ECE 492: Computer Engineering Design Project

Network-Controllable Embedded MP3 Player

An embedded MP3 player with a web application user interface. It decodes MP3 files stored on an SD card and plays the audio output to the audio interface of the DE2 development board.

Group 12
Brady Thornton
Jason Brown

Lab day: Thursday, 2:00-5:00pm

Table of Contents

Abstract 5
Functional Requirements of Project 5
MP3 Playing Requirements 6
User Interface Requirements 6
User Interface Control Minimum Requirements 6
Web Interface Requirements 7
Meeting the Requirements 7
Design and Description of Operation 7
Bill of Materials 8
Reusable Design Units 8
I/O Signals 9
3-Wire SPI SD Card Interface 9
Optrex 16207 LCD Display Core 9
Wolfson WM8731 Audio Codec 10
DM9000A Ethernet Controller 10
Datasheet 10
Power Consumption 10
System Block Diagram 10
Background Reading 11
Software Design 11
Theoretical Software State Machine Model 11
RTOS Tasks and Division of Functionality 12
RTOS Inter-task Message Passing 14
Web Server Task 14
HTTP Communication over Ethernet 14
Public Web API 15
Track Indexing 16
SD Card File System 16
Parsing ID3 Metadata from MP3 Stream Data 17
ID3 Specification and Background 17
Parsing Strategy 19

Calculating Track Length using MPEG Headers 19
Constant Bit Rate Track Length 20
Variable Bit Rate Track Length 20
External Libraries and Components 21
Libraries used inside DE2 21
Libraries used in client-side web application 21
User Interface Design 21
Client-Server Architecture 21
Visual Design 22
Test Plan 29
Software 29
Reliability and Stress Testing 30
Hardware 30
Results of Experiments and Characterization 30
Multitasking Performance 30
Hardware Resources and Utilization 31
SD Card Buffering 31
MP3 Decoding Optimizations 32
References 33
Appendix 35
Quick Start Manual 35
Future Work 36
Hardware Documentation 37
System Block Diagram 37
Clock and Abridged Data Flow Diagram 37
Source Code 39
VHDL 39
Software – On Altera DE2 39
Software – Client Application 40
Self-Evaluation 41

Abstract

The MP3 encoding format is a popular audio format that uses lossy compression techniques to encode and store audio data. It is the de facto standard for storing and distributing digital audio content, especially in consumer personal media players. Our project goal is to design and construct a user-friendly MP3 player on the Altera DE2 development board that is capable of being controlled by any device in a user's local network.

The player reads MP3-encoded data from the onboard SD card slot, decodes the audio stream in software, and sends the decoded samples to the onboard audio codec. This process is done in real time, similar to consumer-grade MP3 playback devices. Additionally, the player acts as a web server and provides a user-friendly web application interface for controlling playback. The player makes use of freely available software libraries and IP cores to facilitate rapid development, such as libmad [11] for MP3 decoding, the InterNiche TCP/IP stack [9], the EFSL embedded file system [8], and the jQuery JavaScript library [17].

A number of our planned features made it into the final design; however, we removed the requirement for hardware control in order to allow for extra web interface development time. Equalization features were also dropped due to the computational demands in software and time constraints in exploring a hardware solution.

This document outlines the design of the system architecture and components required to implement the overall system, as well as the characterization of I/O.

Functional Requirements of Project

The high-level features and requirements of the embedded MP3 player can be described as follows:

- MP3 files stored on an SD card are decoded and played to the onboard audio codec for output
- MP3 track metadata is parsed using each file's embedded ID3 tag information
- Users can control the player using a web application interface over a local network
- The web interface supports multiple simultaneous clients and downloads rich artist metadata automatically when internet access is available to the client.
- Hot-swapping of SD cards is supported

The requirements needed to achieve this functionality are summarized in the tables below. The description column describes how each requirement contributes to the overall functionality of the MP3 player.

MP3 Playing Requirements

Component	Requirements	Functional Description
Onboard SD card reader	-Detect SD card -Read file system -Buffer read MP3 files	The SD card reader is the source of the data for the MP3 player. It is buffer read by the system and provides a constant stream of data to be decoded. A FAT32 file system, provided by EFSL [7,8] is used.
NIOS II CPU & software support	-Decode MP3 frames into PCM audio samples	The MP3 data is decoded before being sent to the audio interface. Decoded data is buffered to the audio codec.
Onboard Wolfson WM8731 Audio Codec	-Send PCM audio samples as output to headphone jack	The resultant PCM samples from the source MP3 are played through the headphone jack.

User Interface Requirements

Component	Requirements	Functional Description
NIOS II CPU & software support	-Index SD card -Parse ID3 metadata for each track and store in sorted data structures for playlist generation	The CPU performs software processing of file data, and performs tasks that respond to control of the player through the user interface.
Optrex 16207 2x16 Digit LCD Display	-Show basic player status updates, such as if it is paused, stopped, or playing.	The LCD screen acts as a basic status display for the MP3 player.
Web Application	-A web application to act as a client, making API requests based on the user's actions to allow for control of the player over the network. -Provide artist metadata such as album art, artist images, artist biographies, and similar artists where available.	The user interface is the primary point of interaction between the user and the player. The client uses the lastFM API to get artist metadata [24].

User Interface Control Minimum Requirements

Similar to the behavior in consumer MP3 players, the embedded MP3 player provides simple controls to the user via the web interface. All controls are handled by the client user interface via API calls to the server:

Play/Stop:	Starts or stops the playback.
Pause Toggle:	Toggles playback between the play state and paused state.

Next/Previous:	Plays the next or previous file on the SD card, if such files are available.
Track Selection:	Plays the track selected by the user through browsing or searching.
Volume Set:	Adjusts the overall volume of the output.

Particular attention was given to designing a pleasant and seamless graphical interface that shows the user their music collection in an elegant and refined way.

Web Interface Requirements

Component	Requirements	Functional Description
Ethernet port interface	-Monitor the Ethernet port for incoming requests -Perform bidirectional communication using the TCP/IP protocol stack and a simple subset of HTTP	The NicheStack [9] TCP/IP stack provides low level communication over Ethernet, and an application specific subset of HTTP has been used to handle web requests appropriately.
Web server	-Serve the appropriate data (HTML, JSON data, JavaScript, etc.) based on the user request -Communicate requests to the MP3 player's playback system to perform functionality such as track changes	The web server services user requests for MP3 player's web interface. Such requests include serving the HTML page (along with associated scripts and data) to the client, and passing client requests for MP3 player control to the playback system.
Web page user interface	-Display available tracks to user -Allow selection of track for playback -Allow other control requests such as volume level changes -Display the currently playing track	The webpage allows the user to operate the MP3 player over the network using a modern web interface.

Meeting the Requirements

All of the requirements necessary for playback were met. The web server requirement was added to allow for a richer control set and added value to the user-player interaction.

The requirements for equalization and hardware control were dropped in an effort to allow for extra time developing the player's core functionality. Much attention was given to player responsiveness, user experience, and proper functionality, and this required more time to complete than anticipated.

Design and Description of Operation

The main functional requirement of playing MP3 files from an SD card has been accomplished utilizing hardware included on the DE2 development board and communication channels (such as the Avalon Bus) in the hardware, as seen in Fig. 1. Components utilized include a 3-wire SPI interface to interface with the onboard SD card hardware slot, utilizing the onboard RAM and a NIOS II CPU to decode MP3 files, and interfacing with the audio codec using Altera's University Program audio core [1] and the University Program Audio and Video Configuration core for configuring the codec over a serial interface [2], to set all the numerous registers appropriately for our needs as per the data sheet [3].

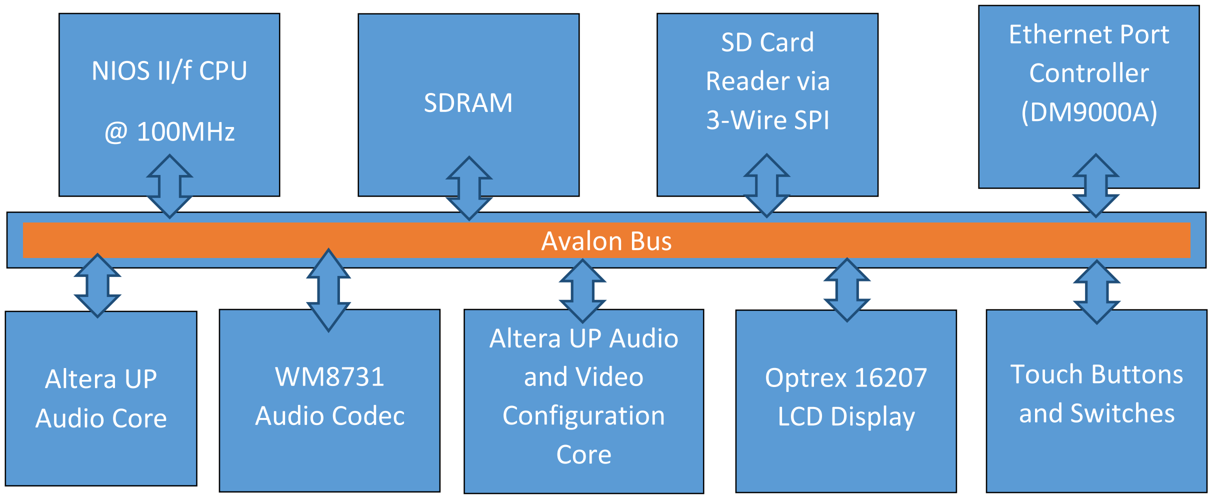

NIOS II/f CPU
@ 100MHz
SDRAM
SD Card Reader via 3-Wire SPI
Ethernet Port Controller (DM9000A)
Avalon Bus
Altera UP Audio Core
WM8731 Audio Codec
Altera UP Audio and Video Configuration Core
Optrex 16207 LCD Display
Touch Buttons and Switches

Figure 1: DE2 hardware communication architecture

Due to the computational nature of decoding MP3 files, we use the most powerful NIOS-II CPU configuration available, the NIOS-II/f. In order to maximize performance, the system clock rate has been increased to 100 MHz as opposed to the 50 MHz used in the introductory labs for the course.

Bill of Materials

Component	Cost
Altera DE2 Development Board	$519.75
2GB SD Card	$10.49
Linksys 4-Port Wired Router	$31.49
Ethernet cable	$0.98
Total	$562.71

Reusable Design Units

A summary of the software libraries used by the MP3 player is shown in the table below. Note that compilation size is calculated with binaries that are compiler optimized (that is, with gcc's "–O3" flag).

Component	Source Size	Compiled Size
libMAD Decoding Library	854KB	95KB
Embedded File System Library	247KB	68KB
NicheStack TCP/IP Stack	116KB	127KB
Total Library Usage	1217KB	290KB

SD card interfacing is done using a 3-wire SPI interface to the onboard SD card in one-bit SD bus mode [6], and a file system library (EFSL, as previously discussed) to support file allocation table (FAT) file systems with higher-level abstraction on file access [7]. EFSL is available under the GNU license, and conveniently has a wrapper written for it for the NIOS-II processor [8], which was employed to save writing our own implementation. As tested, the interface performs fast enough to saturate the MP3 decoder with enough data for real time audio playback.

Lastly, the NicheStack TCP/IP stack [9] is a reusable software library component that provides a basic TCP/IP stack for use with the DM9000A chip on the DE2. The NIOS-II IDE provides wizard generated sample code that implements a (quite limited) subset of the HTTP standard over top of this TCP/IP stack. This code was leveraged and modified to provide web access to static resources via HTTP GET, and perform API calls via HTTP POST.

I/O Signals

Signal	Location
SD Card bus (4 I/O)	Avalon bus to 3-wire SPI core to SD card pin out
Avalon Bus	On-board bus
Optrex 16207 LCD Display (14 I/O)	On-board component to microprocessor
Wolfson WM8731 Audio Codec (21 I/O)	On-board component to microprocessor
Push button signal (1 wire per button)	On-board component to microprocessor
DM9000A Ethernet Controller (23 I/O)	Avalon bus to controller core to Ethernet plug pin out

3-Wire SPI SD Card Interface

SPI Interface	SD Card Pinout
Master-Output-Slave-Input (out)	SD_CMD (in/out)
Master-Input-Slave-Output (in)	SD_DAT (in/out)
Slave Select_n (out)	SD_DAT3 (in/out)
SPI Clock (out)	SD_CLK (out)

Optrex 16207 LCD Display Core

I/O to FPGA	I/O to Hardware
Address (in)	E (out)
Data (in/out)	RS (out)
Control (in/out)	R/W (out)
	DB [8] (in/out)

Wolfson WM8731 Audio Codec

Signal	Type	Description
XTI/MCLK	Input	Master clock in
BCLK	Input/Output	Digital Audio Bit Clock
DACDAT	Input	DAC digital audio input data
DACLRCK	Input/Output	DAC sample rate L/R clock
ADCDAT	Output	ADC digital audio output data
ADCLRCK	Input/output	ADC sample rate L/R clock
CSB	Input	2-3 wire MPU chip select / MPU interface address
SDIN	Input/output	2-3 wire MPU data input
SCLK	Input	2-3 wire MPU clock input

DM9000A Ethernet Controller

Signal	Type	Description
ENET_DATA	Input/output (16 bits)	Data to and from the Ethernet controller
ENET_CS_N	Output	Inverted chip select signal
ENET_CMD	Output	Command select
ENET_INT	In	Interrupt signal
ENET_RD_N	Output	Inverted read select signal
ENET_WR_N	Output	Inverted write select signal
ENET_RST_N	Output	Inverted reset signal
ENET_CLK	Output	Ethernet clock signal (25 MHz from a PLL)

Datasheet

Power Consumption

Using an ammeter in series and a voltmeter in parallel, the power consumption and current drawn by the player was measured. The values were constant during idle, startup, and active states.

Using measured voltage and current, power $P = VI$.

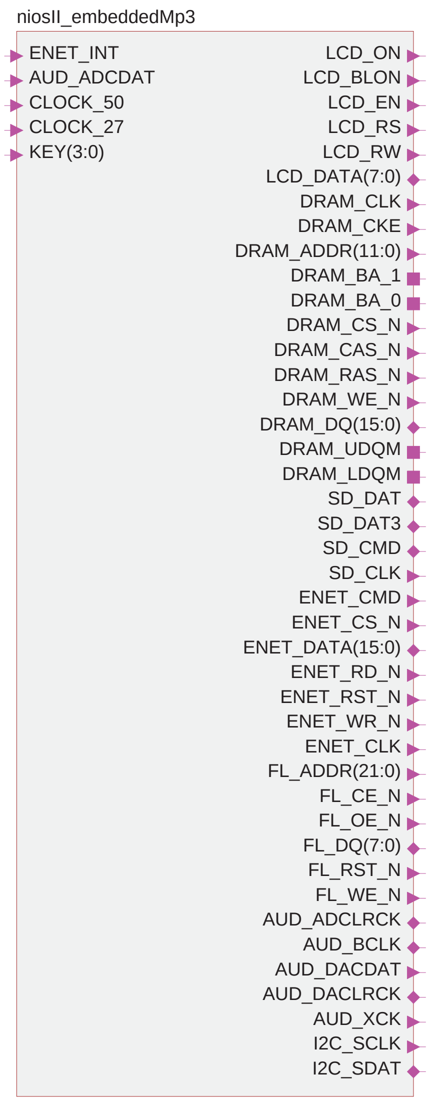
Measure	Value
Current	530 mA
Voltage	9.06 V
Power	4.81 W

System Block Diagram

On the next page is a system block diagram of the system and its external signals. We assume that the system is connected to peripherals on the Altera DE2 board and will be operating at room temperature.

Entity design.niosII_embeddedMp3

- ieee.std_logic_arith.ALL
- ieee.std_logic_unsigned.ALL
- Architecture: structure



Background Reading

In addition to the provided references, the reader can consult [15] to read about different audio encoding standards and how MP3 arose as the most popular audio encoding format. As discussed in the article, the MP3 format has faced some challenges due to its close ties with music piracy. Industries and retailers continuously seek new ways to provide MP3s to consumers through legal distribution channels.

Software Design

Theoretical Software State Machine Model

A desirable modelling construct for a device such as an MP3 player that possesses clearly defined instantaneous states and distinct transition scenarios is a finite state machine. Early in the design process, such an architecture was considered and prototyped in C++ making use of object-oriented constructs such as polymorphism. However, for simpler interoperability with many of the reusable software libraries being leveraged, further development was done in C. However, due to the clean theoretical mapping, the same conceptual design is applied to the C code implementation in the final MP3 player’s playback control task.

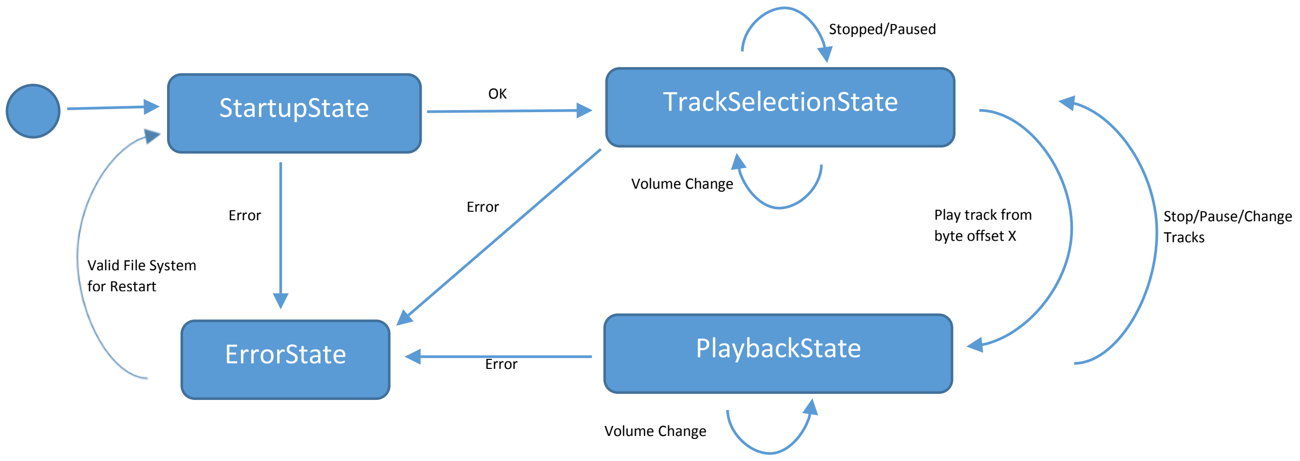


Figure 2: State machine model of an MP3 player

As outlined in Fig. 2, the theoretical layout of an audio playback device as a finite state machine consists of an initial startup state, followed by a track selection state, and a playback state. Additionally, any of these three states can transition to an error state upon encountering a runtime issue.

In the context of our implementation, the startup state encompasses doing an initial indexing of the SD card and its file system to gather track information. This state is also responsible for initializing and configuring any hardware devices such as the audio codec, and launching the web server as a separate task (whose operation is discussed in detail later).

The distinction between the track selection and playback states is most clearly illustrated by the use of the libmad MP3 decoder. There is a logical separation between the track selection state, in which nothing is actively playing, and the playback state, in which MP3 data is being actively decoded and played through the audio codec. We have hence distinguished between user controls that can be done actively during playback, and those which require returning from the MP3 decoding functions of the current track to a non-playing (or what we call track selection) state. It was determined that the only change that can be applied during playback without altering the decoding process is a volume change. All other events such as pausing, stopping, or changing tracks require stopping the decoding of the current track either indefinitely, or for the purpose of restarting decoding on a new file. Track selection state however deals with switching between tracks and restarting from a paused or stopped state by interfacing with the file system and offsets into given files. Note that to provide users with expected functionality in all states, volume change functionality is duplicated here.

It is worth noting that the error state of the player is entered in the case that an unrecoverable error has occurred. Typically, this is indicative of an SD read failure, and as such the error state simply prompts the user to enter a valid SD card and will wait until readable data exists for it to re-enter the startup state.

RTOS Tasks and Division of Functionality

The use of the MicroC/OS-II real-time operating system (RTOS) [25] affords us multitasking within our software design through the use of operating system tasks. Upon startup, a task is created to initialize and startup all other tasks for the system as a dispatcher task. Upon successful completion of this trivial process, the design of the system is such that all work is partitioned into two tasks:

- Player task:

  As discussed above, this task maintains a state machine-like process for indexing the SD card, selecting files, and MP3 decoding and playback. MicroC/OS-II lends itself to having long running tasks implemented as infinite loops, and as such each state in the state machine described above is implemented as a separate infinite loop, with the exception of the startup state which will not infinitely loop if it doesn't reach the track selection state, but rather move to the error state.

  State transitions between the two non-trivial states (i.e., the track selection state and the playback state) occur based on explicit polling of a semaphore-synchronized shared (with the web server task) player data structure that indicates control requests from the user. This polling model is similar in execution to a condition variable concept used by many multi-threaded applications, but does not require long-term blocking (through the use of semaphore "accept" calls rather than "pend"), so responsiveness to audio playback or events such as SD card removal are not inhibited. Given the free time afforded in the track selection state due to lack of decoding being performed, polling is an acceptable mechanism that ensures responsiveness and ease of implementation for features that cannot easily be mapped to hardware interrupts (such as the SPI interface to the SD card for example). In the playback state, polling occurs as the lone task as long as enough decoded audio is buffered for playback in the audio codec's output FIFO buffer. An interrupt is configured from the audio codec hardware core to signal that buffered data in the FIFO is running low. Within the interrupt service routine, the interrupt is disabled (to prevent hammering the ISR while we deal with the lack of buffered audio) and a binary semaphore is posted to indicate to that task that in addition to polling for control requests, time must be spent reading the file further from the SD card, and decoding MP3. Upon re-filling the buffer, the interrupt is re-enabled to allow for signaling once the audio FIFO runs low again. This synchronization of running the decoder is crucial so as not to overrun the output buffer due to faster than real-time decoding speeds.

- Web server task:

  In order to enable timely responses to requests, the web server is run as a separate task in the RTOS. In practice, the operation of the web server is to serve static web pages from the zip file system stored in flash memory in response to HTTP GET requests, and to route HTTP POST requests through to the MP3 player's API handlers. API requests that query for data are responded to by generating a JSON formatted message from the data available in the semaphore-synchronized shared player data structure. For example, requests to get the track list are implemented in this way. API requests to perform an action or control the player work similarly, but write to the shared player data structure to pass the control along to the player task to be handled. Full documentation for the supported API functionalities is given below.

RTOS Inter-task Message Passing

As previously mentioned, at the crux of the player's operation is the player data structure that stores all currently relevant information about the state of the player. The structure can best be understood by viewing the C code definition:

```c
typedef struct
{
    int currentTrack;
    int changeTrackTo;
    int volume;
    int currentPosition;
    unsigned int pausedAtFileOffset;
    bool isPaused;
    bool isStopped;
    bool shouldChangeOffset;
    int pausedPosition;
    bool shouldGoNext;
    bool shouldGoPrev;
    char *trackIndex;
    int timesIndexed;
    bool dataReadError;
} PlayerDataStruct;
```

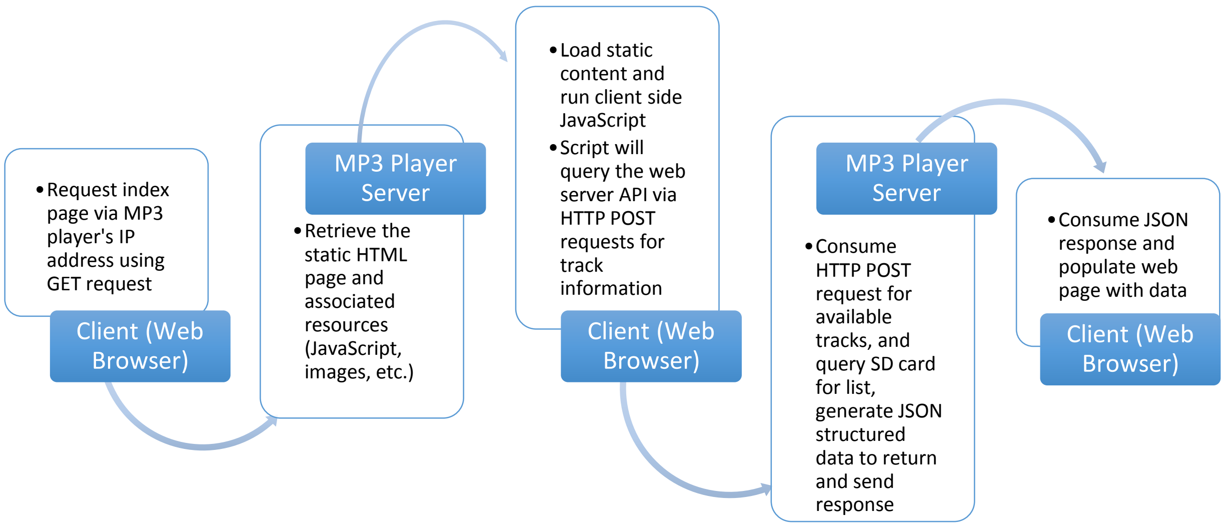
Note that this structure encompasses all necessary information to recover the current status of the player, and is also sufficient for passing messages related to the nature of track iteration or selection, pausing or resuming playback, and other status information such as error indicators, or a count of times the SD card has been indexed.

Message passing between the two major tasks running in MicroC/OS-II on the MP3 player (that is, the player task and the web server task) occurs by sharing a single instance of this structure, taking care to acquire the associated binary semaphore guarding this structure as a mutex. Values set by the web server task in response to an API handler (e.g., shouldGoNext, to indicate the player should switch to the next track) will be noted by the player task during its polling of the structure, and it may respond appropriately. Further, the web server uses the structure to reply to web requests inquiring about the state of the player (e.g., exposing the index of tracks on the SD card to a client via the trackIndex structure member).

Web Server Task

HTTP Communication over Ethernet

To provide users with a web interface to control and interact with the MP3 player, network communication over Ethernet is supported. This is accomplished through the use of the DM9000A chip [4] on the DE2 board. In line with nearly every device hosting a web server, reliable data transfer is to be facilitated by the TCP/IP protocol stack. Altera's NIOS-II IDE contains a sample HTTP server project utilizing the InterNiche TCP/IP implementation [9], which has been modified for the purposes of the MP3 player to support API calls using standard HTTP POST requests [10]. The HTTP GET functionality in the sample code was left largely unchanged and is used to serve web browser clients the static HTML files, images, and scripts needed to populate the web application. A visual description of a typical client request from the web server, requesting the index page is provided in Fig. 3.



Request index page via MP3 player's IP address using GET request
Client (Web Browser)
MP3 Player Server
Retrieve the static HTML page and associated resources (JavaScript, images, etc.)
Load static content and run client side JavaScript
Script will query the web server API via HTTP POST requests for track information
Client (Web Browser)
MP3 Player Server
Consume HTTP POST request for available tracks, and query SD card for list, generate JSON structured data to return and send response
Consume JSON response and populate web page with data
Client (Web Browser)

Figure 3: Data flow between web server API and a user's web browser

Public Web API

In order to service requests from the web based user interface, an important consideration in the design of the software is creating a publicly accessible API to easily query for data from and send control signals to the MP3 player. The functions on this API are responsible for notifying the player task of control requests made through the network, as well as exposing status information to networked clients.

Calls to the API from clients result in the web server task changing the values of the `PlayerData` structure based on the context of the API call. Meanwhile, the decoder task checks `PlayerData` during buffer refilling and after a track has completed playing. In this way, changes to `PlayerData` result in modifying the behavior of playback, based on the logic of the decoding task. This decouples API logic from player logic so that each task may focus on its own requirements instead of needing to know about the internal functionality of other tasks.

The table below indicates the available calls on the API and all expected parameters and data returned. Note that all requests are required to be made in HTTP 1.0/1.1 POST request format, and all returned data is formatted in JavaScript object notation (JSON) structures.

API Resource	Parameters	Description/Return Data
togglePause	N/A	Toggles the player's current playback state between play and pause.
setVolume	(int) vol	Sets the audio codec's volume to the value *vol*. This is in the range [0,100] and is validated client-side. Regardless of malicious intent, volume settings are still restricted server-side to be within [0,100], and hence non-compliant parameters simply produce undefined, but safe behavior.
stopPlayback	N/A	Stops playing the current track.
previousTrack	N/A	Stops the current track and starts playing the previous track in the player's logical play order.
nextTrack	N/A	Stops the current track and starts playing the next track in the player's logical player order.
trackList	N/A	Request for a JSON string containing the current track index.
playerStatus	N/A	Request for a JSON string containing the player's status. This can be used as a heartbeat for a client to periodically ensure synchronization with the player.
playTrack	(int) id	Stops the current track and starts playing a track with id *id*. Note that track id's are defined in the JSON returned via the *trackList* API call.

Track Indexing

In order to provide the user with intuitive information regarding the contents of their music library on the SD card, the contents of the card are indexed as part of startup of the MP3 player. This is accomplished by mounting the card and its file system using EFSL, and iterating the root directory (support for a nested directory structure was not implemented in order to focus on more forward-facing features) in search of files with an "mp3" extension. For each MP3 file found, the ID3 track metadata is parsed (as outlined later in this report) and stored for later user friendly display. Following this, sorting of the indexed tracks is performed first by artist, followed by album title, and finally track number. This allows for continuous playback in an intuitive and natural ordering. Lastly, a JSON string is generated and stored in the player data structure to facilitate fast responses to clients requesting the available tracks from the player.

SD Card File System

To allow for interfacing with the SD card, two approaches were taken. The first was to use Altera's University Program SD Card core [5], which provides a software HAL for high level file system access to the SD card, and an IP core, which abstracts away the serial peripheral interface (SPI) needed to communicate with the SD card in SPI bus mode [6]. While we were able to successfully read data from the SD card (after fixing a bug in the provided HAL), we found that the core did not give us the performance necessary to saturate the MP3 decoder with file data at real time playback speeds. Upon investigation we found that this was due to a number of poor design choices in the core, such as busy waiting on data reads, and inefficient code (e.g. a single value might be calculated several places in a function rather than caching the result). Of chief concern was the inability to clock the core at frequencies other than 50 MHz, which would have limited our entire system. Additionally, it was also noted that the HAL did not function properly for modern SD cards and success was only achieved with an old 512 MB card.

A better alternative to the University Program core is to use the 3-wire SPI interface natively provided in SOPC Builder, directly to the pins in the card. This allows for minimal delay in hardware, customizability of the clock speed, and simple memory mapped read and writing in software.

However, a file system is still desirable so users can configure their music library on a consumer device such as a laptop computer, and then playback with our MP3 player. The Embedded Fileystems Library (EFSL) provides such functionality by providing FAT16 and FAT32 support for embedded devices provided a few functions are written to translate library calls into the memory mapped I/O to the storage device [7]. We found that such a wrapper for the NIOS-II was already publicly available, which made implementation easier [8]. We noted a significant performance improvement with the use of this library and the direct SPI hardware core, and were able to read and write files quickly on SD cards of varying sizes and FAT partitions.

Parsing ID3 Metadata from MP3 Stream Data

ID3 Specification and Background

In order to display useful information about the tracks to the user, the player must extract and parse ID3 tag information from each file during the initial indexing process. Examples of such metadata include the name of the track, artist, and album, and perhaps the track's genre and year of release.

ID3 is a standard for encoding metadata in MP3 files, and there are two encoding formats: ID3v1 and ID3v2. (Although they share similar names, they are somewhat different in their structure).

ID3v1 data was introduced at a time when not all players supported metadata. It consists of a fixed-length header 128 bytes long that is appended to the end of the MP3 file to allow for error-free playback in devices that were not designed to parse it. Valid ID3v1 headers always start with the string "TAG", so the presence of an ID3v1 tag can be checked by examining the bytes corresponding to the header at the correct offset and performing a string comparison.

The structure of an ID3v1 header is shown in the figure below using data from [12]. Bytes are zero-indexed.

ID3v1 Data Specification

0-2	3-32	33-62	63-92	93-96	97-124	125	126	127
"TAG" string	Title	Artist	Album	Year	Comment	Track Valid Bit	Track Number	Genre Code

All characters are encoded in ASCII, with the exception of the track number and genre code. The genre can be looked up by consulting the ID3v1 specification genre lookup table [13] and finding the entry that corresponds to the encoded value.

ID3v2, unlike ID3v1, is a variable-length header prepended to the front of an MP3 file. ID3v2 consists of a header followed by a series of frame-length-data segments, and thus allows for a wider variety of data to be encoded into the stream. A valid ID3v2 header always starts with the "ID3" string, so the presence of an ID3v2 header can be checked by examining the first few bytes of an MP3 file and performing a string comparison. A valid header is then followed by a variable number of frames, ending at an offset specified by the header. The format for the header and frames are shown in the tables below using data from [13].

ID3v2 Header Specification

0-2	3-4	5	6-9
"ID3" string	ID3v2 version	Flags	Size of ID3 data (excl. header)

ID3v2 Frame Specification

0-3	4-7	8-9	10...data size
Frame tag name	Frame data size	Flags	Tag data

Using these standards, a small ID3 parser can be written that reads values at fixed and at variable offsets depending on the ID3 version and places the parsed data into pre-allocated strings. The strategy for doing this is outlined in the next section.

Parsing Strategy

No external ID3 parsing libraries are used. We have implemented our own ID3 parser that is lightweight and supports most of the common and desirable metadata attributes encoded by MP3 encoders. The ID3 parser does not parse fields that the embedded MP3 player does not present to the user.

An MP3 can be encoded with ID3v1 metadata, ID3v2 metadata, both, or neither. Each of these cases must be considered when designing for maximum compatibility with an arbitrary MP3 file.

Since ID3v2 provides possibly richer data than ID3v1, the ID3 indexer checks if the stream contains an ID3v2 header. It will look for all the data that it needs in the ID3v2 frames first. If any of the needed attributes are null or are not included in the ID3v2 header, the parser will then check for the existence of an ID3v1 header. If it exists, the indexer will attempt to fill in missing data fields using the ID3v1 data. If both ID3v2 and ID3v1 tags have been exhausted and attributes are still null, then we can assume that no other metadata can be used to get the missing data and the field is left null.

If the stream does not contain and ID3v2 header, the player will attempt to fill all fields using ID3v1 tag data. As in the first case, if data is missing (or if there is no ID3v1 header), then the fields are left null by the indexer.

One caveat to this hierarchical parsing strategy is that the genre field in ID3v2 can legally contain genre codes specified by the ID3v1 specification. Therefore, the indexer must explicitly check the ID3v2 genre data for the presence of an ID3v1 genre code and use the ID3v1 lookup table to replace it; otherwise, the genre will left as a coded decimal string that is of no real use to the user.

Calculating Track Length using MPEG Headers

An important role of any audio player is keeping track of time and displaying it to the user in some way. Unfortunately, ID3 tags do not specify track lengths. This is due to the variation in encoders and the stream types supported by the MPEG format.

To complicate things further, MPEG streams can be constant bit rate (CBR) or variable bit rate (VBR); therefore, a naïve length calculation using file size cannot always be a reliable way to measure stream duration. It is also impractical to calculate the average bitrate for a variable bitrate stream since it would require parsing every encoded frame of every track during the indexing process, a costly and unnecessary operation.

In order to calculate the length of a track during the indexing process, the indexer must peek into the first encoded frame of the MPEG stream. This is located at an offset equivalent to the ID3v2 data size, or if there is no ID3v2 header, an offset of zero. The length can be calculated using one of two ways depending on the bitrate type.

An MPEG header contains a variety of data about the encoded data stream. In the case of this MP3 player, we choose to support only the most common type of stream: MPEG Layer-III audio sampled at 44 100Hz. The vast majority of MP3 files fit this description, as most CDs are sampled at that frequency. We also support all standard bitrates, including variable bit rates.

Bitrate type can be determined by looking for the presence of a variable bit rate header immediately after the first MPEG header. Similar to ID3v2 tags, variable bit rate headers are identified by a string – in the case of VBR headers, "Xing", "Info", or "VBRI". [14]

Constant Bit Rate Track Length

If the bit rate is a constant number ***b*** kilobits per second, then the length of the track in seconds can be calculated to be

$$L_{CBR} = \frac{(S - I) * 8}{b * 1000}$$

Where L is the calculated length in seconds, S is the size of the file (in bytes), I is the size of the ID3 tag data (in bytes), and *b* is the constant bit rate in kilobits per second. The 8 and 1000 are unit conversion multipliers (8 bits in one byte; 1000 bits in one kilobit. Bit rates use 1000 bits per kilobit, not 1024).

Variable Bit Rate Track Length

If the bit rate is variable, we cannot rely on the bitrate of any particular MPEG frame. Fortunately, the variable bit rate header (added by both of the mainstream MP3 encoders) supplies the total number of MP3 frames *F* in the encoded stream. Then,

$$L_{VBR} = \frac{F}{R}$$

where F is the number of frames in the entire MPEG stream, and R is the number of frames per second in the bit stream.

R can be calculated as

$$R = \frac{f_s}{x}$$

Where f_s is the sampling rate of the MP3 stream and x is the number of samples per frame. Given that our player only officially supports MPEG Layer-III audio sampled at 44 100 Hz, the number of frames per second is always 1152 (as per [14]), and thus R can be calculated to be 38.28125.

External Libraries and Components

Libraries used inside DE2

MicroC/OS-II is used as a real time operating system to run the MP3 player's controller as a task, as well as introduce tasks as needed for I/O operations that can be performed in parallel with other processing. One such example is web server activity, which can serve pages over Ethernet asynchronously with limited effect on MP3 decoding.

As previously discussed, the open-source library libMAD is used for MP3 file decoding, the NicheStack TCP/IP stack is used for network communication over Ethernet, and the Embedded Filesystems Library (EFSL) is used for file allocation table (FAT) file system support for the SD card.

Libraries used in client-side web application

The client-side web-application leverages the industry-standard jQuery JavaScript library [17] to perform all of its asynchronous requests, as well as to support other libraries required by the user interface. jQuery also facilitates visual and content-related functionality such as animations and DOM manipulation for dynamic page content. The other libraries used in the client-side application are jQuery UI [18], jQuery hoverIntent [19], jQuery Coda Slider [20], jQuery Simple Slider [21], and jQuery Timer [22]. Lastly, icon images are used from a batch icon library provided by Adam Whitcroft [23].

User Interface Design

Client-Server Architecture

Due to the limited capabilities of the NicheStack protocol stack in terms of performance, the web application was designed to perform most of the intensive work on the client's side.

To reduce the number of bytes required to load the page, all CSS and HTML files were minified by removing white space, new lines, and tab characters. The Google Closure compiler was used to minify the JavaScript functions [26]. The total size of the web application when minified, and including all server-slide images, is 172KB.

The fundamental design idea behind the client is that of a heartbeat API call. The "playerStatus" API call is performed by all connected clients every second. The player responds with information about which track it is playing, the position in the track, whether it is paused or stopped, the current volume, and the current re-index count.

When the client receives these values, it changes its state to synchronize with them if its own state is different than that of the server. This facilitates multi-client interaction, since if one client changes a parameter such as the volume, other clients will synchronize to the player's values at most one second later.

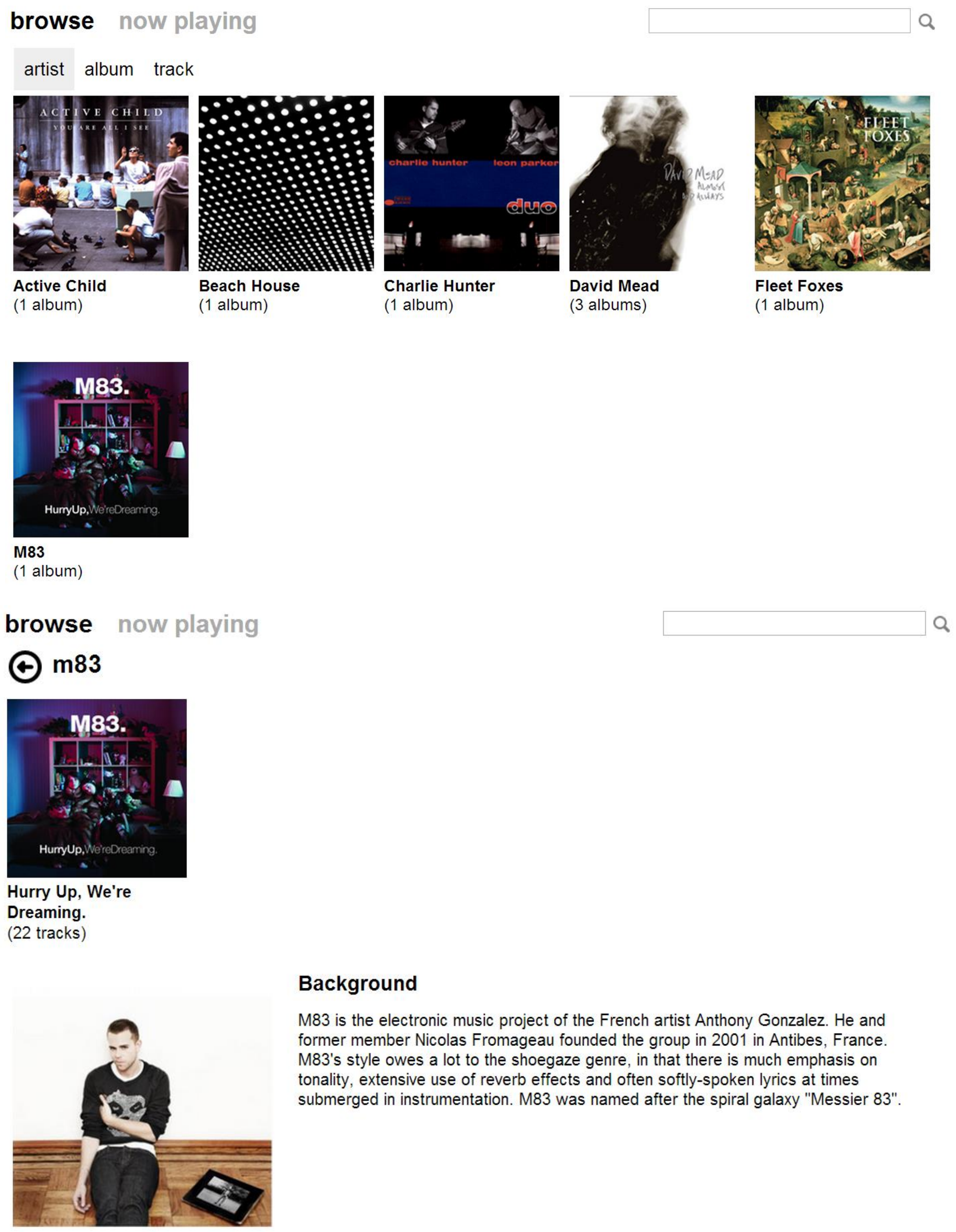
Visual Design

Since MP3 players are common, users have high and rigid expectations about the ease of interaction with the device. This places much responsibility on user interface designers to design interfaces that are consistent with these expectations.

In general, the design of the web application's interface was approached with *Jakob's Law* in mind, which states, "users prefer interfaces that work the same way as other interfaces they already know" [27]. This was mostly applicable to the way navigation was designed and the way the controls work. The control icon behavior most closely resembles YouTube's controller behavior, a behavior that most users will be familiar with.

In addition to incorporating standard UI elements and icons to provide visual cues to users, overall aesthetic and fluidity were considered when designing the application to be pleasant and easy to use. Animations are liberal but are only used when the effect of responsiveness needs to be elicited.

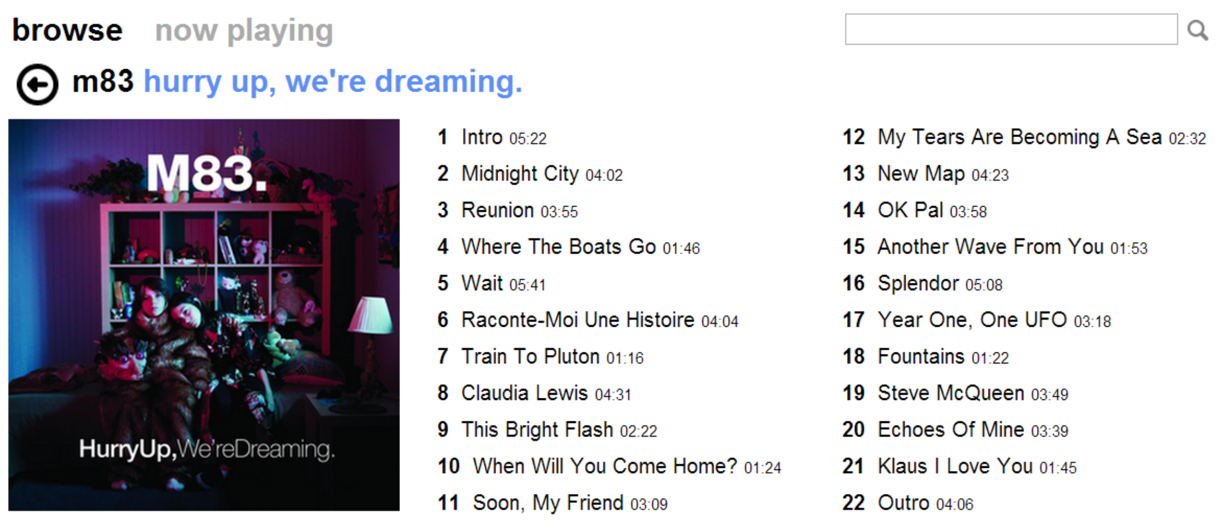
On the next page are screenshots of the user interface (bitmap images are used because they are screenshots of a pixelated display). Following them are the same images with annotations and descriptions of the high-level UI features.



Background

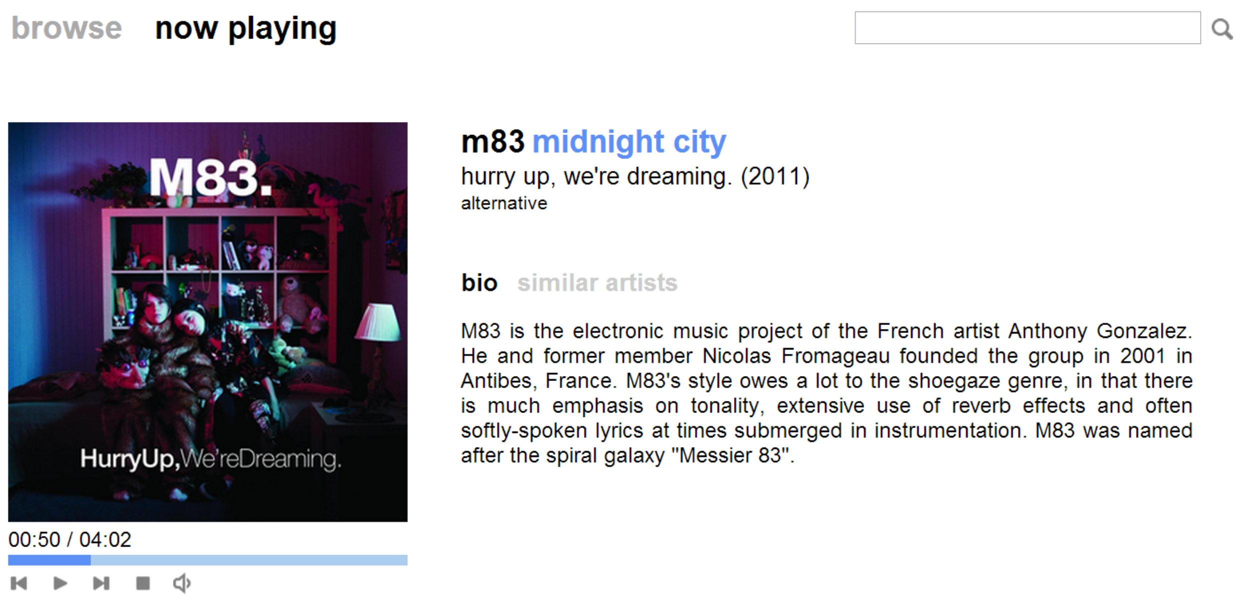
M83 is the electronic music project of the French artist Anthony Gonzalez. He and former member Nicolas Fromageau founded the group in 2001 in Antibes, France. M83's style owes a lot to the shoegaze genre, in that there is much emphasis on tonality, extensive use of reverb effects and often softly-spoken lyrics at times submerged in instrumentation. M83 was named after the spiral galaxy "Messier 83".

Figure 4: Example home page and artist screen for web application.



m83 hurry up, we're dreaming.

1	Intro 05:22	12	My Tears Are Becoming A Sea 02:32
2	Midnight City 04:02	13	New Map 04:23
3	Reunion 03:55	14	OK Pal 03:58
4	Where The Boats Go 01:46	15	Another Wave From You 01:53
5	Wait 05:41	16	Splendor 05:08
6	Raconte-Moi Une Histoire 04:04	17	Year One, One UFO 03:18
7	Train To Pluton 01:16	18	Fountains 01:22
8	Claudia Lewis 04:31	19	Steve McQueen 03:49
9	This Bright Flash 02:22	20	Echoes Of Mine 03:39
10	When Will You Come Home? 01:24	21	Klaus I Love You 01:45
11	Soon, My Friend 03:09	22	Outro 04:06



m83 midnight city
hurry up, we're dreaming. (2011)
alternative

M83 is the electronic music project of the French artist Anthony Gonzalez. He and former member Nicolas Fromageau founded the group in 2001 in Antibes, France. M83's style owes a lot to the shoegaze genre, in that there is much emphasis on tonality, extensive use of reverb effects and often softly-spoken lyrics at times submerged in instrumentation. M83 was named after the spiral galaxy "Messier 83".

00:50 / 04:02

Figure 5: Example album view and now playing screen.

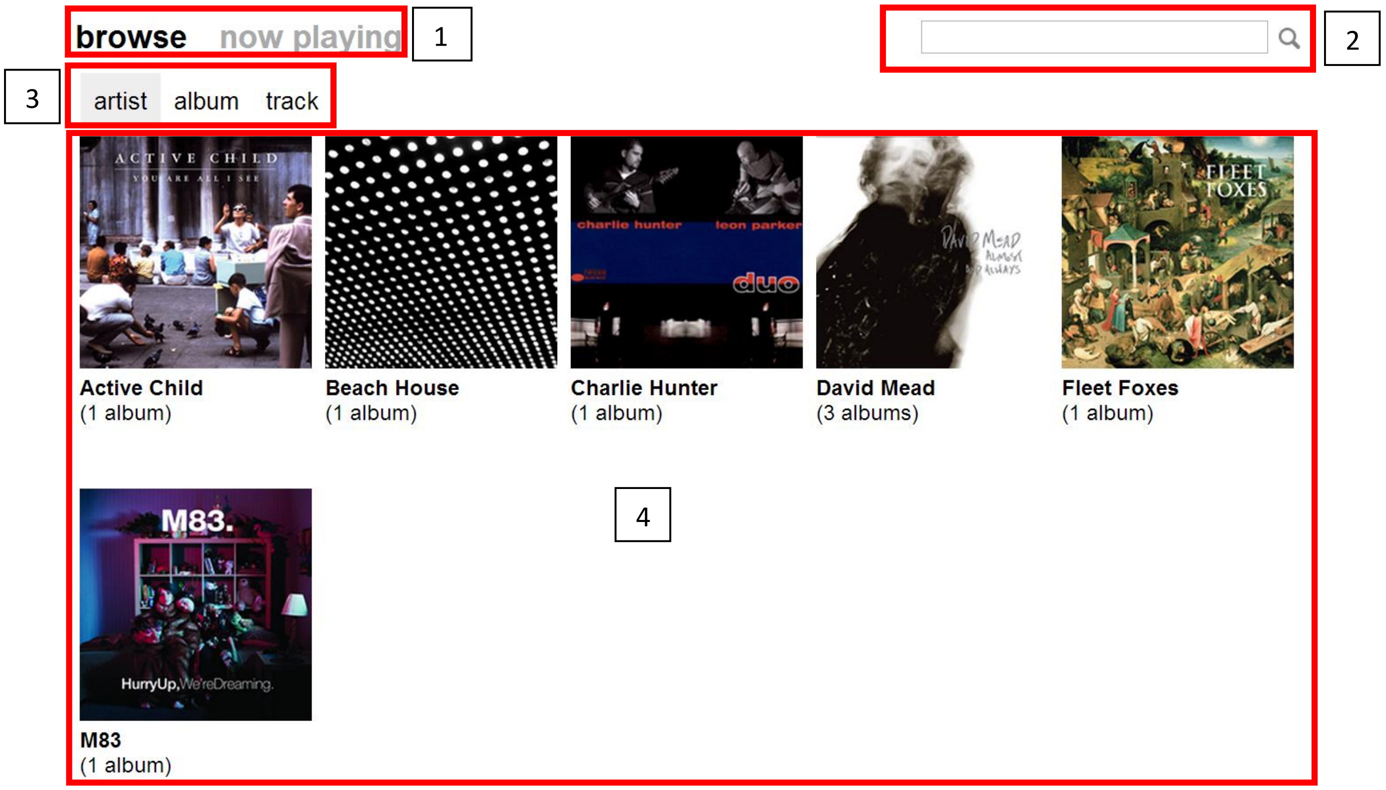
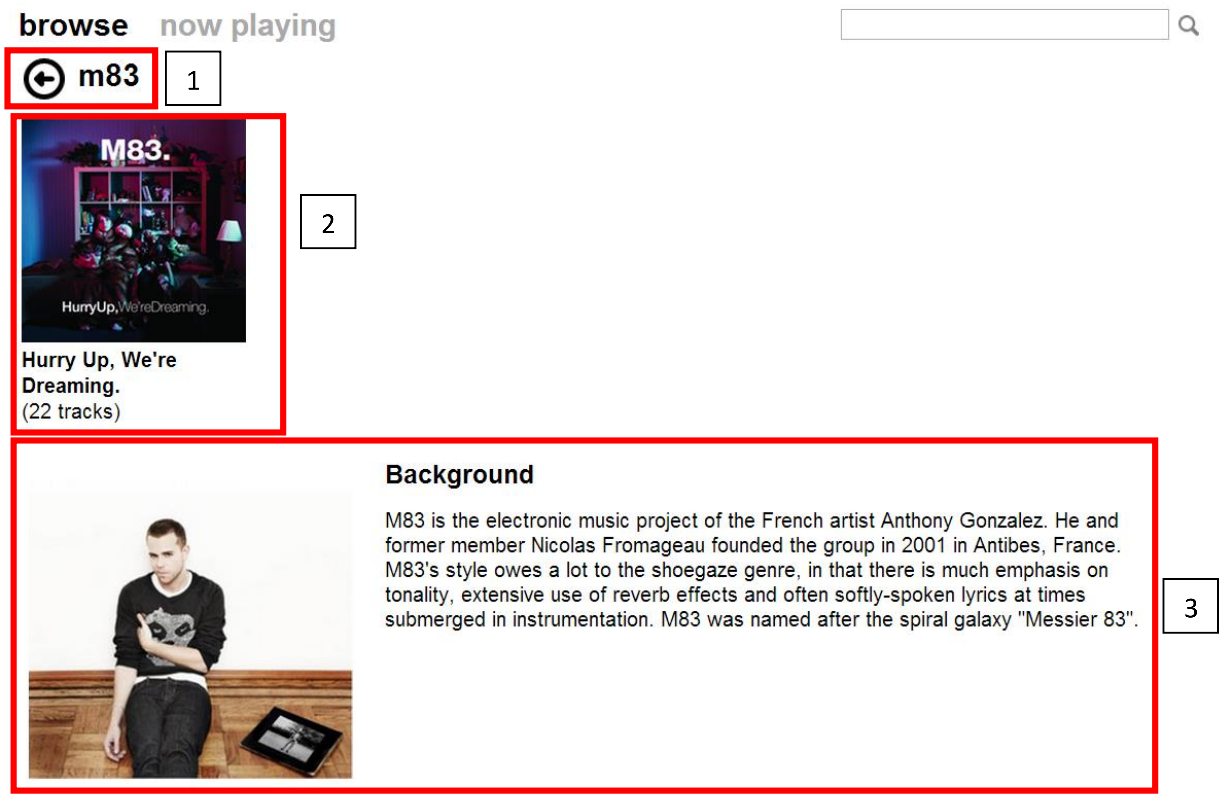


Figure 6: Annotated home screen.

Element	Description
1	Main screen selector, used to switch between browsing the library and viewing the currently playing track.
2	The search pane. The artists, albums, and/or tracks that match a query will be shown to the user, sorted by category. If the query is more than 3 characters in length, the insides of all library metadata strings are searched in addition to the front of the strings. For example, the query "The" will return all artists, albums, and tracks containing the word "The", whereas the query "Th" will return only artists, albums, and tracks beginning with "Th", and not artists, albums or tracks that contain the string "Th" anywhere.
3	The view context selector for the main browsing mode. It allows the user to group the main pane contents by artist, by album, or by track. Results are alphabetized by their respective category depending on the view context.
4	The primary content window. Clicking on a thumbnail will take the user to the thumbnail's logical details page. If an artist is clicked, the artist's albums will be shown. If an album is clicked, the album's tracks will be displayed. If a track is clicked, the player will begin to play the track.



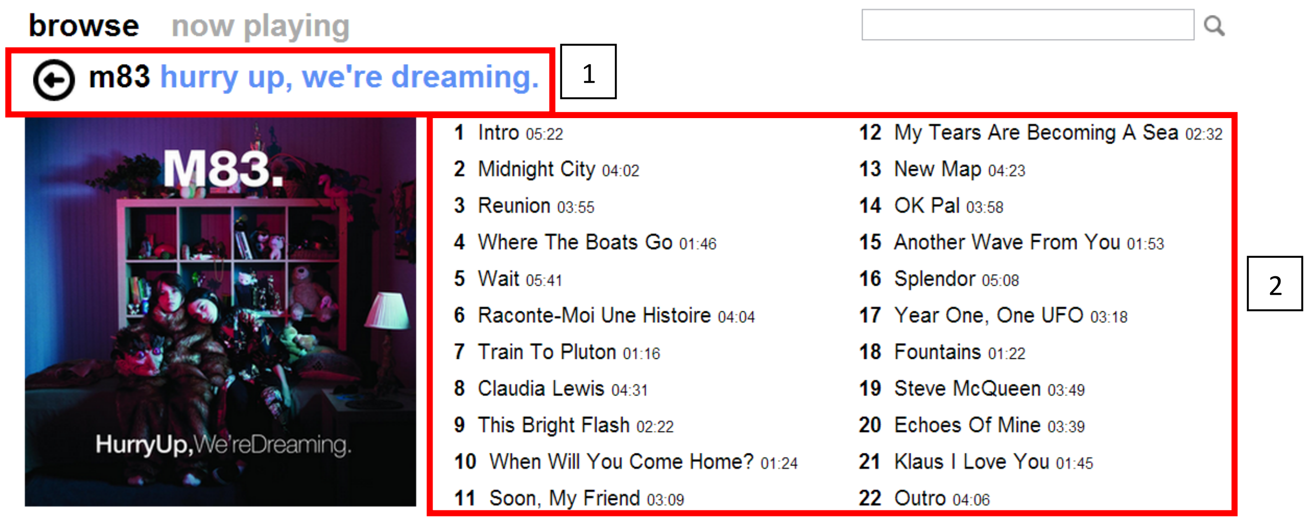
Hurry Up, We're Dreaming.
(22 tracks)

Background

M83 is the electronic music project of the French artist Anthony Gonzalez. He and former member Nicolas Fromageau founded the group in 2001 in Antibes, France. M83's style owes a lot to the shoegaze genre, in that there is much emphasis on tonality, extensive use of reverb effects and often softly-spoken lyrics at times submerged in instrumentation. M83 was named after the spiral galaxy "Messier 83".

Figure 7: Annotated artist view.

Element	Description
1	The artist breadcrumb and back button. In the current context, the back button will return to the main screen. Clicking on an artist's name will bring the user to the artist's album view.
2	The content area for the artist's albums and details about how many tracks for that album are in the library.
3	The artist's background pane. If the client is connected to the internet, the lastFM API [24] will be queried and an artist image will be downloaded and displayed alongside the artist's biography.

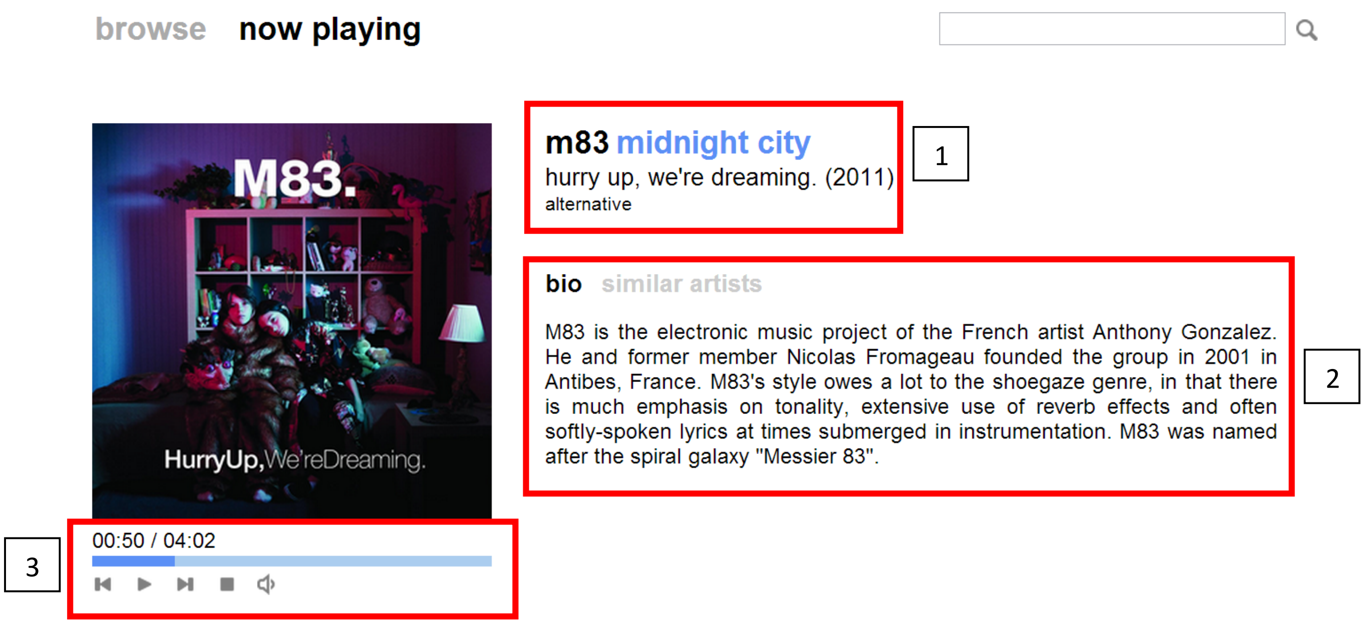


m83 hurry up, we're dreaming.

1	Intro 05:22	12	My Tears Are Becoming A Sea 02:32
2	Midnight City 04:02	13	New Map 04:23
3	Reunion 03:55	14	OK Pal 03:58
4	Where The Boats Go 01:46	15	Another Wave From You 01:53
5	Wait 05:41	16	Splendor 05:08
6	Raconte-Moi Une Histoire 04:04	17	Year One, One UFO 03:18
7	Train To Pluton 01:16	18	Fountains 01:22
8	Claudia Lewis 04:31	19	Steve McQueen 03:49
9	This Bright Flash 02:22	20	Echoes Of Mine 03:39
10	When Will You Come Home? 01:24	21	Klaus I Love You 01:45
11	Soon, My Friend 03:09	22	Outro 04:06

Figure 8: Annotated album view.

Element	Description
1	Artist and album breadcrumbs. In this context, clicking the back button or the artist's name will take the user to the artist's albums page.
2	The track listing of the current album. Clicking any track will make an API call to the player to begin playing the track. The user will then be brought to the "now playing" screen.



m83 midnight city
hurry up, we're dreaming. (2011)
alternative

M83 is the electronic music project of the French artist Anthony Gonzalez. He and former member Nicolas Fromageau founded the group in 2001 in Antibes, France. M83's style owes a lot to the shoegaze genre, in that there is much emphasis on tonality, extensive use of reverb effects and often softly-spoken lyrics at times submerged in instrumentation. M83 was named after the spiral galaxy "Messier 83".

Figure 8: Annotated now playing view.

Element	Description
1	Current track metadata. Clicking on the artist will bring the user to that artist's album view. Clicking on the album will bring the user to that album's track view. Genre and year are shown only if the information is available for the track; otherwise, it is hidden.
2	The artist biography and similar artists pane. The same biography shown in the artist's album view is shown here. Clicking on "Similar Artists" will display artists that the user may also be interested in. If the lastFM API is unreachable, the words "Not available" will populate the pane.
3	The progress bar and player controls. The progress bar is a graphical view of how far the player is into playing the track. The control icons (from left to right) are previous, play/pause, next, stop, and volume control. The controls turn black when hovered over with a mouse. Clicking previous or next will tell the player to start playing the previous or next tracks, respectively. Clicking the play/pause control will toggle the player's pause state. When the player is paused, the icon is a play icon; when the player is playing, the icon is a paused icon. Hovering over the volume control area will bring up a volume slider, which can be clicked or dragged. The speaker icon will change depending on the volume. If the volume icon is clicked, the player will be muted. If it is clicked while muted, the player will set the volume to the volume it was at when it was muted.

Test Plan

Software

In order to facilitate progress on the MP3 player, iterative and regression testing was used throughout development. As expected for a finished project, all tests discussed in this section passed and met any acceptance criteria (logically implied or otherwise). A general approach to designing software with care to minimizing repetition of common tasks and to limiting the responsibilities of code segments to singular or small sets of actions. This allowed for the testing of entire functionalities to often be isolated to testing the correct operation of only a few lines of code.

Hardware independent software within the system, such as the logic representing playback state transitions was tested for correct operation by forcing values of various flags and conditions that are used to indicate state changes. In this way, we were able to confirm expected operation of software flow simply by assuming values of hardware dependent components.

The nature of embedded systems however is such that most software written is inherently dependent on the underlying hardware and devices being interfaced with. Due to this an idealized case of an automated test suite is very difficult to achieve due to hardware requirements. However, manual testing was performed by isolating small units of functionality in the software and verifying operation and expected program flow through the use of printing debug information on the LCD screen or over the JTAG UART.

Some examples of hardware dependent software testing include testing of the Ethernet driver and web server component by sending cURL (a command line tool for transferring data using various protocols) HTTP requests to the player over Ethernet and printing to the console the data received. Correctness of SD card driver software and file system code was verified by both writing and reading files from the card and confirming compliance with the file system as interpreted by a Windows based laptop computer. MP3 decoding was also tested for correctness by decoding an MP3 file and storing it on the SD card to ensure correct operation with various media players on personal computers.

Testing of memory usage was performed through the use of a function provided in MicroC/OS-II named “OS_STK_DATA” [25]. This function allows for examining the amount of stack memory being utilized by each task. This was valuable in ensuring that our code did not overrun the stack memory allotted for each task, even in deeply nested function call chains. Additionally, heap memory usage was tested by adding debug printing statements to indicate the memory addresses and size of each heap allocated memory block within the program. We were able to leverage the output to confirm that memory was not being allocated in any unexpected program control flow cases and that all heap allocated memory was in fact being freed. The information regarding address locations was used to ensure no collisions between the stack and heap were occurring during runtime.

Further reliability checks were also performed as outlined in the below table.

Reliability and Stress Testing

Component Under Test	Test Plan	Acceptance Criteria
Ethernet driver and web server software module	Manual spamming of the refresh key requesting the index HTML page of the player was repeated for indefinite periods of time.	The system should appropriately refuse connections when pushed to a reasonable connection limit, and must remain functional, although certainly less responsive.
Software control request handling	Fast clicking of user interface controls was tested. One such test case was repeatedly hammering the skip track button.	The system should appropriately queue requests and respond in an orderly, though delayed fashion.
MP3 decoder	Playback of high and variable bitrate MP3 files was tested to ensure playback remains acceptable even with increased load.	The system should handle high bitrates without stuttering the audio output.

Hardware

Hardware testing was largely performed tangentially through software. Since all components used in the design were on-board the DE2, facilities for probing with an oscilloscope or other debugging tools were limited. However, through utilization of simple software test cases, the correctness of both hardware and software components could be verified simultaneously.

Results of Experiments and Characterization

Multitasking Performance

Due to the streaming nature of the MP3 data, the multitasking strategy was designed carefully to permit intermittent suspension of the decoder task. It was essential to measure the decoder's optimized performance and to characterize the CPU time spent decoding so that we could ensure the CPU would be able to spend a reasonable amount of time serving web pages and processing (possibly several) client API requests.

Testing was performed by preventing the decoder task from being suspended and by removing all audio codec buffer writes. In this manner, the CPU is solely decoding samples so that a measurement of performance can be completed. The results are summarized in the table below.

gcc Optimization	Stream Type	Actual Length	Decoding Time	Actual/Decode Time Ratio
None @ 100MHz	128kbps CBR	74s	113s	1.88
None @ 100MHz	High VBR	60s	115s	1.92
-O3 @ 100MHz	128kbps CBR	74s	45.5s	0.615
-O3 @ 100MHz	High VBR	224s	151s	0.674

Therefore, we can expect that by using optimizations, it will take approximately 0.67 seconds to decode 1 second of audio samples, which, on average, would allow the HTTP server 0.33 seconds per second of runtime to serve pages and process API requests (in an optimal multitasking scheme).

Hardware Resources and Utilization

All system hardware is well within the limits of the FPGA's capabilities, and utilizes the following resources:

Total logic elements	5,977 / 33,216 (18 %)
Total combinational functions	5,235 / 33,216 (16 %)
Dedicated logic registers	3,582 / 33,216 (11 %)
Total pins	226 / 475 (48 %)
Total virtual pins	0
Total memory bits	261,504 / 483,840 (54 %)
Embedded Multiplier 9-bit elements	12 / 70 (17 %)
Total PLLs	2 / 4 (50 %)

SD Card Buffering

libMAD requires buffered input to the decoding methods in order to provide uninterrupted streaming output. Although a larger buffer size would result in fewer I/O calls to the SD card, the time spent reading the SD card to fill the buffer increases linearly as the buffer size increases.

We found that having an excessively large buffer of (1MB) resulted in large gaps in playback, as the buffering time exceeded the gap between buffered output samples. We also found that decreasing the buffer to a small amount resulted in a large number of unnecessary SD card reads, producing similarly choppy output.

The tradeoff in input buffer size is between buffer filling frequency and time to fill the buffer. Having a medium-sized buffer of 2048 bytes proved to be an adequate tradeoff that resulted in smooth playback with fast buffer refilling.

MP3 Decoding Optimizations

It was calculated that approximately 76 frames' worth of PCM samples must be decoded by the decoder every second. During initial development, system frequency was at 50MHz, and the software was running without any compilation optimizations.

Under these conditions, libMAD was unable to produce samples at a constant rate, and the audio was therefore choppy and of poor quality. We measured that it took approximately 3.1 seconds to decode one second of samples (44100 samples).

The first optimization made was to implement a custom instruction (FMUL), which outsources libMAD's most commonly called and intensive operation (fixed-point 16-bit multiplication) to hardware [16].

Also, system clock frequency was increased to 100MHz and gcc compiler optimization was turned to -O3, the highest level of optimization. libMAD's ability to decode samples increased dramatically and real time playback was achieved.

References

[1] Altera. “Altera University Program Audio IP Core.” Internet: ftp://ftp.altera.com/up/pub/Altera_Material/10.1/University_Program_IP_Cores/Audio_Video/Audio.pdf , July 2010 [February 16, 2013].

[2] Altera. “Altera University Program Audio/Video Configuration Core for DE-Series Boards.” Internet: ftp://ftp.altera.com/up/pub/Altera_Material/10.1/University_Program_IP_Cores/Audio_Video/Audio_and_Video_Config.pdf, July 2010 [February 16, 2013].

[3] Wolfson Microelectronics. “Portable Internet Audio CODEC with Headphone Driver and Programmable Sample Rates.” Internet: http://www.wolfsonmicro.com/documents/uploads/data_sheets/en/WM8731.pdf, October 2012 [January 19, 2013].

[4] Alex Newcomb, Tom Stefanyk. “Application Note – Webserver.pdf”. Internet: http://www.ece.ualberta.ca/~elliott/ece492/appnotes/2012w/Webserver/, February 15, 2012 [February 28, 2013].

[5] Altera. “Altera University Program Secure Data Card IP Core.” Internet: ftp://ftp.altera.com/up/pub/Altera_Material/10.1/University_Program_IP_Cores/Memory/SD_Card_Interface_for_SoPC_Builder.pdf, March 2009 [January 19, 2013].

[6] Wikipedia. “Secure Digital Transfer Modes”. Internet: http://en.wikipedia.org/wiki/Secure_Digital#Transfer_modes, January 2013 [February 2, 2013].

[7] Lennart Ysboodt, Michael De Nil. “Embedded Filesystems Library”. Internet: http://sourceforge.net/projects/efsl/, June 22, 2011 [February 21, 2013].

[8] Marcio Troccoli. “Embedded Filesystems Library”. Internet: http://www.ohloh.net/p/efsl/commits/68269113, November 7, 20015 [February 21, 2013].

[9] Altera. “Chapter 11: Ethernet and the NicheStack TCP/IP Stack – Nios II Edition” in “Nios II Software Developer’s Handbook”. Internet: http://www.altera.com/literature/hb/nios2/n2sw_nii52013.pdf, May 2011 [March 2, 2013].

[10] Wikipedia. “POST (HTTP)”. Internet: http://en.wikipedia.org/wiki/POST_%28HTTP%29, Februrary 2013 [February 28, 2013].

[11] Underbit. “MPEG Audio Decoder”. Internet: http://www.underbit.com/products/mad/, [January 19, 2013].

[12] Wikipedia. “ID3”. Internet: http://en.wikipedia.org/wiki/ID3, February 2013 [February 26, 2013].

[13] M. Nilsson. “ID3 Specification V2.3.0”. Internet: http://id3.org/id3v2.3.0, February 1999. [February 26, 2013].

[14] Konrad Windszus / CodeProject. “MPEG Audio Frame Header”. Internet: http://www.codeproject.com/Articles/8295/MPEG-Audio-Frame-Header#XINGHeader, April 2007 [February 26, 2013].

[15] McCandless, M. "The MP3 revolution," *Intelligent Systems and their Applications, IEEE* , vol.14, no.3, pp.8-9, May/Jun 1999. [March 2013]

[16] Nate Knight. “Embedded MP3 Player”. Internet: http://www.alterawiki.com/wiki/MP3_Player, September 2010. [January 19, 2013].

[17] jQuery. “jQuery Javascript Library”. Internet: http://www.jquery.com, March 2013. [March 2013].

[18] jQuery UI. “jQuery UI Javascript Library”. Internet: http://www.jqueryui.com, March 2013. [March 2013].

[19] Cherne, Brian. “jQuery hoverIntent plugin”. Internet: http://cherne.net/brian/resources/jquery.hoverIntent.html, March 2013. [March 2013].

[20] Batdorf, Kevin. “jQuery Coda Slider”. Internet: http://kevinbatdorf.github.io/codaslider/, October 2012. [March 2013].

[21] Smith, James. “jQuery Simple Slider.” Internet: http://loopj.com/jquery-simple-slider/, February 2013. [March 2013].

[22] Chavannes, Jason. “jQuery Timer.” Internet: http://jchavannes.com/jquery-timer/demo, February 2013. [March 2013].

[23] Whitcroft, Adam. “Batch Icon Library”. Internet: http://adamwhitcroft.com/batch, January 2013. [March 2013].

[24] Last.fm Ltd. “Last.fm Web Services”. Internet: http://www.last.fm/api, 2013. [April 2013].

[25] Micrium. “uC/OS-II Overview”. Internet: http://micrium.com/rtos/ucosii/overview/. 2012. [April 2013].

[26] Google. “Closure Tools”. Internet: https://developers.google.com/closure/. May 2012. [April 2013].

[27] Nielsen, Jakob. “The Need for Web Design Standards”. Internet: http://www.nngroup.com/articles/the-need-for-web-design-standards/. September 2004. [April 2013].

Appendix

Quick Start Manual

Note that where applicable the "Non-persistent" and "Persistent" tags indicate steps (or sub-steps) relevant only to the named mode of operation. All referenced files are relative to the root "niosII_embeddedMp3" directory in the submitted project source archive.

1. Program the board with the appropriate configuration.
    a. *Non-persistent*: Program the FPGA with "niosII_embeddedMp3.sof".
    b. *Persistent*: Program the FPGA with "niosII_embeddedMp3.pof".
2. Write the website files ("./software/mp3_player_syslib/ro_zipfs.zip") into flash memory at offset 0x100000 using the flash programmer.
3. Launch the Nios II IDE using the "./scripts/launch_nios2_ide.sh" script.
4. Ensure the IDE is using the "software" directory as its workspace, and the "mp3_player" project and its associated syslib are available.
5. Right-click on the "mp3_player" project and select "Run as -> Nios2 Hardware"
6. Upon startup of the web server for the first time, a prompt will appear in the console in the IDE prompting for the board's serial number in order to generate the MAC address for the board. Provided the board is used only on the local network and does not require global uniqueness, any 9 digit number will do.
7. The LCD screen will be prompting for insertion of an SD card. While in theory the hardware and software should support any SD card (note, excepting SDHC) using any FAT file system, we have tested and claim only full support for SD cards using FAT32 with a 32 kb cluster size. Any mp3 files in the root directory on the card should be indexed via the MP3 player upon card insertion.
8. Connect to a router via Ethernet on the 192.168.0.0/24 subnetwork. Any client machines can access the MP3 player's user interface by visiting 192.168.0.99 (a hard-coded static IP) with a web browser. Note that Google Chrome is strongly suggested due to its compliance with the emerging web standards used by the user interface.
9. *Persistent*: Write the software elf file ("./software/mp3_player/Debug/mp3_player.elf") into flash memory at offset 0x00 (i.e., no offset) using the flash programmer. Verify software operation is consistent with the steps outlines starting at step 7 upon device reboot. This step is explicitly placed last to discourage skipping step 6, in which launching from the IDE is required to input a 9-digit number for MAC address determination.

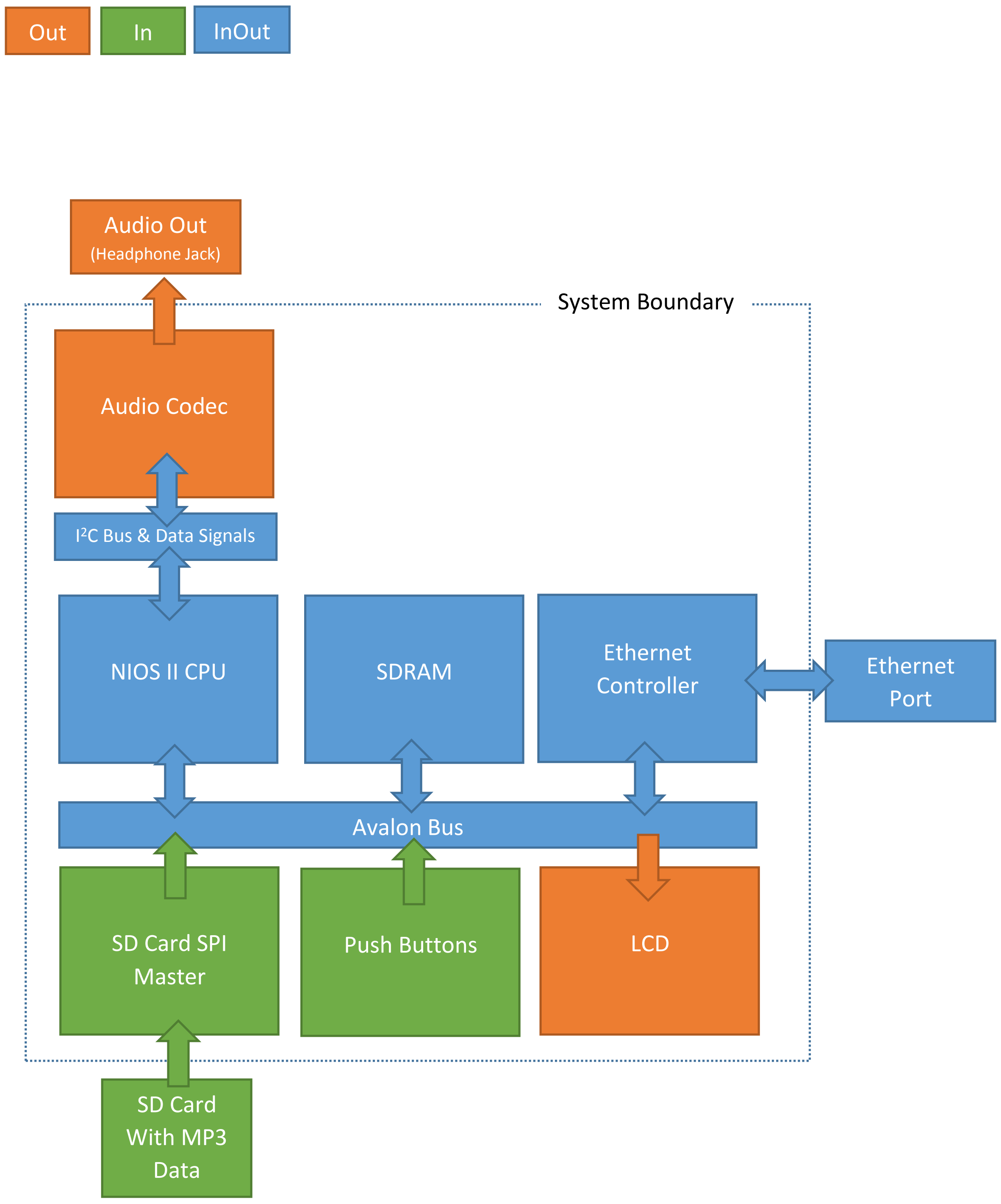
Future Work

Commercial digital music players have many features that could have been implemented in our embedded MP3 player if there were time. In no particular order, here is a list of additional features that could be developed:

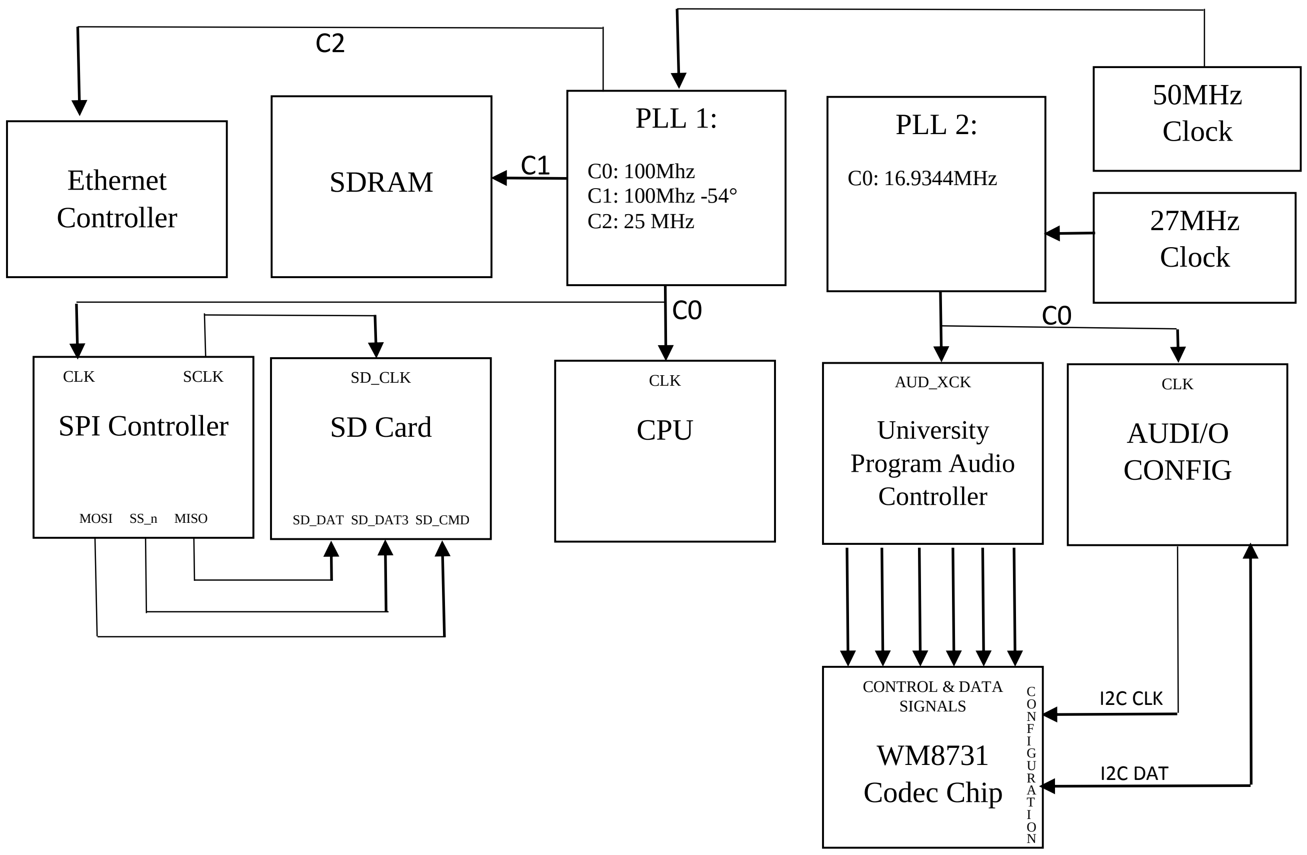
Hardware EQ	Hardware filters could be added before the audio codec to give the user the ability to equalize the music to their preferences.
Touchscreen:	A touchscreen interface for a tactile and more portable user experience.
Network Uploading:	MP3 files could be uploaded to the SD card from a remote network location for playback.
Playlist Creation:	Users can create playlists using the web application that persist across multiple listening sessions.
Network Streaming:	MP3 files could be streamed from a remote network location for playback.
Remote Control:	A wireless remote control for executing control functions remotely. Implementations could either using infrared communication or a simple radio system such as XBee radios.

Hardware Documentation

System Block Diagram



Clock and Abridged Data Flow Diagram



Not shown (to reduce risk of unnecessary clutter) are the memory mapped interfacing between the NIOS II CPU core and the other hardware cores such as the SPI controller, the audio and audio configuration controllers, SDRAM, and the Ethernet controller.

Source Code

An attached archive has been submitted with this report containing the current source code. All files that have been authored by group members contain headers explicitly stating authorship. It can be assumed that files not containing such headers are not authored by us and have been referenced in the Declaration of Original Content.

We maintain full authorship of the files below, except where noted therein; libraries and other files are in the archive but are not summarized here to avoid authorship ambiguity.

VHDL

File	Description	Status
niosEmbeddedMP3.vhd LOC: 315	The toplevel VHDL file for the system.	T

Software – On Altera DE2

File	Description	Status
audioControl.c / h LOC: 142	Contains functionality to configure the audio codec to the required specifications. Contains one function for volume adjustment adapted from an Altera sample project's code.	T
http.c / h LOC: 1419 Modified/Original LOC: ~180	Adapted from the Nios II IDE wizard generated HTTP server to route HTTP POST requests to our web API after some parameter parsing. Authorship primarily Altera's, except as noted in comments.	T
id3.c / h LOC: 834	Contains a lightweight ID3 parsing library for MP3 files. The functions read a provided filename using the SD card's file system library and populate a structure with the desired metadata. Full authorship.	T
id3Genres.c / h LOC: 219	Contains enum values for ID3v1 genre codes. Genre values are derived from the ID3 standard as published on the ID3 organization's site. Full authorship.	T
jsonGenerator.c / h LOC: 101	Generates a JSON-compliant string for a given array of id3 structures. Full authorship.	T
lcd_helper.c / h LOC: 57	Helper functions for writing to the LCD display. Full authorship.	T
madDecoder.c / h LOC: 412	Contains the primary entry points for buffering, decoding, and playing audio samples. Implements callbacks as defined by the libMAD library. Adapted from skeleton code provided by libMAD. Implementation of functions was done by group	T

	members. Other than two small acknowledged functions provided by libMAD, we have full authorship.	
main.c / h LOC: 439	Contains the entry point for execution. Includes definitions for web server and decoder tasks. Sets up RTOS and starts the tasks. Full authorship.	T
player.c / h LOC: 538	Initializes audio devices, sets up SD card file system, and makes calls to parse ID3 data and then play a given track. Contains player state logic for moving between tracks and for pausing playback. Full authorship.	T
trackIndex.c / h LOC: 174	Contains logical helper functions for indexing and sorting an array of parsed Id3 structures. Full authorship.	T
web_api.c / h LOC: 296	Contains request hander functions for client API calls. Full authorship.	T
Total LOC: 3392		

Software – Client Application

File	Description	Status
default.js LOC: 506	Entry point for client-side application. Provides initial API calls to set up the client, as well as event handlers for user actions. Full authorship.	T
nowplaying.js LOC: 265	Contains application logic for handling player status API calls. Populates now playing screen with track metadata and updates control icons based on the player's state. Full authorship.	T
queries.js LOC: 166	Contains functions called when users type data into the search box. Updates application window with relevant search results. Full authorship.	T
thumbnails.js LOC: 451	Contains code to push main navigational elements to the DOM of the web application. Contains state logic for navigational elements like breadcrumbs and the back button. Full authorship.	T
index.html LOC: 93	Contains HTML markup for web application. Full authorship.	T
style.css LOC: 439	Contains information for web application, including layout, margin, text, and image configuration settings. Full authorship.	T
Total LOC: 1920		