

VICTORIA UNIVERSITY OF WELLINGTON

Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Archiving New Zealand Digital Games From 80's and 90's

Cao Yue (Daniel)

Supervisors: Dr. Ian Welch

Submitted in Partial fulfilment of the requirements for
Bachelor of Information Technology.

Abstract

Increasingly people are moving from producing physical artifacts to digital. Digital objects are being produced and updated faster than ever before; however, every time when they are replaced, the valuable knowledge behind them is lost as well. One possible solution is trying to archive digital objects and conserve them for study and research purposes. During the last ten years, many studies have been conducted on the problems of digital archiving and many countries have started experimenting with digital archiving, yet, there is still a gap in archiving software and games. Therefore, this project is devoted to find possible ways of archiving digital games and trying to build a solid foundation for future research on this subject. The potential scope is large so this project focuses upon developing a tool to archive games developed for the Sega SC3000 platform that was a popular computer in New Zealand during the 1980s.

Table of Contents

1 Introduction	1
1.1 Motivation	1
1.2 Research Areas	2
1.3 Report Summary	2
1.4 Report Outline	3
2 Background	5
2.1 Digital Archiving	5
2.1.1 Digital Archiving in General	5
2.1.2 General Requirements for Digital Archiving	6
2.1.3 Digital Archiving in New Zealand	6
2.2 Emulation	7
2.2.1 Needs for an Emulator	7
2.2.2 Game Emulators for SC3000	7
2.2.3 Emulators in Digital Archiving	8
2.3 Metadata	8
2.3.1 Why Need Metadata?	8
2.3.2 Metadata Standards	9
3 Sound Processing	11
3.1 The WAV Format	11
3.2 Input Format for an Emulator	13
3.3 The Tape Format	14
4 Design	15
4.1 Methodology	15
4.2 User Stories	16
4.2.1 Archiving Directly from a Tape	17
4.2.2 Archiving a WAV File	17
4.2.3 Updating an Archived Game with Metadata	18
4.2.4 Updating the Stored Program	18
4.2.5 Extracting Information from the Archive	18

4.3 Metadata.....	19
4.3.1 Archive Data.....	19
4.3.2 Tape Data.....	22
4.3.3 Original WAV Data.....	23
4.3.4 Source Code Data	23
4.3.5 Attachment Data	24
5 Implementation	25
5.1 Class Structure.....	25
5.1.1 System Integrator.....	25
5.1.1.1 Graph Plotter.....	25
5.1.1.2 Program Analyzer	25
5.1.2 Program Converter.....	31
5.1.2.1 Char Converter	31
5.1.2.2 Program Decoder.....	31
5.1.3 XML Generator	33
5.1.4 System Input Interface.....	34
5.1.4.1 Main Input Interface.....	34
5.1.4.2 Subfiles Input Interface.....	36
5.2 File Structure.....	37
6 Testing and System Evaluation	39
6.1 The Test Code Generator	39
6.2 System Evaluation	41
6.2.1 Archiving Directly from a Tape.....	41
6.2.2 Archiving a WAV File.....	42
6.2.3 Updating an Archived Game with Metadata	42
6.2.4 Updating the Stored Program	43
6.2.5 Extracting Information from the Archive	43
7 Conclusions	45
7.1 Contributions.....	46
7.2 Future Work.....	46
8 Bibliography.....	49
9 Appendices	51

9.1 Sound Format	51
9.2 XML Schema	55

List of Tables

<i>Number</i>	<i>Page</i>
Table 1 – Internet Archive Metadata Structure	10
Table 2 – An Example of a BASIC Program Stored in the Input Format.....	32
Table 3 – An Example of a Temporary Vector	40

List of Figures

<i>Number</i>	<i>Page</i>
Figure 1 – Conversions from the Original Program to the Final WAV File.....	11
Figure 2 – WAV File Structure.....	12
Figure 3 – Listing of Archive Data Structure.....	20
Figure 4 – Listing of Tape Data Structure	22
Figure 5 – Listing of Original WAV Data Structure	23
Figure 6 – Listing of Source Code Data Structure.....	24
Figure 7 – Listing of Attachment Data Structure	24
Figure 8 – Decoding Algorithm.....	28
Figure 9 – DFD for Program Analyzer	30
Figure 10 – Main Input Interface	34
Figure 11 – Accessory Input Interface	35
Figure 12 – Tape Input Interface.....	36
Figure 13 – File Structure.....	37
Figure 14 – The SC3000 Manual	51
Figure 15 – The Tape Format	52
Figure 16 – The Input Format for SC3000	53
Figure 17 – The Input Format for SC3000 (Continued).....	54
Figure 18 – The Archive Data Structure	55
Figure 19 – The Tape Data Structure	56
Figure 20 – The Original WAV Data Structure	57
Figure 21 – The Source Code Data Structure.....	58
Figure 22 – The Attachment Data Structure	59

1 Introduction

1.1 Motivation

There was a rapid uptake of digital games in New Zealand in the 1980s because of New Zealand's technology enthusiastic environment. Those games soon became one of the main ways in which people were exposed to the then new technologies [1]. The game industry had rapidly become a fast developing area and many overseas companies had chosen to manufacture in New Zealand. These factories produced a wide range of products from general game consoles to specialised arcade machines [2].

Study has shown that, although some games initially appear to be 'clones' of other well-known games, their code is unique from any other known game. This suggests that although some locally produced games may have borrowed visual design and gameplay elements from other well-known games, they were in fact written locally from scratch [2]. Therefore, games that were available on the market during that period were specifically designed to suit New Zealand players and thus contained distinct New Zealand content. This unique digital cultural heritage fostered a remarkable amount of user communities, which later became the cradle of New Zealand's very own software industry [1].

However, as time goes by, that 'Golden' period has been forgotten by majority of people. The distinct digital heritage is facing the risk of been lost. Some still existing games are only kept by a few private collectors. These games are quite dated for the current hardware technology and are impossible to be played directly on modern computers. Therefore, there is an urgent need of collecting those games together and find an efficient way of archiving them, so they can be conserved and widely distributed for the benefit of the community and study and research purpose in the future.

However the scope of archiving digital games for different consoles is potentially very large, the software used for developing those games various to a great extend. Therefore, this project only focused on archiving the SC3000 games. The project is designed to investigate the possibility of archiving those early digital games, define the requirements that are needed for archiving digital objects, build a well-designed data structure for storing all the related information, and construct a solid foundation for any further research by developing a basic digital archiving system.

1.2 Research Areas

For this project, we define three major research areas. They are: conversion of games from their original format, metadata structure and a tool for the digital archiving.

Conversion of games from their original format is essential for the success of this project. Those early games were originally saved on tapes or cartridges in an analogue format. When those files are digitized, they are saved as Waveform Audio Format (WAV) which is a commonly used sound format. There are three steps in this conversion process. First, binary data should be read from the file, broken into bytes and converted into a series of digital values. This result is the Pulse Code Modulation (PCM) representation of the sampled program. Then these values should be converted to either 1s or 0s according to the amplitude each of them is representing. The second step results a series of 1s and 0s, which will be fed into a data forming process that can help correcting any errors that are introduced by differences in the sampling rate used to create the original analogue representation and the WAV file. The final well formatted result will be then converted into a human-readable format, which would normally be some simple BASIC code.

Metadata structure is another major part in this project. Since the project is called ‘Archiving New Zealand Games’, there is a need for developing a generic metadata structure that can be used for storing general information for a digital object. As we expect to read information about a book in a library before we actually decide to borrow it, the digital archiving system should allow all users to read all the related information about a digital object, such as its name, released data and a brief description etc., before they decide to access that digital object. The metadata structure is designed exactly for this purpose. It is implemented with XML and the details will be discussed later in this document.

Digital archiving system tool is believed to be the backbone of a reliable archiving process. Designing and implementing the digital archiving system tool is considered as one of the major parts of this project. The system that has been developed will be referred as the SC3000 recording system throughout this report. The details of the system design and the class structure will be discussed later in the design and implementation chapter.

1.3 Report Summary

The purpose of the project is to design, develop and implement a tool for archiving BASIC games developed for the SC3000 platform. The tool allows the original game to be migrated to modern platform for archiving purpose by parsing the original game that is stored as a digitized sound file together with the metadata of that game. The tool is designed with a focus on its flexibility, so that the future it can be modified with minimum effort for archiving games that are developed for other platforms.

Additionally, provenance the origin of the archived digital object is important for historians who wish to check the authenticity of the digital object.

1.4 Report Outline

This report will start with an introduction of the modern digital archiving in Chapter 2. It will provide a discussion of general requirements for digital archiving, followed by digital archiving in New Zealand. The chapter also concentrates on discussions of metadata structures for capturing essential preservation information about the project. Chapter 3 introduces the signal processing concepts involved in this project. The concepts include the WAV file format, the input format for SC3000 emulator and the tape format. Chapter 4 is one of the major parts in this report. It describes the development methodology used in this project, followed by five major user stories that were used as guidelines for the system development. The metadata structure developed in this project will be described in detail at the end of this chapter. Chapter 5 concentrates on the implementation of the SC3000 recording system, its class structure and file structure. Chapter 6 introduces the test code generator that is developed in this project, and the final system is also evaluated. Conclusions are given at the end of this report.

2 Background

2.1 Digital Archiving

2.1.1 Digital Archiving in General

Digital archiving means using some reliable technologies to conserve digitalized data for future study and research. In other words, it is a process of ensuring that a digital object or a digitalized object can be accessed over a long period of time [3]. Today massive digital objects are being created everyday. People emphasize heavily on the speed and ease of short-term dissemination of digital objects, but are concerned little about the long-term preservation of that digital information [4].

Digital information, however, is very different from the information that is traditionally recorded on paper or films. Digital information can be very system-dependent, and easily corrupted or altered without recognition [3]. Because of the short natural life of machines, the storage media that can be used for storing digital information have a much shorter life time than the traditional technologies. Digital information also needs to be reviewed regularly, and the accessing technology that it relies on needs to be updated from time to time, in order to keep that digital information always accessible by the latest technology [5].

But on the other hand, digital information also brings many advantages that the traditional archiving method could not ever achieve. It allows for fast searching speed with more accurate results. This is not only for digitalized books, but also pictures, audios and other digital objects. It also allows people to have access to the information whenever and wherever.

Generally, a digital archive file can be classified into one of the following types [5]:

- Electronic databases (Access databases, Oracle databases, Excel spreadsheets etc.)
- Electronic word processed documents
- Digitalized audios and videos (mp3, mpeg etc)
- Software and programs
- And digital images and photographic records (jpeg, bmp etc)

2.1.2 General Requirements for Digital Archiving

Firstly, intellectual property remains as a key issue in digital archiving and the approaches to intellectual property rights differ from country to country [4]. Therefore, before any digital archiving project can be carried out, the project team has to familiarize themselves with the local digital legislation, and resolve intellectual property issues related that project. For instance, in this project the work is carried under the research and study provisions of the New Zealand copyright Act.

Secondly, digital archiving needs a special identification scheme. Once a digital object has been archived, it is essential to assign it an easily recognizable identification [6], so that it can be well kept and managed over a long period of time. In the case of the National Library of New Zealand (NLNZ) which will be introduced in the next section, their first step was to identify a new metadata schema for electronic document.

Thirdly, one should strive to find a storage media that is reliable and can last as long as possible. This is the most crucial part of digital archiving, since a digital storage failure could result losing all the information that is stored in that media. However, even a good storage system cannot last for ever, due to its fragile nature and fast evolving technology. It is believed that storage systems should be updated at least once every five years. [4]

Finally, digital archiving should also be accessible wherever, and whenever. High accessibility is one of the major reasons for digital archiving [3]. People want to enjoy the freedom of accessing useful information without any time or geographic constrains.

2.1.3 Digital Archiving in New Zealand

Until recently the importance of digital archiving has not been recognized in New Zealand. It was partly because people were not aware of the importance and advantages of archiving digital objects and there was no huge demand on digital archiving. Also there was a lack of knowledge on digital archiving, even if some people had thought about the idea of archiving digital objects, they didn't know how they could implement the idea [3].

The National Library of New Zealand (NLNZ) has been charged by the New Zealand government with implementing digital archiving of New Zealand content. Aware of the advantages and efficiency that digital archiving could bring, pressured by the rapid developing information technologies, the National Library of New Zealand decided to launch its first digital archiving project [7]. In November 2002, NLNZ released its first version of a preservation metadata schema, which explained the data fields needed to support the preservation of digital objects in detail, and formulated the basic framework for a database repository and input systems for collecting and storing preservation metadata. It is believed to the foundation for future digital archiving.

The project was finished by 2004, and its implementation turned out to be very successful. It now has 16 metadata records providing access to the all sorts of digital information. In fact, the project had be awarded by The New Zealand government with \$24 million in funding for capturing the digital output of the nation to archive for current and future generations. The next step is implementation of a secure digital archive.

2.2 Emulation

2.2.1 Needs for an Emulator

The project is about archiving New Zealand games from the 80s and 90s. However, as technology is developing at a very rapid speed, all of the old consoles that were used to run those games had be eliminated long ago by the market and replaced by more sophisticated and better developed consoles. Thus, even if one still keeps some original game software from the 80s, one can hardly find the hardware to run it on. This is a serious problem for the project. However, it can be solved by using a console emulator. An emulator is a piece of software that people can use to perform emulations of the hardware used by the original system. It is designed to virtually simulate the original architecture of an arcade or computer system board, and uses the copied read-only memory (ROM) or disk images from the original system to run software [8]. In this case, they are those old console games.

Using an emulator also provides other advantages. Most of time, a programmer that is writing an emulator would design some additional features which the original system did not have due to the limitation of technology back in the old days. Those features can enhance the virtual system of the emulator and deliver a better performance than the original system [8].

2.2.2 Game Emulators for SC3000

The most popular use of emulators is to imitate the experience of running arcade games or console games on personal computers [8]. It pretends to be the hardware for those games and allows you to run the copied ROM images from these original games on your computer. Also it is believed that emulating those old games on a modern computer is much more convenient than running on the original machines.

There are many emulators on the market for different types of machines. For example, Virtual GameBoy Advance (VGBA) allows people to play games for Nintendo's GameBoy Advance on a personal computer; whereas Multiple Arcade Machine Emulator (MAME) can give you the experience you can only get from a 'coin up system' [9]. Most emulators can emulate a number of games, or even a number of similar consoles.

As a very famous console with limited distributions from mid 1980s, SC3000 is a popular target for all the emulator enthusiasts [10]. The most well-known emulators for it are Meka and MasterGear. They are both multi-console emulators in an attempt to mimic all the 8bit Sega Game consoles with a single virtual system [11]. Compared with MasterGear, Meka is open source software which allows for free download. It publishes its code occasionally to allow people to contribute to its development. However, Meka is not as portable as MasterGear, which can be run on many different platforms [11]. Moreover, MasterGear does not only run games, it also allows people to debug Sega software without using costly and cumbersome systems.

Many emulators are able to load games that were originally stored on cartridges; however, few can load games that were stored on tapes.

2.2.3 Emulators in Digital Archiving

The purpose for archiving earlier video games is to endeavor to conserve the dated but valuable digital content for further study and research. However, it won't have any use if only the games are archived. People also need a way to access those games, play with them and learn those games from interactions with the system. As mentioned before, it is not feasible to keep a dated cumbersome machine for running those games, since metal electronic circuits are very fragile and they eventually will be ruined by rust as time goes. Therefore, an emulator which can emulate the original console wherever there is a modern computer, is needed to run the games and be archived together with the games. An emulator doesn't need to worry about the time and whenever the computer technology is upgraded only the emulator needs to be updated instead of redesigning an actual game console.

Apart from its reliability, an emulator also has high portability, which is another goal of the digital archiving: 'making the archiving content easily accessible to the wide public' [1]. A specially designed emulator, for example, can be loaded onto a pocket sized PDA, or even be accessed and distributed across the Internet, allowing far more flexibility and portability than a real console could ever allow.

2.3 Metadata

2.3.1 Why Need Metadata?

When archiving digital games, we need not only the right format for recording games, but also a metadata structure that can be used for storing useful information about game objects. These contents can be stored in various formats, such as ASCII or .doc. However,

XML is considered to be the best option for storing metadata, since it is designed primarily for representing structured documents and data.

Currently, there are many organizations around the world that are experimenting with archiving digital data. Some formal archive procedures have been established specifically for digital data and the required metadata structure has also been developed. This project, therefore, takes advantage of this by carefully studying the metadata structure that is available and redesigning that structure to suit its special needs.

2.3.2 Metadata Standards

The Internet Archive is one of those organizations that are currently experimenting with archiving digital data. It was founded in 1996 in San Francisco. Their primary goal is to build an ‘Internet library’ that can offer permanent access for researchers, historians, and scholars to historical collections that exist in digital format [12]. Their archive objects include texts, audio, moving images, software and archived web pages. The most attractive feature of the Internet Archive is software archive collection. It is designed to preserve historical software and provide access to all kinds of background information on the software. The collection includes a broad range of software related materials including shareware, freeware, speed runs of actual software game play and even previews and promos for software games [12]. Each software is stored in a zip file together with its metadata information and files such as images of documentation or original box covers.

The Internet Archive has developed a fairly mature metadata structure for their archive. Based on their metadata structure, the modified structure for this project is shown in the following table.

Name	Description
Identifier	A file name (a combination of numbers and characters) across the entire archive that can be used for identifying the file uniquely and allow fast retrieving of that file.
Title	The original title for the archive object. For texts, it can be the title of the document; for software, it is the name of that software.
Creator	The name of the company/ organization/ individual that created the object.
Media type	The type of object. For example, an archived game has the type of “software” whereas an archived song has the type of “audio”.
Collector	This field is specially used for rare, dated software that is collected by enthusiasts only for historical reasons. The copyright for the software probably couldn’t be legally established anymore and the ownership therefore becomes more important.
Publisher	The publisher of the object.

Language	The language that is used. Some object may contain different versions for different languages.
Medium	The medium that is used for storing the original object. For example, for certain software, it could be stored on a 3.5inch floppy disk or a compact disk.
Extent	The number of disks that are used for storing the entire object.
System name	The name of the system that the program object is originally executed on. Please note that, this field is only used for software objects.
System model	The model of the system that program object is originally executed on. For example, SC3000. Please note that, this field is only used for software objects.
Disk availability	Indicates if the original disk is available.
Update date	The date that the object is added or updated.
Updater	The last person that updated this object.

Table 1 – Internet Archive Metadata Structure

Apart from this required information, some other information may need to be also identified and structured by using XML. For example, it is useful to list all the additional objects of the archived game, such as box covers and/or documents in an additional table. The table should include the file name and the file type, so that users can see clearly what other files are included in the object file.

The metadata structure used by the Internet Archive is believed to be currently the best practice. It is the most efficient way of sorting object information and provides a clear structure of representing the information when it is needed. This project, therefore, adopts the general structure of the metadata used by the Internet Archive when constructing the archive data structure and adds extra fields peculiar to the type of games we wish to archive.

3 Sound Processing

SC3000 encapsulates its programs in a format specifically designed for SC3000 consoles. This format will be referred as input format throughout this paper. After the program is encapsulated, the file will be transferred to the tape representation and saved on the tape. When those files are digitalized, they are saved as WAV format which is a commonly used sound format. The above procedure can be illustrated in the following picture.

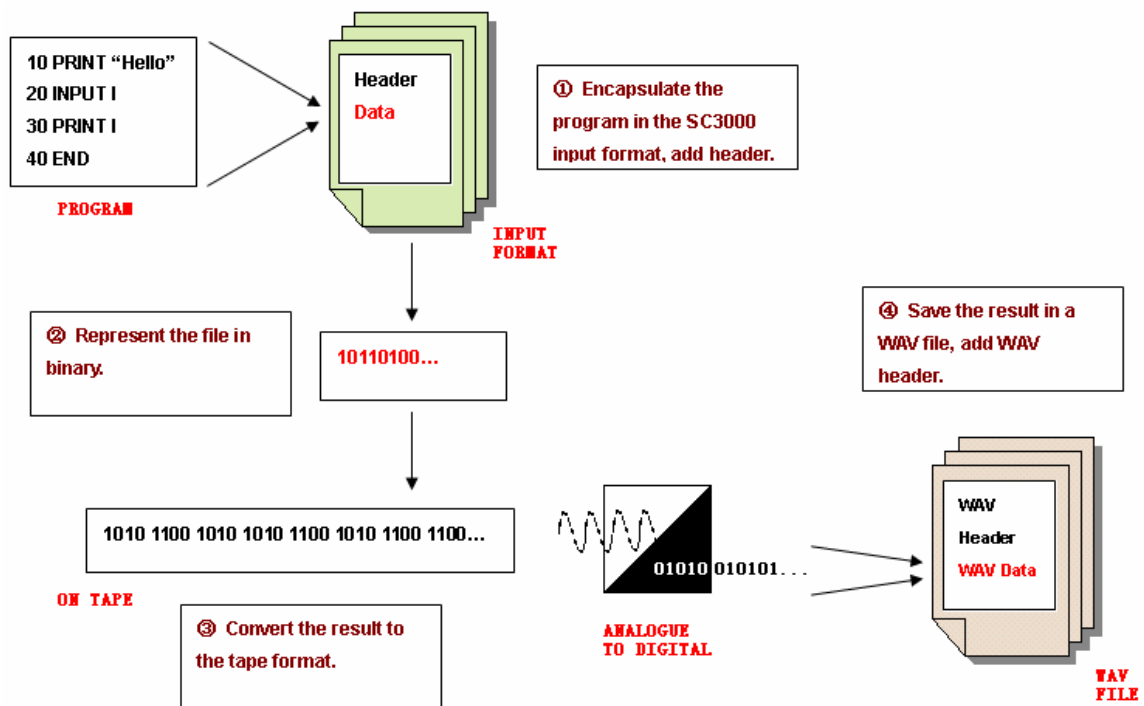


Figure 1 – Conversions from the Original Program to the Final WAV File

In this chapter, the WAV format will be introduced first, followed by the input format for the SC3000 emulator. The conversion to the tape format is relatively straight forward; therefore, it will be discussed at the end of this chapter.

3.1 The WAV Format

WAV (or WAVE), short for Waveform audio format, is a file format developed by Microsoft and IBM for storing audio on PCs. The format was originally developed on Windows 95 operating system [13], but due to the popularity of Windows it quickly became a standard audio format [14]. It is probably the simplest format for storing audio samples since it doesn't require any pre-processing other than just formatting of the raw

data. The file always ends with .wav file extension. The following diagram shows the general structure of a WAV file.

The Canonical WAVE file format

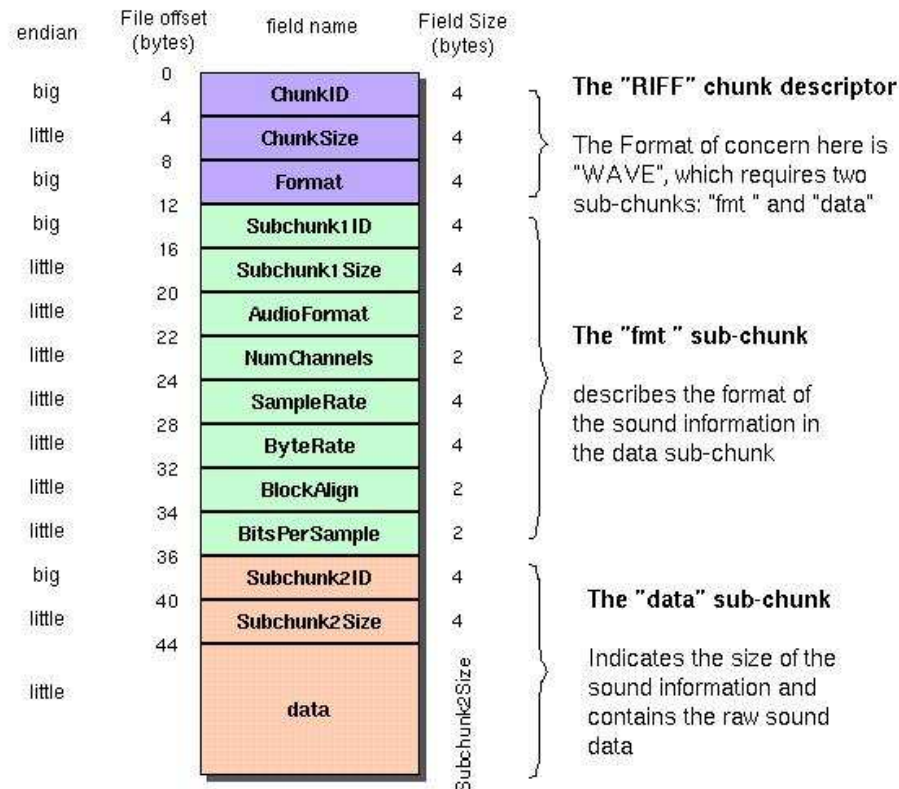


Figure 2 – WAV File Structure (M. Boldin, *WAV File Format*)

The WAV file format is a subset of Microsoft's RIFF specification for the storage of multimedia files, which is a structure of grouping the files contents into a number of separate chunks with each containing its own header and date bytes. This structure allows programs that do not use or recognize a particular type of chunks to easily skip them and continue processing following the known chunks [13].

A common WAV file is often just a RIFF file with a single 'WAVE' chunk which consists of two sub-chunks: an 'fmt' chunk and a "data" chunk.

The "fmt" chunk is usually used for describing the format of the audio information in data chunk. Firstly it contains a sub-chunk ID for identifying itself and a sub-chunk size for informing the program how long the sub-chunk will be. It then contains two bytes for audio format, which specify if there is any compression algorithm used, followed by 2

bytes for indicating number of channels, with 1 for mono and 2 for stereo. Then there are 4 bytes for specifying sample rate and 4 bytes for specifying the byte rate. After that, there are two bytes used for block alignment and another two bytes for describing bits per sample. On the other hand, the “data” chunk is much simpler. It just has the sub-chunk ID and the sub-chunk size as the 'fmt' chunk does, followed by the raw data. The only tricky thing is, if the original sound has two channels, the left sound channel and the right sound channel will be sampled alternately.

Because the WAV format keeps the data as close as possible to the original sound signal, professional users often use it for maximum audio quality. Also for some study or research purposes, WAV audio can be edited or manipulated with relative ease using special software.

3.2 Input Format for an Emulator

Before a WAV file can be examined, however, it needs to be converted into a file with respect to the input format for SC3000 emulator.

The input format for SC3000 emulator is relatively complex.

For a header, at the very beginning, it uses 450 bytes of 1s for synchronization, followed by one start byte which actually identifies the start of the header. After that, it has 16 bytes in ASCII format that describes the program’s name [15], followed by two bytes for describing the program length and another two bytes for identifying the start address for the program. After another byte of parity, the header is finished with some dummy codes.

The data format is quite similar for the actual program field, but is a bit simpler. Again it starts with 450 bytes for synchronization and one start byte, but then what follows them are just pure program codes. When the program codes are finished, they are closed with one parity byte and some dummy codes.

In order to go further with the input format analysis, we need to read from a WAV file bit by bit, extract the data from that file and compare it with the input format that has been described above. Unfortunately, there are quite few Java libraries that I can use for audio processing, particularly for reading and writing from WAV files. However, there are Java libraries available for some codecs, such as Free Lossless Audio Codes (FLAC) [16]. In computer terminology, a codec is usually a program that can perform encoding and decoding on stream of data or signals. It encodes data for transmission or storage, and decodes it for viewing or editing [17]. In this project, some methods from the existing FLAC library were used in the system development stage for reading from WAV files.

The more detailed structure of the format will be illustrated in the appendices.

3.3 The Tape Format

When the encapsulated program is converted to the binary presentation, it becomes a string of 1s and 0s. If we save this string directly onto a tape, a problem can arise when we try to read a bunch of 0s or 1s. Because a typical tape player has no bit counter designed for it, when a bunch of continuous 1s or 0s is read from a tape, all the tape player knows is that there are many 1s, or 0s, but it is impossible for it to determine exactly how many bits of 1s or 0s that has been read.

To solve this problem, binary data is saved onto a tape in the tape format, which represents a 1 as '1010' and a 0 as '1100'. In this way, the tape player always reads a series of continuously changing signals, so that the number of bits that are read can be easily determined.

In order to recognize the bit pattern, a program can be designed to read four bits at a time and compare the read bits with the tape format and convert them to 1 or 0 accordingly. This technique has a drawback. It can only recognize two bit patterns, if a bit is corrupted, it is highly likely that the whole data string would not be recognized. For example, assume there is a binary string '1100 1010 1100 1010' that can be recognized as '0101', however, if the first bit is lost, the string would become '1001 0101 1001 010' which doesn't make sense at all. Therefore, it is better to find a recognizable pattern in the input string first, and then start reading from the recognized bit.

4 Design

This project has gone through a complete software development cycle from the user requirements gathering to the analysis phase, followed by design and implementation of the system and finally ending with the system testing.

Prior to the user requirements gathering, however, there was a need of identifying a suitable software development methodology as a guideline for the project, so that the project could be carried out smoothly. The required methodology was to support a short development time and evolving requirements.

This chapter will discuss the chosen software development methodology at the beginning. Five major user stories identified in this project will then be introduced, which give a very good overview of the system design. Finally, the design for metadata structure will be discussed at the end of this chapter.

4.1 Methodology

Extreme Programming (XP) was selected to be the methodology for the system development, as the Extreme Programming targets for small teams of developers who need to develop software quickly in a changing environment [18]. Extreme Programming also stresses client satisfaction, which is indeed a major focus for this project. The methodology is designed to deliver the software to the client needs when it is needed. Therefore, by implementing this methodology, a simple prototype of the system can be developed easily at the beginning of the project. More user requirements can be then added and developed on the original prototype. This feature also empowers developers to confidently respond to changing client requirements at any stage of the project, even late in the life cycle [18].

The core practices of Extreme Programming that have been adopted for this project can be grouped into three areas (9 practices) as follows [18]:

Fine scale feedback

- Test driven development
- Planning game

Continuous process

- Continuous Integration
- Design Improvement
- Small Releases

Shared understanding

- Simple design
- System metaphor

Some of the aspects of XP have been changed slightly to address the nature of this project. As there is no real client in this project, my supervisor is playing the on-site customer role that is continuously monitoring the system development and refining the system requirements when it is needed.

For the planning game, one or more story cards were completed during each weekly project meeting. A story card either refines the existing user requirements or adds new user requirements. Then the time that is needed for implementing the user requirements was estimated based on these stories. The planning games, the same as the weekly project meetings, were carried out at least once a week during the system development.

As Extreme Programming is a test driven development methodology, a test case will be completed each time prior to the further system development. In this project, testing was carried out every time after the implementation of a major part of the system to ensure that the implemented feature works correctly, and also continuous integration was carried out to make sure that the newly implemented unit could perform as expected when it was added to the existing system.

The system has at least one small release each week, as we focus on implementing just a small requirement each time. In addition, the project design has been improved continuously. After every release of the program, the system performance was assessed by the on-site customer according to the earlier story card. System design was modified if it did not satisfy user's requirements, or when the customer wanted to change the previous system requirement, or when the additional features were required by the customer.

There are five major user stories identified for this project, which will be explained in the next section.

4.2 User Stories

According to the XP methodology, short user stories that address all the major user requirements are the key to the success of this project. After careful deliberation, there are five major user stories with each focusing on a different functionality of the complete system. The stories are: archiving directly from a tape, converting from a WAV file, updating an archived game with metadata, updating the stored program and extracting information from an archive.

4.2.1 Archiving Directly from a Tape

A SC3000 game collector wants to copy one of his games from the original tape and store it on a modern computer. He first connects the output of a SC3000 tape player to the input of a computer, which has the SC3000 recording system activated. He then selects the ‘Record from the tape’ option on the SC3000 recorder together with the desired sample rate, and starts to play the tape. The signal output of the tape will then be sampled according to the chosen sample rate, and converted directly into WAV format for storing on the computer. It also keeps an archive file, which is usually a text file.

Comments:

There are two types of storage devices for the original SC3000 games: cartridges and tapes. For games stored on a cartridge, the read-only memory (ROM) image of that cartridge can be copied directly onto a computer as a .sc file, which is the executable format for SC3000 emulators. However, for games stored on tapes, as they are only saved as an audio representation of the program broken down into binary, they cannot be converted into .sc files directly. In order to better preserve the original program, we need not only a WAV representation of the original audio file, but also a text file just for the program and its relevant metadata. One concern for saving the program as a text file is that as different systems have different binary formats for line-endings, a binary file would be distorted if it is transmitted from its original system to another system which has a totally different format for line-endings. UUencoding is considered to be a better approach because it has no line-endings, thus the binary data stays in the same format no matter what system it is stored on.

4.2.2 Archiving a WAV File

Some SC3000 games have already been recorded into WAV files. The collector of these games wants to archive these files. He activates the SC3000 recording system on his computer and chooses the ‘Archive from WAV file’ option. The recorder will prompt the collector for the name of the input file. The collector enters the file name and clicks on ‘Start’. The SC3000 recording system then starts to analyze the game program, extracts information and creates the XML files for storing that data. The recorder might ask the user to manually enter some of the information needed for the metadata schema.

Comments:

Ideally, users should be allowed to add any type of data to the archive object according to the metadata structure that is defined. In other words, users should be able to, for example, add Joint Photographic Experts Group (JPEG) objects that record the box covers, or Tagged Image File Format (TIFF) objects if they want higher resolution images. They should also be able to add documentations for the games in various formats, such as PDF

Portable Document Format (PDF), or Portable Network Graphics (PNG) which might be a better option as it is not proprietary.

4.2.3 Updating an Archived Game with Metadata

A SC3000 game collector later finds out that some of his information stored for a game is actually inaccurate. He wants to change the metadata stored for that game. He activates the SC3000 recording system and selects the 'Update game information' option. The recorder will prompt the collector for the name of the file he wants to modify. The collector then enters the file name and clicks on 'Open'. All the metadata of that object is then presented to the user. The user makes his change and clicks on "Save". After the confirmation, the new data is saved and overwrites the old data.

Comments:

A user-friendly interface for presenting the metadata is essential.

4.2.4 Updating the Stored Program

A new record of a SC3000 game has just been made, which is sampled at a higher frequency with better quality or has been through signal filters to remove noise. The game collector wants to replace his old record by this new one. He activates the SC3000 recording system and selects the 'Update game record' option. The recorder will ask the collector to locate the new record he wants to store. The collector locates the new record. The recorder will then prompt the collector for the name of the file he wants to modify. The collector then enters the file name. Before overwriting the game record, the recorder checks with the collector if he wants to overwrite the existing record. The collector chooses 'Yes'. After the confirmation, the old record is overwritten by the new record.

Comments:

It might be a good idea to check with the user if he/she wants to update the metadata of the object after the game record is updated. Because some information recorded in metadata, such as 'sampled rate', 'date recorded' etc. may also need to be updated. However, this is only an option and is according to user's preferences.

4.2.5 Extracting Information from the Archive

A SC3000 collector wants to have a copy of all the information for a Sega game. He activates the SC3000 recording system and selects the 'Extract information' option. The recorder will prompt the collector for the name of the file he wants to open. The collector enters the file name and clicks on 'Open'. The recorder then asks the user a file name for this information to be stored in. The user types the name and clicks on 'Ok'. Then the

information is extracted and stored in a file with the name that the user has just entered. This file can be viewed later by the user.

Comments:

There has been discussion over the number of objects allocated for storing each game. Is it better to store all the information as one object, or store different types of information as individual objects? The advantage of storing different types of information as different objects is that it makes extracting information easier. However, it also makes managing archives harder, since a game will be represented by several objects. If one object is lost, the file is incomplete. A better approach is saving all the information related to the same game in a .tar file as only one object, so that all the related information to the game can be stored and managed together with the game itself.

4.3 Metadata

It is critical for an archiving system to store all the important information about a digital object, so that the archived object can be accessed and used later for research and study purposes. As mentioned in the last chapter the information should be carefully selected and normalized, so that relevant information can be managed together. The schema should be designed carefully to address this idea, and make the data management as easy as possible [19]. There is also a need for linking between different schemas, so that each schema can be modified individually whereas all the schemas as one metadata structure still keep the consistencies.

There are five XML schemas identified in this project. They are: Archive Data, Tape Data, Original WAV Data, Source Code Data and Attachment Data. The structure for each of them will be listed and discussed in the remaining part of this section.

4.3.1 Archive Data

Archive Data is the most important schema for the metadata structure in this project. It records all the basic information about an object, in this case the original program of a SC3000 digital game. It provides unique identification for the object, clarifies various object characteristics, its storage medium and also some miscellanea information for keeping track of updating history. At the end, the schema contains four URL links that point to four subfiles with each containing more specific information about one aspect of the digital object. The structure of the schema is shown as below.

PROGRAM

- **objectIdentifier**
 - objectIdValue
 - objectName

- objectCreator
- objectPublisher
- objectCollector
- objectReleasedDate
- **preservationLevel**
- **objectCharacteristics**
 - compositionLevel
 - fileAttribute
 - fileSize
 - sampleRate
 - bitsPerSample
 - totalSamples
 - format
 - formatName
 - formatVersion
 - creatingApplication
 - creatingApplicationName
 - creatingApplicationVersion
 - dateCreatedbyApplication
 - environment
 - environmentCharacteristics
 - environmentNote
 - emulationSoftware
 - swName
 - swVersion
 - swOtherInformation
 - system
 - systemName
 - systemModel
 - accessory
 - accessoryName
 - accessoryModel
 - accessoryAvaliability
- **storage**
 - originalMedium
 - extent
 - diskAvaliability
- **miscellanea**
 - language
 - archivedDate
 - updateDate
 - updater
- **subfiles**
 - tape
 - originalWAV
 - sourceCode
 - attachment

Figure 3 – Listing of Archive Data Structure

objectIdentifier

The objectIdentifier field is used to uniquely identify a digital object in the system. Each object should have a unique identification number and its own program name. Please note that, information of the creator, publisher and collector will also be stored in this field.

preservationLevel

This is a value that indicates the set of preservation functions expected to be applied to the object. It could be seen as the priority level for the preservation. The value should be entered by the updater as an integer number from 1 to 5 with 1 representing the highest priority for preservation. This was designed for the ease of the future archive management, so that whenever there is a conflict in archiving, the archivist should deal with the object that has the highest preservation level first.

objectCharacteristics

This is the biggest part of the schema which records all the technical properties of a digital object.

The compositionLevel is an indication of whether the object has been subject to one or more processes of decoding or unbundling. A digital object can be encoded with compression or bundled with other objects into larger packages. Therefore, it is important to know the order in which these actions are taken, so that the original object can be recovered. Initially all the game objects are assigned with 0 as their compositionLevel, since by default they have no composition level. However, if an object is saved as a compressed file, the compositionLevel should then be changed to 1 or higher depending on the number of compressions that have been applied.

The fileAttribute field describes some basic technical properties of the file, which is useful when decoding the program. The format indicates the format of the file, in this case, it is WAV, and the formatVersion indicates the version of the format that has been used.

When signals are transferred from tape to WAV file, a certain application will be used. The information about the application that was used for this transformation should be archived, because different applications might use different sample rates, and quite likely additional header information will be introduced by the application. The information includes name of the application, its version and the date the file was created by the application.

The environment field was designed for describing hardware and software environments of an object. The definition of the environment here is the surroundings by which user renders and interacts with content [19]. Separation of digital object from its environmental context can result in the content becoming unusable [19]. The environmentCharacteristics defines an assessment of the extent to which the described environment supports its purpose. For example, the value could be 'minimum' or

‘recommended’. It is important that the environment that is established can meet the defined characteristics, so that the digital object can be accessed correctly. In other words, the program can run only if the minimum software and hardware requirements are met. The emulationSoftware is the software that is needed to play the digital game on a modern computer, which in this case is the SC3000 emulator. The system is the one that can support the software that is defined earlier. In this case, it could be WindowsXP or MAC. Also for some games, they might have special accessories, such as handle, shooting gun etc.

storage

Defines the initial storage medium for the object, for example, floppy disks or CDs. Quite often, there are more than one disk needed for storing the complete program. Therefore, the extent is a value that indicates the number of disk exists. It is also useful to know the availability of the original disks.

miscellanea

This field is used for keeping track of the updating history. Language is the language that is used in the original program, for example BASIC. archivedDate indicates the date the object was archived whereas the updateDate is a series of dates that indicates the updating history.

subfiles

This field contains URLs which are acting as links to all the subfiles. There are four subfiles defined. They are: Tape Data, Original WAV Data, Source Code Data and Attachment Data.

4.3.2 Tape Data

Tape Data is designed for storing extra information about tapes where the original digital game was stored on. The structure of the schema is as follows.

TAPE

- createdDate
- archivedDate
- collector
- brandName
- location
- availability
- extent

Figure 4 – Listing of Tape Data Structure

Apart from elements that are identical to the Archive Data schema, such as archivedData, collector, there is also a number of new information added. The createdDate indicates the date that the tape was created. Ideally, this should be the same with

objectReleasedDate; however, if the tape is a copy of the original tape, this difference should be reflected on the tape created date. Apart from its availability, it is also important to know the location of the original tape. It is most likely that the tape will be located in the same place as its archive file; however, it is also possible that the tape is categorized and archived with other audio files.

Please note, that every TAPE element has an URL as its attribute which points back to the location of the Archive Data schema.

4.3.3 Original WAV Data

Original WAV Data contains detailed information about the original WAV file, such as its sample rate, bits per sample etc. The structure of the schema is shown as follows.

ORIGINALWAV

- createdDate
- creator
- software
 - swName
 - swVersion
 - swOtherInformation
- fileSize
- sampleRate
- bitsPerSample
- totalSamples

Figure 5 – Listing of Original WAV Data Structure

The software is the one that was used to convert signals from the original tape to the WAV format. The createdDate is the date the WAV file was created and the creator is the person who was responsible for that conversion process. Please note, that every ORIGINALWAV element has an URL as its attribute which points back to the location of the Archive Data schema.

4.3.4 Source Code Data

This schema is designed for storing source code of the original program. The source code could be either stored in binary code or, more preferably in BASIC, which can be copied directly into the emulator. The structure of the schema is shown as follows.

SOURCECODE

- language
 - languageName
 - languageVersion
 - languageOtherInformation
- recorder
 - recoderName

- recoderVersion
- code

Figure 6 – Listing of Source Code Data Structure

The language element indicates the type of the language that is used for the source code, for example BASIC. The recorder is the one that was used for converting signals back to the original program code. Finally, there is a field for storing the original code itself. Please note, that the same as the last two schemas, every SOURCECODE element has an URL as its attribute which points back to the location of the Archive Data schema.

4.3.5 Attachment Data

Quite often people want to archive different types of information rather than just plain description about the program itself. For digital games, people may also want to archive game posters, box covers, documentations etc. Therefore, there is a need for archiving objects of various formats. The attachment Data schema is designed for this purpose. The structure of the scheme is as follows.

ATTACHMENT

- attachmentType
- attachmentName
- attachmentSource
- attachmentOtherInformation

Figure 7 – Listing of Attachment Data Structure

An attachment should be identified by its name and its type. For example, a picture can have 'box_cover_front' as its name, and .png as its type. A document can have 'BASIC for Starters' as its name, and .pdf as its type. Also the location of the attachments will be recoded as URLs under attachmentSource field. Since it is quite likely that a digital game has more than one attachment, more than one attachment instances are, therefore, allowed under the same attachment tag. Please note, that the same as everyone else, every ATTACHMENT element has an URI as its attribute which points back to the location of the Archive Data schema.

5 Implementation

5.1 Class Structure

There are fifteen classes developed for this project. They can be further grouped into four categories: System Integrator, Program Converter, XML Generator and System Input Interface. Each part is developed and tested independently, but can be integrated together and they will work as one complete system. This design allows each part of the system to work individually; therefore, it gives greater flexibility to the developers who might be involved in further research into this topic.

At the beginning of this section, classes in System Integrator will be introduced, since they are the most important components for the SC3000 recoding system. After that, classes that belong to Program Converter category will be introduced followed by XML generators. The section will finally end with a brief description of all the system input interface classes.

5.1.1 System Integrator

System integrator contains only two classes: Graph Plotter and Program Analyzer. The system integrator group, as its name implies, is responsible for integrating all the system components together and make sure the completed system works as required.

5.1.1.1 Graph Plotter

It is a utility class that can plot out the pulse code modulation (PCM) representation of the signal for analyzing purposes. The X axis represents all the PCMs read from the PCM value file, whereas the Y axis represents the values of those PCMs.

The graphical representation for the signals is quite important in this project, since it gives a virtual representation of the processed data and makes it easier to recognize if an error occurs.

5.1.1.2 Program Analyzer

Program analyzer class contains the main class of the SC3000 recording system. It got its name because it analyzes the input data first, then formats the data and passes the results to the program converter. Program analyzer class is very important not only because it contains the main class of the system, but also because it provides algorithm for formatting the raw data and correcting any corrupted data.

There are many methods developed for this class. The method `analyze()` is used as glue for gathering all the methods together and making sure that the input data is analyzed and formatted correctly one step after another.

As the sequence of calling methods is crucial in this class, I will continue my discussion in the rest of this section based on the order these methods are called.

loadHeader()

Refer back to the WAV format, as it is shown in figure 2, at the beginning of a file, there are 44 bytes in the header. `loadHeader()` method is designed for reading only the WAV header. It does this by reading only the first 352 bits in a WAV file, and recording them in a text file.

formatData()

Before we can process the header, it is better if we can format the header first, and put the name of the value before the actual value. For example, before the sample rate value, we want to put a string 'Sample Rate' and before the total sample value we want to put a string 'Total Sample' in front of it. In this way, if we want to get the sample rate of the file later in the program execution, we only need to find the string 'Sample Rate' and followed by which should be the binary representation of the file sample rate.

readHeader()

After the header is formatted, we can start to extract information that is important for reading the data chunk, namely the format of the file, sample rate, byte rate, bits per sample and sub-chunk size. These values will be stored as global variables and will be used in various functions. Please note, that the sub-chunk size representing the size of the data, which can give us a pretty good indication of the end of a file when we are loading the raw data from the original WAV file.

loadData()

After the size of data is found, we can start to load the data. `loadData()` method takes a file name and the file size (the size of data) as its arguments, read the file bit by bit according to the specified file size, and output the binary result to a text file. This text file is called the 'raw data'.

convertLittle()

According to the header, we know that there are two bytes per sample. It is also understood that WAV files recode their data fields according to the little endian format with the lower bytes before the higher bytes. Thus, number 6 in little endian would be represented as '00000110 00000000'. `convertLittle()` is therefore developed for processing the little endian data and converting the result to integers. The final result should be the pulse code modulation (PCM) representation of the original program and will be saved in a text file called 'PCM values'.

findMark()

The PCM values are the numeric representations of the amplitudes of the original tape signals [20]. We have to find a reasonable benchmark so that for a PCM value that is less than the benchmark, we can interpret it as a 0, whereas if the value is greater than the benchmark, we can interpret it as a 1. In findMark() method, we believe that taking the average for the first 5000 PCM values should result a pretty accurate benchmark. The value will be saved as a global variable and used in decode2Tape() method.

decode2Tape()

This method reads all the values from the PCM values file, compares each of them with the benchmark value and interprets a PCM value as 1 or 0 accordingly. The result should be the same as the data that is recoded on the original tape; therefore, it is saved in a text file called 'tape data'.

In Shannon's sampling theorem it states that 'Exact reconstruction of a continuous-time baseband signal from its samples is possible if the signal is bandlimited and the sampling frequency is greater than twice the highest frequency'. However, as we found out in this project, the original signal was sampled at the sample rate that is 18.125 times the highest frequency of the original signal. As the sampling rate is not multiple of the highest frequency, it could result some signal distortions, and also because those tapes are quite old, the tape data we have produced is very likely to be incorrect. Therefore, we need an algorithm for detecting the errors and correcting them automatically. The algorithm is illustrated in the following diagram.

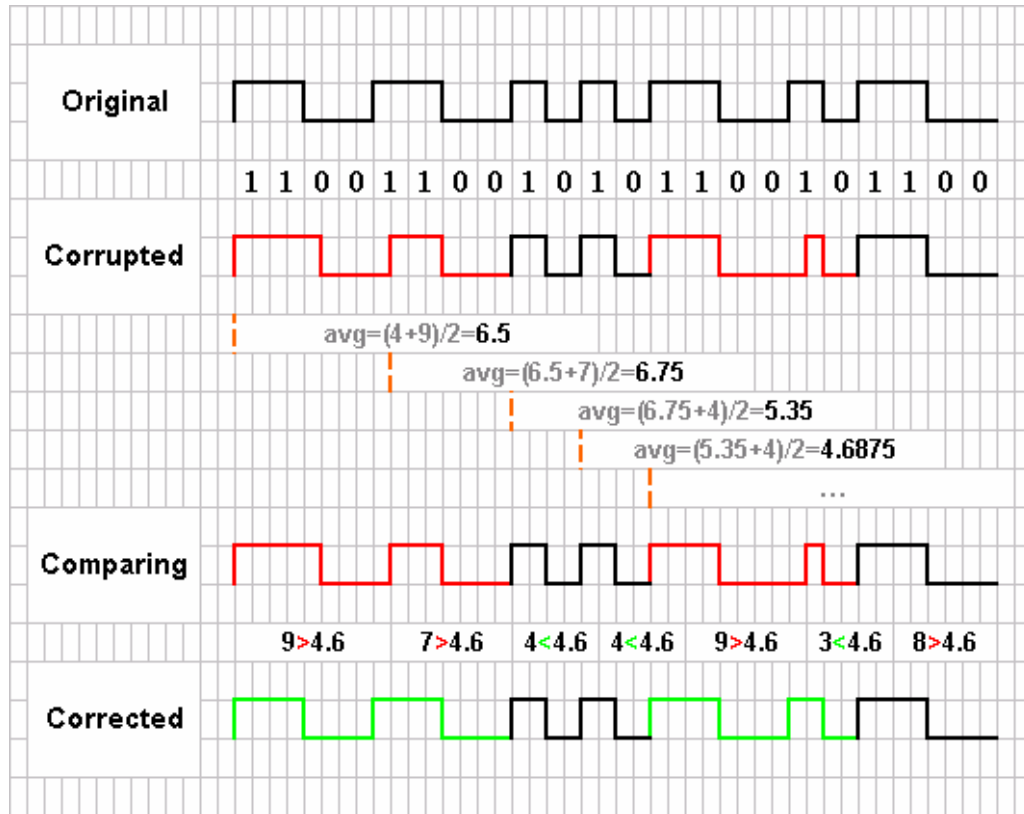


Figure 8 – Decoding Algorithm

Firstly, we need to get an average length for all the signal waves. After that we start from the beginning of the file, compare the length of a complete wave with the average length, if the wave is greater than the average length, we interpret it as '1100', otherwise, we interpret it as '10'. However, at the beginning we need an estimated average value to start with, which in practice should equal to twice the length of a single bit. In this example, by looking at the original signal pattern (As shown in figure 8), we can see it is $2 \times 2 = 4$, whereas in the project, it is $4 \times 2 = 8$.

measureLength()

This method is used for measuring the average wave length. The length for a complete wave can be gained by setting a counter every time when the program encounters a 1, continuously increasing the counter every time when a bit is read, and stopping the counter when the program encounters another 1. The value from the counter should be the length of that wave.

formatTape()

formatTape() reads from the tape data file, formats the data and saves the results. By saying 'format' I mean that any errors or corruptions in the tape data should be detected and corrected according to the algorithm described above. The method allows the

program to read a complete wave from tape data file, compares the length of that wave with the average length. If the wave length is greater than the average length, the system interprets it as '1100'; otherwise, the system interprets it as '10'. The results should be the exact binary representation of the program when it is recoded on a tape, and thus will be saved as 'binary on tape' file.

ConvertTape()

Recall the tape format from the signal processing chapter. Each 1 will be represented as '1010' on the tape whereas each 0 will be represented as '1100'. This convertTape() method is, therefore, used for converting '1100' and '1010' back to 0s and 1s respectively. The result file is saved in a text file and is called 'converted tape' data.

This should be the final binary version of the original BASIC program. The data will then be passed to the program decoder for decoding into a human-readable BASIC program.

The above procedure can be illustrated in the data flow diagram in the following page.

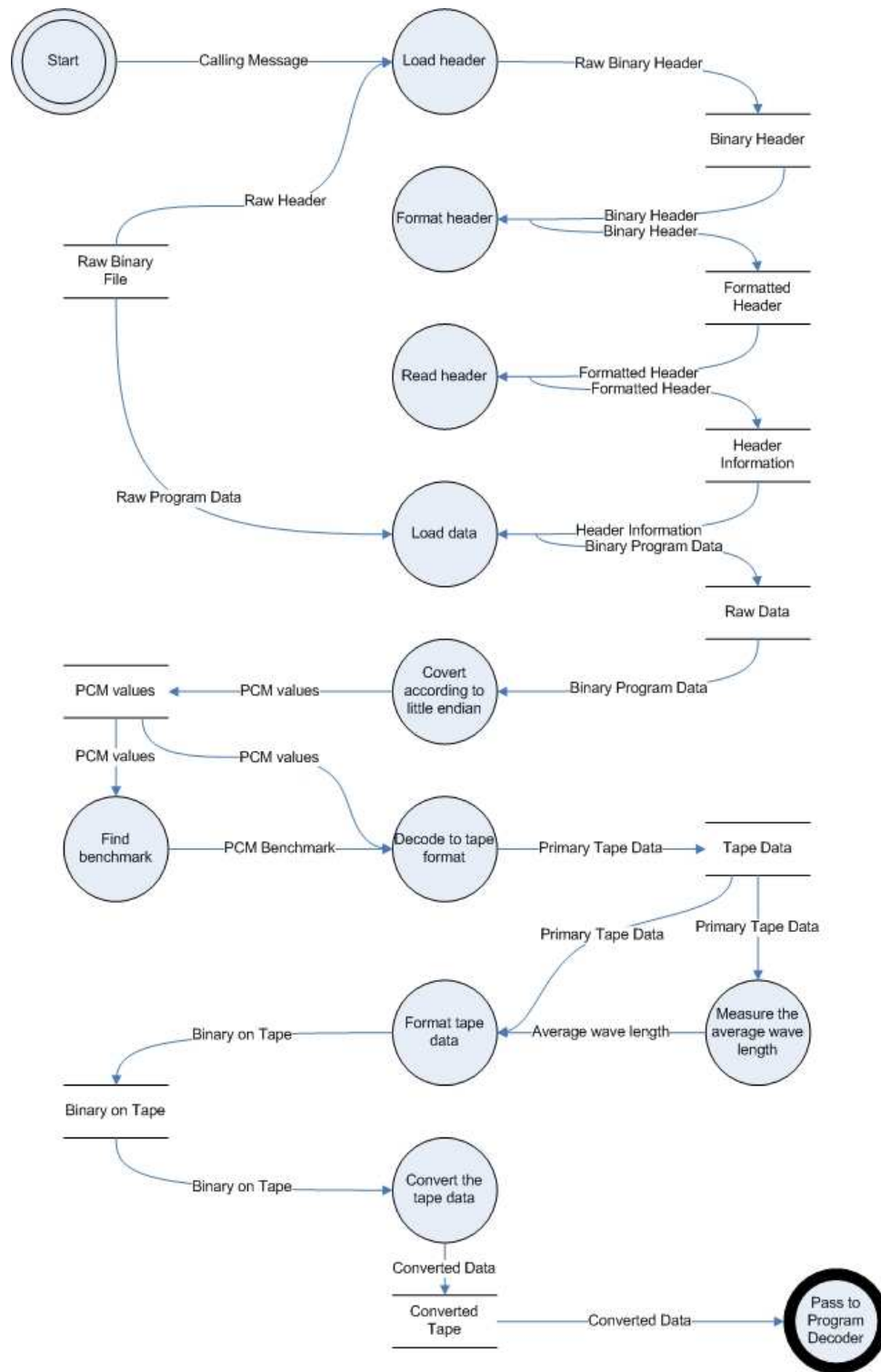


Figure 9 – DFD for Program Analyzer

5.1.2 Program Converter

Program converter is another major category in the system class design. It contains two classes. They are Char Converter and Program Decoder.

5.1.2.1 Char Converter

In this project, many conversions are needed between binary numbers and human-readable letters or phrases. The conversion between a character and its ASCII representation is quite straight forward, however, SC3000 also has its own manual for converting between its commands and the corresponding binary numbers. The CharConverter class, is therefore, designed to facilitate this conversion. The class has two major methods: IntToCommand() and CommandToInt().

For IntToCommand(), it takes an integer as input and returns a string representing the corresponding SC3000 command. For example, if it has a number 136 as its input, firstly it converts the decimal number to the hexadecimal number, which is Hx88. It then looks up its table for the corresponding string, in this case it is the 'LOAD' command. If it cannot find a string for the input value, it simply returns the decimal number as a string. For CommandToInt(), the underline process is exactly the same as IntToCommand(), but works backwards. It takes a string which represents a SC3000 command, and returns corresponding an integer value according to SC3000 manual. If the input string cannot be recognized, it simply returns -1.

There are also some utility methods designed for this class, such as isInt() for recognizing an integer, isChar() for recognizing a character and isPunc() for recognizing any punctuation.

5.1.2.2 Program Decoder

Program decoder is developed for processing a 'cleaned' tape format data and decoding it into a human-readable BASIC program. The word 'cleaned' here means that all the corrupted data should be corrected (by using the algorithm that will be described later in this section) and the binary data should follow the tape format closely.

As discussed in the signal processing chapter, the tape format is quite different from the original binary representation of the program. When the program is recorded onto the tape, some overhead information was added, and each '1' was represented by '1010', whereas each '0' was represented by '1100'. The method formatTape() is designed to convert each '1100' string back to '0' and each '1010' string back to '1', so we can obtain the original binary representation of the program. It does this by reading from a file that contains only the cleaned tape data. The output from this method will be written into a new file for further decoding.

At this stage, it is necessary to refer back to the input format that has been discussed in the signal processing chapter. A BASIC program can be separated into many blocks with each block representing a line of BASIC code. When the program is encapsulated in the input format, before saving a block in the program field the SC3000 console adds a number indicating the current position within the program field, followed by which is a 4 byte binary number representing the line number in the BASIC program. These can be considered as the 'block leading information'. The real program code then comes after the leading information. Please note that, each block representing a line of code and is ended with a byte representing decimal 13, which in ASCII code is carriage return.

For instance, the following two lines of BASIC code when stored in the input format would look like the table below.

```
10 PRINT "A"
20 END
```

00000001	1
00000000	
00000000	
00000000	
00001010	10
00000000	
00000000	
00000000	
00001000	Blank Space
10010001	PRINT
00001000	Blank Space
00100010	"
01000001	A
00100010	"
00001101	Carriage Return
00010001	17
00000000	
00000000	
00000000	
00010100	20
00000000	
00000000	
00000000	
00001000	Blank Space
10011000	END
00001101	Carriage Return

Table 2 – An Example of a BASIC Program Stored in the Input Format

In decodeTape() method, the formatTape() method will be called first, and then the resulted file will become the input file for this method. The method continuously reads

blocks which are separated by the binary representation of 13, processes the block leading information and decodes the program.

The program is represented by a series of numbers and a number can represent a command, or an ASCII code or just a real number. Fortunately, the integer representation for ASCII code is in the range between 0 and 127, and according to the manual, the integer representation for SC3000 is in the range between 130 and 230. Thus, if a number is less than 127, it will be decoded as ASCII code; otherwise, it will be decoded as a SC3000 command by using the CharConverter class. It is also understood that a real number will be represented in decimal-digital format. For example, number '101' will become '00110001 00110000 00110001' following the ASCII code representation.

The decoded program will be saved into a file as well as printed on the screen after the program execution.

5.1.3 XML Generator

There are five XML Generator classes, with each one responding to a user input interface. The basic idea for this design is to allow each metadata schema have its own input interface and XML generator, so the system components can be tested individually and the XML data files can be generated without interfering with each other. The five XML generators are: MainXMLWriter, tapeXMLWriter, wavXMLWriter, sourceXMLWriter and attachmentXMLWriter.

For the MainXMLWriter, it takes three vectors, namely: information array, system array and accessory array, as its arguments and produces a XML file according to the Archive Data schema. For other XML writers, each one takes a vector that is created by its corresponding user interface class as an argument, and produces the XML files according to the XML schema respectively. For example, tapeXMLWriter produces its XML file according to Tape Data schema, whereas wavXMLWriter produces its XML file according to WAV Data schema.

There is a Java-based, open source framework available for reading, writing, navigating and editing XML called dom4j [21]. Dom4j is fairly easy to use. It combines features of the Document Object Model (DOM) and Simple Application Programming Interface for XML (SAX), and includes support for XML Path Language (XPath), Java 2 Collections, Java API for XML Parsing (JAXP), Transformation API for XML (TRaX) and Extensible Style Language Transformations (XSLT) [21]. In this project, dom4j library has been used heavily for creating and editing XML files. Other methods such as createPrettyPrint() from OutputFormat class are also used for printing out the created XML files at the end of system execution for debugging purpose.

The implementations for the XML generators are quite straight forward, since input data has been collected, reformatted and passed by the corresponding user interface classes, all

that the generators have to do is to read information from the vectors and create XML files accordingly.

5.1.4 System Input Interface

System input interface category represents the classes that implement the graphic user interface (GUI). They are designed to guide users through the input process, display and record the correct information. There are five classes in this category: Main Input Interface, Tape Input Interface, WAV Input Interface, Source Input Interface and Attachment Input Interface.

5.1.4.1 Main Input Interface

It is designed for collecting all the information needed for the Archive Data schema. Since there are more than thirty elements in that schema, it is reasonable for the main input interface to have multiple pages, so that a page of the interface would not be crowded by all the input text fields and the information that is closely related can be displayed and edited on the same page. Users can use the navigation buttons to move between different pages. The system does not only provide normal functions such as ‘save’, ‘clear’ and ‘exit’, but also allows the interface to be modified by users according to their particular need. A screen print of the interface is shown as follows.

The screenshot displays a window titled "Tape Information Input Window I" with a subtitle "File General Information (Page One)". The interface is organized into several sections with yellow backgrounds and black text labels. The "Object Identifier" section includes fields for ID (000001), Name, and Released Date (2000-01-30). Below this is the "Preservation Level" section with fields for Preserve Level and Composition Level (1). The "File Information" section contains fields for File Size, Sample Rate, Bits Per Sample, Total Samples, Format Name, and Format Version. The "Application Creation Information" section has fields for Application Name, Application Version, and Created Date (2000-01-30). The "Environment Information" section includes fields for Environment Characteristics and Environment Notes. At the bottom, there is a row of buttons: "Clear All", "Clear P1", "P1=", "P2=", "P3=", "P4=", "Save", and "Exit".

Figure 10 – Main Input Interface

Please note that user have the option of cleaning only the current page or all the pages.

As discussed before, a digital game can have an emulator that can run on multi-platforms, also the game can have many accessories. Therefore, the interface has to change its appearance to suit this particular need.

In the system, on the second page of the main input interface, users have the option to verify the number of systems the emulator can be executed on, and the number of accessories the original game had. The third page and the fourth page will then be constructed according to these two numbers. For instance, if a user typed a number '5' in the 'Number of accessories' field, when he clicks on P4, the page should look like the one shown in the following picture.

The screenshot shows a window titled "Tape Information Input Window IV" with a subtitle "File General Information (Page Four)". The main area is yellow and contains the text "Please specify all the accessories...". Below this are five sections, each labeled "Accessory 1" through "Accessory 5". Each section has two input fields: "Accessory Name" and "Accessory Model", followed by a checkbox labeled "Available". The data entered in the first three sections is as follows:

Accessory	Accessory Name	Accessory Model	Available
Accessory 1	Control Handle	1.0	<input checked="" type="checkbox"/>
Accessory 2	Control Handle	2.1	<input type="checkbox"/>
Accessory 3	Shooting Gun		<input checked="" type="checkbox"/>
Accessory 4			<input type="checkbox"/>
Accessory 5			<input type="checkbox"/>

At the bottom of the window, there is a row of buttons: "Clear All", "Clear P4", "=P1=", "=P2=", "=P3=", "=P4=" (highlighted in yellow), "Save", and "Exit".

Figure 11 – Accessory Input Interface

All the information is recorded in vectors so that they can be passed around as a single variable; however, there are three different vectors. The information vector is used for recording general information about a digital object, such as its name, creator, publisher and technical statistics etc. The system vector is used for recoding all the systems that the emulator can be executed on, whereas the accessory vector is used for recoding information for various accessories. These vectors are created while the interface is being displayed, so that all the information that has been entered can be displayed on the interface. Users can modify the information in these vectors until they click on the 'save' when all the information is passed to the XML creators.

Problems exist if users first define the number of accessories to be 5, but later decide it should really be 3. In this situation, the length of the accessory vector will be truncated to 3, which means if the user ever entered information for all the five accessories, the information for the first three accessories will be kept, whereas all the information for last two accessories will be lost.

5.1.4.2 Subfiles Input Interface

As mentioned before, there are four XML subfiles, which are: Tape Data file, WAV Data file, Source Data file and Attachment Data file. Since the amount of information collected is quite small, the place needed for displaying a complete interface for a sub-file is significantly smaller than the main input interface. Thus, all four sub-file interfaces are linked together for easy management, and users can switch among them by using the navigation buttons.

A screen shot of the tape input interface is shown as below.

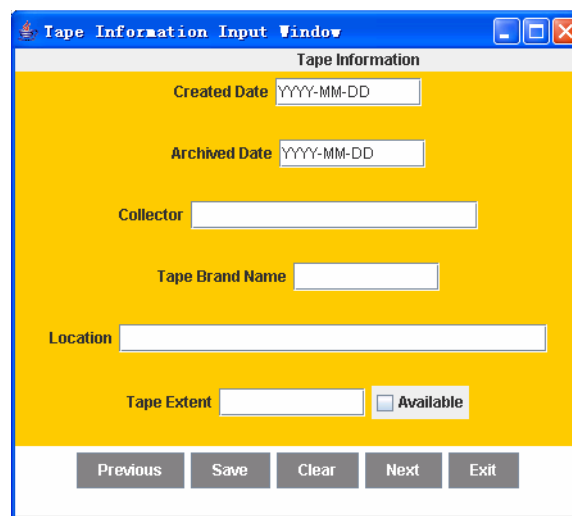


Figure 12 – Tape Input Interface

All the information is collected by vectors within the class and passed to its own XML creator respectively by calling the corresponding XML writer. For example, in the tape input interface class, a vector will be used for storing all the information collected for the original tape. This vector will then be passed to the tapeXMLWriter class for generating Tape Data file; however, if this was in WAV information interface class, the vector would be passed to the wavXMLWriter class instead.

Please note, that in each of these four classes, there exist utility methods such as inputGuard() and dateGuard() to make sure that the collected information is valid. Since these subfiles use different interface classes and pass the collected information to different creators for creating XML files, we cannot use a single vector for storing all the

information for these four subfiles. Every time when users switch to another interface by clicking on the navigation button, a new interface opens up and the old one is closed down. Therefore, the methods `saveGuard()` are used for making sure that users save their current data before they switch to the other input interface.

5.2 File Structure

When storing a digital game, many related files need to be archived; therefore, we need a well designed file structure that groups the closely related information together for easy management.

The recommended file structure is shown in the diagram below.

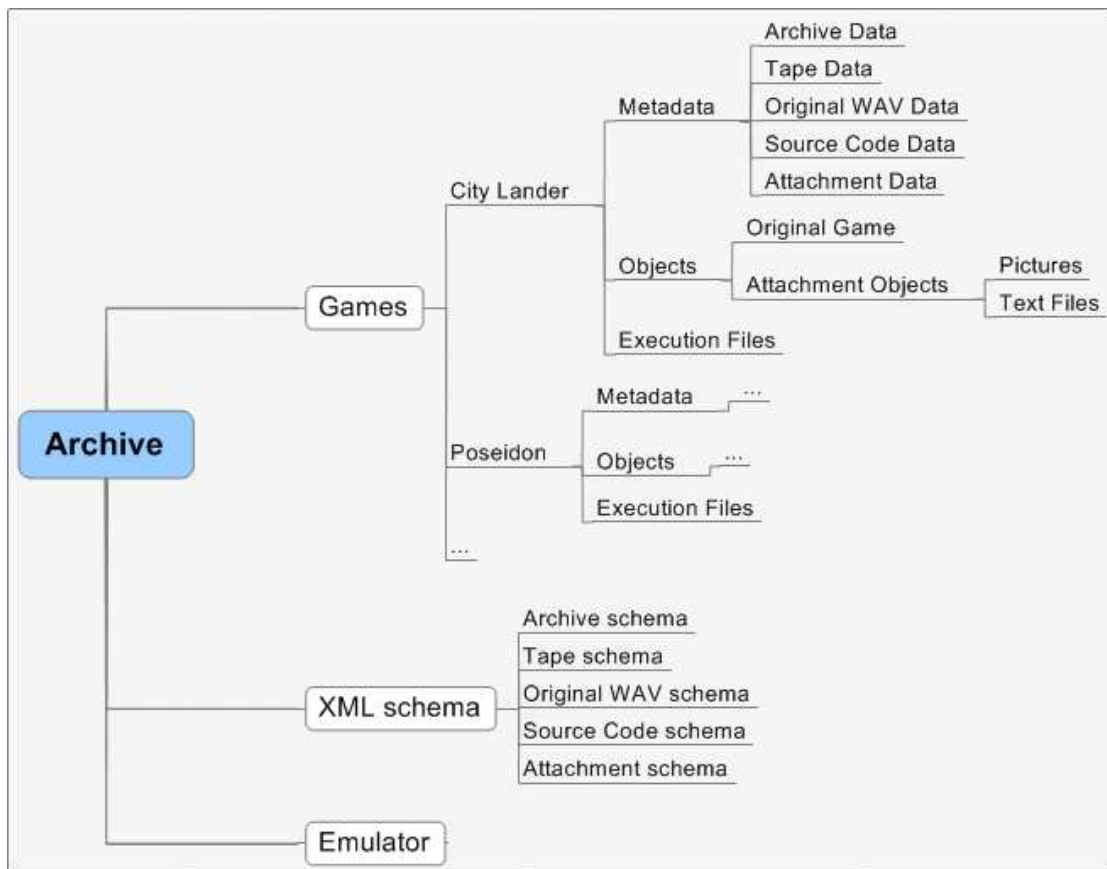


Figure 13 – File Structure

In the general digital archive directory, there exist three sub-directories: the XML schema that contains five different schemas used in this project, the emulator and the games.

All the digital games and their related information should be stored within its own directory that forms a part of the 'Games' directory. Each game directory contains the objects, its metadata and execution files. The metadata directory contains all the XML files that have been generated by the system. The objects directory contains the digital game itself together with its attachment objects. Recall that, the local URLs of these attachment objects should be stored in the Attachment Data XML file as links to the resources. The execution directory contains all the files generated during the archiving program execution. This directory can be stored together with the game or simply deleted after the execution.

In the XML schema directory, there are the five schemas that are used for validating the XML files generated by the system. There are discussions on whether the XML schema should be stored in a central location or be stored in a distributed manner. In this design, as it is shown, each general digital archive directory should contain a copy of the XML schemas and the emulator, so that greater flexibility can be guaranteed for each archive center while it still ensures that those archive centers can follow the agreed archiving data structure.

After all files are in the right place, the whole game directory can be tarred up with a checksum to allow corruption to be detected.

The feasibility of this file structure has already been tested in this project. Methods from File class in JRE System library (Version 1.5.0) can be used for getting a file path and thus locating a file. The mkdir() method can be used for making new directories. A move method can be implement easily by synchronized copying of a file from one directory to another and then deleting the old file by calling delete() method on a File object. It is thus believed that implementing this file structure can become a part of the future system development.

6 Testing and System Evaluation

6.1 The Test Code Generator

The methodology used in this project determined that an excellent test code generator, which can generate code as close to the original tape data as possible, is critical for the success of the system. This test code generator is thus developed for this purpose and used heavily throughout the whole project developing process for testing and debugging the system.

This test code generator allows a user to write a BASIC program and save it in a pure text file. It reads from the saved file, converts the program to the binary representation line by line, adds the block leading information for each line and then encapsulates the whole program in the SC3000 input format and converts the final result to the tape format. However, the generator does not attempt to convert the final result to PCM and encapsulate the PCM values in WAV format, since developing such a function is time consuming and error prompting, and it involves deciding a preferable 'bits per sample' value and sample rate, which if it did not choose correctly, would make to generated code rather useless.

For demonstrating purposes, the rest of this section will take the following BASIC program as an example.

```
10 PRINT "ENTER A NUMBER"  
20 INPUT I  
30 IF I>=(100-20)*2 THEN GOTO 40 ELSE GOTO 50  
40 PRINT "NO PRINT!"  
50 PRINT I  
60 END
```

The method `readCodeFile()` is used to read BASIC program line by line. When a new line is read, the line number will be striped off, converted to binary and passed to the method `processCode()` together with program code itself.

The method `processCode()` firstly breaks the program code into pieces according to the blank spaces and then stores all the pieces in a temporary vector. All the additional information, such as blank spaces and carriage returns, will also be added. For example,

after breaking down into pieces, the first line of the sample program will look like the following table.

0	PRINT
1	BLANKSPACE
2	"ENTER
3	BLANKSPACE
4	A
5	BLANKSPACE
6	NUMBER"
7	CARRIAGERETRUR

Table 3 – An Example of a Temporary Vector

The table will then be processed line by line, and the binary value will be added together as a string and returned to the caller.

Problems exist when processing a print command that prints the value that might be recognized as a SC3000 command. Take the 4th line of the sample code for example; it wants to print a sentence with the command comparable word "PRINT" in it. To solve this problem, every time when the generator encounters a quotation mark, it starts to code each character in its corresponding ASCII format. This process ends when the generator encounters another quotation mark.

There is also another problem needed to be considered when implementing the generator. Take the third line of the sample code for instance. In that line, it is specified that $I \geq (100 - 20) * 2$. The tricky part is, in the SC3000 manual, it treats $>$, $=$ and \geq as three individual mathematical operators, each with a different value. Therefore, ' \geq ' cannot be converted into two ASCII codes, but rather as one SC3000 command. To solve this problem, the generator has to recognize the operators first, which are '+-*/<>=' (Note that, brackets are not recognized as operators), it keeps adding the operators to a string until it encounters a non-operator value. Then it will pass the string to CharConverter class for getting its corresponding integer value.

All the converted value will then be added together with the line number attached at the very beginning (As shown in table 2). When the readFile() method received the result from the processCode() method, it counts the number of bytes in the result, and determines the position number for that block. If the first block has 7 bytes, then the next block should have the position number 8.

When the lines of a BASIC program are read and converted to their binary representation, the result will be written in a new file. The convertTape() method will read from the file, encapsulates it in the SC3000 input format and finally converts it to the tape format. The result will be saved in a file and ready for testing purposes.

Please note, that the method `fillingBinary()` is used for filling up the empty space in a byte. An ASCII code only has 7 bits, therefore, `fillingBinary()` is needed to fill up the one bit left with 0. Also recall that a line number is represented by 4 bytes, which are 32 bits, apart from the number itself, the rest of the space should be filled up with 0s.

6.2 System Evaluation

The system evaluation was carried out after the development phase was completed. Because the entire project development was based on the five major user stories, the system was also evaluated by using those user stories as guidelines. In the rest of this section, the system performance for different user requirements will be described and compared with the original user stories.

In the project, the main menu is implemented with the command line structure. Because the project is not concentrated on the user input interface, the command line structure can perform actions exactly the same as a normal graphic user interface (GUI), and allows me to spend minimum time on the GUI design while concentrating on the system architecture.

6.2.1 Archiving Directly from a Tape

When a user wants to archive directly from a tape, he connects the tape player with a computer, and activates an audio recording program, such as 'GoldWave' or 'Audacity'. He then starts to play the tape and the selected audio recording program will save the program signals in a WAV file on the computer. After the entire file is saved, the user activates the SC3000 recording program and archives the WAV file.

Evaluation:

This user story is not implemented as a part of the SC3000 recording system, since many programs have already been developed for recording tape signals. To avoid duplication work, the best approach is therefore for users to manually record the tape signals into a WAV file and then use the SC3000 recording system for archiving the saved file. The details of the archiving procedure will be described in 'Archiving a WAV file' user stories.

Overall, the user requirements in this story are somewhat satisfied with the help of another application.

6.2.2 Archiving a WAV File

A user wants to archive a WAV file. He activates the SC3000 recording system on his computer and types ‘-a wav’ (Archive from WAV file) in the command line. The SC3000 recording system prompts the user for the name of the input file. The user enters the file name and clicks on ‘Start’. The SC3000 recording system then starts to analyze the game program extracts information and creates the XML files for storing those data. The recorder also asks the user to manually enter some of the information needed for the metadata schemas.

Evaluation:

It is to the user’s preference if he wants to enter the additional information for the game, however, if the user chooses not to enter the information, the only information that will be stored in the metadata files is primary technical statistics, such as sample rate, bits per sample and total samples. Therefore, when the user chooses not to enter the additional information, the system generates a warning message and asks if the user wants to carry on his decision.

To archive other types of objects, such as pictures or documentations, users have to move those files to the ‘Attachment Object’ folder under ‘Objects’ directory, as described in the File Structure section. However, if the file structure described is implemented, the moving of those attachment files can be done automatically by the SC3000 recording system.

Generally speaking, apart from the need for users to manually move the attachment files to the correct folder and update the links in the Attachment Data files, all the other requirements in the original user stories can be satisfied by the SC3000 recording system.

6.2.3 Updating an Archived Game with Metadata

A SC3000 game collector later finds out that some of his information stored for a game is actually inaccurate. He wants to change the metadata stored for that game. He activates the SC3000 recording system and types ‘-u infor’ (Update game information) in the command line. The recorder will prompt the collector for the name of the file he wants to modify. The collector then enters the file name and clicks on ‘Open’. All the metadata of that object is then presented to the user. The user makes his change and clicks on ‘Save’. After the confirmation, the new data is saved and overwrites the old data.

Evaluation:

The performance of SC3000 recording system is identical to the original user story. Therefore, all the requirements described in this user story can be fulfilled by the SC3000 recording system.

6.2.4 Updating the Stored Program

A game collector wants to replace his old record by a better version. He activates the SC3000 recording system and types '-u game' (Update game record) in the command line. The recorder asks the collector to locate the new record he wants to store. The collector locates the new record. The recorder will then prompt the collector for the name of the file he wants to modify. The collector then enters the file name. After the confirmation, the old data is overwritten by the new data.

Evaluation:

As discussed in the related user story, it might be a good idea to ask the user if he/she wants to update the metadata of the object after the game record is updated. In practice, some technical information, such as sampled rate, is extracted from the new WAV file and recorded in the corresponding metadata file automatically once the user has located the new WAV file. However, the user has to manually move the new WAV file to the correct folder, and delete the old file. The automation of this procedure can be improved, again, when the file structure is implemented.

The technical information, such as sample rate, bits per sample can be updated once the new WAV file is located; however, the user has to manually replace the old WAV file by the new one. This appears to be the only unfulfilled requirement for this user story.

6.2.5 Extracting Information from the Archive

A SC3000 collector wants to save all the information about a Sega game so that he can view the information later on. He activates the SC3000 recording system and types '-r infor' (Extract information) in the command line. The recorder will prompt the collector for the name of the file he wants to open. The collector enters the file name and clicks on 'Open'. The recorder then asks the user a file name for this information to be stored in. The user types the name and clicks on 'Ok'. Then the information is extracted and stored in a file with the name that the user has just entered. This file can be viewed later by the user.

Evaluation:

The system performance is identical to the original user story. The extracted information will be saved in a text file for users' review. All the requirements in this user story are fulfilled.

The discussion about saving all the objects related to a game as one compressed file results in an alternative archiving standard for the game files. Recall from the File Structure section, all the information about a digital game will be stored in the directory with the name of that game. To achieve a higher level of consistency, the directory can be compressed as a .tar file, so that all the files in that directory can be accessed or managed as one entity.

7 Conclusions

This project has investigated the feasibility of archiving digital objects, especially digital games from the 80s and 90s. In order to archive those games, we do not only need to store the games themselves, but also need a well developed metadata structure for storing information that is related to those games. This information is essential for future study and research purposes, because it contains data such as ‘sample rate’, ‘bits per sample’, etc. without which it would be impossible to decode the WAV file correctly.

The project developed five XML schemas which are efficient for storing all types of the information mentioned above. Archive Data is designed to be the main data schema which contains URL links pointed to the other four sub-schemas. The same idea is used in Attachment Data: when the locations of the attachments are stored, they are stored as URLs of the attachment objects. By using URLs, links between files can be established while the objects can be grouped by their types in a better file structure.

Since there are many files needed to be stored, a file structure is needed for the ease of management and to make sure that objects of the same type are stored in the same directory. The feasibility of developing this file structure has been tested in this project and the idea of implementing the file structure seems practical.

The SC3000 recording system is developed as a part of this project. The system went through a typical XP development process by identifying user stories first, building up a test code generator and continuously identifying, designing, implementing and testing the new components throughout the project. Because of the development methodology that has been chosen for this project, we decided to make sure that each type of XML schema has its own input interface and its own XML file generator, so that every time a new system component is developed, it can be tested individually by using the test code, and integrated into the existing SC3000 recording system.

The system was tested on the test code that has been generated; however, we are still experiencing problems when the system is tested on the real program data. One possible reason is that for the real program it actually contains the unrecorded data at the beginning of the program and some dummy data at the end. When the test code is created, however, these two chunks of data were not added. Although the system is designed to find a meaningful pattern before it starts to load data, it is believed that because of these dummy data the system cannot input the real program data correctly; therefore, further analysis cannot be carried out.

7.1 Contributions

Firstly, in this project a basic archiving system is developed as a foundation for any further research on this topic.

Secondly, a well-designed metadata scheme is presented, which can be used for storing information of a digital object together with that object, so that the information is consistent and easier to manage. A general research was conducted prior to the designing of the scheme, which included topics such as the advantage of using a metadata scheme, the storage structure of a scheme and any related work on digital archiving. The scheme is implemented in XML and verified by a development tool called 'XMLSpy'. The metadata can be populated by using the archiving system, which takes user input and converts it to a valid XML file automatically. Details on this feature will be discussed later in this document.

Thirdly, an in-depth study of how to read games stored in sound files and extract original source code representation was carried out in this project. Topics include the WAV format that is used for storing the original data, the open source FLAC format that can be used for data storage in the future and the input format for the emulator that is used for this project. Some algorithms have been developed for the purpose of converting between different signal formats, for example, the conversion from a PCM sampled signal to its binary format. Issues such as error detection and correction are considered thought-out the analyze and implement phases. For the testing purpose, a test code generator has also been developed.

7.2 Future Work

At present, when a game is archived, users have to move files manually to the correct archive directory. It is important that the file structure described in this project can be implemented as a part of the future work, so that sorting out the archived files can be done automatically by the system. The checksum fields should be added for protecting the integrity of the archived files, also digital signatures could be added to the file for security purposes.

As mentioned in the conclusion, the SC3000 recording system is still experiencing problems when archiving the real digital games. The junk data has to be cleared out before the real program data can be processed. The existing system can already process the program data; therefore, once the junk data is cleared out as a part of the future work, the SC3000 recording system can then be used on the real program data without any modification to the existing system.

The current system is able to process SC3000 games that are stored as source code; however, some SC3000 games are stored as compiled programs on the original tape.

Therefore, processing the compiled programs can also be considered as a part of the future work

The SC3000 recording system is designed and implemented especially for SC3000 games. It might be a good idea to extend its flexibility by allowing it to process digital games that were created for other consoles. As the system is developed by using XP methodology, each component of the system can be executed independently. To change the SC3000 recording system for other type of games, the developer only needs to change the program decoder class so that it can decode the program according to the new format.

8 Bibliography

- [1] M. Swalwell, I. Welch, C. Susan, *A Proposal for Community Partnership Fund.*, The Digital Strategy, Wellington, NZ, 2005.
- [2] C. Dean, *Conserving Early New Zealand Digital Games*, Faculty of Humanities and Social Sciences, Victory University of Wellington, 2005.
- [3] D. Flecker, *Digital Archiving: What is Involved?*, available from <http://www.educause.edu/ir/library/pdf/ERM0316.pdf>, last accessed: August 2006.
- [4] G.M. Hodge, *Best Practices for Digital Archiving an Information Life Cycle Approach*, available from <http://www.dlib.org/dlib/january00/01hodge.html>, last accessed: September 2006.
- [5] *Digital Archiving: Procedures and Guidelines*, available from <http://www.english-heritage.org.uk/server/show/nav.001002003009001002004>, last accessed: October 10 2006.
- [6] A. Fienlander, *Summary of Findings-Center for Information Strategy and Policy*, available from <http://flac.sourceforge.net/>, last accessed: August 2006.
- [7] *Archives New Zealand*, available from <http://www.e.govt.nz/resources/research/ready-access-2004/chapter5.1.html>, last accessed: October 2006.
- [8] Rothenberg, J. (1995). *Ensuring The Longevity of Digital Documents*. Scientific American, 272(1), 42-47.
- [9] M. Fayzullin, *Emulators for Windows and Msdos*, available from <http://fms.komkon.org/EmuWindows/>, last accessed: September 2006.
- [10] *Technical Page for SC3000*, available from <http://homepages.ihug.co.nz/pinwhiz/sc3000.htm>, last accessed: September 2006.
- [11] M. Fayzullin, *Master System/ Gamegear Emulator*, available from <http://fms.komkon.org/MG/>, last accessed: June 2006.
- [12] *Internet Archive Structure*, available from <http://www.archive.org/about/about.php>, last accessed: September 2006.
- [13] *Wav Frequently Asked Questions*, available from <http://www.mp3-converter.com/faq/wav.htm>, last accessed: September 2006.
- [14] M. Boldin, *WAV File Format Description*, available from <http://rti.etf.bg.ac.yu/rti/ir1pp2/domaci/WavFileFormat.html>, last accessed: October 15 2006.

- [15] R. Mavati, *Sega SC3000-SC7000 Technical Page*, available from <http://membres.lycos.fr/mavati/sf7000/>, last accessed: September 2006.
- [16] *What is FLAC?*, available from <http://flac.sourceforge.net/>, last accessed: July 2006.
- [17] *Codec*, available from <http://en.wikipedia.org/wiki/Codec>, last accessed: October 20 2006.
- [18] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley, 2002.
- [19] *Data Dictionary for Preservation Metadata*, The PREMIS Working Group, OCLC and RLG, 2005.
- [20] M. Bosi, *WAV PCM Sound File Format*, available from <http://ccrma.stanford.edu/CCRMA/Courses/422/projects/WaveFormat/>, last accessed: August 2006.
- [21] *Dom4j: The Flexible XML Framework for Java*, available from <http://www.dom4j.org/>, last accessed: September 2006.

9 Appendices

9.1 Sound Format

ROM ROUTINES: These routines are used by the cpu when it communicates with the devices connected to it. These routines can be called independently by the programmer, using a CALL statement. Table XXX3 lists some important ROM routines.

Table XXX1. RESERVED RAM AREAS.

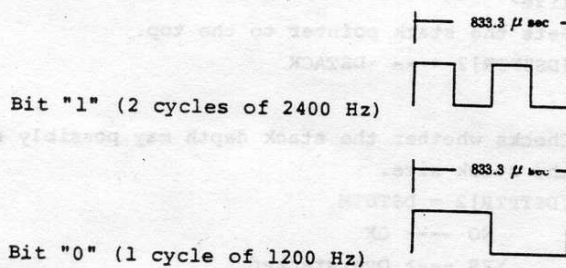
Hex Address	Purpose
8160/8161	Start of Basic program
8162/8163	End of Basic program
8164/8165	String Storage pointer
8166/8167	Top of String Storage
8168/8169	Top of Memory pointer
82A2	Program found flag, 0=found
82A3	Filename being loaded (16 bytes)
83A3	Filename being saved (16 bytes)
8B30	Basic Stack Area
8B36	8H80 bytes. Write to VRAM 8H1800+
9336	Screen control
9339	Color text screen byte
933A	Color graphics screen byte
9364	8H80 bytes VRAM stores 8H1800+ here
9411	Top range of cursor
9412	Bottom range of cursor
9413	8 bytes for storage of PATTERN command
9420	8H28 bytes for storage of VRAM data
9460 " 9480	INKEY\$ Storage area
9484	Cursor, 0=normal, 2=graphics
9485	1=lowercase, 0=uppercase
9486	keybeep, 0=no beep, 1=no beep
9489	Cursor position X value
948A	Cursor position Y value
948E	Time\$ seconds
948F	Time\$ minutes
9490	Time\$ hours
9744/9745	Address of DATA byte

Table XXX2. BASIC KEYWORDS.

82 LIST	A0 ON	C1 *	BC MOTOR
83 LLIST	A1 RETURN	C2 /	BD FN
84 AUTO	A2 ERASE	C3 MOD	BE TO
85 DELETE	A3 CURSOR	C4 +	E2 STEP
86 RUN	A4 TP	C5 -	E3 THEN
87 CONT	A5 RESTORE	C6 < > or > <	E4 TAB
88 LOAD	A6 SCREEN	C7 > = or = >	E5 SPC
89 SAVE	A7 COLOR	C8 < = or = <	CC NOT
8A VERIFY	A8 LINE	C9 >	CD AND
8B NEW	A9 SOUND	CA <	CE OR
8C RENUM	AA BEEP	CB =	CF XOR
90 REM	AB CONSOLE	CC 8080 ABS	8081 RND
91 PRINT or ?	AC CLS	CD 8082 SIN	8083 COS
92 PRINT or L?	AD OUT	CE 8084 TAN	8085 ASN
93 DATA	AE CALL	CF 8086 ACS	8087 ATN
94 DEF	AF POKE	D0 8088 LOG	8089 LGT
95 INPUT	B0 PSET	D1 808A LIT	808B EXP
96 READ	B1 PRESET	D2 808C RAD	808D DEG
97 STOP	B2 PAINT	D3 808E PT	808F SCR
98 END	B3 BLINK	D4 8090 INT	8091 SCN
99 LET	B4 POSITION	D5 8092 ASC	8093 LEN
9A DIM	B5 HOOKY	D6 8094 VAL	8095 PEEK
9B FOR	B6 SPRITE	D7 8096 INP	8097 PRE
9C NEXT	B7 PATTERN	D8 8098 VPEEK	8099 STICK
9D GOTO	B8 CIRCLE	D9 809A STRIG	80A0 CHR\$
9E GOSUB	B9 BCIRCLE	DA 80A1 HEX\$	80A2 INKEY\$
9F GO	BA MAG	DB 80A3 LEFT\$	80A4 RIGHT\$
CO ^	BB VPOKE	DC 80A5 MIDS	80A6 STR\$
		DD 80A7 TIME\$	

Figure 14 – The SC3000 Manual

Supplementary Chart 12. Format for Recording on Cassette Tape
[Representation of 1 bit]

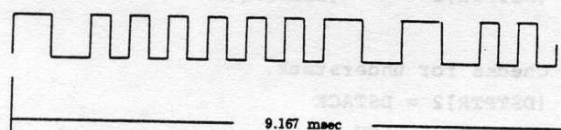


[Representation of 1 byte]

Example

For 1-byte data, 27H

START BIT	BIT0	BIT1	BIT2	BIT4	BIT4	BIT5
0	1	1	1	0	0	1



BIT6	BIT7	STOP BIT	STOP BIT
0	0	1	1

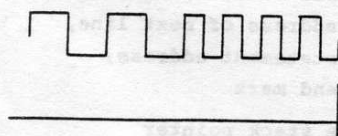
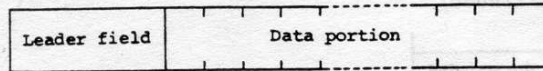


Figure 15 - The Tape Format

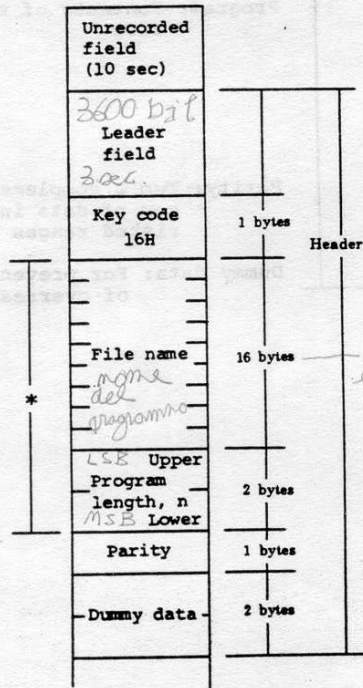
[General format]



Leader field: 3600 records of bit 1 (=450 bytes)

Data portion: A collection of byte data

[Format for save of text]



Key code: For identification of header portion of BASIC text

File name: 16 char. max. A shorter name is followed by spaces to fill 16 positions.

Program length: 16-bit binary

Parity: Two's complement of sum of data in asterisked ranges

Dummy data: For prevention of overread

(continues)

Figure 16 - The Input Format for SC3000

(continued)

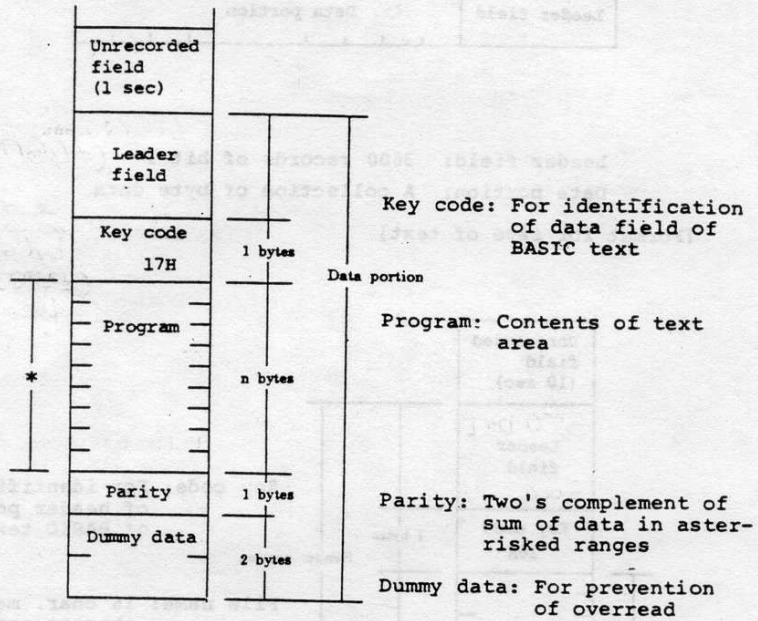


Figure 17 - The Input Format for SC3000 (Continued)

9.2 XML Schema

D:\Daniel.C.Doe\My Documents\MyProject\TestA...\archivedata.xsd



2006-10-12 15:10:30

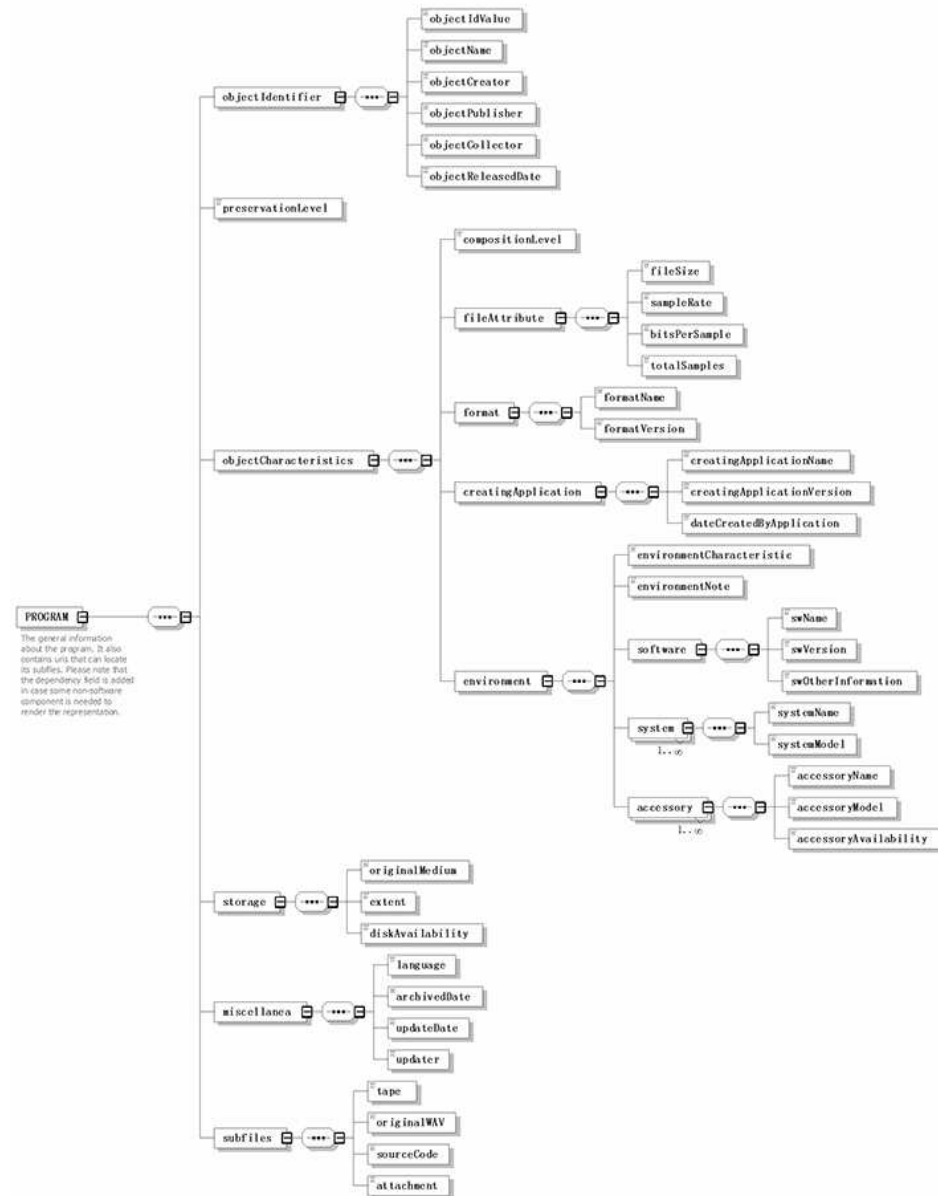


Figure 18 – The Archive Data Structure

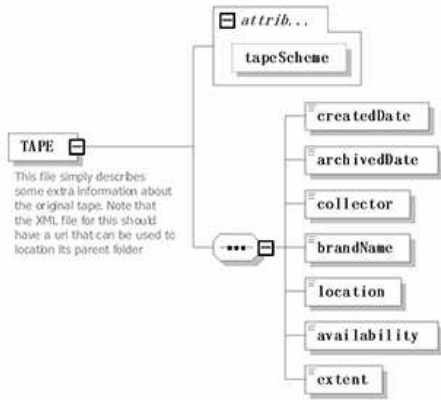


Figure 19 – The Tape Data Structure

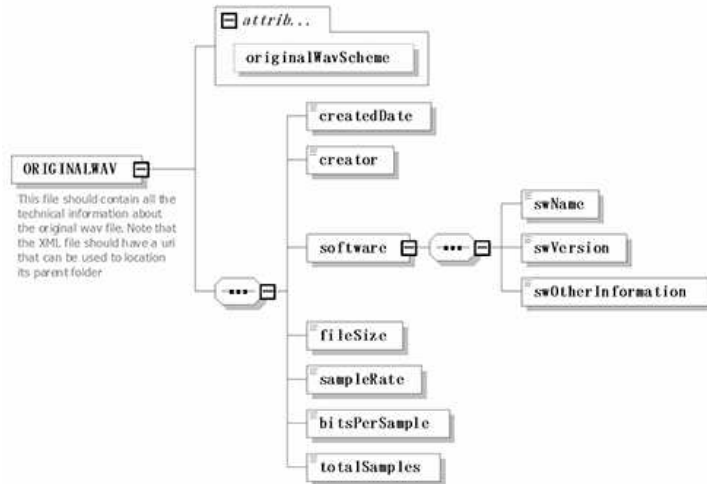


Figure 20 – The Original WAV Data Structure

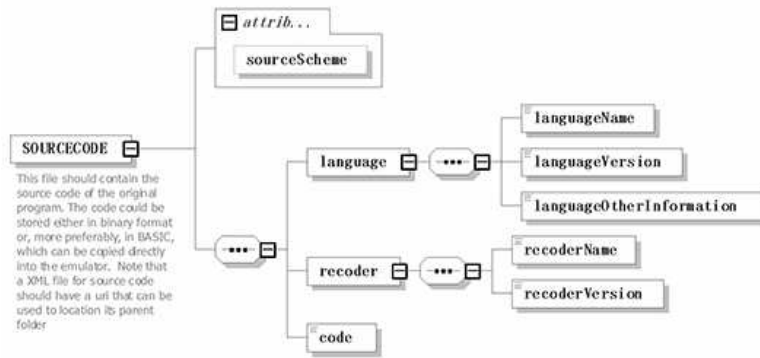


Figure 21 – The Source Code Data Structure

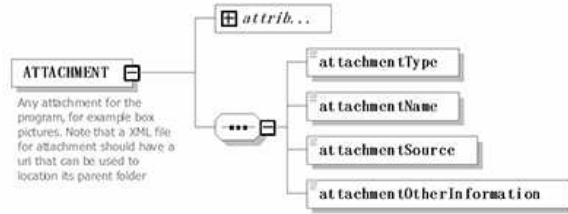


Figure 22 – The Attachment Data Structure