DATA COMPRESSION ENGINE ENHANCEMENT USING HUFFMAN CODING ALGORITHM

KAN FU GIANG

UNIVERSITI TEKNOLOGI MALAYSIA
UNIVERSITI TEKNOLOGI MALAYSIA

DECLARATION OF THESIS / UNDERGRADUATE PROJECT PAPER AND COPYRIGHT

Author’s full name : Kan Fu Giang

Date of birth : 22 August 1986

Title : Data Compression Engine Enhancement Using Huffman Coding Algorithm

Academic Session: 2010/2011

I declare that this thesis is classified as :

☐ CONFIDENTIAL (Contains confidential information under the Official Secret Act 1972)*

☐ RESTRICTED (Contains restricted information as specified by the organization where research was done)*

☑ OPEN ACCESS I agree that my thesis to be published as online open access (full text)

I acknowledged that Universiti Teknologi Malaysia reserves the right as follows:

1. The thesis is the property of Universiti Teknologi Malaysia.
2. The Library of Universiti Teknologi Malaysia has the right to make copies for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

Certified by :

[Signature]

SIGNATURE

[New IC No./Passport No.]

860822-59-5251

Prof. Dr. Mohamed Khalil B Mohd Hani

NAME OF SUPERVISOR

Date : 20 MAY 2011

Date : 20 May 2011

NOTES : * If the thesis is CONFIDENTIAL or RESTRICTED, please attach with the letter from the organization with period and reasons for confidentiality or restriction.
“I hereby declare that I have read this project report and in my opinion this project report is sufficient in terms of scope and quality for the award of the Bachelor of Engineering (Electrical – Microelectronics)”

Signature : 
Name of Supervisor : Prof. Dr. Mohamed Khalil Bin Mohd. Hani
Date : 20 MAY 2011
DATA COMPRESSION ENGINE ENHANCEMENT USING HUFFMAN CODING ALGORITHM

KAN FU GIANG

A project report submitted in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering (Electrical – Microelectronics)

Faculty of Electrical Engineering
Universiti Teknologi Malaysia

MAY 2011
I declare that this project report entitled “Data Compression Enhancement Using Huffman Coding Algorithm” is the result of my own research except as cited in references. The project report has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature : …………………………………………
Name : Kan Fu Giang…………………………
Date : 20 May 2011…………………………
Specially dedicated to
my family and friends
ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my supervisor, Professor Dr. Mohamed Khalil Hani bin Mohd Hani for giving me a chance to work with him. His guidance, advices and encouragement has lead to the fruitful completion of the project. I’ve learnt not only in academic wise but also in philosophies of life. All these have unleashed my potential which pushes me to greater height.

I would like to convey my gratefulness and appreciation to those who have helped me directly or indirectly. Of particular mention is Mr Liew Tek Yee for his guidance, advices and precious time. The project and thesis would not be concrete without his patience and dedication.

Last but not least, I would like to express my appreciation to my family who are always there supporting me in all aspects.
ABSTRACT

As the usage of computer and networking increased day by day, data storage and transmission have been the issue among people because it cost money. There is a need to do data compression to increase effective communication bandwidth and effective storage capacity. This thesis proposed a widely used data compression technique for textual file and its enhancement using Huffman Coding. The LZSS compression algorithm implementation and its enhancement are model in C language. The design is further verified by experiment on the compression and decompression process with a sample text file. Performance analysis is done by comparing the compression ratio before and after enhancement. Result of the analysis showed that the enhancement using Huffman Coding is successful.
ABSTRAK

TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARATION</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>ABSTRAK</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>LIST OF SYMBOLS</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF APPENDICES</td>
<td></td>
<td>xii</td>
</tr>
</tbody>
</table>

1 INTRODUCTION 1
1.1 Problem Statement 1
1.2 Objectives 2
1.3 Scope of Work 2
1.4 Contributions of the Project 3
1.5 Organization of Project 3

2 BACKGROUND AND LITERATURE REVIEW 4
2.1 Data Compression Background 4
2.2 Data Compression Algorithm 7
2.3 Application of Data compression 8
## LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE NO</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Compression ratio comparison</td>
<td>20</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE NO</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Lossless compression algorithm</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Lossy compression algorithm</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Project Workflow</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Dictionary and Incoming Source Data in Array</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Source data for verification</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>Compressed data for LZSS compression</td>
<td>18</td>
</tr>
<tr>
<td>5.3</td>
<td>Restored data for LZSS compression</td>
<td>18</td>
</tr>
<tr>
<td>5.4</td>
<td>Compressed data for Huffman Coding compression</td>
<td>19</td>
</tr>
<tr>
<td>5.5</td>
<td>Restored data for Huffman Coding compression</td>
<td>19</td>
</tr>
<tr>
<td>5.6</td>
<td>Compressed data for enhanced LZSS compression</td>
<td>20</td>
</tr>
<tr>
<td>5.7</td>
<td>Restored data for enhanced LZSS compression</td>
<td>20</td>
</tr>
</tbody>
</table>
## LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC</td>
<td>Intergrated Circuit</td>
</tr>
<tr>
<td>LZ77</td>
<td>Lempel-Ziv-77</td>
</tr>
<tr>
<td>LZSS</td>
<td>Lempel-Ziv-Storer-Szymanski</td>
</tr>
</tbody>
</table>
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Example of LZSS Compression Algorithm</td>
<td>25-31</td>
</tr>
<tr>
<td>B</td>
<td>LZSS Source Code in C</td>
<td>32-37</td>
</tr>
<tr>
<td>C</td>
<td>Huffman Coding Source Code in C</td>
<td>38-50</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

This project proposes the implementation of a data compression engine. The data compression engine is modeled in C behavioral model. This first chapter includes the problem statement, objectives, scope of project and project contributions. Meanwhile the thesis organization is at the end of this chapter.

1.2. Problem Statement

As the usage of computer and others digital technology products with the same kind increase, most of the documents and information are being stored in the digital form. Many people tend to digitalize everything in the life to make life became simple and easier. Moreover, internet and networking also contribute to the increment of this phenomenon. Therefore, data storage and data transmission have become the main concern of these technologies products’ users.

Data transmission and storage cost money. The more information being dealt with, the more it costs. In spite of this, most digital data are not stored in the most compact form (Steven, 2007). In order to get the most compact form, we need to compress the digital data. This process is called data compression. Since all sources information have inherent redundancies with them (Shanon, 1948), data compression technique tends to remove the inherent redundancies by representing data in fewer bits. Therefore, it increases the effective storage capacity. Thus, reduce the
consumption of expensive resources, such as hard disk space and transmission bandwidth.

Until today, many data compression techniques have been developed. These techniques have different performance based on the characteristics of the source data. However, research is still being constructed to seeking for any improvement for the data compression.

1.6 Objectives

From the issues that discussed in the previous section, the objectives of this project were set out bellow:

1. To explore the data compression design in C behavioral model.

2. To enhance the data compression engine using Huffman coding.

3. To study the compression ratio before and after the enhancement.

1.7 Scope of Work

In this project, the data compression engine is designed in C behavioral model. LZSS data compression is used as a case study to explore and study about the data compression process.

After that, Huffman coding is used to enhance the data compression engine. Meanwhile, the performance is examined based on the compression ratio before and after the enhancement.
1.8 Contributions of the Project

This project work has led to following outcomes:

1. The LZSS data compression has been introduced and explored as a learning process for the data compression technique.

2. Huffman coding has been proposed as an enhancement for the LZSS data compression.

3. This project also leads to exploration of between Huffman coding and LZSS data compression. The compression ratio before and after enhancement is compared and verified.

1.9 Organization of Project

This project is organized into 6 chapters. The first chapter introduces the motivation, project background, problem statement, project objectives and scope of work. Project contribution is review before summarizing the organization of the project.

Literature background and literature review are provided in chapter two. Summary of the literature review is given to clarify the research rationale

Chapter three describes the project methodology. This chapter shows the workflow of project, verification methods and together with the description of tools and techniques required for this project.
Chapter four describes about design of the LZSS and Huffman coding including the algorithms, flow charts and block diagram of the overall system design.

Chapter five reports the simulation results of the design. The results are analyzed to validate the performance of the data compression engine.

Chapter six summarizes the project work and states all deliverables of the project. Recommendation for potential works all provided.
CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

Data compression background related to this work is presented in this chapter. Data compression algorithms and previous works are also described briefly in this chapter.

2.5 Data Compression Background

Data compression is the science and art of messaging data from a given source data in such a way as to obtain a simple representation with at most tolerable lost of fidelity [Nelson, 1996]. The source data can take many forms such as text, image, sound or any combination of all these types such as video. Data compression is popular for two reasons:

(1) People like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression seems useful because it delays this inevitability [David Salomon, 2007].

(2) People hate to wait a long time for data transfers. When sitting at the computer, waiting for a Web page to come in or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait [David Salomon, 2007].
There are two types of data compression in common, which are lossless compression and lossy compression [Ida Mengyi Pu, 2006]. Lossless compression is able to reconstruct exactly the original data from the compressed data. There is no data loss during the compression process. It is also known as reversible compression. Relatively, lossy compression allows some information loss during the compression process to exchange for better compression effectiveness. Lossless compression is suitable for textual files and executable files which require no information loss. While lossy compression is best for multimedia files such as image, video and audio because we need to discard noise from the source data.

Many compression algorithms have been developed since the introduced of the data compression. Most of these algorithms can be divided into two categories, statistical compression and dictionary-based compression. In the statistical
compression, we need a prior knowledge about the characteristic of the source data to generate predefined encoding table that will be reuse in the decompression phase. The encoding table has to be sent separately to the decompression controller. These kind of data compression algorithm (e.g., Huffman coding) are more efficient in compression and decompression as long as the symbol probability distribution are constant. In contrast to statistical compression, dictionary-based compression is universal data compression algorithm. In this case, data compression algorithm does not require a prior knowledge of source data [Rissanen, 1983]. It can learn the characteristic of the source data and generate a dictionary as a reference for the coded-word during the compression phase. Moreover, the dictionary is not necessary to sent to the decompression controller because it can be regenerated itself from the decompression algorithm during the decompression phase.

2.6 Data Compression Algorithm

The Lempel–Ziv (LZ) compression methods are among the most popular algorithms for lossless storage. Most of the widely used data compression algorithms are based on LZ compression [Lempel and Ziv 1977, 1978]. In the LZ compression, a dictionary is produced based on the previous seen symbols in the source data to detect the repeating symbols in the incoming data. Then the detected repeating symbols are replaced with smaller sized codeword. LZ compression is dictionary-based compression algorithm mentioned above. So, LZ compression and its variants are universal type of data compression algorithms. Lempel-Ziv-77 (LZ77) algorithm proposed by Abraham Lempel and Jacob Ziv in 1977 is the first LZ compression algorithm [Lempel and Ziv 1977].

Lempel-Ziv-Storer-Szymanski (LZSS) algorithm is derived from LZ77 which has better compression performance [Storer and Szymanski 1982]. Hence, we are using LZSS as the data compression model in this project. In general, the length in the LZ77/LZSS codeword is not uniform. A smaller length occurs more frequently, and this suggests that Huffman coding [Huffman 1952] can be used to encode further the codeword to yield additional saving in the compression. Huffman
coding is a variable length entropy encoding algorithm which attempts to reduce the amount of bits required to represent a string of symbols. Shorter length codeword replaces frequently used symbol while seldom used symbol will has longer codeword. In many practical applications, Huffman coding is usually applied after an encoding process that produce non-uniformly distributed codeword for better compression performance. Therefore, Huffman coding is proposed for enhancement of the data compression model in this project.

A high speed data compression and decompression cores were designed and developed by Universiti Teknologi Malaysia [Yeem Kah Meng, 2002]. The hardware was designed based on of LZSS algorithm and Huffman coding. It was designed as a parameterized module for easy configurability that can provide suitable compromise between constraints of hardware resources, processing speed and compression saving. An improved version data compression engine was introduced in 2007 by Universiti Teknologi Malaysia [Roslin bin Mohd Sabri, 2007] to compensate drawbacks such as portability to any logic programmable logic device and solved the abnormal behavior of the decompression processor core.

2.7 Application of Data Compression

Data compression is used when we need to reduce the redundancy of a given source data. Source data in this compacted form will enhance the data storage and data transmission. As a result, higher data transfer rate can be achieved [Hifn, 1998]. Hence, data compression is applicable in many fields. Lossy image compression is used in digital cameras to increase storage capacities with minimal degradation of picture quality. Lossless data compression is used in many applications. For example, it is used in the popular ZIP file format and in the Unix tool gzip. It is also often used as a component within lossy data compression technologies. Besides that, cryptosystems often use lossless data compression before encryption in for added security. An example of this application is proposed which using Huffman coding for data reduction before the encryption [Nilkesh Patra and Sila Siba Sankar, 2007].
Data compression is also playing a vital role in IC testing field. When the design of the IC become more complex, larger test data volume will be required. Result in more power consumption because of the larger test data volume. This situation may cause damage to the circuit-under-test. To solve this problem, test data compression is applied [Chandra, 2002].

2.8 Summary

Data compression is widely use in various applications especially in the field of data storage and high speed communication. Lossless and lossy are two main classes of data compression techniques that have its respective field of application. LZSS is use in this project because it is a widely use universal data compression algorithm with good compression ratio. Huffman coding is usually applied after an encoding process that produce non-uniformly distributed codeword for better compression performance. Hence, it is suitable to use for enhancement of data compression model in this project.

To conclude, background and application of data compression are covered in this chapter followed by some previous works so as motivation to the project.
CHAPTER 3

METHODOLOGIES, TOOLS AND TECHNIQUES

This chapter includes an overview of design methodology used in this project. Project procedures are discussed followed by tools and techniques.

3.4 Project Procedures

This project can be generally divided into two modeling parts which are LZSS compression and LZSS compression enhanced with Huffman Coding. This project is divided into two phases; the first phase is essentially the study of algorithm to be implemented and literature review of previous work. Second phase of the system consists of design implementation where the algorithm is to be implemented in C Language.
Firstly, the project is started by defining the objective from the problem statement. Then, the identification for scope of work is done. Next is the study and learning of tools and techniques required to facilitate the project. In this project, the tool used is Microsoft Visual Studio 2010 and the technique used is C Language Programming. The data compression engine is implemented into two models using C Language. One of the models is LZSS compression and the other is LZSS compression enhanced using Huffman Coding. Lastly, the performance analysis on compression ratio of the both models is done to verify the significant of the enhancement. Literature review and documentation are all along the project.

C codes packets of the LZSS and Huffman Coding compression algorithms are downloaded from the internet. These C codes packets are used to study the algorithms of LZSS and Huffman Coding compression. Besides that, they are also used to generate executable files for LZSS compression and Huffman Coding compression systems. Both executable files (LZSS.exe and Huffman.exe) are use in the experiment to calculate the compression ratios for both data compression models. Performance analysis is done based on the compression ratios. Textual file is used as
input source data in the experiment to get the compression ratio because both LZSS and Huffman Coding are Lossless compression algorithms.

3.5 Tool and Technique

Besides knowledge, tool and technique are essential to conduct this project. This section describes briefly the tool and technique used in the project.

3.5.1 Microsoft Visual Studio 2010

Microsoft Visual Studio 2010 is the latest version of programming tool developed by Microsoft. It is a powerful tool use for compiling and debugging software systems and applications that design in many programming languages. This tool is used in this project because it is easy to use and famous among the programmer. Debugging is done using this tool to motivate the learning process of compression algorithms. Executable files of compression engine used for experiment are also built by this tool.

3.6 Summary

Methodology and project workflow of the project are explained and verification method of the data compression’s performance is discussed in this chapter. Tool and technique required to accomplish this project are introduced and briefly explained.
CHAPTER 4

DESIGN OF DATA COMPRESSION ALGORITHM

The details of the LZSS and Huffman Coding data compression algorithms are presented in this chapter.

4.1 LZSS Algorithm

LZSS is a compression algorithm which based on the dictionary model, is one kind of improvement algorithm based on the LZ77 compression algorithm foundation. It attempts to replace a string of symbols with a shorter fixed length codeword which is a reference to a dictionary location for the same string. The codeword is a combination of the flag and (position, length) pair. The flag shows than whether a string is a literal or not. Position is the length between the match string and its reference string. Length is the length of the match string.

4.3.1 LZSS Compression Algorithm

At the beginning of the LZSS compression, the source data is stored to an array. This array will be a combination of dictionary and incoming source data differential by a position pointer. The position pointer is moved from the beginning to the end of the array during the compression. Data in front of the position pointer is
incoming source data. Where else, the data behind the position pointer is treated as a reference (dictionary) to the incoming source data.

As the position pointer move, data from the incoming source is compared with string in dictionary to find the longest match string. The longest match string is boundary by a predefined maximum match length. If a match string does not exist or its length is less than the minimum match length, it is output as a literal in the dictionary. Otherwise, the match string will be replaced by a fixed length LZSS codeword. After that, the position pointer will moving forward to the incoming source data which has not been compared. The matching process will be repeated until the position pointer reach the end of the array.

The LZSS compression algorithm can be simplify into the following steps:
1. Initialize the dictionary to a known value.
2. Read an uncoded string that is the length of the maximum allowable match.
3. Search for the longest matching string in the dictionary.
4. If a match is found greater than or equal to the minimum allowable match length:
   a. Write the encoded flag, then the offset and length to the encoded output.
   b. Otherwise, write the uncoded flag and the first uncoded symbol to the encoded output.
5. Shift a copy of the symbols written to the encoded output from the unencoded string to the dictionary.
6. Read a number of symbols from the uncoded input equal to the number of symbols written in Step 4.

![Diagram showing Dictionary and Incoming Source Data in Array](image-url)
7. Repeat from Step 3, until all the entire input has been encoded.

4.3.2 LZSS Decompression Algorithm

LZSS decompression is the reverse of the compression process. The process of decompression is simple than compression because there is no search and match process. If the encoded data is a literal, just output it to the restored data. If it is a LZSS codeword, original string is recovered by refer to the (position, length) pair in the codeword.

The LZSS decompression algorithm is summarize into the following steps:
1. Initialize the dictionary to a known value.
2. Read the encoded/not encoded flag.
3. If the flag indicates an encoded string:
   a. Read the encoded length and offset, then copy the specified number of symbols from the dictionary to the decoded output.
   b. Otherwise, read the next character and write it to the decoded output.
4. Shift a copy of the symbols written to the decoded output into the dictionary.
5. Repeat from Step 2, until all the entire input has been decoded.

4.2 Huffman Coding Algorithm

Huffman Coding is an entropy encoding algorithm. Codeword used in the compression is stored in a code table that has to be transferred to the decompression block as a reference in the decompression process. The code table content the information on the symbols exists in the source data and its probabilities and coded words. It is generated based on a binary tree of probability. Below are the steps taken to generate a binary tree:
1. Sort source outputs in decreasing order of their probabilities.
2. Merge the two least-probable outputs into a single output whose probability is the sum of the corresponding probabilities.
3. If the number of remaining outputs is more than 2, then go to step 1.
4. Arbitrarily assign 0 and 1 as codewords for the two remaining outputs.
5. If an output is the result of the merger of two outputs in a preceding step, append the current codeword with a 0 and a 1 to obtain the codeword of the preceding outputs and repeat step 5. If no output is preceded by another output in a preceding step, then stop.

During the compression process, each symbol in the source data is replaced with its codeword by referring to the code table. It is the reverse for decompression process. The encoded data is compare bit by bit with the codeword in the code table to recovered the symbol that represented by its codeword.

4.4 Summary

The detail of LZSS and Huffman Coding compression algorithms are presented in this chapter. LZSS compression and decompression algorithms have summarized into several steps. Formation of Huffman binary tree is explained through several steps. In the next chapter, result and performance analysis is presented.
CHAPTER 5

DESIGN VERIFICATION AND PERFORMANCE ANALYSIS

This chapter reports the results obtain on testing the C behavioral model of data compression before and after enhancement. It starts with the design verification, followed by performance analysis.

5.2 Design Verification

The design verification is done by conducting experiment on the designed data compression engine using both executable files of LZSS and Huffman Coding in the Command Prompt. The input data of the experiment is a text file with redundancy. Sizes and contents of input and output files are compared to verify the design.

Figure 5.1: Source data for verification
5.1.1 LZSS Compression Engine

Input file of this experiment is infile.txt with size of 8265 bytes. LZSS executable file (LZSS.exe) is switched into compression mode using Command Prompt to get the encoded file (encodefile.txt) of compression process with size of 2120 bytes. After that, LZSS.exe is switched into decompression mode to do decompression on the encoded file. Output file obtained from the decompression process is outfile.txt with size of 8265 bytes. There is a decreasing amount of the file size after compression process and both of the input and output files are identical. Hence, the design of LZSS compression engine is verified. Figure 5.1-5.3 shows the input file, encoded file and output file of the LZSS compression engine.

![Figure 5.2: Compressed data for LZSS compression](image1)

![Figure 5.3: Restored data for LZSS compression](image2)

5.1.2 Huffman Coding Compression Engine

Input file of this experiment is infile.txt with size of 8265 bytes. Huffman Coding executable file (Huffman.exe) is switched into compression mode using Command Prompt to get the encoded file (encodefile.txt) of compression process with size of 4151 bytes. After that, Huffman.exe is switched into decompression...
mode to do decompression on the encoded file. Output file obtained from the decompression process is outfile.txt with size of 8265 bytes. There is a decreasing amount of the file size after compression process and both of the input and output files are identical. Hence, the design of Huffman Coding compression engine is verified. Figure 5.1, 5.4 and 5.5 shows the input file, encoded file and output file of the Huffman Coding engine.

![Figure 5.4: Compressed data for Huffman Coding compression](image1)

There is a decreasing amount of the file size after compression process and both of the input and output files are identical. Hence, the design of Huffman Coding compression engine is verified. Figure 5.1, 5.4 and 5.5 shows the input file, encoded file and output file of the Huffman Coding engine.

![Figure 5.5: Restored data for Huffman Coding compression](image2)

5.1.3 LZSS Compression Enhancement with Huffman Coding

Input file of this experiment is infile.txt with size of 8265 bytes. The input file is compressed using LZSS compressor followed by Huffman Coding compressor to obtain the encoded file (encodefile.txt) with size of 1848 bytes. Then, output file (outfile.txt) with size of 8265 bytes is obtained after applying Huffman Coding decompression followed by LZSS decompression to the encoded file. There is a decreasing amount of the file size after compression process and both of the input and output files are identical. Hence, the design of enhanced LZSS compression engine is verified. Figure 5.1, 5.6 and 5.7 shows the input file, encoded file and output file of the enhanced LZSS compression engine.
5.2 Performance Analysis

The performance analysis of this project is compression ratio comparison between LZSS, Huffman Coding and enhanced LZSS compression engines. Compression ratio is the ratio of size difference before and after compression to the size before compression.

\[
\text{Compression Ratio} = \frac{\text{Size before compression} - \text{Size after compression}}{\text{Size before compression}} \times 100\%
\]

Table 5.1: Compression ratio comparison

<table>
<thead>
<tr>
<th></th>
<th>Compression Ratio, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZSS</td>
<td>73.35</td>
</tr>
<tr>
<td>Huffman Coding</td>
<td>49.78</td>
</tr>
<tr>
<td>Enhanced LZSS</td>
<td>77.64</td>
</tr>
</tbody>
</table>
From the Table 5.1, better compression ratio is obtained if enhanced LZSS is used in the compression. These compression ratios are based on the same input file (infile.txt).

5.3 Summary

Results of design verification and performance analysis are presented in this chapter. LZSS, Huffman Coding and enhanced LZSS compression engines are verified to function correctly and as expected. In performance analysis, the compression ratio is better when enhanced LZSS compression engine is used. In the next chapter, the conclusion of this project is presented.
CHAPTER 6

CONCLUSIONS

This chapter concludes the thesis by summarizing the research findings, which followed by suggestion future work.

6.1 Concluding Remarks

Data storage and data transmission cost money. Moreover, there often have redundancies in the data that we used. To solve these problems, data compression has to be applied. LZSS is a widely use universal data compression algorithm. Huffman Coding is usually used after LZSS encoding to get a better compression saving. Therefore, LZSS and Huffman Coding are used in this project to implement a better data compression engine.

Based on the performance analysis in the previous chapter, it is proofed that enhanced LZSS compression offers better compression saving. The enhancement using Huffman Coding algorithm is successful.
6.2 Recommendation for Future Work

Future enhancement can be done based on the work in this thesis.

Verification should not be limited to text files. Other types of source data such as, PDF file, executable file, word file and programming code can be included in the verification to have a more accurate and trustable result.

Compression time should be included in the performance analysis. Compression time is another measure of the compression quality. Speed is an important issue that people concern when involving data transmission.
REFERENCES


APPENDIX A

EXAMPLE OF LZSS COMPRESSION ALGORITHM

This appendix explains how LZSS compression and decompression works by providing simple example.

A.1 LZSS Encoding Example

1. Input Data = "ABCDABCA"
   - Let’s start at the first byte "A". Have this symbol seen before? No. Encode it as a literal byte. "0" (to indicate literal) followed by the data "A"
   Output data = 0 01000001 (0"A")

2. Input Data = "ABCDABCA"
   - Next byte is "B". Have this symbol seen before? No. Encode it as literal
   Output data = 0 01000001(0"A") 0 01000010(0"B")

3. Input Data = "ABCDABCA"
   - Next byte is "C" Have this symbol seen before? No. Encode it as literal
   Output data = 0 01000001(0"A") 0 01000010(0"B") 0 01000011(0"C")

4. Input Data = "ABCDABCA"
- Next byte is "D" Have this symbol been seen before? No. Encode it as literal
Output data = 0 01000001(0"A") 0 01000010(0"B") 0 01000011(0"C")
0 01000100(0"D")

5. Input Data = "ABCDABCA"

- Next byte is "A" Have this symbol been seen before? YES. OK, when did this symbol appears previous to this? It was 4 bytes ago (offset=4). OK, how similar is the data 4 bytes ago to the current data? Well, three bytes are the same "ABC". So have encode this as "1" (to indicate a match" followed by (4, 3).
Output data = 0 01000001(0"A") 0 01000010(0"B") 0 01000011(0"C")
0 01000100(0"D") 1 0100 0011 (1 43)

6. Input Data = "ABCDABCA"

- Next byte is "A" Have this symbol been seen before? YES. 3 bytes ago. But this time only 1 byte can be matched. Encode it as "1" followed by (3, 1).
Output data = 0 01000001(0"A") 0 01000010(0"B") 0 01000011(0"C")
0 01000100(0"D") 1 0100 0011 (1 43) 1 0011 0001 (1 31)

A.2 LZSS Encoding Example (with Minimum Match Length Limitation)
Assume the input data is “AABBCBBAABC” because this example is able to show the LZSS algorithm with match length limitation. Again, let * be the data position. In this case, minimum match length is 2 bytes hence matches that are less than 2 bytes will not be represent in (position, length) pair.

1. Input Data = "AABBCBBAABC"

- Let’s start at the first byte "A". Have this symbol seen before? No. Encode it as a literal byte. "0" (to indicate literal) followed by the data "A"
Output data = 0 01000001 (0"A")
2. Input Data = "AABBCBBAABC "

- Next byte is "A". Have this symbol seen before? Yes, but it can only copy 1 byte. The match length is less than minimum march length (2). So encode it as literal
  Output data = 0 01000001(0"A") 0 01000001(0"A")

3. Input Data = "AABBCBBAABC "

- Next byte is "B". Have this symbol seen before? No. Encode it as literal
  Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B")

4. Input Data = "AABBCBBAABC "

- Next byte is "B". Have this symbol seen before? Yes, but it can only copy 1 byte. The match length is less than minimum march length (2). So encode it as literal
  Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B") 0 01000010(0"B")

5. Input Data = "AABBCBBAABC "

- Next byte is "C". Have this symbol seen before? No. Encode it as literal
  Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B") 0 01000010(0"B") 0 01000011(0"C")

6. Input Data = "AABBCBBAABC "

- Next byte is "B". Have this symbol seen before? Yes. Where has this symbol seen before? 3 bytes before the current position. It can copy 2 bytes from that position. Since the match length is 2 which is the minimum match length. It fulfills the requirement. Encode it as (position, length) pair.
Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B")
0 01000010(0"B") 0 01000011(0"C") 1 00110010(3, 2)
The data position is thus increased by 2 bytes instead by 1 bytes.

7. Input Data = "AABBCBBAABC"
- Next byte is "A". Have this symbol seen before? Yes. Where has this symbol
seen before? 7 bytes before the current position. It can copy 3 bytes from that
position. Since the match length is 3 which is more than the minimum match
length. It fulfills the requirement. Encode it as (position, length) pair.
Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B")
0 01000010(0"B") 0 01000011(0"C") 1 00110010(3, 2) 1 01110011(7, 3)
The data position is thus increased by 3 bytes instead by 1 bytes.

8. Input Data = "AABBCBBAABC"
- Next byte is "C". Have this symbol seen before? Yes, but it can only copy 1
bytes. The match length is less than minimum march length (2). So encode it
as literal
Output data = 0 01000001(0"A") 0 01000001(0"A") 0 01000010(0"B")
0 01000010(0"B") 0 01000011(0"C") 1 00110010(3, 2) 1 01110011(7, 3) 0 01000011(0"C")
A.3 LZSS Decoding Example

Explanation of LZSS decompression algorithm

The compressed data which obtained from above is $0"A"\ 0"B"\ 0"C"\ 0"D"\ 1(4,\ 3)\ 1(3,\ 1)$. Now let decode it or decompress it using LZSS decompression algorithm. Let * indicate the data position.

```
<table>
<thead>
<tr>
<th>'A'</th>
<th>'B'</th>
<th>'C'</th>
<th>'D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10000001</td>
<td>0 10000010</td>
<td>0 10000011</td>
<td>0 1000100</td>
</tr>
</tbody>
</table>
```

The data position is now point to 0, which mean that it is a LZSS Literal. Hence, take one byte of data and put it into output file as shown below.

01000001

Let’s check the next byte if it is LZSS literal or LZSS (Offset, Length) pair.

```
<table>
<thead>
<tr>
<th>'A'</th>
<th>'B'</th>
<th>'C'</th>
<th>'D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10000001</td>
<td>0 10000010</td>
<td>0 10000011</td>
<td>0 1000100</td>
</tr>
</tbody>
</table>
```

Since it is 0, hence it is LZSS literal. Hence take one byte and put into output file as shown below.

01000001 01000010

Let’s check the next byte if it is LZSS literal or LZSS (Offset, Length) pair.

```
<table>
<thead>
<tr>
<th>'A'</th>
<th>'B'</th>
<th>'C'</th>
<th>'D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10000001</td>
<td>0 10000010</td>
<td>0 10000011</td>
<td>0 1000100</td>
</tr>
</tbody>
</table>
```

Since it is 0, hence it is LZSS literal. Hence take one byte and put into output file as shown below.

01000001 01000010 01000011

Let’s check the next byte if it is LZSS literal or LZSS (Offset, Length) pair.
Since it is 0, hence it is LZSS literal. Hence take one byte and put into output file as shown below.

01000001 01000010 01000011 01000100

Let’s check the next byte if it is LZSS literal or LZSS (Offset, Length) pair.

Hey, it is 1! Hence it is LZSS (Offset, Length) pair. Now go back 4 bytes (offset), and take a length, 3 bytes (Length) then put into output file.

01000001 01000010 01000011 01000100 01000001 01000010 01000011 01000001

Let’s check the next byte if it is LZSS literal or LZSS (Offset, Length) pair.

Again, it is 1! Hence it is LZSS (Offset, Length) pair. Now go back 3 bytes (offset), and take a length, 1 bytes (Length) then put into output file.

01000001 01000010 01000011 01000100 01000001 01000010 01000011 01000001 01000010 01000001

Or “ABCDABCA”

Original data is obtained.
APPENDIX B

LZSS SOURCE CODE IN C

lzss_compress.cpp
// (c)2002 Jonathan Bennett (jon@hiddensoft.com)
#include <stdio.h>
#include <windows.h>
#include "lzss.h"

int HS_LZSS::Compress(unsigned char *bData, unsigned char *bCompressedData,
unsigned long nDataSize, unsigned long *nCompressedSize)
{
    unsigned int nOffset, nLen;
    // Set up initial values
    m_nDataStreamPos = 0;       // We are at the start of the input data
    m_nCompressedStreamPos = 0; // We are at the start of the compressed data
    m_nCompressedLong = 0;      // Compressed stream temporary 32bit value
    m_nCompressedBitsUsed = 0;  // Number of bits used in temporary value
    m_bCompressedData = bCompressedData; // Pointer to our output buffer
    m_bData = bData;            // Pointer to our input buffer
    m_nDataSize = nDataSize;    // Store the size of our input data
    m_nCompressedSize = 0;      // No compressed data yet
    // If the input file is too small then there is a chance of
    // buffer overrun, so just abort
    if (m_nDataSize < HS_LZSS_MINDATASIZE)
        return HS_LZSS_E_BADCOMPRESS;
    // Write out our file header info so the decompressor doesn't have to
    // guess about things like uncompressed buffer sizes
    // LZSS ID (4 bytes)
    CompressedStreamWriteBits('L', 8);
    CompressedStreamWriteBits('Z', 8);
    CompressedStreamWriteBits('S', 8);
    CompressedStreamWriteBits('S', 8);
    // Uncompressed size (4 bytes)
    CompressedStreamWriteBits( (m_nDataSize >> 16) & 0x0000ffff, 16);
    CompressedStreamWriteBits( m_nDataSize & 0x0000ffff, 16);
    // Window and match bits (not yet implemented)
    CompressedStreamWriteBits( HS_LZSS_WINDOWBITS, 8);
    CompressedStreamWriteBits( HS_LZSS_MATCHBITS, 8);
    // Loop around until there is no more data
    while (m_nDataStreamPos < m_nDataSize)
    {
        if ( (m_nCompressedSize >= (m_nDataSize - 16)) )
            return HS_LZSS_E_BADCOMPRESS;
     // Search for matches
        if ( FindMatches(&nOffset, &nLen) == true )
        {
            // A match was found, write it out
            CompressedStreamWriteBits(HS_LZSS_MATCH, 1);
            CompressedStreamWriteBits(nOffset, HS_LZSS_WINDOWBITS);
            CompressedStreamWriteBits(nLen, HS_LZSS_MATCHBITS);
        }
        else
        {
            // No matches were found, just write out the literal byte
            CompressedStreamWriteBits(HS_LZSS_LITERAL, 1);
            CompressedStreamWriteBits(m_bData[m_nDataStreamPos++], 8);
        }
    }
}
void HS_LZSS::CompressedStreamWriteBits(unsigned int nValue, unsigned int nNumBits)
{
  if (nNumBits == 0)
  {
    m_nCompressedLong = m_nCompressedLong << (32 - m_nCompressedBitsUsed);
    m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)(m_nCompressedLong >> 24);
    m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)(m_nCompressedLong >> 16);
    m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)(m_nCompressedLong >> 8);
    m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)m_nCompressedLong;
    m_nCompressedSize = m_nCompressedSize + 4;
    m_nCompressedBitsUsed = 0;
    return;
  }
  while (nNumBits > 0)
  {
    nNumBits--;
    m_nCompressedLong = m_nCompressedLong << 1;
    m_nCompressedLong = m_nCompressedLong | (long)((nValue >> nNumBits) & 0x00000001);
    m_nCompressedBitsUsed++;
    if (m_nCompressedBitsUsed == 32)
    {
      m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)(m_nCompressedLong >> 24);
      m_bCompressedData[m_nCompressedStreamPos++] = (unsigned char)(m_nCompressedLong >> 16);
      m_nCompressedSize = m_nCompressedSize + 2;
      m_nCompressedBitsUsed = m_nCompressedBitsUsed - 16;
    }
  }
}

bool HS_LZSS::FindMatches(unsigned int *nOffset, unsigned int *nLen)
{
  unsigned long nTempWPos, nWPos, nDPos; // Temp Window and Data position markers
  unsigned long nMaxWPos, nDataStreamPos, nDataSize; // Pure temp values for speeding up loops
  unsigned long nTempLen; // Temp vars
  unsigned long nBestOffset, nBestLen; // Stores the best match so far
  unsigned char *bData; // Reset all variables
  nBestOffset = 0;
  nBestLen = HS_LZSS_MINMATCHLEN - 1;
  if (m_nDataStreamPos < HS_LZSS_MINMATCHLEN)
    return false; // No matches
  if (m_nDataStreamPos < HS_LZSS_WINDOWLEN)
    nWPos = 0;
  else
    nWPos = m_nDataStreamPos - HS_LZSS_WINDOWLEN;
  // Store some temp values to speed up the loop (a little)
  nDataStreamPos = m_nDataStreamPos;
  nMaxWPos = nDataStreamPos - HS_LZSS_MINMATCHLEN;
  bData = m_bData;
  nDataSize = m_nDataSize;
  // Main loop
  while (nWPos <= nMaxWPos)
  {
    if (bData[nDataStreamPos] == bData[nWPos])
    {
      nTempWPos = nWPos;
      nDPos = nDataStreamPos;
      nTempLen = 0;
      while (bData[nTempWPos] == bData[nDPos]) &
      {
        nTempWPos++;
        nTempLen++;
        nDPos++;
      } // End while
      // See if this match was better than previous match
      if (nTempLen > nBestLen)
      {
if (nTempLen >= HS_LZSS_MATCHLEN)
{
    nTempLen = HS_LZSS_MATCHLEN;
    nBestLen = nTempLen;
    nBestOffset = nDataStreamPos - nWPos;
    break;  // Force the while loop to end
}
    nBestLen = nTempLen;
    nBestOffset = nDataStreamPos - nWPos;
}
} // End if
nWPos++;
} // End while
if ( (nBestOffset == 0) )
    return false;  // No match
else
{
    m_nDataStreamPos = m_nDataStreamPos + nBestLen;
    *nOffset = nBestOffset;
    *nLen = nBestLen;
    return true;  // Match! :)
}
} // FindMatches()

lzss_uncompress.cpp
// (c)2002 Jonathan Bennett (jon@hiddensoft.com)
#include <stdio.h>
#include <windows.h>
#include "lzss.h"

void HS_LZSS::CompressedStreamReadBits(unsigned int *nValue, unsigned int nNumBits)
{
    unsigned long nTemp;
    // Ensure that the high order word of our bit buffer is blank
    m_nCompressedLong = m_nCompressedLong & 0x000ffff;
    while (nNumBits > 0)
    {
        nNumBits--;
        // Check if we need to refill our decoding bit buffer
        if (m_nCompressedBitsUsed == 0)
        {
            // Fill the low order 16 bits of our long buffer
            nTemp = (unsigned long)m_bCompressedData[m_nCompressedStreamPos++];
            m_nCompressedLong = m_nCompressedLong | (nTemp << 8);
            nTemp = (unsigned long)m_bCompressedData[m_nCompressedStreamPos++];
            m_nCompressedLong = m_nCompressedLong | nTemp;
            m_nCompressedBitsUsed = 16;
        }
        // Shift the data into the high part of the long
        m_nCompressedLong = m_nCompressedLong << 1;
        m_nCompressedBitsUsed--;
    }
    *nValue = (unsigned int)(m_nCompressedLong >> 16);
} // CompressedStreamReadBits()

int HS_LZSS::GetUncompressedSize(unsigned char *bCompressedData, unsigned long *nUncompressedSize)
{
    unsigned int nTemp;
    char szAlgID[4+1];
    // Set up initial values
    m_nCompressedStreamPos = 0;  // We are at the start of the compressed data
    m_nCompressedLong = 0;  // Compressed stream temporary 32bit value
    m_nCompressedBitsUsed = 0;  // Number of bits used in temporary value
    m_bCompressedData = bCompressedData;  // Pointer to our input buffer
    // Check that data is a valid LZSS stream and return the length of it
    if (strstr(szAlgID, "LZSS") )
        return HS_LZSS_E_NOTLZSS;  // Wasn’t a valid LZSS stream

    CompressedStreamReadBits(&nTemp, 8);  //szAlgID[0] = (char)nTemp;
    CompressedStreamReadBits(&nTemp, 8);  //szAlgID[1] = (char)nTemp;
    CompressedStreamReadBits(&nTemp, 8);  //szAlgID[2] = (char)nTemp;
    CompressedStreamReadBits(&nTemp, 8);  //szAlgID[3] = (char)nTemp;
    szAlgID[4] = 0;
    if ( strncmp(szAlgID, "LZSS") )
        return HS_LZSS_E_NOTLZSS;  // Wasn’t a valid LZSS stream

    nTemp = (unsigned int)CompressedStreamReadBits(&nTemp, 8);  //szAlgID[5] = (char)nTemp;
    nTemp = (unsigned int)CompressedStreamReadBits(&nTemp, 8);  //szAlgID[6] = (char)nTemp;
    if ( strncmp(szAlgID, "LZSS") )
        return HS_LZSS_E_NOTLZSS;  // Wasn’t a valid LZSS stream

    nTemp = (unsigned int)CompressedStreamReadBits(&nTemp, 8);  //szAlgID[7] = (char)nTemp;
    *nUncompressedSize = nTemp;
} // GetUncompressedSize()
// Uncompressed size (4 bytes)
CompressedStreamReadBits( &nTemp, 16);
*nUncompressedSize = ((unsigned long)nTemp) << 16;
CompressedStreamReadBits( &nTemp, 16);
*nUncompressedSize = *nUncompressedSize | (unsigned long)nTemp;
return HS_LZSS_E_OK;
} // GetUncompressedSize()

int HS_LZSS::Uncompress(unsigned char *bCompressedData, unsigned char *bData)
{
    unsigned int nTemp;
    unsigned int nOffset, nLen;
    unsigned long nTempPos;
    // Set up initial values
    m_nDataStreamPos = 0; // We are at the start of the input data
    m_nCompressedStreamPos = 0; // We are at the start of the compressed data
    m_nCompressedLong = 0; // Compressed stream temporary 32bit value
    m_nCompressedBitsUsed = 0; // Number of bits used in temporary value
    m_bCompressedData = bCompressedData; // Pointer to our input buffer
    m_bData = bData; // Pointer to our output buffer
    m_nDataSize = 0; // We will get this from the input
    // Skip the LZSS alg ID (4 bytes)
    CompressedStreamReadBits(&nTemp, 16);
    CompressedStreamReadBits(&nTemp, 16);
    // Get the uncompressed size (4 bytes)
    CompressedStreamReadBits( &nTemp, 16);
    m_nDataSize = ((unsigned long)nTemp) << 16;
    CompressedStreamReadBits( &nTemp, 16);
    m_nDataSize = m_nDataSize | (unsigned long)nTemp;
    // Skip the window and bit lengths (not yet implemented) 2 bytes
    CompressedStreamReadBits( &nTemp, 16);
    // Perform decompression until we fill our predicted buffer
    while(m_nDataStreamPos < m_nDataSize)
    {
        // Read in the 1 bit flag
        CompressedStreamReadBits(&nTemp, 1);
        // Was it a literal byte, or a (offset,len) match pair?
        if (nTemp == HS_LZSS_MATCH)
        {
            // Read the offset and length
            CompressedStreamReadBits(&nOffset, HS_LZSS_WINDOWBITS);
            CompressedStreamReadBits(&nLen, HS_LZSS_MATCHBITS);
            // Write out our match
            nTempPos = m_nDataStreamPos - nOffset;
            while (nLen > 0)
            {
                nLen--;
                m_bData[m_nDataStreamPos++] = m_bData[nTempPos++];
            }
        }
        else
        {
            // Output a literal byte
            CompressedStreamReadBits(&nTemp, 8);
            m_bData[m_nDataStreamPos++] = (unsigned char)nTemp;
        }
    }
    return HS_LZSS_E_OK;
} // Uncompress()

main.cpp
#include <stdio.h>
#include <windows.h>
#include "lzss.h"
int main(int argc, char* argv[])
{
    HS_LZSS oLZSS; // Our compression class
    FILE *fp;
    unsigned char *bmyData;
    unsigned char *bmyCompressedData;
    unsigned long nCompressedSize;
    unsigned long nUncompressedSize;
    int nRes;
    if (argc != 4)
printf("Usage: LZSS <c|u> <infile> <outfile>\n");
return 0;

// Compress function
if (!strcmp("c", argv[1]))
{
    if ( (fptr = fopen(argv[2], "rb")) == NULL)
    {
        printf("Error opening input file.\n");
        return 0;  // Error
    }
    fseek(fptr, 0, SEEK_END);
    nUncompressedSize = ftell(fptr);
    printf("Input file size : %d\n", nUncompressedSize);
    bmyData = (unsigned char *)malloc(nUncompressedSize);
    bmyCompressedData = (unsigned char *)malloc(nUncompressedSize);
    fseek(fptr, 0, SEEK_SET);
    fread(bmyData, sizeof(unsigned char), nUncompressedSize, fptr);
    fclose(fptr);
    nRes = oLZSS.Compress(bmyData, bmyCompressedData, nUncompressedSize, &nCompressedSize);
    if (nRes != HS_LZSS_E_OK)
    {
        printf("File not worth compressing.\n");
        return 0;
    }
    if ( (fptr = fopen(argv[3], "w+b")) == NULL)
    {
        printf("Error opening output file.\n");
        free(bmyData);
        free(bmyCompressedData);
        return 0;  // Error
    }
    fwrite(bmyCompressedData, sizeof(unsigned char), nCompressedSize, fptr);
    fclose(fptr);
    printf("Output file size: %d\n", nCompressedSize);
    printf("Compression ratio: %f\%
", 100 - ((float)nCompressedSize / (float)nUncompressedSize)*100.0));
    free(bmyData);
    free(bmyCompressedData);
    return 0;
}

// Uncompress function
if (!strcmp("u", argv[1]))
{
    if ( (fptr = fopen(argv[2], "rb")) == NULL)
    {
        printf("Error opening input file.\n");
        return 0;  // Error
    }
    fseek(fptr, 0, SEEK_END);
    nCompressedSize = ftell(fptr);
    printf("Input file size : %d\n", nCompressedSize);
    bmyCompressedData = (unsigned char *)malloc(nCompressedSize);
    fseek(fptr, 0, SEEK_SET);
    fread(bmyCompressedData, sizeof(unsigned char), nCompressedSize, fptr);
    fclose(fptr);
    nRes = oLZSS.GetUncompressedSize(bmyCompressedData, &nUncompressedSize);
    if (nRes != HS_LZSS_E_OK)
    {
        printf("Error not a valid LZSS file.\n");
        return 0;
    }
    bmyData = (unsigned char *)malloc(nUncompressedSize);
    oLZSS.Uncompress(bmyCompressedData, bmyData);
    if ( (fptr = fopen(argv[3], "w+b")) == NULL)
    {
        printf("Error opening output file.\n");
        free(bmyData);
        free(bmyCompressedData);
        return 0;  // Error
    }
    fwrite(bmyData, sizeof(unsigned char), nUncompressedSize, fptr);
    fclose(fptr);
    printf("Output file size: %d\n", nUncompressedSize);
free(bmyData);
free(bmyCompressedData);
}

return 0;

}

LZSS.h
#endif __HS_LZSS_H
#define __HS_LZSS_H

// Error codes
#define HS_LZSS_E_OK 0 // OK
#define HS_LZSS_E_BADCOMPRESS 1 // Compressed file would be bigger than source!
#define HS_LZSS_E_NOTLZSS 2 // Not a valid LZSS data stream

// Stream flags
#define HS_LZSS_LITERAL 0 // Just output the literal byte
#define HS_LZSS_MATCH 1 // Output a (offset, len) match pair

// Compression options, these will be user selectable in a later version
#define HS_LZSS_WINDOWLEN 1023 // Sliding window size (10 bits, 0-1023)
#define HS_LZSS_WINDOWBITS 10 // Num of bits to this is
#define HS_LZSS_MATCHLEN 7 // Maximum size of match (3 bits, 0-7)
#define HS_LZSS_MATCHBITS 3 // Num of bits that this is
#define HS_LZSS_MINMATCHLEN 3 // Minimum match size (3 bytes or not efficient)
#define HS_LZSS_MINDATASIZE 32 // Arbitrary minimum data size that we should attempt to compress

class HS_LZSS
{
public:

    // Functions
    int Compress(unsigned char *bData, unsigned char *bCompressedData,
                 unsigned long nDataSize, unsigned long *nCompressedSize);
    int Uncompress(unsigned char *bCompressedData, unsigned char *bData);
    int GetUncompressedSize(unsigned char *bCompressedData, unsigned long *nUncompressedSize);

private:

    // Variables
    unsigned long m_nDataStreamPos; // Current position in the data stream
    unsigned long m_nCompressedStreamPos; // Current position in the compressed stream
    unsigned char *m_bData;
    unsigned char *m_bCompressedData;
    unsigned long m_nDataSize; // The size of our uncompressed data
    unsigned long m_nCompressedSize; // The size of our compressed data

    // Temporary variables used for the bit operations
    unsigned long m_nCompressedLong; // Compressed stream temporary 32bit value
    int m_nCompressedBitsUsed; // Number of bits used in temporary value

    // Functions
    bool FindMatches(unsigned int *nOffset, unsigned int *nLen); // Searches for pattern matches
    // Bit operation functions
    void CompressedStreamWriteBits(unsigned int nValue, unsigned int nNumBits);
    void CompressedStreamReadBits(unsigned int *nValue, unsigned int nNumBits);

};
#endif
APPENDIX C

HUFFMAN CODING SOURCE CODE IN C

<optlist.h>
#ifndef OPTLIST_H
#define OPTLIST_H

#define OL_NOINDEX -1
typedef struct option_t
{
    char option;
    char *argument;
    int argIndex;
    struct option_t *next;
} option_t;

option_t *GetOptList(int argc, char *const argv[], char *const options);
void FreeOptList(option_t *list);
#endif /* ifndef OPTLIST_H */

<optlist.cpp>
#include "optlist.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

option_t *MakeOpt(const char option, char *const argument, const int index);
option_t *GetOptList(const int argc, char *const argv[], char *const options)
{
    int nextArg;
    option_t *head, *tail;
    int optIndex;
    nextArg = 1;
    head = NULL;
    tail = NULL;
    while (nextArg < argc)
    {
        if ((strlen(argv[nextArg]) > 1) && ('-' == argv[nextArg][0]))
        {
            optIndex = 0;
            while ((options[optIndex] != '0')
                && (':' == options[optIndex]))
            {
                do
                {
                    optIndex++;
                } while ((options[optIndex] != '0')
                    && (':' == options[optIndex]));
            }
            if (options[optIndex] == argv[nextArg][1])
            {
                if (NULL == head)
                {
                    head = MakeOpt(options[optIndex], NULL, OL_NOINDEX);
                }
                else
                {
                    tail = MakeOpt(options[optIndex], tail, OL_NOINDEX);
                }
            }
        }
        nextArg++;
    }
    return head;
}

void FreeOptList(option_t *list)
{
    option_t *ptr;
    while (list != NULL)
    {
        ptr = list;
        list = list->next;
        free(ptr);
    }
}

option_t *MakeOpt(const char option, char *const argument, const int index)
{
    option_t *opt = (option_t *)malloc(sizeof(option_t));
    opt->option = option;
    opt->argument = argument;
    opt->argIndex = index;
    opt->next = NULL;
    return opt;
}

option_t *GetOptList(int argc, char *const argv[], char *const options)
{
    int nextArg;
    option_t *head, *tail;
    int optIndex;
    nextArg = 1;
    head = NULL;
    tail = NULL;
    while (nextArg < argc)
    {
        if ((strlen(argv[nextArg]) > 1) && ('-' == argv[nextArg][0]))
        {
            optIndex = 0;
            while ((options[optIndex] != '0')
                && (':' == options[optIndex]))
            {
                do
                {
                    optIndex++;
                } while ((options[optIndex] != '0')
                    && (':' == options[optIndex]));
            }
            if (options[optIndex] == argv[nextArg][1])
            {
                if (NULL == head)
                {
                    head = MakeOpt(options[optIndex], NULL, OL_NOINDEX);
                }
                else
                {
                    tail = MakeOpt(options[optIndex], tail, OL_NOINDEX);
                }
            }
        }
        nextArg++;
    }
    return head;
}

void FreeOptList(option_t *list)
{
    option_t *ptr;
    while (list != NULL)
    {
        ptr = list;
        list = list->next;
        free(ptr);
    }
}

option_t *MakeOpt(const char option, char *const argument, const int index)
{
    option_t *opt = (option_t *)malloc(sizeof(option_t));
    opt->option = option;
    opt->argument = argument;
    opt->argIndex = index;
    opt->next = NULL;
    return opt;
}
tail = head;
}
else
{
    tail->next = MakeOpt(options[optIndex], NULL, OL_NOINDEX);
    tail = tail->next;
}
if ('=' == options[optIndex + 1])
{
    if (strlen(argv[nextArg]) > 2)
    {
        tail->argument = &argv[nextArg][2];
        tail->argIndex = nextArg;
    }
    else if (nextArg < argc)
    {
        nextArg++;
        tail->argument = argv[nextArg];
        tail->argIndex = nextArg;
    }
}
}
nextArg++;
}
return head;
}

option_t *MakeOpt(const char option, char *const argument, const int index)
{
    option_t *opt;
    opt = malloc(sizeof(option_t));
    if (opt != NULL)
    {
        opt->option = option;
        opt->argument = argument;
        opt->argIndex = index;
        opt->next = NULL;
    }
    else
    {
        perror("Failed to Allocate option_t");
    }
    return opt;
}

void FreeOptList(option_t *list)
{
    option_t *head, *next;
    head = list;
    list = NULL;
    while (head != NULL)
    {
        next = head->next;
        free(head);
        head = next;
    }
    return;
}

<huflocal.h>
#ifndef _HUFFMAN_LOCAL_H
#define _HUFFMAN_LOCAL_H
#include <limits.h>
#if (UCHAR_MAX != 0xFF)
#error This program expects unsigned char to be 1 byte
#endif
#if (UINT_MAX != 0xFFFFFFFF)
#error This program expects unsigned int to be 4 bytes
#endif
/* system dependent types */
typedef unsigned char byte_t;       /* unsigned 8 bit */
typedef unsigned int count_t;       /* unsigned 32 bit for character counts */
typedef struct huffman_node_t
Huffman Node Structure:

```c
struct huffman_node_t {
    int value; /* character(s) represented by this entry */
    count_t count; /* number of occurrences of value (probability) */
    char ignore; /* TRUE -> already handled or no need to handle */
    int level; /* depth in tree (root is 0) */
    struct huffman_node_t *left, *right, *parent;
};
```

Huffman Node Definitions:

```c
#define FALSE 0
#define TRUE 1
#define NONE -1
#define COUNT_T_MAX UINT_MAX /* based on count_t being unsigned int */
#define COMPOSITE_NODE -1 /* node represents multiple characters */
#define NUM_CHARS 257 /* 256 bytes + EOF */
#define EOF_CHAR (NUM_CHARS - 1) /* index used for EOF */
#define max(a, b) ((a)>(b)?(a):(b))
```

Huffman Tree Generation:

```c
huffman_node_t *GenerateTreeFromFile(FILE *inFile)
{
    huffman_node_t *huffmanTree; /* root of huffman tree */
    int c;
    for (c = 0; c < NUM_CHARS; c++)
    {
        if ((huffmanArray[c] = AllocHuffmanNode(c)) == NULL)
        {
            for (c--; c >= 0; c--)
                free(huffmanArray[c]);
            return NULL;
        }
        huffmanArray[c].count = 1;
        huffmanArray[c].ignore = FALSE;
    }
    huffmanTree = BuildHuffmanTree(huffmanArray, NUM_CHARS);
    return huffmanTree;
}
```

Huffman Node Allocation:

```c
huffman_node_t *AllocHuffmanNode(int value)
{
    huffman_node_t *ht;
    ht = (huffman_node_t *)(malloc(sizeof(huffman_node_t)));
    if (ht != NULL)
    {
        ht->value = value;
        return ht;
    }
```
ht->ignore = TRUE; /* will be FALSE if one is found */
/* at this point, the node is not part of a tree */
ht->count = 0;
ht->level = 0;
ht->left = NULL;
ht->right = NULL;
ht->parent = NULL;
}
else
{
    perror("Allocate Node");
    return ht;
}

static huffman_node_t *AllocHuffmanCompositeNode(huffman_node_t *left,
huffman_node_t *right)
{
    huffman_node_t *ht;
    ht = (huffman_node_t *)(malloc(sizeof(huffman_node_t)));
    if (ht != NULL)
    {
        ht->value = COMPOSITE_NODE;  /* represents multiple chars */
        ht->ignore = FALSE;
        ht->count = left->count + right->count;  /* sum of children */
        ht->level = max(left->level, right->level) + 1;
        /* attach children */
        ht->left = left;
        ht->left->parent = ht;
        ht->right = right;
        ht->right->parent = ht;
        ht->parent = NULL;
    }
    else
    {
        perror("Allocate Composite");
        return NULL;
    }
    return ht;
}

void FreeHuffmanTree(huffman_node_t *ht)
{
    if (ht->left != NULL)
        FreeHuffmanTree(ht->left);
    if (ht->right != NULL)
        FreeHuffmanTree(ht->right);
    free(ht);
}

static int FindMinimumCount(huffman_node_t **ht, int elements)
{
    int i;
    /* array index */
    int currentIndex = NONE;  /* index with lowest count seen so far */
    int currentCount = INT_MAX;   /* lowest count seen so far */
    int currentLevel = INT_MAX;   /* level of lowest count seen so far */
    /* sequentially search array */
    for (i = 0; i < elements; i++)
    {
        if ((ht[i] != NULL) && (!ht[i]->ignore) &&
            (ht[i]->count < currentCount ||
            (ht[i]->count == currentCount && ht[i]->level < currentLevel)))
        {
            currentIndex = i;
            currentCount = ht[i]->count;
            currentLevel = ht[i]->level;
        }
    }
    return currentIndex;
}

huffman_node_t *BuildHuffmanTree(huffman_node_t **ht, int elements)
{
    int min1, min2;  /* two nodes with the lowest count */
    for (;

{ min1 = FindMinimumCount(ht, elements); if (min1 == NONE) break;
  ht[min1]->ignore = TRUE; /* remove from consideration */
 min2 = FindMinimumCount(ht, elements);
 if (min2 == NONE) break;
  ht[min2]->ignore = TRUE; /* remove from consideration */
 if ((ht[min1] = AllocHuffmanCompositeNode(ht[min1], ht[min2])) == NULL) return NULL;
 ht[min2] = NULL;
}
return ht[min1];

<huffman.h>
#ifndef _HUFFMAN_H_
define _HUFFMAN_H_
/* traditional codes */
int HuffmanShowTree(char *inFile, char *outFile); /* dump codes */
int HuffmanEncodeFile(char *inFile, char *outFile); /* encode file */
int HuffmanDecodeFile(char *inFile, char *outFile); /* decode file */
/* canonical code */
int CHuffmanShowTree(char *inFile, char *outFile); /* dump codes */
int CHuffmanEncodeFile(char *inFile, char *outFile); /* encode file */
int CHuffmanDecodeFile(char *inFile, char *outFile); /* decode file */
#endif /* _HUFFMAN_H_ */

<huffman.cpp>
#include <stdio.h>
#include <stdlib.h>
#include "huflocal.h"
#include "bitarray.h"
#include "bitfile.h"
typedef struct code_list_t
{ byte_t codeLen; /* number of bits used in code (1 - 255) */
  bit_array_t *code; /* code used for symbol (left justified) */
} code_list_t;
static int MakeCodeList(huffman_node_t *ht, code_list_t *codeList);
static void WriteHeader(huffman_node_t *ht, bit_file_t *bfp);
static int ReadHeader(huffman_node_t **ht, bit_file_t *bfp);
int HuffmanEncodeFile(char *inFile, char *outFile)
{ huffman_node_t *huffmanTree; /* root of huffman tree */
  code_list_t codeList[NUM_CHARS]; /* table for quick encode */
  FILE *fpIn, *fpIn;
  bit_file_t *bfpOut;
  int c;
  if ((fpIn = fopen(inFile, "rb")) == NULL)
  { perror(inFile);
    return FALSE;
  }
  if (outFile == NULL)
    bfpOut = MakeBitFile(stdout, BF_WRITE);
 else
  { if (bfpOut = BitFileOpen(outFile, BF_WRITE)) == NULL)
    { perror(outFile);
      fclose(fpIn);
      return FALSE;
    }
  if ((huffmanTree = GenerateTreeFromFile(fpIn)) == NULL) return FALSE;
  for (c = 0; c < NUM_CHARS; c++)
  { codeList[c].code = NULL;
    codeList[c].codeLen = 0;
  }
if (!MakeCodeList(huffmanTree, codeList))
    return FALSE;
WriteHeader(huffmanTree, bfpOut);
rewind(fpIn);               /* start another pass on the input file */
while((c = fgetc(fpIn)) != EOF)
{
    BitFilePutBits(bfpOut,
        BitArrayGetBits(codeList[c].code),
        codeList[c].codeLen);
}
BitFilePutBits(bfpOut,
    BitArrayGetBits(codeList[EOF_CHAR].code),
    codeList[EOF_CHAR].codeLen);
for (c = 0; c < NUM_CHARS; c++)
{
    if (codeList[c].code != NULL)
        BitArrayDestroy(codeList[c].code);
}
fclose(fpIn);
BitFileClose(bfpOut);
FreeHuffmanTree(huffmanTree);     /* free allocated memory */
return TRUE;

int HuffmanDecodeFile(char *inFile, char *outFile)
{
    huffman_node_t *huffmanTree, *currentNode;
    int i, c;
    bit_file_t *bfp;
    FILE *fp;
    if ((bfp = BitFileOpen(inFile, BF_READ)) == NULL)
    {
        perror(inFile);
        return FALSE;
    }
    if (outFile == NULL)
    {
        fp = stdout;
    }
    else
    {
        if ((fp = fopen(outFile, "wb")) == NULL)
        {
            BitFileClose(bfp);
            perror(outFile);
            return FALSE;
        }
    }
    for (i = 0; i < NUM_CHARS; i++)
    {
        if ((huffmanArray[i] = AllocHuffmanNode(i)) == NULL)
        {
            for (i--; i >= 0; i--)
                free(huffmanArray[i]);
            BitFileClose(bfp);
            fclose(fp);
            return FALSE;
        }
    }
    if (!ReadHeader(huffmanArray, bfp))
    {
        for (i = 0; i < NUM_CHARS; i++)
            free(huffmanArray[i]);
        BitFileClose(bfp);
        fclose(fp);
        return FALSE;
    }
    if ((huffmanTree = BuildHuffmanTree(huffmanArray, NUM_CHARS)) == NULL)
    {
        FreeHuffmanTree(huffmanTree);
        BitFileClose(bfp);
        fclose(fp);
        return FALSE;
    }
    currentNode = huffmanTree;
    while ((c = BitFileGetBit(bfp)) != EOF)
if (c != 0)
    currentNode = currentNode->right;
else
    currentNode = currentNode->left;
if (currentNode->value != COMPOSITE_NODE)
    {  
        if (currentNode->value == EOF_CHAR)
            break;
        fprintf(currentNode->value, fpOut);  /* write out character */
        currentNode = huffmanTree;  /* back to top of tree */
    }
BitFileClose(bfpIn);
fclose(fpOut);
FreeHuffmanTree(huffmanTree);  /* free allocated memory */
return TRUE;
}

int HuffmanShowTree(char *inFile, char *outFile)
{
    FILE *fpIn, *fpOut;
    huffman_node_t *huffmanTree;  /* root of huffman tree */
    huffman_node_t *htp;          /* pointer into tree */
    char code[NUM_CHARS - 1];  /* 1s and 0s in character's code */
    int depth = 0;                    /* depth of tree */
    if ((fpIn = fopen(inFile, "rb")) == NULL)
        {  
            perror(inFile);
            return FALSE;
        }
    if (outFile == NULL)
        fpOut = stdout;
    else
        {  
            if ((fpOut = fopen(outFile, "w")) == NULL)
                {  
                    perror(outFile);
                    fclose(fpIn);
                    fclose(fpOut);
                    return FALSE;
                }
        }
    if ((huffmanTree = GenerateTreeFromFile(fpIn)) == NULL)
        {  
            fclose(fpIn);
            fclose(fpOut);
            return FALSE;
        }
    fprintf(fpOut, "Char  Count      Encoding\n");
    fprintf(fpOut, "-----  ----------  ----------------\n" );
    htp = huffmanTree;
    for(;;)
        {  
            while (htp->left != NULL)
                {  
                    code[depth] = '0';
                    htp = htp->left;
                    depth++;
                }
            if (htp->value != COMPOSITE_NODE)
                {  
                    if (depth == 0)
                        {  
                            code[depth] = '0';
                            depth++;
                        }
                    code[depth] = EOF;
                    if (htp->value != EOF_CHAR)
                        {  
                            fprintf(fpOut, "0x%02X  %10d %s\n",
                                    http->value, http->count, code);
                        }
                    else
                        fprintf(fpOut, "EOF  %10d %s\n", http->count, code);
                }
        }
}
while (htp->parent != NULL)
{
    if (htp != htp->parent->right)
    {
        code[depth - 1] = '1';
        htp = htp->parent->right;
        break;
    }
    else
    {
        depth--;
        htp = htp->parent;
        code[depth] = '0';
    }
    if (htp->parent == NULL)
     break;
}
fclose(fpIn);
fclose(fpOut);
FreeHuffmanTree(huffmanTree); /* free allocated memory */
return TRUE;

static int MakeCodeList(huffman_node_t *ht, code_list_t *codeList)
{
    bit_array_t *code;
    byte_t depth = 0;
    if((code = BitArrayCreate(256)) == NULL)
    {
        perror("Unable to allocate bit array");
        return (FALSE);
    }
    BitArrayClearAll(code);
    for(;;)
    {
        while (ht->left != NULL)
        {
            BitArrayShiftLeft(code, 1);
            ht = ht->left;
            depth++;
        }
        if (ht->value != COMPOSITE_NODE)
        {
            codeList[ht->value].codeLen = depth;
            codeList[ht->value].code = BitArrayDuplicate(code);
            if (codeList[ht->value].code == NULL)
            {
                perror("Unable to allocate bit array");
                BitArrayDestroy(code);
                return (FALSE);
            }
            BitArrayShiftLeft(codeList[ht->value].code, 256 - depth);
        }
        while (ht->parent != NULL)
        {
            if (ht != ht->parent->right)
            {
                BitArraySetBit(code, 255);
                ht = ht->parent->right;
                break;
            }
            else
            {
                depth--;
                BitArrayShiftRight(code, 1);
                ht = ht->parent;
            }
        }
        if (ht->parent == NULL)
         break;
    }
    BitArrayDestroy(code);
    return (TRUE);
}
static void WriteHeader(huffman_node_t *ht, bit_file_t *bfp)
{
    int i;
    for(;;)
    {
        while (ht->left != NULL)
            ht = ht->left;
        if ((ht->value != COMPOSITE_NODE) &&
            (ht->value != EOF_CHAR))
        {
            BitFilePutChar(ht->value, bfp);
            BitFilePutBits(bfp, (void *)&(ht->count), 8 * sizeof(count_t));
        }
        while (ht->parent != NULL)
        {
            if (ht != ht->parent->right)
            {
                ht = ht->parent->right;
                break;
            }
            else
                ht = ht->parent;
        }
        if (ht->parent == NULL)
            break;
    }
    BitFilePutChar(0, bfp);
    for(i = 0; i < sizeof(count_t); i++)
        BitFilePutChar(0, bfp);
}

static int ReadHeader(huffman_node_t **ht, bit_file_t *bfp)
{
    count_t count;
    int c;
    int status = FALSE; /* in case of premature EOF */
    while ((c = BitFileGetChar(bfp)) != EOF)
    {
        BitFileGetBits(bfp, (void *)&count, 8 * sizeof(count_t));
        if ((count == 0) && (c == 0))
        {
            status = TRUE;
            break;
        }
        ht[c]->count = count;
        ht[c]->ignore = FALSE;
    }
    ht[EOF_CHAR]->count = 1;
    ht[EOF_CHAR]->ignore = FALSE;
    if (status == FALSE)
        fprintf(stderr, "error: malformed file header.\n");
    return status;
}

<bitfile.h>
#define _BITFILE_H_
#define _BITFILE_H_
#include <stdio.h>
typedef enum
{
    BF_READ = 0,
    BF_WRITE = 1,
    BF_APPEND = 2,
    BF_NO_MODE
} BF_MODES;
struct bit_file_t;
typedef struct bit_file_t bit_file_t;
bit_file_t *BitFileOpen(const char *fileName, const BF_MODES mode);
bit_file_t *MakeBitFile(FILE *stream, const BF_MODES mode);
int BitFileClose(bit_file_t *stream);
FILE *BitFileToFILE(bit_file_t *stream);
/* toss spare bits and byte align file */
int BitFileByteAlign(bit_file_t *stream);
/* fill byte with ones or zeros and write out results */
int BitFileFlushOutput(bit_file_t *stream, const unsigned char onesFill);
/* get/put character */
int BitFileGetChar(bit_file_t *stream);
int BitFilePutChar(const int c, bit_file_t *stream);
/* get/put single bit */
int BitFileGetBit(bit_file_t *stream);
int BitFilePutBit(const int c, bit_file_t *stream);
/* get/put number of bits (most significant bit to least significant bit) */
int BitFileGetBits(bit_file_t *stream, void *bits, const unsigned int count);
int BitFilePutBits(bit_file_t *stream, void *bits, const unsigned int count);
int BitFileGetBitsInt(bit_file_t *stream, void *bits, const unsigned int count,
const size_t size);
int BitFilePutBitsInt(bit_file_t *stream, void *bits, const unsigned int count,
const size_t size);
#endif /* _BITFILE_H_ */

<bitfile.cpp>
#include <stdlib.h>
#include <errno.h>
#include "bitfile.h"
typedef enum
{
    BF_UNKNOWN_ENDIAN,
    BF_LITTLE_ENDIAN,
    BF_BIG_ENDIAN
} endian_t;
struct bit_file_t
{
    FILE *fp;                   /* file pointer used by stdio functions */
    endian_t endian;            /* endianess of architecture */
    unsigned char bitBuffer;    /* bits waiting to be read/written */
    unsigned char bitCount;     /* number of bits in bitBuffer */
    BF_MODES mode;              /* open for read, write, or append */
};
/* union used to test for endianess */
typedef union
{
    unsigned long word;
    unsigned char bytes[sizeof(unsigned long)];
} endian_test_t;
int BitFilePutBitsLE(bit_file_t *stream, void *bits, const unsigned int count);
int BitFilePutBitsBE(bit_file_t *stream, void *bits, const unsigned int count,
const size_t size);
int BitFileGetBitsLE(bit_file_t *stream, void *bits, const unsigned int count);
int BitFileGetBitsBE(bit_file_t *stream, void *bits, const unsigned int count,
const size_t size);
bit_file_t *BitFileOpen(const char *fileName, const BF_MODES mode)
{
    char modes[3][3] = {'rb', 'wb', "ab"}; /* binary modes for fopen */
    bit_file_t *bf;
    bf = (bit_file_t *)malloc(sizeof(bit_file_t));
    if (bf == NULL)
        errno = ENOMEM;
    else
    {
        bf->fp = fopen(fileName, modes[mode]);
        if (bf->fp == NULL)
        {
            free(bf);
            bf = NULL;
        }
    }
    return (bf);
}
int BitFilePutBitsBE(bit_file_t *stream, void *bits, const unsigned int count, const size_t size)
{
    unsigned char *bytes, tmp;
    int offset, remaining, returnValue;
    if (count > (size * 8))
        return EOF;
    bytes = (unsigned char *)bits;
    offset = size - 1;
    remaining = count;
    while (remaining >= 8)
    {
        returnValue = BitFilePutChar(bytes[offset], stream);
        if (returnValue == EOF)
            return EOF;
        remaining -= 8;
        offset--;
    }
    if (remaining != 0)
    {
        tmp = bytes[offset];
        tmp <<= (8 - remaining);
        while (remaining > 0)
        {
            returnValue = BitFilePutBit((tmp & 0x80), stream);
            if (returnValue == EOF)
                return EOF;
            tmp <<= 1;
            remaining--;
        }
    }
    return count;
}

<bitarray.h>
#ifndef BIT_ARRAY_H
#define BIT_ARRAY_H
struct bit_array_t;
typedef struct bit_array_t bit_array_t;
bit_array_t *BitArrayCreate(unsigned int bits);
void BitArrayDestroy(bit_array_t *ba);
/* debug functions */
void BitArrayDump(bit_array_t *ba, FILE *outFile);
/* set/clear functions */
void BitArraySetAll(bit_array_t *ba);
void BitArrayClearAll(bit_array_t *ba);
void BitArraySetBit(bit_array_t *ba, unsigned int bit);
void BitArrayClearBit(bit_array_t *ba, unsigned int bit);
/* raw bit access */
void *BitArrayGetBits(bit_array_t *ba);
/* bit test function */
int BitArrayTestBit(bit_array_t *ba, unsigned int bit);
/* copy functions */
void BitArrayCopy(bit_array_t *dest, const bit_array_t *src);
bit_array_t *BitArrayDuplicate(const bit_array_t *src);
/* logical operations */
void BitArrayAnd(bit_array_t *dest, const bit_array_t *src1, const bit_array_t *src2);
void BitArrayOr(bit_array_t *dest, const bit_array_t *src1, const bit_array_t *src2);
void BitArrayXor(bit_array_t *dest, const bit_array_t *src1, const bit_array_t *src2);
void BitArrayNot(bit_array_t *dest, const bit_array_t *src);
/* bit shift functions */
void BitArrayShiftLeft(bit_array_t *ba, unsigned int shifts);
void BitArrayShiftRight(bit_array_t *ba, unsigned int shifts);

void BitArrayIncrement(bit_array_t *ba);
void BitArrayDecrement(bit_array_t *ba);

int BitArrayCompare(const bit_array_t *ba1, const bit_array_t *ba2);

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>
#include <string.h>
#include "bitarray.h"

/* make CHAR_BIT 8 if it's not defined in limits.h */
#ifndef CHAR_BIT
#warning CHAR_BIT not defined. Assuming 8 bits.
#define CHAR_BIT 8
#endif

/* position of bit within character */
#define BIT_CHAR(bit)         ((bit) / CHAR_BIT)

/* array index for character containing bit */
#define BIT_IN_CHAR(bit)      (1 << (CHAR_BIT - 1 - ((bit) % CHAR_BIT)))

/* number of characters required to contain number of bits */
#define BITS_TO_CHARS(bits)   ((((bits) - 1) / CHAR_BIT) + 1)

/* most significant bit in a character */
#define MS_BIT                (1 << (CHAR_BIT - 1))

struct bit_array_t
{
    unsigned char *array;       /* pointer to array containing bits */
    unsigned int numBits;       /* number of bits in array */
};

bit_array_t *BitArrayCreate(unsigned int bits)
{
    bit_array_t *ba;
    ba = (bit_array_t *)malloc(sizeof(bit_array_t));
    if (ba == NULL)
        errno = ENOMEM;
    else
    {
        ba->numBits = bits;
        ba->array = (unsigned char *)malloc(sizeof(unsigned char) *
            BITS_TO_CHARS(bits));
        if (ba->array == NULL)
            {
                errno = ENOMEM;
                free(ba);
                ba = NULL;
            }
    }
    return(ba);
}

.............................
.............................
.............................

int BitArrayCompare(const bit_array_t *ba1, const bit_array_t *ba2)
{
    int i;
    if (ba1 == NULL)
    {
        if (ba2 == NULL)
            return 0;  /* both are NULL */
        else
            return -(ba2->numBits); /* ba2 is the only Non-NULL */
    }
    if (ba2 == NULL)
        return (ba1->numBits); /* ba1 is the only Non-NULL */
    if (ba1->numBits != ba2->numBits)
    {
        return(1);  i = BIT_CHAR(ba1->numBits - 1); i++
    }
if (ba1->array[i] != ba2->array[i])
    return(ba1->array[i] - ba2->array[i]);
return 0;
}