

Shortcut	Description
o	Open a Daisy DTB.
p	Start playback of the loaded DTB.
space	Toggle pause.
s	Stop playback.
left arrow	Seek to previous passage.
right arrow	Seek to next passage.
up arrow	Seek to previous chapter.
down arrow	Seek to next chapter.
b	Seek to beginning of the DTB.
c	Print the current playback position.
j	Jump to a specific playback position.
l	List chapters.
m	Print the menu.
q	Quit program.

Table H.1: Shortcuts

Daisygui user manual

H.8 What is daisygui? Where can I get it?

Daisygui is a free Linux program for playing Daisy Digital Talking Books. Daisygui is a front-end for the libdaisy library, created for those who wants a simple graphical user interface.
You can download the latest version of daisygui and libdaisy from the DaisyPlayer Project's web-page (http://developer.skolelinux.no/info/studentgrupper/2006-hig-daisyplayer/).

H.9 Getting started

Once you have installed daisygui you can start it by typing *daisygui* in a console.
When the program has started you will see something like figure H.2 below.

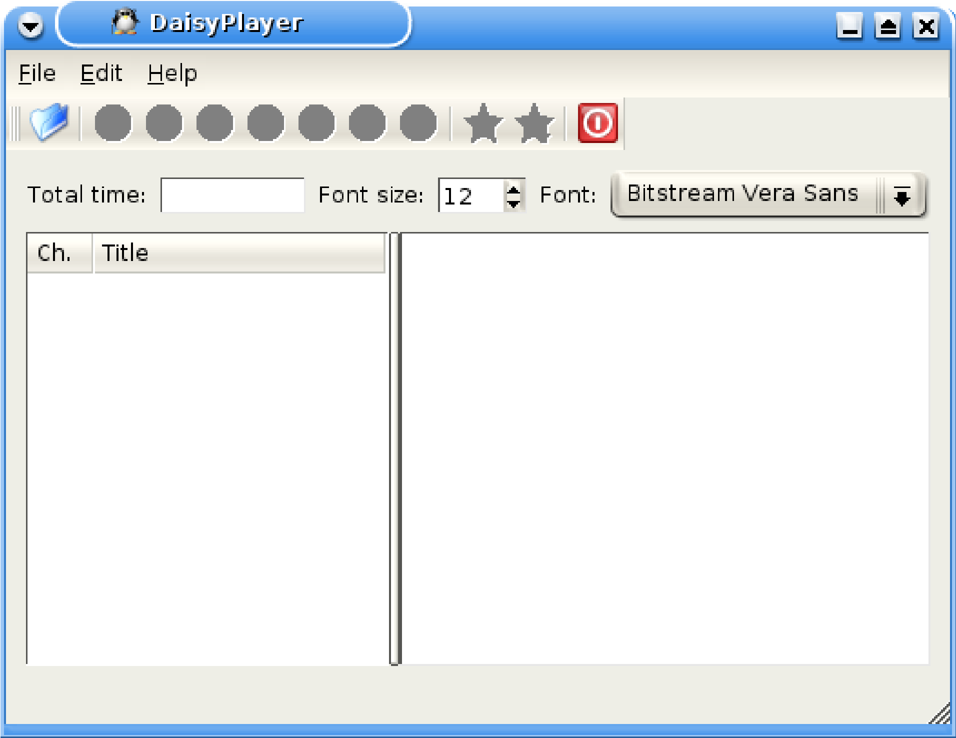

Figure H.2: Daisygui screenshot.

At the top of the window you have a toolbar with buttons for opening a Daisy DTB, seeking to the previous chapter, seeking to the previous passage, starting playback, toggling pause, stopping playback, seeking to the next passage, seeking to the next chapter, bookmarking current playback position, jumping to last bookmark and exiting the program. Some of the buttons will not be available at all time, e.g. play will not be available if no book is loaded.
Below the toolbar you have box showing the total playback time when a Daisy DTB is loaded, a box showing the font size, which you can increase or decrease, and a drop-down box where you can choose between different fonts to use in the center area.
In the left area you have a box for listing all the chapters in a loaded Daisy DTB. When a DTB is loaded, it will show a list of all chapters with its chapter number and title.
The center area is for outputting the text from the current passage when playing a Daisy DTB.
The bottom field of the main window shows status- and tooltip messages. Try moving the mouse pointer over some of the buttons to see the corresponding tooltip.
You can change the language for the user interface by clicking *Help* -> *Language* -> *your-language*, or by pressing *Alt+H+L*.

H.10 Open a Daisy DTB

The first step for playing a Daisy DTB is to load the book. You can load a book by pressing and you will see a window like figure H.3. Browse to the DTB's ncc.* file (Daisy 2.02) or *.ncx file (Daisy Z39.86-2005) and press open. The tooltip field on the bottom of the main window will tell if the book was opened successfully or not.

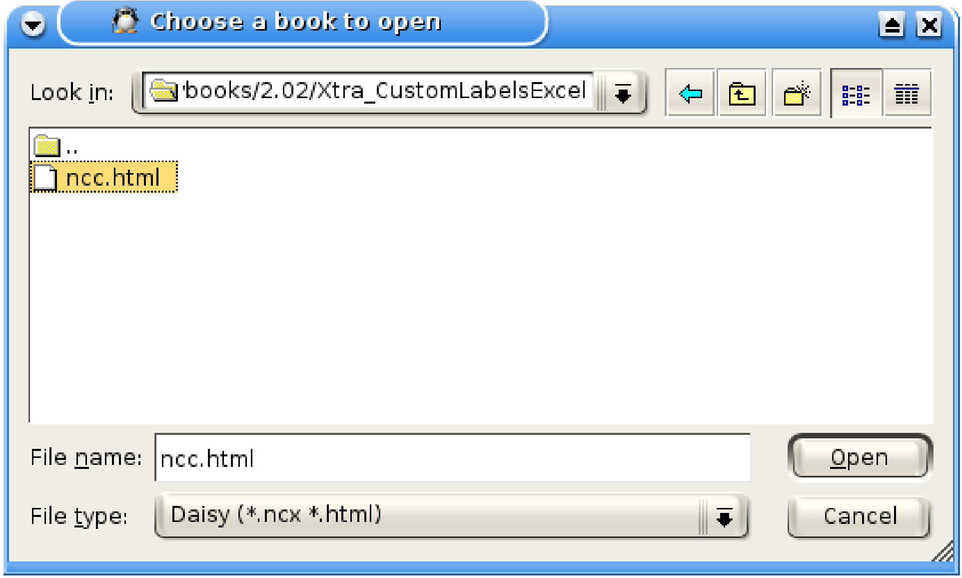

Figure H.3: Daisygui screenshot.

H.11 Playback

The playback functionality will be available after a book is successfully loaded.

H.11.1 Play

You can start the playback by clicking on the button. The program will start playing the passages in chronological order outputting the corresponding text to the center area as a passage is played. If the book does not contain any text, the audio will play without any text being outputted. If there is only text in the book, one passage at the time is outputted to the screen. The text will normally be outputted synchronized with the audio, so when there is no audio present, the user must navigate to the next passage when he is done reading the last output.

H.11.2 Pause

When a book is playing, you can toggle pause by clicking the button. By clicking the button once more, the playback will continue.

H.11.3 Stop

When a book is playing, you can stop playback by clicking the button. To restart the playback from the last passage after a stop, simply click the button.

H.11.4 Seek

Seeking is possible as soon as a book is loaded. You can seek while the book is playing, stopped or paused.
You can seek to the next passage by clicking the button, previous passage by clicking the button, next chapter by clicking the button and previous chapter by clicking the button. If you try seeking forward beyond the last passage or chapter you will get a *end of book* message in the tooltip area and if you try seeking backwards beyond the start of the book, playback will jump to the beginning.

H.12 Bookmarking

You can add a bookmark to the current playback position by clicking the button. When you want to jump back to the bookmarked position, simply click the button.

H.13 Book indexing

When a Daisy DTB is loaded, the left area will show a list of all chapters with its corresponding chapter number and title. To jump directly to a specific chapter, simply double-click the entry in the list corresponding to the chapter you want to jump to.

H.14 Shortcut keys

Shortcut	Description
Ctrl+O	Open a Daisy DTB.
Ctrl+P	Start playback of the loaded DTB.
Ctrl+Space	Toggle pause.
Ctrl+S	Stop playback.
Ctrl+Left	Seek to previous passage.
Ctrl+Right	Seek to next passage.
Ctrl+Up	Seek to previous chapter.
Ctrl+Down	Seek to next chapter.
Ctrl+B	Set bookmark.
Ctrl+L	Load bookmark.
Ctrl+A	About dialog.
Ctrl+X	Exit program.

Table H.2: Shortcuts

Appendix I

CD contents

The following CD contains a dump of our repository with all the project files. For more detail about the structure, please refer to README on the root of the CD.