VEHICLE TELEMATICS PLATFORM

Leander Perera
B.S., University of Maryland, Germany, 2000

PROJECT

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

SPRING
2012
VEHICLE TELEMATICS PLATFORM

A Project

by

Leander Perera

Approved by:

_____________________________, Committee Chair
Jinsong Ouyang, Ph.D.

_____________________________, Second Reader
Ahmed Salem, Ph.D.

__________________________
Date
Student: Leander Perera

I certify that this student has met the requirements for format contained in the University format manual, and that this Project is suitable for shelving in the Library and credit is to be awarded for the Project.

___________________________, Graduate Coordinator _______________________

Nik Faroughi, Ph.D.    Date
Abstract

of

VEHICLE TELEMATICS PLATFORM

by

Leander Perera

Telematics combines communications and information technology to provide location, mapping, traffic, weather, entertainment and internet connectivity to motor vehicles. Similar to the recent developments in smart phones, the advancement in cellular infrastructure and cutting edge electronics has made the concept of a connected car a reality. The requirements for such systems however, are increasingly complex and costly. Manufacturers can no longer be dependent on proprietary designs to drive market differentiation and deliver successful products. Mission critical systems in motor vehicles require a well-established, tested and proven framework in order to ensure safety and reliability [3].

The advent of many open source software and hardware solutions on the market has enabled the possibility of leveraging off of proven robust solutions that can be applied to automotive designs. One such solution is the choice of real time operating systems. Companies can use Linux distributions optimized for real time performance in order to reach timing requirements for a particular solution. In addition, cross-platform
frameworks like GStreamer, VideoLAN, and GPAC exist for audio and multimedia services within a motor vehicle.

From a hardware perspective, companies like Gumstix offer reference designs of development boards consisting of all peripherals required to prototype solutions. The specifications of these designs are provided so that customized hardware can be designed with any additional interfacing to external peripherals such as the CAN (Controller Area Network) bus and engine control modules added.

This Master’s project report describes my implementation of a vehicle entertainment component for a telematics system using open and proven industry standards. The design and implementation of a prototype is discussed in detail with results and future work discussed for future researchers to further build on the prototype.

_________________________, Committee Chair
Jinsong Ouyang, Ph.D.

_________________________
Date
ACKNOWLEDGEMENTS

I would like to thank my Project Advisor, Dr. Jinsong Ouyang for his insight, guidance and time in helping me complete this project. I am also grateful to my second reader, Dr. Ahmed Salem, for his valuable time reviewing and commenting on my project.

My heartfelt love, gratitude and respect also go to my father Rienzie and mother Celene for providing me with the strong educational foundation, guidance and encouragement to pursue my graduate education. I would also like to thank my brother Lysander for his encouragement during the course of my graduate education.

A very special thank you also goes to my mother-in-law, Christine for her gentle prodding and encouragement during the completion of my project.

I would also like to express my everlasting love and appreciation to my wife Nidhana for her constant encouragement and understanding as I spent hours working on getting the prototype to work and completing the project report. I could never have completed this project without her loving support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 OVERVIEW</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Hardware Components</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Board Setup and Diagnostics</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Software Components</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Software Setup and Configuration</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Basic Operation</td>
<td>10</td>
</tr>
<tr>
<td>3 ARCHITECTURE AND DESIGN</td>
<td>12</td>
</tr>
<tr>
<td>3.1 User Interface Layer</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Communications Layer</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Media Interface Layer</td>
<td>19</td>
</tr>
<tr>
<td>4 IMPLEMENTATION</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Passenger Unit</td>
<td>23</td>
</tr>
</tbody>
</table>
4.2 Head Unit ...........................................................................................................31

5 SETUP AND CONFIGURATION .............................................................................37

5.1 Build Environment Setup ..................................................................................39
5.2 Setting Up the microSDHC Card .......................................................................43
5.3 Setting Up the Development Workspace .........................................................47
5.4 Building the Head Unit and Passenger Unit Components .............................50
5.5 Configuration and Execution ............................................................................51

6 RESULTS AND FUTURE WORK ...........................................................................53

Appendix Accompanying CD ..................................................................................55

References ..................................................................................................................56
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In-vehicle Network</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Network Topology</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Overo EVM Pack from Gumstix</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>CD Player User Interface</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Digital Audio Player User Interface</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>UI Top-level Class Diagram</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Top-level Finite State Machine</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>CD/Digital Audio Payback Finite State Machine</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>Communication Channels</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>Wave File Playback Pipeline</td>
<td>21</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Vehicle telematics bridges information technology with telecommunications to provide services including GPS navigation, in-vehicle entertainment, vehicle tracking, engine control, and Internet connectivity to motor vehicles. The integration with in-vehicle subsystems such as the engine control modules will also enable drivers to retrieve detailed diagnostic and service information regarding the operation of their vehicle [2]. At present basic information including tire pressure, fuel economy and oil life is presented. More detailed information is only available with expensive electronic diagnostics equipment available at automotive repair shops. Integrating rich user interface elements with powerful micro controllers to the vehicle diagnostic functions would enable users to have the information needed to be better informed about repairs needed ahead of time.

The proliferation of communications technologies like WiMAX, and LTE is improving network availability and bandwidth introducing a host of new possibilities to motor vehicles. Some of the features available in today’s smart phones such as streaming audio and video capabilities, navigation and search services can be adapted to motor vehicles as well. Today’s navigation systems deliver content in DVD format which can become out of date rather quickly. Internet connectivity will enable services where users will be able to get download updated maps and directions on the fly. Improved network connectivity
will also enable passengers to listen to internet radio stations and stream movie content from services such as Netflix, and communicate over internet telephony networks.

The master’s project focuses on the implementation of a vehicle telematics platform on which the vehicle entertainment subsystem is implemented as a proof of concept. The vehicle entertainment system consists of a distributed in-vehicle network that is capable of streaming CD audio, MP3, FLAC, OGG, and WAV audio formats to one or more passenger seat-back units. The hardware and software architecture together with the design and implementation of the system is discussed in detail and future enhancements to the system with external interfaces to other platform components such as GPS integration is discussed at the end.
Chapter 2

OVERVIEW

An in-vehicle telematics and infotainment system consists of many separate interconnected components. Figure 1 below shows such a system where a central control unit is connected to each of the different devices and handles all the interactions.

![In-vehicle Network Diagram]

Figure 1 - In-vehicle Network

There can be multiple busses and networks within the vehicle. The CAN network in the above example is a serial bus that connects the vehicle subsystems like the automatic breaking system (ABS) and transmission system to the engine control module (ECM).
The CAN protocol provides a reliable packet based connection that guarantees reliable communication between critical components of the vehicle [6]. An internal Ethernet network is also required for high-bandwidth applications including the transfer of audio and video data. A GPRS modem connects the vehicle to an external 4G wireless network.

The prototype described on this report consists of the head unit (Central Control Unit) with a hard drive for storing digital audio files and a CDROM drive for CD audio content. The head unit streams digital audio to one or more passenger units (the in-dash and seat-back units that deliver personalized audio and control interfaces to each of the passengers. The TCP/IP in-vehicle network connects each of the passenger units to the head unit. Each node on the in-vehicle network establishes connections during initialization.

As shown in Figure 2, the head unit and each of the passenger units are connected to a switch on the network. Communication between each of the nodes is bi-directional. Requests are sent in messages to the head unit while the head unit initiates the transfer of audio data streams to each requesting client.
2.1 Hardware Components

This prototype uses an open reference design from Gumstix, a company specializing in solutions for embedded systems development. A Gumstix Overo Fire module with Chestnut board and 4.3” LCD touchscreen is used for the passenger unit. The Overo module has an ARM Cortex-A8 based CPU with DSP and graphics acceleration capabilities providing ample power for running the user interface, media playback applications and any future applications that may be added. The module has an 8 GB MicroSD card to store OS and program data, USB ports for connecting external peripherals, and 802.11 wireless and wired Ethernet connectors for establishing connections with other devices. The LCD touchscreen has a resolution of 480x272 and is connected to the Chestnut board. The Overo EVM Pack packaged with all required peripherals as shown in Figure 3 below was used in this project.
2.1.1 Board Setup and Diagnostics

The first step before starting development work on the system is to get all the peripherals setup correctly according to the online documentation provided by Gumstix [9]. The Overo Fire module in Figure 3 has two 70 pin connectors that align along the white outline on the Chestnut43 LCD control module. Once correctly aligned, firmly push the Overo module into the Chestnut43 board until the two boards click into place. The ribbon connector on the LCD touchscreen module then connects into a flat plastic connector on the back of the Chestnut43 board. There is a black plastic bar that locks the ribbon cable in place securely. Gently slide the black bar of the connector out and slide the ribbon cable into the connector slot. Once the ribbon cable is in place, push the bar back in to secure the ribbon cable in place.
As the next step, connect the packaged 5V power supply to the power connector on the Chestnut43 board and connect the unit to a wall outlet. The Overo module ships with 512MB of double data rate (DDR) memory and 512MB of NAND flash for program storage. The NAND flash is factory loaded with a boot loader and Angstrom Linux distribution. During startup, the touchscreen will show the boot progress and load a touchscreen calibration utility with step by step directions on calibrating the touchscreen. A desktop environment with tools and utilities is displayed to the user after touchscreen calibration is successfully completed. The user can now use the touchscreen to navigate around the desktop, starting applications and changing system configuration settings. This functionality ensures that the hardware setup is functioning correctly as needed to continue with software setup and configuration described in section 2.2.2.

The Ethernet cable can be connected to the head unit to enable Internet connectivity and streaming content from the head unit to the client. The Gumstix module also has an 802.11 Wi-Fi module that enables wireless communication. In addition, the USB cable serves as a terminal connection to the development system. The Kermit communications program is used to connect to the target device. A Linux console is displayed on the Kermit terminal session allowing configuration and setup from the development system.

As stated earlier, the Overo module, Chestnut43 board and touchscreen make up the passenger units that can be installed in the dashboard and seat backs of a vehicle. The head unit on this prototype is a laptop running the Ubuntu distribution of Linux and all
support files and software needed to serve requests from each of the passenger units. In a production environment, dedicated hardware with customized OS and firmware to run the server-side services will be used in place of the laptop. The laptop also serves as the development system.

2.2 Software Components

As touched on in the previous section, the Linux operating system is used on both the head unit and passenger units due to its open standards and proven industry acceptance. The passenger unit uses the Angstrom distribution of Linux that is used in many embedded devices including the Sharp Zaurus and HP iPAQ. The head unit uses an Ubuntu distribution which runs the server-side hunitapp component that enables streaming of CD audio and digital media files.

The audio playback capabilities requires the ability to decode the CD audio, MP3, WAV, FLAC, and OGG formats and send them to the audio hardware on the platform. GStreamer is a plugin-based framework with codecs capable of performing the encoding and decoding of the audio formats that are used in this prototype [4].

The GUI on the passenger units is built using Qt embedded, a light weight, cross-platform user interface. Qt is an object oriented cross-platform graphical user interface framework used in developing applications [5]. The framework provides classes for buttons, windows, frames and other commonly used graphical elements. The embedded release of the framework provides a light weight implementation of the same
functionality that does not require a fully-fledged graphical user environment such as GNOME or KDE to run on the target platform. The release uses the Linux frame buffer device instead to write to the graphics hardware directly to render user interface elements.

Qt is capable of interfacing with many input devices including the touchscreen used in the prototype.

2.2.2 Software Setup and Configuration

Section 2.1 used the factory supplied flash image to boot up and verify the functionality of the board. The accompanying CD contains the flash image with the passenger unit software components that were developed for this project. It is recommended that the factory NAND flash image be left intact in case we need to boot from it. The microSDHC card is used instead to host the OS and software components for the project.

The microSDHC card needs to be partitioned before first use. Please follow the instructions in section 4.2 to partition the card and write the flash image with the passenger unit software into it. The software for the head unit needs to be installed and configured next according to the instructions provided in section 4.4. Since the head unit and passenger units communicate via TCP/IP, the IP address and port numbers need to be specified in the head unit and passenger unit configuration files as described in section 4.5.
2.3 Basic Operation

Once the passenger unit and head unit software components are in place, the head unit server component can be started by running the hunitapp command from the workspace/hunitapp directory. The head unit then awaits connections from passenger units. The head unit can initiate audio streams to the passenger units based on requests received.

During boot, the passenger unit establishes a connection with the head unit and presents the user interface shown in figure 4 below.

![CD Player User Interface](image)

Figure 4 - CD Player User Interface

The top three buttons in figure 4 select between CD audio, digital audio files, and digital video playback. The video playback feature is beyond the scope of this project, but can be added in the future when needed. The navigation buttons at the bottom allow the user to skip, play, or pause audio tracks. The scroll bar is used to seek within a track.
The digital audio playback interface is selected by clicking on the corresponding button at the top. As shown in Figure 5 below, the digital audio player lists out the available tracks with title and artist.

![Digital Audio Player User Interface](image)

**Figure 5 - Digital Audio Player User Interface**

Similarly to the CD audio interface, the passenger can use the button bar at the bottom to navigate between tracks. The scroll bar on the bottom is used to seek within the selected track. The passenger may also click on the track title to play the corresponding track. Clicking on the CD audio button at the top left corner would suspend the currently playing digital audio file and transition back to the CD audio interface. If a particular CD audio track was playing before the switch, it will resume playback at the position it left off.
Chapter 3

ARCHITECTURE AND DESIGN

The prototype uses a layered design partitioned into three main layers. The user interface layer handles the event handling required to service interactions with the user. The communications layer consists of client and server interfaces that enable communications between the passenger and head units. The third layer is the media interface layer that performs audio encoding/decoding and interfaces with the audio hardware on the passenger unit. This section will discuss the design and architecture of these layers. The implementation will be covered on the next chapter.

3.1 User Interface Layer

The user interface layer consists of the CD player and digital audio player screens that were shown on figures 4 and 5 in section 2.3. The third screen for video playback currently shows a placeholder that can be extended to show video clips in the future. Each of the three screens is implemented in separate C++ classes that contain object references to UI elements including buttons, frames, labels and sliders. Each class also implements state machine behavior and event handling for each user interface.
Figure 6 above shows a UML diagram of the top level class hierarchy. The ContainerUI class is the top level class that contains references to the CDPlayerUI and DAPlayerUI classes. The StreamerGSI and Requestor classes are references shared between both the CDPlayerUI and DAPlayerUI classes.

All behavior in the user interface is driven by state machines. The ContainerUI class has open* functions for each screen. The openCDPlayer function for instance initializes the CD player user interface and starts its state machine. The closeCDPlayer function will suspend the playback of CD audio and hide the CD player user interface. These functions get called during state transitions within the top level state machine. The top-level state machine, shown on Figure 7 below, defines the user interface behavior for selecting between CD audio and digital media files.
Figure 7 - Top-level Finite State Machine

This state machine can be further extended to include digital video playback and internet audio sources among other features. Corresponding classes to drive those features can be added with their own state machines as needed to define their behavior. The transitions shown in the figure are triggered based on user interactions. The CD audio and digital media playback interfaces each have their own state machines which become active when the respective audio format is selected from the user interface. The prototype also adds a provision on the top-level state machine for the future addition of digital video playback.

The CDPlayerUI class contains methods that draw the user interface elements for the CD player and provide state machine behavior for the user interactions with the interface. It adds a button panel at the bottom with player controls and a status display that shows the CD track position and time. The track time and position are updated via calls from the media interface layer. The CDPlayerUI class calls corresponding methods in the
StreamerGSI class to start/stop audio playback in response to user interface actions.

There is also a reference to a Requestor object that is used to send messages to the head unit based on user interface actions. The CDPlayerUI can for instance send a request to the head unit to start streaming out the next CD track if the user touches the next button on the player navigation controls. The state machine has states for pause, play, skip forward, skip backwards and seek.

The DAPlayerUI class provides the user interface for controlling the playback of digital audio files. The class presents the navigation button bar at the bottom similar to the CD audio interface to maintain consistency. A list of available audio files is presented to the passenger together with the navigation buttons at the bottom similarly to the CDPlayerUI class, the DAPlayerUI class maintains references to a StreamerGSI and Requestor instances. The StreamerGSI instance controls the audio stream from the head unit while the Requestor sends messages to the head unit. The Requestor and StreamerGSI classes will be covered in more detail in the corresponding sections for the Communications and Media Interface layers.

The CD and digital audio file playback state machines shown in Figure 8 below have the same states even though the events on the state machine are triggered by different user interface elements.
On the CD audio state machine, the pause, play, skip and seek events are triggered by the respective buttons on the button bar and slider above the button bar. The behavior for the button bar and seek slider remain the same for the digital audio file playback interface to maintain consistency with the CD audio interface.

Each state machine begins at the pause state as illustrated in Figure 8. The transition to each of the other states is achieved via interactions with the user interface. When the user selects a different audio source by switching from CD audio to digital audio file
playback, the state of the current state machine is saved before switching over to the audio file playback state machine. The same functionality exists when switching back from audio file playback to CD audio playback. This allows the user to resume playback of a previously selected audio source after switching back to the corresponding state machine.

Each interaction with the user interface results in an event. These events are known as signals in Qt framework terminology [5]. A signal is associated with a slot which is a function implemented to handle the event. In the case of the play button click for instance, a clicked event is generated. The clicked event for the button is tied to a state transition to either the play or pause state based on which state the state machine is in as shown on Figure 8 above.

3.2 Communications Layer

A service-based architecture based on the Jini middleware technology was considered [1]. However, the technology was not used in this project due to its dependence on Java which makes it difficult to integrate with the functionality provided by the media and user interface layers. A custom solution using the boost ASIO libraries were used instead as it provided the functionality that was needed.

Communication between the head unit and each of the passenger units is defined in the communications layer. A session between the head unit and passenger unit consists of the establishment of a message channel and an audio channel. The message channel is
bidirectional and is used to send messages between the passenger and head units. The passenger unit may use the message channel to send requests to the head unit and the head unit may use it to send status information and stream information to the passenger unit. The audio channel is a one way channel that is used to send audio streams from the head unit to the passenger unit in order to fulfill the requests. Separate sessions are maintained so that each passenger can have a separate personalized audio stream to listen to via headphones. The communication channels between the head unit and passenger unit is shown on Figure 9 below.

![Communication Channels](image)

**Figure 9 - Communication Channels**

There are currently two types of requests that can be sent by passenger units to the head unit via the message channel. A source change request specifies which audio source (CD audio or digital audio files) the head unit needs to use for the session with the passenger unit. Once the source has been selected, the passenger unit sends an item request that specifies which audio track or file needs to be served by the head unit. Once the item request is received, the head unit begins decoding audio content from either an audio CD or digital audio file, encodes it into the FLAC audio format to be transmitted via TCP/IP and initiates a stream to the passenger unit via the audio channel. The FLAC audio format
is a lossless format which preserves the audio quality, but compresses the data so it can be transmitted over the network. The passenger unit then decodes the received stream and sends it to the audio hardware for playback. Encoding to FLAC is only needed for CD audio and WAV audio streams in order to compress the data. MP3 and OGG streams are sent directly to the passenger unit for decode without compression since they are already compressed.

The passenger unit and head unit each have client and listener interfaces. The client interface on the passenger unit enables sending messages to the head unit. The passenger unit also has a listener interface that will accept incoming audio streams from the head unit. The head unit’s listener interface in turn accepts requests from the passenger unit, processes them and initiates a connection to the passenger unit in order to send the audio stream.

The communications layer consists of two classes. The Requestor class enables the passenger unit to send messages to the head unit via the message channel. The Listener class provides the functionality to handle the messages sent by passenger units.

3.3 Media Interface Layer

The media interface layer uses the GStreamer framework to setup audio pipelines that are needed to decode, encode, and playback audio streams. A pipeline is a concept used by GStreamer to indicate an end-to-end path from audio source to destination [4]. The media interface layer exists on both the head unit and the passenger units. Each of these
modules carries out audio processing and creates separate pipelines for each supported audio format.

A pipeline consists of elements, bins, pads and can also contain other pipelines. An element carries out reading, processing, or writing data. The data can be read from a file, or another element. The processing within the element can include adding various effects or decoding a particular file format. The processed data can then be written to a file or sent to another element. A bin contains many elements that can be dynamically put together based on the input to form a pipeline. A pad is contained within an element and can be a source pad that receives data or a sink pad that sends data.

At the head unit, the decoding of CD audio, MP3, WAV, FLAC and OGG contains logic to construct separate pipelines. For each pipeline, the decoded content is post-processed, encoded and sent via TCP/IP to the passenger unit. The passenger unit has its own decoder pipeline which will decode and post-process the audio as needed and send the data to the audio hardware. Figure 8 below illustrates how a WAV audio stream is sent from the head unit to a passenger unit.
Figure 10 - Wave File Playback Pipeline

The pipeline above each has five elements, file source and TCP server source being two of these elements. Within the head unit, the file source element provides an interface into a file where the contents of an audio file such as a WAV file can be read. The WAV Parser then extracts the audio samples from the wave file and passes it down into the audio converter which performs any required format conversions such as changes in channel count and sample rate. The FLAC encoder then encodes the stream into the lossless FLAC format before passing it down to the TCP client sink, which sends the audio stream via TCP/IP.

On the passenger unit, a TCP server source receives the audio data over the network. The decode bin element would then decode the input stream to raw audio samples and passes it through the audio converter for any format conversions. Any required equalization can then be applied using the ten band graphic equalizer element before sending it to the
ALSA (Advanced Linux Sound Architecture) driver which sends the audio to the hardware.

The media interface layer consists of three classes that interface with GStreamer. The CDPlayerGSI and DAPlayerGSI classes are used by the head unit to decode CD audio and digital audio files and encode them into formats that can be sent through the network. The StreamerGSI class as indicated earlier is used by the passenger unit to receive, decode and send the audio data to the audio driver for playback.
Chapter 4

IMPLEMENTATION

The previous chapter provided insight into how the various components of the project fit together. Details of the interactions between each component were also discussed. This chapter drills into more details on the implementation of each of the components.

The implementation of the project is done using C++. All of the main components described in chapter 3 are implemented in C++ classes. The head unit and passenger units are separate software applications that are compiled, linked and deployed separately.

4.1 Passenger Unit

The passenger unit application uses the Qt framework for the graphical user interface. The first step in the initialization is to instantiate a QApplication object as shown below:

```cpp
QApplication a(argc, argv);
```

The QApplication object is a class in the Qt framework that initializes the main event loop in the graphical user interface enabling application control flow and event handling. The object needs to be created before painting any other GUI components. The `argc` and `argv` arguments are passed from the main function to the application object to handle special command line processing for the Qt GUI. In a Unix/Linux environment for instance, the –display switch can be specified in the passenger unit application command line to direct which X display is to be used to present the graphical user interface.
The passenger unit has its own configuration file, punitapp.cfg as shown in the listing below. This file specifies the hostnames/IP addresses and port numbers of the head unit and listener on the passenger unit that awaits audio streams from the head unit as described in section 3.2.

```
# Passenger Unit Configuration File
# Hostname/IP address of head unit
hunit_address = localhost
# Control port used by head unit to receive messages
control_port = 3000
# Listening port to receive audio streams sent by head unit
audio_port = 4000
# Listening host interface (host interface to bind the
# audio_port to)
audio_host = localhost
```

The `hunit_address` setting in the punitapp.cfg file specifies the IP address or host name of the head unit. The `control_port` parameter specifies the TCP/IP port used to setup the message channel between the head unit and the passenger unit as described in section 3.2.

The `audio_port` and `audio_host` parameters are used by the audio listener interface on the passenger unit awaiting audio streams sent by the head unit.

The punitapp.cfg configuration file is read as illustrated in the listing below right after the QApplication object is created.

```cpp
// Read and parse punitapp.cfg configuration file
PunitConfig config;
try {
```
config.readConfig("punitapp.cfg");
}
catch(FileNotFoundException ex) {
    std::cout << "ERROR: Exception occurred - " << ex.what() << endl;
    exit(1);
}

// Get each of the properties from the punitapp.cfg file
std::string hunitAddress = config.getHunitAddress();
int controlPort = config.getControlPort();
std::string audioHost = config.getAudioHost();
int audioPort = config.getAudioPort();

A PunitConfig object is created and the readConfig method is called to read the configuration settings from the configuration file. The PunitConfig class is implemented in PunitConfig.hpp and provides the functionality to parse the puintapp.cfg file. The class uses the boost program_options library internally to parse each of the settings out and extract the configuration parameters. The values are returned through the getHunitAddress, getControlPort, getAudioHost, and getAudioPort member functions of the PunitConfig class as shown in the listing above.

The readConfig member of the PunitConfig class may throw a FileNotFoundException which is caught in the exception handler try/catch block shown above. In this event, the application would provide the user with an error message and gracefully exit. This situation is never expected to occur in a production environment as the puintapp.cfg file should have been deployed along with the other components on the passenger unit.

After the configuration file is successfully read, the passenger unit can use the parameters to initialize the remaining components and bring up the user interface. The first step in
this sequence is to create the StreamerGSI GStreamer interface that will listen in for audio streams delivered by the head unit.

```c
StreamerGSI streamer;
streamer.setAudioHost(audioHost, audioPort);
```

The `setAudioHost` method is used to specify the `audio_port` and `audio_host` parameters from the punitapp.cfg configuration file. The StreamerGSI object initiates an audio listener on the specified host and port to receive audio streams from the head unit.

The Requestor class is then instantiated to establish the message channel between the passenger unit and head unit. The IP address or hostname of the head unit and control port number are specified as the parameters. The Requestor is used by the passenger unit application to send requests to the head unit.

```c
Requestor requestor(hunitAddress, controlPort);
```

The `hunit_address` and `control_port` parameters from the punitapp.cfg file are passed into the constructor of the Requestor object. These parameters are used to setup the message channel.

The third step involves creating the top-level container user interface implemented within the ContainerUI class. The ContainerUI class contains the CD player (CDPlayerUI class) and digital audio player (DAPlayerUI) class references that implement the user interfaces for the CD and digital audio player respectively. The StreamerGSI and Requestor objects
are passed in as parameters to the constructor of the ContainerUI class. The following listing illustrates how the top-level container user interface is created.

```cpp
ContainerUI container(NULL, &streamer, &requestor);
container.setGeometry(0, 0, 480, 272);
container.show();
```

The `streamer` and `requestor` parameters to the constructor specify the StreamerGSI and Requestor object instances. The `NULL` parameter specifies that the container user interface does not have a parent window. The `setGeometry` method restricts the size of the container window to the resolution of the LCD display. The resolution on the LCD display is 480 x 272 pixels. The `show` method then renders the entire user interface.

The passenger unit application finally calls the `exec` member of the QApplication class to start the event loop that handles all messages and events from the user interface and the media interface layers. Event handlers in the ContainerUI, StreamerGSI and Requestor handle the processing of the events directing behavior of the application.

The constructors of the CDPlayerUI and DAPlayerUI initialize all private members and call the `initPlayerControls` and `initStateMachine` members to initialize the player graphical user interface elements and state machines that drive the behavior of these interfaces as described in section 3.1. The CDPlayerUI also calls the `initStatusDisplay` member to initialize the CD track number and duration display that gets updated every
second. The DAPlayerUI calls the `initMediaList` method to show the track titles and artist information on the status display.

The `initStateMachine` member of both user interfaces initializes and defines the state machine behavior. The state machine and separate states are defined for pause, play, skip and seek as shown below.

```cpp
QState *pauseST = new QState();
QState *playST = new QState();
QState *skipFWDST = new QState();
QState *skipBWDST = new QState();
QState *seekST = new QState();
```

All classes instantiated above are part of the Qt framework. The QStateMachine class requires the parent to be passed in as a parameter. The `this` pointer passed in refers to the CDPlayerUI or DAPlayerUI class instance that defines the state machine. The transition from one state to another is specified as shown in the following code snippet.

```cpp
// Transition from pause to play state when play/pause button is clicked
pauseST->addTransition(m_btnPlayPause, SIGNAL(clicked()), playST);
```

This statement specifies that a transition from the pause state to the play state is defined when the button identified by `m_btnPlayPause` is clicked. The `m_btnPlayPause` is a reference to a custom QPixButton object that is derived from QPushButton that is part of the Qt framework. The QPixButton adds support for an image that can be displayed in place of text. The following code fragments define further state transitions.
// Transition from play to pause state when play/pause button is clicked
playST->addTransition(m_btnPlayPause, SIGNAL(clicked()), pauseST);

// Transition from play state to "skip forward" state when skip forward button is clicked
playST->addTransition(m_btnSkipFWD, SIGNAL(pressed()), skipFWDST);
skipFWDST->addTransition(m_btnSkipFWD, SIGNAL(released()), playST);

// Transition from play state to "skip backward" state when skip backward button is clicked
playST->addTransition(m_btnSkipBWD, SIGNAL(pressed()), skipBWDST);
skipBWDST->addTransition(m_btnSkipBWD, SIGNAL(released()), playST);

The events that need to be called need to be defined once the state transitions have been specified. Qt uses signals for events that are triggered by different actions. Slots are functions that get called as a result of a signal that occurred. The events for the state transitions are defined as follows.

// Connect all state transition signals to actions(slots) when player buttons are set
connect(playST, SIGNAL(entered()), this, SLOT(playRequested()));
connect(pauseST, SIGNAL(entered()), this, SLOT(pauseRequested()));
connect(skipFWDST, SIGNAL(entered()), this,
     SLOT(skipFWDRequested()));
connect(skipBWDST, SIGNAL(entered()), this,
     SLOT(skipBWDRequested()));

This associates that the play state transition will trigger the playRequested function to be called. Similarly, the pause state transition will trigger the pauseRequested function to be called. The this pointer passed in indicates that the parent class generates the signals. In this case a signal is generated when the state machine transitions to a particular state and the associated function callback is invoked. The callback function that gets invoked is known as a slot in Qt terminology.
Once the event handlers are specified, each state can be added to the state machine as follows.

```cpp
machine->addState(pauseST);
machine->addState(playST);
machine->addState(skipFWDST);
machine->addState(skipBWDST);
machine->addState(seekST);
```

The initial state is set to the pause state since no audio should be playing. Clicking on the play/pause button will begin playback. State machine transitions are started by calling the start method in the QStateMachine instance.

All event generation and event handling begin after the state machine is running and the application’s main event thread has started by a call to the QApplication start member function as described earlier.

The Requestor and StreamerGSI are both used to control the playback of audio streams. The StreamerGSI needs to be in the playing state in order to receive an audio stream from the head unit and begin playing back the audio. The StreamerGSI provides play, pause and reset methods to control the stream states. Once the StreamerGSI is in the play state, the Requestor can be used to request for a particular source or media file index. To illustrate, the following sequence is used to request a CD audio track within the CDPlayerUI class.

```cpp
m_streamerGSI->reset();
m_trackCount = m_requestor->changeSourceToCD();
m_streamerGSI->reset();
m_requestor->getMediaIndex(m_currentTrack);
```
The reset methods need to be called on the StreamerGSI class as shown to reset the connection state and have the StreamerGSI class listen in for new connections. In the above example, the `changeSourceToCD` member function on the Requestor causes the head unit to switch over to CD audio and start streaming the first audio track. The second reset then re-initializes the connection state and the `getMediaIndex(…)` method then requests the track that was requested. This will cause the StreamerGSI class to start streaming the new content.

The Requestor class also has a `changeSourceToDA` to change the audio source to digital audio file playback. In this case, the `getMediaIndex` method requests the digital audio file similar to how a CD audio track is requested.

4.2 Head Unit
The head unit application runs in a console displaying incoming connection information and metadata for the current audio stream. During startup, the head unit application reads its configuration settings from `hunitapp.cfg` which contains the passenger unit IP address or hostname, control port and audio port to send audio streams to. Listed below is the `hunitapp.cfg` file for illustration.

```yaml
# # Head Unit Configuration #
# # Hostname / IP address of passenger unit
punit_address = 172.16.0.2
```
# Control port used to receive control messages from passenger unit
control_port = 3000

# Port to send audio streams sent to passenger unit
audio_port = 4000

Parsing and reading in the configuration settings is carried out by the HunitConfig class.

The class extracts each of the above parameters and provides the `getPunitAddress`, `getControlPort`, and `getAudioPort` methods so that the client can use these settings. The following code snippet illustrates how the HunitConfig class is used to extract the settings needed for the application.

```cpp
// Read configuration from hunitapp.cfg
HunitConfig config;
try {
    config.readConfig("hunitapp.cfg");
} catch (FileNotFoundException ex) {
    std::cout << "ERROR: Exception occurred - " << ex.what() << std::endl;
    exit(1);
}

string punitAddress = config.getPunitAddress();
unsigned int controlPort = config.getControlPort();
unsigned int audioPort = config.getAudioPort();
```

The code is very similar to the passenger unit application, except that the head unit uses the HunitConfig class that implements logic specific to configuration parameters required for the head unit. The exception handling code ensures that the hunitapp.cfg settings are correctly read before continuing. In the event of an error reading the configuration settings, the application displays the error message and reason for the error and gracefully exits.
After the configuration settings have been successfully read, the head unit initiates the message channel by instantiating a Listener object as follows:

```cpp
Listener listener(punitAddress, audioPort, controlPort);
```

Each of the parameters to the constructor was read from the hunitapp.cfg file. The listener uses the `punitAddress` and `audioPort` to initiate audio streams to the passenger unit. The `controlPort` parameter specifies the port that the listener will listen in on for requests. A call to the `start` method in the `Listener` class then starts the listener loop. The `Listener` will then await requests from passenger units and serve the requested audio content.

Most of the communication, audio processing and streaming are carried out within the `Listener`. The Boost ASIO library is used for the communication between the head unit and the passenger unit. The ASIO library provides features for marshaling and un-marshalizing structures and sending the data across the network.

The listener receives the source change messages from the passenger units requesting that the audio source be changed. In the event of a CD audio request, the head unit would send a source change message with `SOURCE_CD_PLAYER` as the message type. If the digital audio player was playing, it is paused by a call to the DAPlayerGSI pause method as illustrated below:

```cpp
// Stop any currently playing digital audio audio (if any)
if (currentSource == SOURCE_DAPLAYER)
    m_daPlayerGSI.pause();

currentSource = SOURCE_CDPLAYER;
    m_cdPlayerGSI.play();
```
The *currentSource* variable helps keep track of the currently playing media source. After the operation is complete, the head unit sends a response message back to the passenger unit.

The first step in handling any request is for the head unit to listen in on a socket and accept any incoming connections. An acceptor object is created in the constructor of the listener as illustrated below.

```
m_pAcceptor = new boost::asio::ip::tcp::acceptor(m_Service,  
    boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(),  
    controlPort));
```

The acceptor takes three parameters. The first parameter, *m_service*, is of type `boost::asio::io_service` which provides core functionality for asynchronous I/O services. The TCP version and port (*controlPort* parameter) that is used to establish the message channel are passed as well.

The main server loop is then started. The first step within the server loop is to accept a connection from a passenger unit. This is done by creating a socket using the I/O service that was created. The serialized message structure then needs to be read from the socket and deserialized in order to interpret it. The following code fragments illustrate accepting a connection and reading the serialized message structure from the passenger unit.
// Accept a connection from a client
boost::asio::ip::tcp::socket socket(m_Service);
m_pAcceptor->accept(socket);

// Read data from socket
boost::system::error_code error;
boost::array<char, MAX_MSG_LENGTH> serMessage;
rwsize = socket.read_some(Boost::asio::buffer(serMessage), error);
serMessage.at(rwsize) = '\0';

The serialized message structure is given a maximum length of MAX_MSG_LENGTH.

This maximum length is adequate to store the serialized sequence of bytes for the
serialized message. The serialized message is NULL terminated after it has been read.

This message now needs to be de-serialized as illustrated in the following code fragment:

// De-serialize received data
std::istringstream archiveInputStream(serMessage.c_array());
boost::archive::text_iarchive iarchive(archiveInputStream);
iarchive >> message;

The serialized string is treaded as a stream of characters. A istringstream object is created
with the serialized array and a text_archive object is created using the input string stream.
The >> operator on the text_iarchive will unmarshal the string into a message object that
it can be interpreted in the subsequent steps.

After the message is handled, a response is sent back to the passenger unit. The response
structure is populated as follows and marshaled before sending to the passenger unit.

// Create the response structure
CDSourceResponse response;
response.trackCount = m_cdPlayerGSI.getTrackCount();

// Serialize the response before writing to the socket
A string can be generated from the marshaled data represented in the text_oarchive object by calling the \texttt{str()} method. The write method can then be used to transmit the marshaled string to the passenger unit as illustrated below.

```cpp
std::ostringstream archiveOutputStream;
boost::archive::text_oarchive oarchive(archiveOutputStream);
oarchive << response;
```

```cpp
// Get string representation of text_oarchive object
serResponse = archiveOutputStream.str();
```

```cpp
// Write the response to the socket
boost::asio::write(socket, boost::asio::buffer(serResponse),
                    boost::asio::transfer_all());
```

The \texttt{transfer\_all()} function indicates that the write operation should continue until all data has been transferred or until an error is encountered.
Chapter 5

SETUP AND CONFIGURATION

The GNU C/C++ compiler is used to build the source code for the head unit and passenger unit components. Since the passenger unit hardware is based on the ARM architecture, all code for the passenger unit needs to be cross-compiled for that architecture. A cross-compilation tool chain with all dependencies needs to be built. The cross-compilation tool chain is then used to build all frameworks and libraries and their dependencies along with the GUI and media playback application stack running on the passenger unit. This is a complex manual process where the sources for all dependencies need to be correctly configured and built for the target architecture. A means to automate this process using a build tool was needed in order to make the build process more consistent.

OpenEmbedded is a cross-platform build system capable of cross-compiling code for multiple architectures [8]. The build system uses bitbake as the build tool which is similar to tools like make, but uses files called recipes that can specify package dependencies including those that are found in version control systems such as SVN, CVS and git. The tool can download source revisions from these version control systems and compile the sources to build a complete Linux distribution and applications on a file system that can be written to disk and booted from. Subsequent compilations are faster since the tool
keeps track of which components need to be rebuilt and rebuilds only those that are needed.

The OpenEmbedded build tool was used to cross-compile the Linux kernel, utilities, GStreamer, Qt Embedded and all dependencies that were needed for the passenger unit software to be built. Three OpenEmbedded recipes needed to be written, one for building the passenger unit modules and the other to build the custom init process which spawns the passenger unit app during startup. The third recipe builds calls the other two recipes to build the passenger unit app and init process, then runs the packing process to build a complete file system with the entire passenger unit software stack in place.

All source code for the project is checked into a SVN repository. The recipe files for the project then pull the source code for the project directly from the SVN repository and then start the build process. The recipe files also contain steps to build a Linux ext3 file system image to be written to the micro secure digital card. The file system image includes all the passenger unit components and dependencies together with the Qt libraries and GStreamer codecs and libraries. The Linux file system image is then used to boot the Gumstix overo module. The USB console connection is used to monitor the boot process to debug any issues and setup the module.

The steps for setting up the build environment are described in the next few sections.
5.1 Build Environment Setup

The first step in the build process is to setup the OpenEmbedded build system correctly on the development system. The build file system requires about 4GB of working space on the development system since OpenEmbedded builds the Linux kernel and various packages including GStreamer, the Qt runtime environment, utilities and tools that need to run on the target platform. A separate 4GB partition was used on the development system to keep the build system separate from the rest. The partition was mounted at /media/hdd and will be referenced during the steps described below.

1.) Create the /media/hdd/overo-oe subdirectory within the mounted partition.

   mkdir /media/hdd/overo-oe

2.) The bitbake build tool needs to be installed within the overo-oe directory. Ensure that you are in the /media/hdd/overo-oe directory. Download and install the bitbake tool as follows:

   cd /media/hdd/overo-oe
   wget http://download.berlios.de/bitbake/bitbake-1.8.18.tar.gz
   tar -xvzf bitbake-1.8.18.tar.gz
   mv bitbake-1.8.18 bitbake
   rm bitbake-1.8.18.tar.gz

3.) The next step is to download and install the OpenEmbedded files on the development. Follow the steps below to checkout the mainline of OpenEmbedded. The OpenEmbedded repository will be cloned to the org.openembedded.dev directory.
git clone git://gitorious.org/gumstix-oe/mainline.git
org.openembedded.dev

4.) Change to the org.openembedded.dev directory and set the repository to point to the overo branch:

```bash
cd org.openembedded.dev
git checkout --track -b overo origin/overo
```

5.) The configuration files from the OpenEmbedded distribution now needs to be copied over to the /media/hdd/overo-oe directory.

```bash
cd /media/hdd/overo-oe
cp org.openembedded.dev/contrib./gumstix/build .
```

6.) The environment variables in the /media/hdd/overo-oe/build/profile file needs to be set before running bitbake builds. This can be achieved by appending its contents to the bash profile (~/.bashrc).

```bash
cp ~/.bashrc ~/.bashrc.bak
cat /media/hdd/overo-oe >> ~/.bashrc
```

7.) The configuration and setup can be tested by building a console image to be run on the Gumstix Overo as follows:

```
bitbake omap3-console-image
```

Step 7 described above creates a usable console only image that can be used to ensure that the hardware and software configuration is correctly setup. The next step will use this image to boot the Gumstix Overo module with peripherals to ensure proper functionality of the hardware.
Once the image has been written to the microSDHC card, make sure that the following connections are made between the development system and Gumstix overo board (the target):

- USB connection from development system connected to “Console” port on Gumstix Overo module.
- Ethernet cable connecting development system to Gumstix Overo module. This is required for later steps.
- Power adapter connected to Gumstix Overo module. The Gumstix Overo module can also be powered by a USB connection from the development system.

The Kermit program is used to establish a terminal connection to the target (Gumstix Overo module) and monitor the boot process. It is recommended that the terminal connection be established prior to powering up the target. The USB serial port on the development system is identified by /dev/ttyUSB0. The following command line is used to connect to the target device:

```
kermit -l /dev/ttyUSB0
```

The following Kermit commands are then entered at the Kermit command line to establish the console connection.

```
C-Kermit>set flow-control none
C-Kermit>set carrier-watch off
C-Kermit>set speed 115200 /dev/ttyUSB0, 115200 bps
C-Kermit>connect
```
Supply power to the Gumstix Overo module using a USB connection or a power adapter once the Kermit connection has been established. Console messages will be displayed on the Kermit screen as the board boots. A login screen will be displayed on the touchscreen after the device completes the boot process. This login screen cannot be used without a USB keyboard attached to the USB port on the target. The Gumstix documentation also states that a compatible USB hub needs to be connected in order to get keyboard and mouse input to work. These peripherals were not used in this project since the USB serial connection using Kermit worked fine.

Building the Qt embedded meta toolchain can be carried out once the console boot image can be used successfully to boot the target. The Qt embedded meta toolchain is an SDK with libraries including directFB, GStreamer, tslib, and glib are needed to link into cross-compiled code to be run on the ARM based Gumstix Overo module. This step also requires some dependencies that were missing on the target system. These dependencies are installed using the following commands.

```
sudo apt-get install libtiff-tools
sudo apt-get install libtiff-dev
sudo apt-get install libmng-dev
sudo apt-get install libdbus-1-dev
sudo aptitude install xorg-dev
```

Now that the dependencies are in place, the Qt embedded SDK can be built using bitbake as follows:

```
bitbake meta-toolchain-qte
```
The Qt embedded SDK can be found in the /media/hdd/overo-oe/tmp/deploy/glibc/sdk directory. The filename of the package created was angstrom-2010.7-test-20100925-i686-linux-armv7a-linux-gnueabi-toolchain-qte-4.6.3.tar.bz2 on the development system.

```bash
sudo tar -C / -xjf angstrom-2010.7-test-20100925-i686-linux-armv7a-linux-gnueabi-toolchain-qte-4.6.3.tar.bz2
```

The above command creates a /usr/local/angstrom/arm directory on the development machine. The OpenEmbedded tools can now reference the SDK during the build process as needed.

5.2 Setting Up the microSDHC Card

The Gumstix Overo, Chestnut board and Touchscreen needs to be assembled together as described in the documentation. Section 2.1.1 discussed using the factory provided NAND flash image to boot the board for the first time. Alternatively, testing the hardware can be carried out by writing the console image created in the previous section and following the directions below. The reader may alternatively choose to install one of the pre-built images found on the Gumstix support site to ensure proper hardware functionality.

1.) Use the microSDHC card reader or adapter provided and attach it to the development system.
2.) The microSDHC card needs to be partitioned for use the first time. Two partitions need to be created on the card: the first partition is a FAT partition that will contain the boot loader while the second partition will contain the ext3 file system with Linux kernel and applications. The microSDHC card is detected as /dev/mmcblk0 on the development system. Partitioning is carried out as follows:

```
# sudo fdisk /dev/mmcblk0
```

Use the o option to create the FAT partition. Next, have a look at the partition table using the p option. Note the card size returned in bytes. For the 8GB card used on this project, the number of bytes came out to be 7,973,371,904. The number of bytes is needed in the next step to calculate the number of cylinders in the disk geometry.

3.) Enter expert mode in fdisk by entering x at the fdisk prompt. Next specify the number of heads as 255 and the number of sectors to 63.

4.) The number of cylinders is calculated using the number of bytes reported in step 2 as follows:

```
floor(bytes reported in step 2 / heads / sectors / sector_size)
```

Which works out to be:

```
floor(7,973,371,904 / 255 / 63 / 512) = 969
```

5.) Enter the number of cylinders calculated in step 4 above (969) by using fdisk option c.
6.) A 32MB primary partition now needs to be created. Enter r to return to fdisk’s main mode. Then enter n to create a new partition and select p for primary partition. Select partition number as 1. The first cylinder for the partition should be specified as 1 followed by the size as +32M. This will create a 32 MB primary partition.

7.) The partition type now needs to be changed to FAT32. Type t at the fdisk command prompt to set the partition type. Enter c as the hex code to set the partition type to FAT32.

8.) Partition 1 now needs to be marked as bootable. Enter a at the fdisk prompt and enter 1 as the partition. This will make the FAT32 partition as the active boot partition.

9.) A second partition can now be created with the remaining space to host the ext3 file system. Specify n at the fdisk prompt to create a new partition. Select p as the next command to specify the creation of another primary partition. Specify the partition number as 2. Next, specify the first cylinder of the partition as 6. Specify the last cylinder of the partition as 969.

10.) Use the p command at the fdisk prompt to print out the partition table. The following information is displayed for the 8GB microSDHC card used in this project.

Disk /dev/mmcblk0: 7973 MB, 7973371904 bytes
255 heads, 63 sectors/track, 969 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x3995f398

<table>
<thead>
<tr>
<th>Device</th>
<th>Boot</th>
<th>Start</th>
<th>End</th>
<th>Blocks</th>
<th>Id</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/mmcblk0p1</td>
<td>*</td>
<td>1</td>
<td>5</td>
<td>40131</td>
<td>c</td>
<td>W95 FAT32 (LBA)</td>
</tr>
<tr>
<td>/dev/mmcblk0p2</td>
<td>6</td>
<td>969</td>
<td>7743330</td>
<td>83</td>
<td>Linux</td>
<td></td>
</tr>
</tbody>
</table>

11.) Write the partition table into the microSDHC card by typing `w` at the fdisk prompt.

12.) Now that the partitions are created, create the FAT and ext3 filesystems on the first and second partitions using the following commands:

```
sudo mkfs.vfat -F 32 /dev/mmcblk0p1 -n FAT
sudo mkfs.ext3 /dev/mmcblk0p2
```

13.) There are three boot files that now need to be copied over to the FAT partition created above. This step needs to be done only once since the contents of the FAT partition do not change during subsequent builds of the project. The MLO boot loader file needs to be copied over first due to some constraints on the mask boot ROM in the OMAP processor [9].

```
sudo mount /dev/mmcblk0p1 /media/card
sudo cp MLO-overo /media/card/MLO
sudo cp u-boot.bin /media/card/u-boot.bin
sudo cp uImage.bin /media/card/uImage
sudo umount /dev/mmcblk0p1
```

14.) A root filesystem can now be written to the ext3 partition as follows. The root filesystem image from section 4.1 step 7 can be written to the ext3 partition.

```
sudo mount /dev/mmcblk0p2 /media/card
cd /media/card
sudo tar -xvaf /path/to/some-image.tar.bz2
sudo umount /dev/mmcblk0p2
```
15.) In order to ensure that the u-boot environment variables are refreshed, boot into the u-boot prompt during system boot by hitting enter at the console and entering the following commands:

```
nand erase 240000 20000
reset
```

The boot process can now be monitored as described in section 4.1 using the USB serial connection to debug any issues with the boot process.

5.3 Setting Up the Development Workspace

The development workspace is a directory from where the source code for the passenger units and head unit gets built. This directory is off the user’s home directory. The SVN repository can be found on the accompanying CD for the project. The repository needs to be accessible using an SVN URL like `svn://127.0.0.1/…` for flexibility. Changing the SVN repository location would then require that the IP address/hostname portion of the URL to be updated. A file:/// specification could have been used with the absolute pathname, but this would require that the entire pathname be hardcoded within the recipes used for the project.

Making the SVN repository accessible via a SVN URL requires that the SVN server run on the development system. This can be accomplished using the following command line:

```
svnserve -d -r ~/svnroot
```
The above command starts the SVN daemon on the development machine using the SVN repository located in the user’s home directory.

The following steps can be used to setup the workspace once the SVN repository is available via SVN URL:

1.) Create the workspace directory within the user’s home directory:

   mkdir ~/workspace

2.) The source code in the SVN repository contains directories for branches, test, and trunk. The branches directory is for any development that needs to branch off the mainline. The source code in the mainline can be checked out using the following command:

   svn co svn://127.0.0.1/trunk mproject

   This command checks out the sources in the trunk to the mproject directory. The directory hierarchy of the project is as follows:

   ┌───coms
   │   ├───doc
   │   │   └───exceptions
   │   │       └───gsint
   │   │           └───histlog
   │   │               └───hunitapp
   │   │                   └───punitapp
   │   │                           └───backgrounds
   │   │                               └───images
   │   └───utils

   The coms directory contains the communication layer classes including the listener and requestor. The code in this directory is common to both the passenger unit and head unit.
The doc directory contains project design notes. The exceptions directory has custom exception classes that are needed for error recovery. At present, a `FileNotFoundException` class is defined for handling an instance where a required configuration file is missing. The gsint directory contains all the files that are needed for the GStreamer interface. These include the CDPlayerGSI, DAPlayerGSI and StreamerGSI classes described in the design and implementation chapter. The histlog directory contains a history logger that logs debug information for debug builds of the application. The hunitapp directory contains code for the head unit software components while the punitapp directory contains the code for the passenger unit software components. Finally, the utils directory directory contains header files containing custom data types that are common to both the head unit and passenger unit applications.

The OpenEmbedded recipe files for the project needs to be checked out into a subdirectory within the /media/hdd/overo-oe directory. This can be accomplished as follows:

```
cd /media/hdd
svn co svn://127.0.0.1/oe/trunk user.location
```

This will check out the three recipes required to build the passenger unit, filesystem image, and custom init process into the following hierarchy.

This will enable cross compiling the passenger unit components and building the filesystem to be written to the microSDHC card.
5.4 Building the Head Unit and Passenger Unit Components

Both the head unit and passenger unit components can be built to run on the x86 development machine. This is ideal for development and testing since all components can be tested on the x86 platform prior to cross-compilation using the OpenEmbedded build system.

The punitapp contains a build.sh shell script that calls the Qt qmake tool that auto-generates the makefile based on the specifications on the punitapp.pro project file. The project file specifies all the source files needed to build the project. The file also contains any defines, dependent library references and packaging directives needed for building the components. Type the following commands to build the passenger unit components.

```
# cd ~/workspace/punitapp
# ./build.sh
# make
```

This will build an executable called punitapp which is the passenger unit application that can playback content streamed from the head unit.

The hunitapp contains a Makefile that can be used to generate the components that run on the head unit. The following commands can be used to build these components.

```
# cd ~/workspace/hunitapp
# make
```

This will build an executable called hunitapp which is the head unit application that can serve requests from the passenger units.
The passenger unit components can be cross-compiled to run on the ARM architecture using the OpenEmbedded build environment as follows:

```
# cd /media/hdd/overo-oe
# bitbake punitapp-image
```

This will cross-compile the passenger unit application and packages it in a filesystem archive filename is `punitapp-image-overo.tar.bz2` and can be found in the `/media/hdd/overo-oe/tmp/deploy/glibc/images/overo` directory after the build is completed. This archive can be written to the microSDHC card as described in section 3.2.

### 5.5 Configuration and Execution

Both the passenger unit and head unit have two configuration files, `punitapp.cfg` and `hunitapp.cfg` that needs to be setup prior to execution. The `punitapp.cfg` configuration file has settings for the passenger unit and is shown below:

```
# # Passenger Unit Configuration
#
#
# Hostname/IP address of head unit
hunit_address = 172.168.0.1

# Control port used by head unit to receive messages
control_port = 3000

# Listening port to receive audio streams sent by head unit
audio_port = 4000

# Listening host interface
#   (host interface to bind the audio_port to)
audio_host = 172.168.0.2
```
In the above example configuration file, the passenger unit is using IP address 172.168.0.2 while the head unit is using 172.168.0.1. The hunit_address specifies the IP address that the head unit is using to receive requests, while the audio_host specifies the IP on which the passenger unit is listening in on for audio streams.

The hunitapp.cfg configuration file specifies settings that are to be used on the head unit.

Listed below is an example configuration file:

```
# Head Unit Configuration
#
# Hostname / IP address of passenger unit
punit_address = 172.16.0.2
#
# Control port used to receive control messages from passenger unit
control_port = 3000
#
# Port to send audio streams sent to passenger unit
audio_port = 4000
```

In the above example, the punit_address setting specifies the IP address of the passenger unit so that audio streams can be sent.

The localhost can be used for test purposes where the passenger unit and head unit both execute on the development machine.
Chapter 6

RESULTS AND FUTURE WORK

The prototype designed for the master’s project is currently able to playback CD audio, WAV, MP3, OGG, and FLAC file formats successfully. CD audio and WAV require very high bandwidth requirements to stream the audio through the network. Encoding these formats into the lossless FLAC format made it possible to compress the audio data and sending it over the network without losing any quality.

All of the audio processing including decoding, encoding, and processing is carried out by GStreamer plugins. The workloads carried out by these plugins can be very CPU intensive. The ARM processors have dedicated hardware for carrying out a lot of these operations including encoding and decoding of audio and performing DSP functions thus freeing the CPU for other tasks. These require the plugins to be modified to offload CPU bound workloads to the dedicated hardware in order to have them use less CPU time.

Texas Instruments currently runs the TI DaVinci project that aims at writing versions of plugins optimized to run on their OMAP platforms [7].

These optimized modules were not integrated into the project due to some unresolved compilation and packaging errors encountered while using the OpenEmbedded build system. Integrating these modules may work with later releases of the OpenEmbedded build tools.
The Linux OS and Open Source software components used provides flexibility, reliability and support for vendors to customize and differentiate their products. Similarly, the open hardware specifications used enables manufacturers to extend the reference design by adding additional hardware capabilities such as interfacing with the vehicle CAN bus to be able to retrieve information from other vehicle subsystems such as the engine control module (ECM).

The proposed solution can be extended to enhance security within the vehicle network. This is especially important as the system becomes capable of running other third party applications. Marcario and Torchiano proposes such a solution using the Google Android framework where each application is segregated and provided with clearly defined interfaces for communication with other applications and peripherals.

As the coverage provided by 4G network infrastructure such as WiMAX and LTE become widely supported, more usage models for vehicle connectivity will improve providing an echo system for more in-vehicle applications such as on-demand video and telephony.
APPENDIX

Accompanying CD

The accompanying CDROM contains all the source code and SVN repository for the project. The project source code is the latest revision that was checked out of the SVN repository for convenience. The SVN repository can be used to look at the history and evolution of the project over time.

All source files for the project is contained within the mproject subdirectory. The oe directory contains the OpenEmbedded recipes for passenger unit application. The svnroot directory hosts the SVN repository for the project.
REFERENCES


