Implementing Associative Coder of Buyanovsky (ACB) data compression
by Sean Michael Lambert

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
In 1994 George Mechislavovich Buyanovsky published a basic description of a new data compression algorithm he called the “Associative Coder of Buyanovsky,” or ACB. The archive program using this idea, which he released in 1996 and updated in 1997, is still one of the best general compression utilities available. Despite this, the ACB algorithm is still barely understood by data compression experts, primarily because Buyanovsky never published a detailed description of it. ACB is a new idea in data compression, merging concepts from existing statistical and dictionary-based algorithms with entirely original ideas. This document presents several variations of the ACB algorithm and the details required to implement a basic version of ACB.

IMPLEMENTING ASSOCIATIVE CODER OF BUYANOVSKY (ACB) DATA COMPRESSION

by

Sean Michael Lambert

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY-BOZEMAN
Bozeman, Montana

April 1999

APPROVAL

of a thesis submitted by

Sean Michael Lambert

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Brendan Mumey — (Signature) — 4/21/99 — Date

Approved for the Department of Computer Science

J. Denbigh Starkey — (Signature) — 4/21/99 — Date

Approved for the College of Graduate Studies

Graduate Dean — (Signature) — 4-22-99 — Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University-Bozeman, I agree that the Library shall make it available to borrowers under rules of this Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Sean Ladd

Date 4/21/99

TABLE OF CONTENTS

TABLE OF CONTENTS — Continued

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

In 1994 George Mechislavovich Buyanovsky published a basic description of a new data compression algorithm he called the "Associative Coder of Buyanovsky," or ACB. The archive program using this idea, which he released in 1996 and updated in 1997, is still one of the best general compression utilities available. Despite this, the ACB algorithm is still barely understood by data compression experts, primarily because Buyanovsky never published a detailed description of it. ACB is a new idea in data compression, merging concepts from existing statistical and dictionary-based algorithms with entirely original ideas. This document presents several variations of the ACB algorithm and the details required to implement a basic version of ACB.

CHAPTER 1

INTRODUCTION

Most modern data compression techniques focus on one of two basic concepts. One of these ideas is to statistically model the data, in an attempt to predict the next symbol based on the symbols which have come before. The other idea, which was pioneered by Jacob Ziv and Abraham Lempel in the late 1970's, involves replacing strings of symbols with dictionary references. The Associative Coder of Buyanovsky (ACB) combines portions of each of these concepts.

George Mechislavovich Buyanovsky, first published a general description of his new algorithm in the Russian periodical Monitor in 1994. It went relatively unnoticed until he produced an archive utility based on his idea in 1996. This archive utility topped the Archive Comparison Test (ACT) lists in several categories at its debut, and remains near the top of these lists today. About this time Buyanovsky posted a few articles outlining the technique to the Usenet newsgroup comp.compression. These basic descriptions were vague, however, and language barriers prevented more in-depth discussions on the topic. Buyanovsky's desire to profit from his invention may have also stifled his interest in sharing the details of ACB, although he has not discouraged further research.

The purpose of this study is to bring ACB to the attention of the data compression community by providing an accurate description of the algorithm, some of its variations, and details needed for its implementation. This thesis should be viewed as a point of departure for further research in this area.

CHAPTER 2

COMMON DATA COMPRESSION TECHNIQUES

This chapter provides a brief overview of the common lossless data compression techniques in use today. This overview will help to illustrate the differences and similarities between ACB and well established data compression algorithms.

2.1 Statistical Modeling Methods

Data is usually stored in a computer by assigning each symbol a fixed-size binary code. For example, ASCII code is a standard of representation for symbols such as the letters of the alphabet. Each character is assigned an eight-bit binary code. Statistical models change this paradigm by assigning shorter codes to more-frequently-used symbols and longer codes to less-frequently-used symbols. For example, if the model was English text, an 'e' would be represented with fewer bits than an 'x'.

2.1.1 Shannon-Fano and Huffman Coding

The Shannon-Fano and Huffman codes are popular choices for variable-size symbol coding. Each provides a heuristic for determining which bit patterns to assign to the symbols of a particular set, given the expected frequency of each symbol.

Compression is then simply a matter of replacing the old fixed-size bit patterns with the new variable-sized bit patterns. The drawback of these methods is that the distribution of the symbol frequencies does not necessarily match the distribution of the bit pattern lengths. This is because the bit pattern lengths must be integral numbers of bits, while the frequencies may be fractional numbers.

2.1.2 Arithmetic Coding

Arithmetic coding improves on the previous methods by allowing a symbol to use a fractional number of bits. This is accomplished by transforming the string of symbols to be compressed into a single real number, which has a very high precision. The number can be any number in a range, and that range is reduced as each symbol is encoded. This is most easily described by an example:

A language has 3 symbols: 'a', 'b', and 'c'. Their probability of appearance has been determined to be 0.5, 0.3, and 0.2 respectively. The output number will be between 0 and 1, so the initial range is (0, 1). As each symbol is encoded, the range is narrowed to the portion of the previous range matching the probability of the symbol's appearance. For this example the first symbol to be encoded is 'a', so the range becomes (0, 0.5). A 'b' would have reduced the range to (0.5, 0.8), and a 'c' would have reduced the range to (0.8, 1). The final encoded number will be in one of these ranges, allowing the decoder to determine what the first character of the string is. The probabilities are scaled to the new range, and these two steps are repeated until the entire string is encoded. The encoder then outputs a number in the final range, using the smallest possible number of

bits. Continuing the example, the next characters are 'b', 'a', and 'c'. The range is narrowed to (0.25, 0.4), (0.25, 0.325), and then (0.31, 0.325).

Arithmetic coders use several tricks to maintain the range without causing the variables used to underflow their precision. As long as the same trick is used by the decoder, the string can be restored. Arithmetic coding also allows the frequency distribution of the symbols to change at any time, as long as the information needed to change the distribution is also available to the decoder. This flexibility allows adaptive models to be constructed, which modify themselves based on the frequency of the symbols in the current string which have already been encoded.

It is also possible to change the frequency distribution based on the current context, or the preceding symbols. For example, in English if the characters 'zo' are the last two characters seen, the next character is very likely to be 'o' and not as likely to be 'e'. A statistical model which considers only the current character is called an order-0 model. A model is called order-*n* if it considers the previous *n* symbols when determining the frequency distribution of the current character.

2.1.3 Prediction by Partial String Matching (PPM)

Higher-order models produce better results than lower-order models, but they require more computation. PPM is a method that uses a higher-order model when possible, and lower-order models otherwise. For example, a particular PPM encoder tries to find an order-3 match for the current context and next character. If one cannot be found in the previously encoded text, an order-2 model is used, and an escape character

is generated to let the decoder know that the model has changed. This subtraction is repeated until the current character can be properly encoded.

2.2 Dictionary Methods

Dictionary methods of data compression substitute substrings of the string to be compressed with fixed-length indices into a dictionary. The indices, of course, must be shorter than the substrings they replace for compression to occur. The differences between the various dictionary techniques mostly have to do with the selection of the substrings which make up the dictionary.

2.2.1 Ziv-Lempel 77 (LZ77)

Many compression algorithms used today are based on the LZ77 algorithm. The dictionary for this method is a sliding window into the context, or a buffer containing the last k symbols. The compressor attempts to find the longest match between the string to be encoded, which is in the look-ahead buffer, and a portion of the sliding window. If a match is found, the compressor outputs a fixed-length index into the sliding window, followed by the length of the match, followed by the first unmatched character. Longer sliding windows and look-ahead buffers increase the compression and the computation required, similar to the way higher-order statistical models achieve better compression but require more computation than lower-order models. Longer sliding windows also require larger indices, which reduce the effectiveness of the substitution. Some compression utilities that use variations of the LZ77 algorithm are *arj*, *lha*, and *zip*.

2.2.2 Ziv-Lempel 78 (LZ78)

The LZ78 method uses an actual dictionary, rather than the sliding window that LZ77 uses. This dictionary is built as the encoding proceeds, using only previously encoded symbols. The decoder is then able to build its dictionary in the same way. The encoder collects symbols to be encoded until it has found a substring which is not in the dictionary. Since this substring was built one symbol at a time, there is a substring in the dictionary which matches all but the last character of the substring to be encoded. The compressor outputs the index of the existing dictionary entry followed by the unmatched character. Both the encoder and decoder now add to their dictionaries the substring made from the existing dictionary entry plus the unmatched character. Increasing the size of the LZ78 dictionary has similar consequences to increasing the LZ77 sliding window size. However, unlike LZ77, the LZ78 dictionary will become full. Often LZ78 compressors will switch to a static dictionary at that point, and monitor the compression rate. If the compression rate becomes too low, the entire dictionary is deleted and built up again. This is one of the most popular compression algorithms, and is used by the UNIX compress utility.

CHAPTER 3

ASSOCIATIVE CODER OF BUYANOVSKY (ACB)

There are several variations of the ACB algorithm, written by Buyanovsky and others. In a postscript to one of his letters Buyanovsky writes, "Let me remind [you] again this is a simplified version of the associative coder, and it is fairly easy to invent tens of variations of this scheme[.]" This chapter will illustrate the concept that Buyanovsky feels is the central idea of the associative coder, and will present some variations of his algorithm.

3.1 Some Definitions

The following definitions and Figure 1 will aid in the description of the algorithm variations:

- Let S be the string of symbols to be encoded. Its length is represented by N.
- Let S[*n*] be the nth symbol in S.
- The **context** at S[*n*] is the part of S including and preceding S[*n*], beginning with the most recent symbol and receeding into the past, i.e. {S[*n*], S[*n*-1], S[*n*-2], ..., S[2], S[1]}.
- The **content** at S[*n*] is the part of S following S[*n*], beginning with the next symbol and proceeding into the future, i.e. {S[*n*+1], S[*n*+2], S[*n*+3], ..., S[N-1], S[N]}.

Figure 1. Context and Content.

```
                    <-- context|content -->
S: ...11101000101011101100|10110100100111101011...
                          S[n]^
```

3.2 The Concept

Like LZ77, ACB uses a sliding window dictionary. ACB also outputs a dictionary reference and a length. The dictionary entries, however, are not referenced by fixed-size references into the window. The probability that each index will be chosen is estimated, and the choice is statistically encoded. Thus ACB combines ideas from both statistical modeling and dictionary compression techniques. The statistical concept, which Buyanovsky feels is the root concept of ACB, was unfortunately overlooked by all other references.

3.3 The Dictionary

The dictionary is actually a list of pointers into the sliding window. Buyanovsky specifies that the dictionary be sorted by context. The sliding window is considered to be a circular buffer so that the context and content of all entries are unbounded. Sorting the dictionary by context allows the associative list to be built more efficiently, but does not affect the compression ratio achieved by ACB.

3.4 The Associative List

The dictionary reference which is chosen actually comes from a subset of the dictionary called the **associative list**. The associative list contains all indices from the dictionary whose contexts match the current context by at least k symbols. Furthermore, the probability that each member of the associative list will be chosen is based on the length of this context match. Therefore, if the current context matches a large portion of the dictionary, it is assumed to be likely that the content following the dictionary reference will match the current content. These probabilities are used to arithmetically encode the index of the associative list member with the longest content match with the current content. The length of this match is also encoded, as in LZ77.

3.5 The Difference Bit

When using a binary language, if the associative list is sorted by content then it may be possible to encode much of the content match length using one bit. The **difference bit** is the value of the first bit of the current content which does not match the content of the selected associative list member. This bit will provide more information, however, when the current content is compared to the sorted associative list (Figure 2.)

If the difference bit is a '0', then the current content falls between the encoded index and the index preceding it. If the difference bit is a '1', then the current content falls between the encoded index and the index following it. Zero or more bits will match between the contents of these surrounding associative list members. Those bits

(indicated by 'x' in Figure 2) must also match the current content, so they do not need to be counted when encoding the length of the current content match.

If the current content falls before the first associative list entry or after the last entry, there will not be a second content to match against. In these cases there are assumed entries of all '0's and all '1's at the appropriate ends of the list. These entries are not numbered and cannot be matched directly to the current content.

Since the current content is known to match the encoded index, the next bit of the content must also match. The remainder of the content match (indicated by'z' in Figure 2) is then output. However, only the bits which do not match the difference bit need to be counted, since one of those bits is the first bit which does not match.

Figure 2. Use of the Difference Bit and Sorted Associative List.

```
                d=0                                  d=1
i-1  :xxx...xx0yyy...                 i   :xxx...xx0zzz...zz0...
S[n]:xxx...xx1zzz...zzd...            S[n]:xxx...xx0zzz...zzd...
i    :xxx...xx1zzz...zz1...           i+1 :xxx...xx1yyy...
```

3.6 Variations of ACB

The variations of ACB presented in this section are all of the previously published ACB algorithms along with the algorithm developed by the author of this thesis. They have been numbered somewhat arbitrarily by the author of this thesis.

3.6.1 ACB1

This is the algorithm given by Buyanovsky in his 1994 Monitor article. However, it is not the algorithm that the ACB archive utility uses. Buyanovsky states that the newer algorithm is thirty to forty times faster than this one, but shares the same basic premises. ACB1 is the algorithm described above, listed here in detail. The dictionary is considered to be a binary string rather than a character string, so there is a pointer into the sliding window for each bit.

1) Maintain a sliding-window dictionary, sorted by context.
2) Build an associative list containing all dictionary entries matching the current context by at least k bits.
3) Assign each member of the associative list a weight based on the length of the context match with the current context.
4) Sort the associative list by content.
5) Find the associative list entry which has the longest content match with the current content.
6) Statistically encode this index using the weights found above.
7) Encode the difference bit.
8) Encode the match length, modified as described in section 3.5.
9) Update the dictionary, move the current pointer ahead, and repeat.

3.6.2 ACB2

This algorithm is a simplified version of ACB1. It was originally described by Buyanovsky in an article posted to the newsgroup comp.compression on September 18, 1996 (Original Message-ID: <AAX10GoKlE@acb.alma-ata.su>)

ACB2 is exactly the same as ACB1, except that no length is encoded. ACB2 relies on the index and the difference bit alone to imply the match length. The problem with this method is that after a long match, the same dictionary entry will be selected as the most likely to match the next entry. It is apparent that if the same entry is selected

consecutively, it is very likely that the second encoding will match very few or no new bits. For this reason, ACB1 is preferred.

3.6.3 ACB3

This algorithm was posted to the newsgroup comp.compression.research on May 5, 1996, by Leonid A. Broukhis. (Original Message-ID: <4mhjvp$70u@net.auckland.ac.nz>) It is also presented as ACB compression in David Salomon's book Data Compression. This algorithm does not use an associative list, and works with a character-based dictionary.

It is interesting to note that the difference bit and its use were also described by Broukhis, but he did not mention it in any of his examples. This may be because problems arise when mixing character-based and bit-based models (described in section 1.4.) The difference bit's use would have also been problematic in ACB3 because the dictionary is sorted by context and not content.

1) Maintain a sliding-window dictionary, sorted by context.
2) Find the dictionary entry which has the longest context match with the current context.
3) Find the dictionary entry which has the longest content match with the current content.
4) Encode the index of the content match as an offset from the context match. If this number is consistently small then compression can be achieved by statistically encoding this offset.
5) Encode the length of the content match.
6) Encode the first unmatched byte.
7) Update the dictionary, move the current pointer ahead, and repeat.

3.6.4 ACB4

Salomon described a second algorithm in his book Data Compression. It was

presented it as an ACB variation, though it is closer to ACB1 than ACB3 is. This variation uses a character-based dictionary and associative list, but does not use any form of statistical modeling.

The difference bit is explained by Salomon and used in this method, but there is a problem combining a character-based dictionary with the difference bit concept. The problem is that the end of any particular match is very likely to occur in the middle of a byte, rather than at byte boundaries. It is unlikely that the contexts of the dictionary entries would have much meaning in this case.

1) Maintain a sliding-window dictionary, sorted by context.
2) Build an associative list containing all dictionary entries matching the current context by at least k bits.
3) Sort the associative list by content.
4) Find the associative list entry which has the longest content match with the current content.
5) Encode the index of the content match as a fixed-length index.
6) Encode the difference bit.
7) Encode the match length, modified as described in section 3.5.
8) Update the dictionary, move the current pointer ahead, and repeat.

3.6.5 ACB5

This algorithm was developed by the author of this thesis, and is the algorithm discussed in detail in chapter 4. It is much like ACB1, but uses a character-based dictionary. The associative list is a fixed size, listing the best m context matches, rather than all context matches of length k.

1) Maintain a sliding-window dictionary, sorted by context. Implemented dictionary size was 1024 bytes.
2) Build an associative list containing the m dictionary entries which most closely match the current context. Implemented associative list had 256 entries.
3) Assign each member of the associative list a weight based on the length of the

context match with the current context.
4) Sort the associative list by content.
5) Find the associative list entry which has the longest content match with the current content.
6) Statistically encode this index using the weights found above.
7) Encode the difference bit.
8) Encode the match length in whole bytes, modified as described in section 3.5.
9) Encode the first unmatched byte.
10) Update the dictionary, move the current pointer ahead, and repeat.

3.7 Example Using ACB5

To better illustrate how ACB works, a short example of ACB5 follows. Each iteration of the algorithm produces a token to be arithmetically encoded. The token contains an associative list index *i*, a difference bit *d*, a match length *l*, and an unmatched character *c*. This example will show how the token (*i*, *d*, *l*, *c*) is generated, but will not show the encoding of the token. The encoding process is explained in detail in chapter 4.

Figure 3 shows the sliding window which is used in the example. The current context ends with the 'i' in "willson.", which means the current content is "llson.". The length of the dictionary is 23 characters in this example.

Figure 3. Example Sliding Window and Current Content.

Figure 4 shows the dictionary sorted by context. Remember that the context is read from right to left in this example. Since the sliding window is kept in a circular buffer, the content and context can be imagined to repeat indefinitely. The contexts and contents of the dictionary entries in Figure 4 have been repeated for one entire buffer length to illustrate this. The ten dictionary entries which most closely match the current context are selected to become the associative list. These ten entries are marked with asterisks, and the current content/context is marked S.

Figure 4. Example Dictionary Sorted by Context.

	<-- context	content -->	
1	illwillstillkilljillwib	illwillstillkilljillwib	
2	llwillstillkilljillwibi	llwillstillkilljillwibi	*
3	llwibillwillstillkillji	llwibillwillstillkillji	*
4	lljillwibillwillstillki	lljillwibillwillstillki	*
5	llkilljillwibillwillsti	llkilljillwibillwillsti	*
6	llstillkilljillwibillwi	llstillkilljillwibillwi	*
7	billwillstillkilljillwi	billwillstillkilljillwi	S
8	illwibillwillstillkillj	illwibillwillstillkillj	*
9	illjillwibillwillstillk	illjillwibillwillstillk	*
10	lwillstillkilljillwibil	lwillstillkilljillwibil	*
11	lwibillwillstillkilljil	lwibillwillstillkilljil	*
12	ljillwibillwillstillkil	ljillwibillwillstillkil	*
13	lkilljillwibillwillstil	lkilljillwibillwillstil	*
14	lstillkilljillwibillwil	lstillkilljillwibillwil	
15	willstillkilljillwibill	willstillkilljillwibill	
16	wibillwillstillkilljill	wibillwillstillkilljill	
17	jillwibillwillstillkill	jillwibillwillstillkill	
18	killjillwibillwillstill	killjillwibillwillstill	
19	stillkilljillwibillwill	stillkilljillwibillwill	
20	tillkilljillwibillwills	tillkilljillwibillwills	
21	illkilljillwibillwillst	illkilljillwibillwillst	
22	illstillkilljillwibillw	illstillkilljillwibillw	
23	ibillwillstillkilljillw	ibillwillstillkilljillw	

Figure 5 shows the associative list sorted by content. The context match length with the current context follows each associative list entry. This match length is used to arithmetically encode the chosen entry. The longest context match is with entry number 6, so this line is estimated to be the most probable content match by both the encoder and decoder. The longest content match *is* with entry number 6, so the encoded index *i* is 6. The current content is lexicographically smaller than (i.e. would have been sorted before) the content of associative list entry number 6, so the difference bit *d* is '0'. The content match length *l* is three characters, and the first unmatched character *c* is 'o'.

Figure 5. Example Associative List Sorted by Content.

```
        <-- current context|current content -->
  S billwillstillkilljillwi|llson.

                                                context
                  <-- context|content -->         match
  1 illjillwibillwillstillk|illjillwibillwillstillk   0
  2 illwibillwillstillkillj|illwibillwillstillkillj   0
  3 ljillwibillwillstillkil|ljillwibillwillstillkil   0
  4 lljillwibillwillstillki|lljillwibillwillstillki   1
  5 llkilljillwibillwillsti|llkilljillwibillwillsti   1
* 6 llstillkilljillwibillwi|llstillkilljillwibillwi   5
  7 llwibillwillstillkillji|llwibillwillstillkillji   1
  8 llwillstillkilljillwibi|llwillstillkilljillwibi   1
  9 lwibillwillstillkilljil|lwibillwillstillkilljil   0
 10 lwillstillkilljillwibil|lwillstillkilljillwibil   0
```

Figure 6 shows how the match length is decided, and how the match length can be reduced by matching associative list entries. Since the difference bit is '0', associative list entries 6 and 5 are compared. They match each other for a length of two characters, so those two characters must also match the current content. The decoder can perform a similar operation, so the match length *l* is reduced from three to one. Thus the token (*i*, *d*, *l*, *c*) which the encoder will output is (6, 0, 1, 'o') for this match.

Figure 6. Example Match Length Reduction.

```
5 llkill
S llson.
6 llstil
```

Once the token is encoded, the current context is moved ahead by four characters to end with 'o'. The dictionary is adjusted by removing the entries which have been shifted out of the sliding window, and by adding the entries which have been shifted in. The process described above is then repeated until the entire string S has been encoded.

CHAPTER 4

IMPLEMENTATION

ACB5 was implemented in order that its details, successes, and failures could be presented here. This chapter will present the implementation decisions that were made, some problems encountered, and places where the code could be improved.

The main purpose of this implementation was to produce a simple, readable, working implementation of ACB. Execution speed was considered to be second in importance to simplicity, so much of the code is inefficient in this sense. Furthermore, a character-based model was selected for clarity, though a bit-based model would have performed better.

4.1 The Arithmetic Coder

Since a statistical model was needed to encode the associative list index of the longest content match, arithmetic coding was selected. As arithmetic coding itself was not the main focus of the implementation, an off-the-shelf arithmetic coder was used. This code came from Mark Nelson's The Data Compression Book. Nelson's code was easy to understand, well commented, and provided a serviceable user interface as well as an arithmetic coder. The code was modified to use an ACB model, rather than the static

statistical model the code was originally written for.

The arithmetic coder prevented simple file access, since it dealt with the file one bit at a time. The most reasonable solution was to use the arithmetic coder for all input and output. At any given time, all that is needed to arithmetically encode or decode is a list of probabilities and symbols. Thus, all that was needed was a table of probabilities for every set of symbols used by ACB. The implemented version of ACB used four different sets of symbols: an index, the difference bit, the match length, and the first unmatched character.

4.1.1 Modeling The Index

The probabilities for the associative list indexes are decided by ACB, so their implementation was straightforward. However, ACB does not provide an exact rule for the generation of the probabilities. This implementation simply counts the match lengths in characters and adds one, scaling the counts as needed. The match length was increased by one so that non-matching strings which were included in the associative list would have a non-zero probability of occurrence. This method of predicting the probabilities was arbitrarily chosen, and it may be possible to increase compression by applying a different function of the match length in this case.

4.1.2 Modeling The Difference Bit

The difference bit was originally encoded using a fixed model which weighted the symbols '1' and '0' equally. However, the method of selecting among associative list members with similar match lengths produced a bias toward '1' which was observed to

be about twice as prevalent as '0'. The model was then changed to be a simple adaptive arithmetic model. This bias is an artifact of the character-based model, and should not occur in a bit-based model.

4.1.3 Modeling The Match Length

A problem arises when attempting to model the match length. Since the length is potentially infinite, a fixed-size representation is inadequate. In addition, there is a strong bias toward smaller numbers, so a proportional representation is desired. The chosen design was an adaptive arithmetic coding of the numbers 0 to L. If the length was L or greater, L was encoded and subtracted from the length. The remainder of the length was then encoded the same way. During decoding, if a length of L was decoded, another length was decoded and added to the first, until the length decoded at one of the steps was not L. This provided a way for very large numbers to be encoded, with the bias toward smaller numbers.

The selection of L was important to the compression ratio. A large L requires a large probability table to be maintained, with the majority of the probabilities being very low. Since the model used integer values to store the probabilities, large numbers of small probabilities reduced the effectiveness of the arithmetic coding. In contrast, if L is chosen to be very small, then any lengths larger than L are lengthened by having to be encoded in segments. Empirical data showed that 15 would be a better value for L than 31 or the originally chosen 63, but the study was not exhaustive. The ideal L is a function of the amount of repetition in the string to be encoded. If the string has a lot of

long repeated substrings, a larger L is advantageous. If the string has little repetition, or many short repeated substrings, L should be smaller. Since this implementation was designed to be a general data compression utility, the amount of repetition in the strings to be compressed was not knowable in advance.

Obviously, the length can be very long, but it will not be truly infinite. Furthermore, since the length is decoded and stored in a variable, the size of that variable ultimately decides the maximum length of a content match. This implementation used short integers, which are often 16 bytes long. No overflow protection was included in the implementation, but a match of over two gigabytes would be required to cause an overflow. Files of this size could also threaten to overflow any of the other adaptive arithmetic models in this implementation.

Again, this method of encoding the length was arbitrarily chosen, because it met the minimum requirements of the algorithm. It may be possible to improve compression by improving or replacing this method.

4.1.4 Modeling The Unmatched Character

The model of the first unmatched character is the part of this implementation which can be improved upon the most. There are 256 values for a character, and it is difficult to know which characters are more likely in this case. If a given character has been the first unmatched character many times in the past, it is actually less likely that it will be so in the future. This is because it now appears more often in the dictionary and is likely to be matched. However, there are some characters which are simply more

likely to appear in a given file, and others which will not appear at all.

Even more information can be incorporated in the probability estimation for these characters if the previous symbols are also considered. The character in the matched string which did not match the character in question should have a probability of zero, or the match length would have been one longer. Some information can also be gained from the difference bit, which could serve to rule out several other characters as well. This is the main difficulty with the use of a character-based model combined with the difference bit concept. This implementation rounded the match length down to the nearest whole byte, disregarding some of the match. This information was then redundantly encoded as part of the first unmatched character. An alternate method of combining a character-based dictionary and the difference bit concept might be to treat the match as a binary string, and then output enough bits at the end to advance the current pointer to a byte boundary.

This implementation did not attempt to balance any of these issues, and simply weighted all of the 256 possibilities equally, with a very small weight being given to the End-Of-Stream symbol. The disappointing performance of this implementation is most likely due to this simplification of the problem. Character-based models are naturally less efficient in this way, although their efficiency should approach that of a bit-based model if enough attention is paid to details.

4.2 Dictionary Organization

A circular buffer was used for the dictionary, allowing the use of a fixed-size

array for the data storage. This organization simplified the addition and removal of characters to and from the dictionary. In order to find the associative list without searching the entire dictionary, the dictionary elements were kept in a sorted list as well as in the order they appeared in the string to be encoded. To accomplish this each entry kept a pointer to the previous and next entry in sorted order. These pointers were implemented as integer indices into the circular buffer. As entries were added to the dictionary, they were "bubbled" up from the end of the sorted order using an insertion sort.

The dictionary was implemented with a 1024-byte dictionary. Larger dictionaries produced slightly better results, but greatly increased the execution time. A more efficient implementation should consider the use of a larger dictionary.

4.3 Building The Associative List

In an attempt to improve on ACB1, ACB5's associative list uses a fixed size rather than a minimum match length to determine its members (although it is unclear at this time whether this is actually an improvement.) The goal of this change is to reduce the frequency of sparsely populated lists and keep down the list size in the case of a highly repetitive input string. The dictionary entries with the longest context match had been sorted adjacent to the current entry. So, beginning with the newest addition to the dictionary and moving outward using the sorted order, these adjacent entries were added to the associative list. The entries were sorted by content using insertion sort as they were added to the associative list.

If the match length reached the size of the dictionary, then the entries were considered to be identical. Including two or more identical entries to the associative list would have reduced the compression achieved, so identical entries were not permitted.

The size of the associative list used by this implementation was 256 entries. Surprisingly, increasing the associative list size improved compression, even when the associative list was increased to the size as the dictionary. However, this increase in associative list size seriously impacted the execution time. This result suggests an alternate implementation method which treats the entire dictionary as an associative list, eliminating the context sort.

4.4 Using The Associative List

The longest content match was found by starting at the ends of the list and narrowing the focus until the longest match was found. This is a simple, accurate, and slow procedure. Replacing this algorithm could increase the overall speed of the implementation but would not improve the compression achieved.

The difference bit was used, and the match length was reduced by the length of the match between the selected entry and the appropriate adjacent entry. If the current content fell after the last associative list entry, or before the first entry, it was matched with imaginary entries of all '1's or all '0's, as described in section 3.5. The only difference between the description in chapter 3 and the implementation was that the match lengths were byte units rather than bit units.

If two indices produced the same match length, and the current content fell

between them, then either could have been arbitrarily chosen. This implementation specified that the first entry was to be chosen in such cases, resulting in a difference bit of '1'. This partly explains why the difference bit was more often '1' than '0'. If the match lengths are instead compared at a bit level one of the matches will always be longer, so this bias will not be an issue.

If the associative list was empty, a special case matched the current content against imaginary strings of all '1's and all '0's. Since there was no index to encode in that case, a special probability table was passed to the arithmetic encoder which had only one symbol, which occurred with a probability of one.

CHAPTER 5

CONCLUSIONS

ACB combines the efficiency of statistical modeling with the string substitution power of a dictionary compression method. It resembles PPM because it favors longer context matches over shorter ones. It resembles LZ77 because it uses references into a sliding window. But more importantly, it presents a new direction for data compression research.

5.1 Performance

A comparison of this implementation of ACB5 to several commercial compressors is shown in Tables 1-5. The compressors compared on these tables were used by the author of this thesis to compress the files from the Calgary Corpus, Canterbury Corpus, and a proposed addendum to the Canterbury Corpus consisting of large files. These corpuses are commonly accepted as good representations of "general" files which might be compressed, and are therefore often used to compare general compression utilities. The original and compressed file sizes are given in bytes.

This implementation did not perform as well as was expected, but did achieve compression. Several reasons for its poor performance have been given throughout

chapter 4, including the use of a character-based rather than a bit-based model and several shortcuts which were taken during the arithmetic encoding process. This implementation did, however, compress data as well as the fixed-model arithmetic coder from which this implementation's arithmetic coding portion was derived, suggesting that ACB is worthy of further study. Buyanovsky's ACB compression utilities, on the other hand, performed better than any of the other compressors.

Not only does this implementation perform relatively poorly, but it is also very slow. No attempt was made to reduce the execution time of this implementation, since such optimization often reduces code readability, and since this implementation was intended as a learning tool rather than a practical utility. The times in Table 2 were generated on a 350 MHz K6-2 PC.

5.2 Further Research

Several points of departure are suggested by this thesis for future research. Changes to the ACB algorithms themselves as well as studies of specific implementation details could provide progress. Implementation details which could be studied include: alternate methods of encoding portions of the ACB token, bit-based models vs. byte-based models, minimum match length vs. fixed-size associative lists, alternate methods of estimating associative list index probability, and speed optimization. Larger scale ideas which might be of interest include: application of optimizations originally developed for other compression methods, new dictionary organizations, and different probability estimation techniques. Furthermore, no implementation of the other

variations of ACB have been documented in detail. Obviously, there are potentially years of research on ACB related topics which has yet to be done.

Perhaps the performance of Buyanovsky's ACB compression utility provides the most motivation for further research. In the same postscript mentioned earlier Buyanovsky writes, "[Although] at first glance the published algorithm and the one actually programmed are different, their [evolution] is obvious." This utility is an order of magnitude faster and more effective than the implementation presented here, so this faster algorithm is also of great interest. Perhaps one day Buyanovsky can be convinced to explain the details of its implementation.

Table 1. Compression Utilities Compared

Abbreviation	Utility (flags used)
original	Original uncompressed file.
acb	ACB5 implementation presented in this document.
ari	Nelson's static model arithmetic coder.
zip	PkZip 2.04g (-ex).
gz	Gzip 1.2.4 (-9).
rar	WinRar 95 2.02 (a -md1024 -m5 -s).
123	Buyanovsky's ACB 1.23c (uf).
200	Buyanovsky's ACB 2.00c (uf).
These abbreviations are used in Tables 2-5.	

Table 2. Compression Time (Hours : Minutes : Seconds).

File Name	File Size	acb	ari	zip	gz	rar	123	200
bible.txt	4,077,775	2:00:06	0:00:06	0:00:10	0:00:11	0:04:36	0:03:45	0:03:27

Table 3. Compression Results for the Calgary Corpus.

File	Original	acb	ari	zip	gz	rar	123	200
bib	117,541	71,188	76,918	35,701	35,564	33,710	27,178	26,923
book1	785,393	551,450	451,299	315,893	315,826	280,540	224,656	222,802
book2	626,490	388,823	380,057	209,624	209,175	185,411	149,881	147,984
geo	102,400	91,966	73,054	68,810	68,540	67,127	60,039	58,313
news	387,168	268,963	253,410	146,418	146,444	128,558	110,287	109,422
obj1	21,504	14,414	16,372	10,412	10,320	9,987	9,529	9,413
obj2	246,814	132,154	194,260	81,238	81,066	75,355	69,225	67,903
paper1	54,411	33,405	34,350	18,867	18,758	18,580	15,798	15,607
paper2	83,930	53,474	49,039	30,020	29,915	29,242	24,325	24,041
paper3	47,626	31,404	28,220	18,374	18,276	18,217	15,562	15,409
paper4	13,580	8,644	8,173	5,708	5,624	5,694	4,892	4,847
paper5	12,274	7,638	7,764	5,149	5,061	5,116	4,508	4,470
paper6	39,124	23,409	24,859	13,618	13,506	13,446	11,531	11,398
pic	513,216	73,333	108,475	52,513	52,367	52,544	48,270	47,817
progc	41,098	23,381	27,035	13,539	13,357	13,411	11,654	11,570
progl	73,890	32,530	44,740	16,326	16,267	16,198	13,694	13,503
progp	51,345	22,295	31,782	11,392	11,275	11,075	9,382	9,301
trans	94,487	51,213	65,504	19,494	19,132	18,277	15,244	15,097
Total	3,312,291	1,879,684	1,875,311	1,073,096	1,070,473	982,488	825,655	815,820

Table 4. Compression Results for the Canterbury Corpus.

File	Original	acb	ari	zip	gz	rar	123	200
alice29.txt	152,089	96,369	87,191	54,299	54,166	51,674	42,121	41,712
asyoulik.txt	129,301	83,759	78,715	49,060	49,071	47,678	38,717	38,330
cp.html	25,248	14,042	16,735	8,062	8,029	8,089	7,1 45	7,100
fields.c	11,581	5,096	7,430	3,236	3,160	3,196	2,768	2,747
grammar.lsp	3,815	1,753	2,341	1,337	1,257	1,317	1,114	1,105
kennedy.xls	1,029,744	223,322	478,043	204,386	203,292	133,021	158,818	162,457
lcet10.txt	426,754	271,367	249,749	144,663	144,268	128,283	103,946	102,666
plrabn12.txt	481,861	332,464	274,359	194,228	194,481	177,383	143,042	141,568
ptt5	513,216	73,333	108,475	52,515	52,368	52,545	48,261	47,817
sum	38,240	19,851	26,340	12,966	12,772	12,260	11,628	11,431
xargs.1	4,339	2,557	2,789	1,873	1,789	1,846	1,631	1,621
Total	2,816,188	1,123,913	1,332,167	726,625	724,653	617,292	559,191	558,554

Table 5. Compression Results for the Large File Addendum to the Canterbury Corpus.

File	Original	acb	ari	zip	gz	rar	123	200
bible.txt	4,077,775	2,221,834	2,235,424	1,181,186	1,181,190	994,339	757,878	751,694
e.coli	4,638,690	2,169,160	1,160,431	1,306,264	1,301,070	1,294,574	1,162,342	1,159,782
world192.txt	2,473,400	1,640,627	1,551,321	723,473	722,651	546,761	422,307	418,414
total	11,189,865	6,031,621	4,947,176	3,210,923	3,204,911	2,835,674	2,342,527	2,329,890

BIBLIOGRAPHY

Buyanovsky, George M. "ACB, DMC, PPM, LZ - Classification, Idea, Comparison." http://www.cs.pdx.edu/~idr/unbzip2.cgi?compression/acb_compress.html.bz2 (9/15/98).

Buyanovsky, George M., tr. Suzdaltsev, Yuriy "Associative Coding." Monitor. Moscow, Russia. August 1994: 10-19.

Buyanovsky, George M. "Re: Algorithm of ACB." http://www.its.caltech.edu/~bloom/news/bygeorge.html (9/15/97).

Broukhis, Leonid A. "Leo's ACB." http://www.its.caltech.edu/~bloom/news/leoacb.html (9/15/97).

"Calgary Corpus." ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/ (5/8/90).

"Canterbury Corpus." http://corpus.canterbury.ac.nz/ (11/3/98).

Gilchrist, Jeff. "Archive Comparison Test 2.0 (ACT)" http://www.geocities.com/SiliconValley/Park/4264/act.html (3/15/99)

Huffman, David. "A Method for the Construction of Minimum Redundancy Codes." Proceedings of the IRE. 1952. 40(9): 1098-1101.

Nelson, Mark, Jean-loup Gailly. The Data Compression Book. 2nd ed. New York, NY: M&T Books, 1996.

Salomon, David. Data Compression the Complete Reference. Rensselaer, NY: Springer-Verlag New York, Inc., 1997.

Ziv, Jacob, Abraham Lempel. "A Universal Algorithm for Sequential Data Compression" IEEE Transactions on Information Theory. 1977. IT-23(3): 337-343.

Ziv, Jacob, Abraham Lempel. "Compression of Individual Sequences via Variable-Rate Coding." IEEE Transactions on Information Theory. 1978. IT-24(5): 530-536.

APPENDIX

SOURCE CODE FOR ABC5 IMPLEMENTATION

```
#************************** Start of MAKEFILE ************************
#
# ACB Compressor
# Sean Lambert
# 3/15/99
#
# ACB compressor:   make acb-c
# ACB expander:     make acb-e
#
#*********************************************************************


acb-c:  bitio.o errhand.o main-c.o arith-ge.o arith-gm.o acb.o
        gcc -Wall -O3 -o acb-c.exe bitio.o errhand.o main-c.o \
        arith-ge.o arith-gm.o acb.o

acb-e:  bitio.o errhand.o main-e.o arith-ge.o arith-gm.o acb.o
        gcc -Wall -O3 -o acb-e.exe bitio.o errhand.o main-e.o \
        arith-ge.o arith-gm.o acb.o


bitio.o:    bitio.c bitio.h errhand.h
            gcc -c -Wall -O3 bitio.c

errhand.o:  errhand.h
            gcc -c -Wall -O3 errhand.c

main-c.o:   main-c.c bitio.h errhand.h main.h
            gcc -c -Wall -O3 main-c.c

main-e.o:   main-e.c bitio.h errhand.h main.h
            gcc -c -Wall -O3 main-e.c

arith-ge.o: arith-ge.c bitio.h arith-g.h
            gcc -c -Wall -O3 arith-ge.c

arith-gm.o: arith-gm.c arith-g.h acb.h
            gcc -c -Wall -O3 arith-gm.c

acb.o:      acb.c acb.h
            gcc -c -Wall -O3 acb.c


#************************** End of MAKEFILE ************************
```

```
/*************************** Start of MAIN.H ***************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 */

#ifndef _MAIN_H
#define _MAIN_H

#ifdef __STDC__

void CompressFile( FILE *input, BIT_FILE *output, int argc,
    char *argv[] );
void ExpandFile( BIT_FILE *input, FILE *output, int argc,
    char *argv[] );

#else   /* __STDC__ */

void CompressFile();
void ExpandFile();

#endif  /* __STDC__ */

extern char *Usage;
extern char *CompressionName;

#endif  /* _MAIN_H */

/**************************** End of MAIN.H ****************************/
```

```
/*************************** Start of BITIO.H *************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 */

#ifndef _BITIO_H
#define _BITIO_H

#include <stdio.h>

typedef struct bit_file {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
} BIT_FILE;

#ifdef __STDC__

BIT_FILE     *OpenInputBitFile( char *name );
BIT_FILE     *OpenOutputBitFile( char *name );
void          OutputBit( BIT_FILE *bit_file, int bit );
void          OutputBits( BIT_FILE *bit_file,
                          unsigned long code, int count );
int           InputBit( BIT_FILE *bit_file );
unsigned long InputBits( BIT_FILE *bit_file, int bit_count );
void          CloseInputBitFile( BIT_FILE *bit_file );
void          CloseOutputBitFile( BIT_FILE *bit_file );
void        FilePrintBinary( FILE *file, unsigned int code, int bits );

#else   /* __STDC__ */

BIT_FILE     *OpenInputBitFile();
BIT_FILE     *OpenOutputBitFile();
void          OutputBit();
void          OutputBits();
int           InputBit();
unsigned long InputBits();
void          CloseInputBitFile();
void          CloseOutputBitFile();
void          FilePrintBinary();

#endif  /* __STDC__ */

#endif  /* _BITIO_H */

/*************************** End of BITIO.H **************************/
```

```
/************************* Start of ERRHAND.H *************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 */

#ifndef _ERRHAND_H
#define _ERRHAND_H

#ifdef __STDC__

void fatal_error( char *fmt, ... );

#else   /* __STDC__ */

void fatal_error();

#endif  /* __STDC__ */

#endif  /* _ERRHAND_H */

/************************** End of ERRHAND.H **************************/
```

```
/************************* Start of ARITH-G.H *************************
 *
 * This is a generic arithmetic coder by Sean Lambert based on:
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 */

#ifndef _ARITHG_H
#define _ARITHG_H

/*
 * The SYMBOL structure is what is used to define a symbol in
 * arithmetic coding terms.  A symbol is defined as a range between
 * 0 and 1.  Since we are using integer math, instead of using 0 and 1
 * as our end points, we have an integer scale.  The low_count and
 * high_count define where the symbol falls in the range.
 */

typedef struct {
    unsigned short int low_count;
    unsigned short int high_count;
    unsigned short int scale;
} SYMBOL;

extern short int *totals;
extern short int number_of_symbols;
extern unsigned char encoding_complete;
extern unsigned char decoding_complete;

/*
 * Function prototypes, with or without ANSI prototypes.
 */

#ifdef __STDC__

void initialize_encoder_model( FILE *input, FILE *output );
int get_int_to_encode( void );
void adjust_encoder_model( SYMBOL *s );
void initialize_decoder_model( FILE *input, FILE *output );
void put_decoded_int( int c );
void adjust_decoder_model( SYMBOL *s );

#else

void initialize_encoder_model();
int get_int_to_encode();
void adjust_encoder_model();
void initialize_decoder_model();
void put_decoded_int();
void adjust_decoder_model();

#endif

#endif  /* _ARITHG_H */

/************************* End of ARITH-G.H *************************/
```

```
/************************* Start of ACB.H ****************************/
/*
 * ACB Compressor
 * Sean Lambert
 * 3/15/99
 */
/*********************************************************************/


#ifndef _ACB_H
#define _ACB_H


/* constants */
#define INDEX_STEP          ( 1 )
#define DIFFERENCE_BIT_STEP ( 2 )
#define LENGTH_STEP         ( 3 )
#define CHARACTER_STEP      ( 4 )

#define LENGTH_BREAKPOINT           ( 15 )
#define ASSOCIATIVE_LIST_LENGTH     ( 256 )
#define NOTHING_TO_ENCODE_LENGTH    ( 1 )

#define END_OF_STREAM   ( 256 )


/* function prototypes */
#ifdef __STDC__

void initialize_dictionary( void );
void initialize_totals( void );
void find_next_token( void );
void put_decoded_token( void );
void adjust_totals( void );

#else

void initialize_dictionary();
void initialize_totals();
void find_next_token();
void put_decoded_token();
void adjust_totals();

#endif /* __STDC__ */


/* global variables */
extern FILE *input_file;
extern FILE *output_file;

extern int token_index;
extern int token_difference_bit;
extern int token_length;
extern int token_character;

extern int modified_length;

extern int current_associative_list_length;
```

```
extern short int index_totals[];
extern short int difference_bit_totals[];
extern short int length_totals[];
extern short int character_totals[];
extern short int nothing_to_encode_totals[];


#endif  /* _ACB_H */


/************************** End of ACB.H ****************************/
```

```
/************************ Start of MAIN-C.C *************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 *
 * This is the driver program used when testing compression algorithms.
 * In order to cut back on repetitive code, this version of main is
 * used with all of the compression routines.  It in order to turn into
 * a real program, it needs to have another module that supplies one
 * routine and two strings, namely:
 *
 *     void CompressFile( FILE *input, BIT_FILE *output,
 *                        int argc, char *argv );
 *     char *Usage;
 *     char *CompressionName;
 *
 * The main() routine supplied here has the job of checking for valid
 * input and output files, opening them, and then calling the
 * compression routine.  If the files are not present, or no arguments
 * are supplied, it prints out an error message, which includes the
 * Usage string supplied by the compression module.  All of the
 * routines and strings needed by this routine are defined in the
 * main.h header file.
 *
 * After this is built into a compression program of any sort, the
 * program can be called like this:
 *
 *   main-c infile outfile [ options ]
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"
#include "errhand.h"
#include "main.h"

#ifdef __STDC__

void usage_exit( char *prog_name );
void print_ratios( char *input, char *output );
long file_size( char *name );

#else

void usage_exit();
void print_ratios();
long file_size();

#endif

int main( argc, argv )
int argc;
char *argv[];
{
    BIT_FILE *output;
    FILE *input;

    setbuf( stdout, NULL );
```

```
    if ( argc < 3 )
        usage_exit( argv[ 0 ] );
    input = fopen( argv[ 1 ], "rb" );
    if ( input == NULL )
        fatal_error( "Error opening %s for input\n", argv[ 1 ] );
    output = OpenOutputBitFile( argv[ 2 ] );
    if ( output == NULL )
        fatal_error( "Error opening %s for output\n", argv[ 2 ] );
    printf( "\nCompressing %s to %s\n", argv[ 1 ], argv[ 2 ] );
    printf( "Using %s\n", CompressionName );
    CompressFile( input, output, argc - 3, argv + 3 );
    CloseOutputBitFile( output );
    fclose( input );
    print_ratios( argv[ 1 ], argv[ 2 ] );
    return( 0 );
}

/*
 * This routine just wants to print out the usage message that is
 * called for when the program is run with no parameters.  The first
 * part of the Usage statement is supposed to be just the program
 * name.  argv[ 0 ] generally holds the fully qualified path name
 * of the program being run.  I make a half-hearted attempt to strip
 * out that path info and file extension before printing it.  It should
 * get the general idea across.
 */
void usage_exit( prog_name )
char *prog_name;
{
    char *short_name;
    char *extension;

    short_name = strrchr( prog_name, '\\' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, '/' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, ':' );
    if ( short_name != NULL )
        short_name++;
    else
        short_name = prog_name;
    extension = strrchr( short_name, '.' );
    if ( extension != NULL )
        *extension = '\0';
    printf( "\nUsage:  %s %s\n", short_name, Usage );
    exit( 0 );
}

/*
 * This routine is used by main to get the size of a file after
 * it has been closed.  It does all the work, and returns a long.  The
 * main program gets the file size for the plain text, and the size of
 * the compressed file, and prints the ratio.
 */
#ifndef SEEK_END
#define SEEK_END 2
#endif
```

```
long file_size( name )
char *name;
{
    long eof_ftell;
    FILE *file;

    file = fopen( name, "r" );
    if ( file == NULL )
        return( 0L );
    fseek( file, 0L, SEEK_END );
    eof_ftell = ftell( file );
    fclose( file );
    return( eof_ftell );
}

/*
 * This routine prints out the compression ratios after the input
 * and output files have been closed.
 */
void print_ratios( input, output )
char *input;
char *output;
{
    long input_size;
    long output_size;
    int ratio;

    input_size = file_size( input );
    if ( input_size == 0 )
        input_size = 1;
    output_size = file_size( output );
    ratio = 100 - (int) ( output_size * 100L / input_size );
    printf( "\nInput bytes:        %ld\n", input_size );
    printf( "Output bytes:       %ld\n", output_size );
    if ( output_size == 0 )
        output_size = 1;
    printf( "Compression ratio:  %d%%\n", ratio );
}

/************************* End of MAIN-C.C *************************/
```

```
/************************** Start of MAIN-E.C ************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 *
 * This is the driver program used when testing compression algorithms.
 * In order to cut back on repetitive code, this version of main is used
 * with all of the expansion routines.  It in order to turn into a real
 * program, it needs to have another module that supplies a routine and
 * two strings, namely:
 *
 *     void ExpandFile( BIT_FILE *input, FILE *output, int argc, char
*argv );
 *     char *Usage;
 *     char *CompressionName;
 *
 * The main() routine supplied here has the job of checking for valid
 * input and output files, opening them, and then calling the
 * compression routine.  If the files are not present, or no arguments
 * are supplied, it calls the Usage() routine, which is expected to
 * print out the compression type.  All of these routines are defined
 * in the main.h header file.
 *
 * After this is built into an expansion program of any sort, the
 * program can be called like this:
 *
 *   expand infile outfile [ options ]
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"
#include "errhand.h"
#include "main.h"

#ifdef __STDC__

void usage_exit( char *prog_name );

#else

void usage_exit();

#endif

int main( argc, argv )
int argc;
char *argv[];
{
    FILE *output;
    BIT_FILE *input;

    setbuf( stdout, NULL );
    if ( argc < 3 )
    usage_exit( argv[ 0 ] );
    input = OpenInputBitFile( argv[ 1 ] );
    if ( input == NULL )
    fatal_error( "Error opening %s for input\n", argv[ 1 ] );
```

```
    output = fopen( argv[ 2 ], "wb" );
    if ( output == NULL )
    fatal_error( "Error opening %s for output\n", argv[ 2 ] );
    printf( "\nExpanding %s to %s\n", argv[ 1 ], argv[ 2 ] );
    printf( "Using %s\n", CompressionName );
    argc -= 3;
    argv += 3;
    ExpandFile( input, output, argc, argv );
    CloseInputBitFile( input );
    fclose( output );
    putc( '\n', stdout );
    return( 0 );
}

/*
 * This routine just wants to print out the usage message that is
 * called for when the program is run with no parameters.  The first
 * part of the Usage statement is supposed to be just the program
 * name.  argv[ 0 ] generally holds the fully qualified path name
 * of the program being run.  I make a half-hearted attempt to strip
 * out that path info before printing it.  It should get the general
 * idea across.
 */
void usage_exit( prog_name )
char *prog_name;
{
    char *short_name;
    char *extension;

    short_name = strrchr( prog_name, '\\' );
    if ( short_name == NULL )
    short_name = strrchr( prog_name, '/' );
    if ( short_name == NULL )
    short_name = strrchr( prog_name, ':' );
    if ( short_name != NULL )
    short_name++;
    else
    short_name = prog_name;
    extension = strrchr( short_name, '.' );
    if ( extension != NULL )
    *extension = '\0';
    printf( "\nUsage:  %s %s\n", short_name, Usage );
    exit( 0 );
}

/************************** End of MAIN-E.C ************************/
```

```
/************************** Start of BITIO.C *************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 *
 * This utility file contains all of the routines needed to implement
 * bit oriented routines under either ANSI or K&R C.  It needs to be
 * linked with every program used in the entire book.
 */
#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"
#include "errhand.h"

#define PACIFIER_COUNT 2047

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
    return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
    if ( bit_file->mask != 0x80 )
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n" );
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
{
```

```
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void OutputBit( bit_file, bit )
BIT_FILE *bit_file;
int bit;
{
    if ( bit )
        bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
    if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
        fatal_error( "Fatal error in OutputBit!\n" );
    else
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
        putc( '.', stdout );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    }
}

void OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;

    mask = 1L << ( count - 1 );
    while ( mask != 0) {
        if ( mask & code )
            bit_file->rack |= bit_file->mask;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 ) {
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
        fatal_error( "Fatal error in OutputBit!\n" );
        else if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0
)
        putc( '.', stdout );
        bit_file->rack = 0;
            bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
    int value;

    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
            fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
        putc( '.', stdout );
```

```
    }
    value = bit_file->rack & bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 )
    bit_file->mask = 0x80;
    return( value ? 1 : 0 );
}

unsigned long InputBits( bit_file, bit_count )
BIT_FILE *bit_file;
int bit_count;
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0) {
    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
        fatal_error( "Fatal error in InputBit!\n" );
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
        putc( '.', stdout );
    }
    if ( bit_file->rack & bit_file->mask )
            return_value |= mask;
        mask >>= 1;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 )
            bit_file->mask = 0x80;
    }
    return( return_value );
}

void FilePrintBinary( file, code, bits )
FILE *file;
unsigned int code;
int bits;
{
    unsigned int mask;

    mask = 1 << ( bits - 1 );
    while ( mask != 0 ) {
        if ( code & mask )
            fputc( '1', file );
        else
            fputc( '0', file );
        mask >>= 1;
    }
}

/*************************** End of BITIO.C ***************************/
```

```
/************************* Start of ERRHAND.C ************************
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 *
 * This is a general purpose error handler used with every program in
 * the book.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "errhand.h"

#ifdef __STDC__
void fatal_error( char *fmt, ... )
#else
#ifdef __UNIX__
void fatal_error( fmt, va_alist )
char *fmt;
va_dcl
#else
void fatal_error( fmt )
char *fmt;
#endif
#endif
{
    va_list argptr;

    va_start( argptr, fmt );
    printf( "Fatal error: " );
    vprintf( fmt, argptr );
    va_end( argptr );
    exit( -1 );
}

/************************** End of ERRHAND.C *************************/
```

```
/************************** Start of ARITH-GE.C ************************
 *
 * This is a generic arithmetic coder by Sean Lambert based on:
 *
 * Code by Mark Nelson from The Data Compression Book 2nd Ed.
 *
 * Compile with BITIO.C, ERRHAND.C, ARITH-GM.C and either MAIN-C.C or
 * MAIN-E.C
 *
 * This file contains the encoding/decoding engine.  It has been
 * separated from the data modeler to simplify data model
 * modifications.
 */
#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"
#include "arith-g.h"

/*
 * Internal function prototypes, with or without ANSI prototypes.
 */

#ifdef __STDC__

void convert_int_to_symbol( int symbol, SYMBOL *s );
void get_symbol_scale( SYMBOL *s );
int convert_symbol_to_int( int count, SYMBOL *s );
void initialize_arithmetic_encoder( void );
void encode_symbol( BIT_FILE *stream, SYMBOL *s );
void flush_arithmetic_encoder( BIT_FILE *stream );
short int get_current_count( SYMBOL *s );
void initialize_arithmetic_decoder( BIT_FILE *stream );
void remove_symbol_from_stream( BIT_FILE *stream, SYMBOL *s );

#else

void convert_int_to_symbol();
void get_symbol_scale();
int convert_symbol_to_int();
void initialize_arithmetic_encoder();
void encode_symbol();
void flush_arithmetic_encoder();
short int get_current_count();
void initialize_arithmetic_decoder();
void remove_symbol_from_stream();

#endif

/*
 * These four variables define the current state of the arithmetic
 * coder/decoder.  They are assumed to be 16 bits long.  Note that
 * by declaring them as short ints, they will actually be 16 bits
 * on most 80X86 and 680X0 machines, as well as VAXen.
 */
static unsigned short int code;  /* The present input code value */
static unsigned short int low;  /* Start of the current code range */
static unsigned short int high;  /* End of the current code range */
long underflow_bits;  /* Number of underflow bits pending */
```

```
/*
 * This compress file routine is a fairly orthodox compress routine.
 * It first initializes the arithmetic encoder and model.  It then
 * encodes all numbers given to it by the model until it is finished.
 * The output stream is then flushed, and we exit.
 * Note that an extra two bytes are output.  When decoding an arithmetic
 * stream, we have to read in extra bits.  The decoding process takes
 * place in the msb of the low and high range ints, so when we are
 * decoding our last bit we will still have to have at least 15 junk
 * bits loaded into the registers.  The extra two bytes account for
 * that.
 */
void CompressFile( input, output, argc, argv )
FILE *input;
BIT_FILE *output;
int argc;
char *argv[];
{
    int c;
    SYMBOL s;

    initialize_encoder_model( input, output->file );
    initialize_arithmetic_encoder();

    while ( ! encoding_complete ) {
        c = get_int_to_encode();
        convert_int_to_symbol( c, &s );
        encode_symbol( output, &s );
        adjust_encoder_model( &s );
    }
    flush_arithmetic_encoder( output );
    OutputBits( output, 0L, 16 );

    while ( argc-- > 0 ) {
        printf( "Unused argument: %s\n", *argv );
        argv++;
    }
}

/*
 * This expand routine is also very conventional.  It initializes
 * the decoder and the model, then sits in a loop decoding numbers.
 * When the model sets decoding complete, it means we can close
 * up the files and exit.  Note decoding a single number is a three
 * step process:  first we determine what the scale is for the current
 * symbol by looking at the difference between the high an low values.
 * We then see where the current input values fall in that range.
 * Finally, we look in our totals array to find out what symbol is
 * a match.  After that is done, we still have to remove that symbol
 * from the decoder.  Lots of work.
 */

void ExpandFile( input, output, argc, argv )
BIT_FILE *input;
FILE *output;
int argc;
char *argv[];
```

```
{
    SYMBOL s;
    int c;
    int count;

    initialize_decoder_model( input->file, output );
    initialize_arithmetic_decoder( input );

    while ( ! decoding_complete ) {
        get_symbol_scale( &s );
        count = get_current_count( &s );
        c = convert_symbol_to_int( count, &s );
        remove_symbol_from_stream( input, &s );
        put_decoded_int( c );
        adjust_decoder_model( &s );

    }

    while ( argc-- > 0 ) {
        printf( "Unused argument: %s\n", *argv );
        argv++;
    }
}

/*
 * Everything from here down defines the arithmetic coder section
 * of the program.
 */

/*
 * This routine must be called to initialize the encoding process.
 * The high register is initialized to all 1s, and it is assumed that
 * it has an infinite string of 1s to be shifted into the lower bit
 * positions when needed.
 */
void initialize_arithmetic_encoder()
{
    low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/*
 * At the end of the encoding process, there are still significant
 * bits left in the high and low registers.  We output two bits,
 * plus as many underflow bits as are necessary.
 */
void flush_arithmetic_encoder( stream )
BIT_FILE *stream;
{
    OutputBit( stream, low & 0x4000 );
    underflow_bits++;
    while ( underflow_bits-- > 0 )
        OutputBit( stream, ~low & 0x4000 );
}

/*
 * Finding the low count, high count, and scale for a symbol
```

```
 * is really easy, because of the way the totals are stored.
 * This is the one redeeming feature of the data structure used
 * in this implementation.
 */
void convert_int_to_symbol( c, s )
int c;
SYMBOL *s;
{
    s->scale = totals[ number_of_symbols ];
    s->low_count = totals[ c ];
    s->high_count = totals[ c + 1 ];
}

/*
 * Getting the scale for the current context is easy.
 */
void get_symbol_scale( s )
SYMBOL *s;
{
    s->scale = totals[ number_of_symbols ];
}

/*
 * During decompression, we have to search through the table until
 * we find the symbol that straddles the "count" parameter.  When
 * it is found, it is returned. The reason for also setting the
 * high count and low count is so that symbol can be properly removed
 * from the arithmetic coded input.
 */
int convert_symbol_to_int( count, s)
int count;
SYMBOL *s;
{
    int c;

    for ( c = number_of_symbols - 1 ; count < totals[ c ] ; c-- )
        ;
    s->high_count = totals[ c + 1 ];
    s->low_count = totals[ c ];
    return( c );
}

/*
 * This routine is called to encode a symbol.  The symbol is passed
 * in the SYMBOL structure as a low count, a high count, and a range,
 * instead of the more conventional probability ranges.  The encoding
 * process takes two steps.  First, the values of high and low are
 * updated to take into account the range restriction created by the
 * new symbol.  Then, as many bits as possible are shifted out to
 * the output stream.  Finally, high and low are stable again and
 * the routine returns.
 */
void encode_symbol( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;
/*
```

```
     * These three lines rescale high and low for the new symbol.
     */
        range = (long) ( high-low ) + 1;
        high = low + (unsigned short int)
                     (( range * s->high_count ) / s->scale - 1 );
        low = low + (unsigned short int)
                     (( range * s->low_count ) / s->scale );
    /*
     * This loop turns out new bits until high and low are far enough
     * apart to have stabilized.
     */
        for ( ; ; ) {
    /*
     * If this test passes, it means that the MSDigits match, and can
     * be sent to the output stream.
     */
            if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
                OutputBit( stream, high & 0x8000 );
                while ( underflow_bits > 0 ) {
                    OutputBit( stream, ~high & 0x8000 );
                    underflow_bits--;
                }
            }
    /*
     * If this test passes, the numbers are in danger of underflow, because
     * the MSDigits don't match, and the 2nd digits are just one apart.
     */
            else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
                underflow_bits += 1;
                low &= 0x3fff;
                high |= 0x4000;
            } else
                return ;
            low <<= 1;
            high <<= 1;
            high |= 1;
        }
    }

    /*
     * When decoding, this routine is called to figure out which symbol
     * is presently waiting to be decoded.  This routine expects to get
     * the current model scale in the s->scale parameter, and it returns
     * a count that corresponds to the present floating point code:
     *
     *  code = count / s->scale
     */
    short int get_current_count( s )
    SYMBOL *s;
    {
        long range;
        short int count;

        range = (long) ( high - low ) + 1;
        count = (short int)
                ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );
        return( count );
    }
```

```
/*
 * This routine is called to initialize the state of the arithmetic
 * decoder.  This involves initializing the high and low registers
 * to their conventional starting values, plus reading the first
 * 16 bits from the input stream into the code value.
 */
void initialize_arithmetic_decoder( stream )
BIT_FILE *stream;
{
    int i;

    code = 0;
    for ( i = 0 ; i < 16 ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
    low = 0;
    high = 0xffff;
}

/*
 * Just figuring out what the present symbol is doesn't remove
 * it from the input bit stream.  After the character has been
 * decoded, this routine has to be called to remove it from the
 * input stream.
 */
void remove_symbol_from_stream( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;

/*
 * First, the range is expanded to account for the symbol removal.
 */
    range = (long)( high - low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );
/*
 * Next, any possible bits are shipped out.
 */
    for ( ; ; ) {
/*
 * If the MSDigits match, the bits will be shifted out.
 */
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
        }
/*
 * Else, if underflow is threatening, shift out the 2nd MSDigit.
 */
        else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 ) {
            code ^= 0x4000;
            low   &= 0x3fff;
            high  |= 0x4000;
        } else
 /*
```

```
 * Otherwise, nothing can be shifted out, so I return.
 */
            return;
        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        code += InputBit( stream );
    }
}
/************************** End of ARITH-GE.C ************************/
```

```
/************************** Start of ARITH-GM.C **********************/
/*
 * ACB Compressor
 * Sean Lambert
 * 3/15/99
 *
 * This is a data modler by Sean Lambert which plugs into a generic
 * arithmetic coder adapted from code by Mark Nelson.
 *
 * This portion of the modler contains the interface with the coder.
 *
 * Compile with BITIO.C, ERRHAND.C, ARITH-GE.C, ARITH-GM.C and either
 * MAIN-C.C or MAIN-E.C
 */
/*********************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include "arith-g.h"
#include "acb.h"


short int *totals;
short number_of_symbols;
unsigned char encoding_complete;
unsigned char decoding_complete;

char *CompressionName = "ACB model with arithmetic coding";
char *Usage = "in-file out-file\n\n";

static int current_step;


/*********************************************************************
    Prepare the model for its first run through the encoding loop.
*********************************************************************/
void initialize_encoder_model( input, output )
FILE *input;
FILE *output;
{
    input_file = input;
    encoding_complete = 0;

    initialize_dictionary();
    initialize_totals();

    /* The first part of a token is the index. */
    totals = nothing_to_encode_totals;
    number_of_symbols = NOTHING_TO_ENCODE_LENGTH;
    current_step = INDEX_STEP;
}


/*********************************************************************
    At the start of a token, find that token.  Then encode each part
    of the token separately.
```

```
    The length is a special case:
        If the length is less than 63, encode it,
        else encode 63 and remove 63 from the length.
        Repeat until the entire length is output.

    This is because the length is expected to be very small, but can
    be large in some cases.  Since the length is an int, maximum
    length is usually about 2 billion.
*****************************************************************/
int get_int_to_encode()
{
    int c = 0;

    switch ( current_step )
    {
        case INDEX_STEP:
            find_next_token();
            c = token_index;
            break;

        case DIFFERENCE_BIT_STEP:
            c = token_difference_bit;
            /* The original length is needed to update the dictionary
                later, so use a copy here. */
            modified_length = token_length;
            break;

        case LENGTH_STEP:
            if ( modified_length < LENGTH_BREAKPOINT )
            {
                c = modified_length;
                modified_length = -1;
            }
            else
            {
                c = LENGTH_BREAKPOINT;
                modified_length -= LENGTH_BREAKPOINT;
            }
            break;

        case CHARACTER_STEP:
            c = token_character;
            break;
    }

    return( c );
}


/*****************************************************************
    Adjust the model only after the entire token has been written.
    Otherwise, prepare to encode the next part of the token.
*****************************************************************/
void adjust_encoder_model( s )
SYMBOL *s;
{
    switch ( current_step )
    {
```

```
        case INDEX_STEP:
            totals = difference_bit_totals;
            number_of_symbols = 2;
            current_step = DIFFERENCE_BIT_STEP;
            break;

        case DIFFERENCE_BIT_STEP:
            totals = length_totals;
            number_of_symbols = ( LENGTH_BREAKPOINT + 1 );
            current_step = LENGTH_STEP;
            break;

        case LENGTH_STEP:
            /* Proceed only if all length segments are done. */
            if ( modified_length == -1 )
            {
                totals = character_totals;
                number_of_symbols = 257;
                current_step = CHARACTER_STEP;
            }
            break;

        case CHARACTER_STEP:
            if ( token_character == END_OF_STREAM )
            {
                encoding_complete = 1;
            }
            else
            {
                adjust_totals();
                current_step = INDEX_STEP;

                /* Make sure there is something to encode. */
                if ( current_associative_list_length == 0 )
                {
                    totals = nothing_to_encode_totals;
                    number_of_symbols = NOTHING_TO_ENCODE_LENGTH;
                }
                else
                {
                    totals = index_totals;
                    number_of_symbols =
                        current_associative_list_length;
                }
            }
            break;
    }
}


/*********************************************************************
    Prepare the model for its first run through the decoding loop.
*********************************************************************/
void initialize_decoder_model( input, output )
FILE *input;
FILE *output;
{
    output_file = output;
```

```
    decoding_complete = 0;

    initialize_dictionary();
    initialize_totals();

    /* The first part of a token is the index. */
    totals = nothing_to_encode_totals;
    number_of_symbols = NOTHING_TO_ENCODE_LENGTH;
    current_step = INDEX_STEP;
}


/*****************************************************************
    Decode a token step by step, then output the complete token.
*****************************************************************/
void put_decoded_int( c )
int c;
{
    switch ( current_step )
    {
        case INDEX_STEP:
            token_index = c;
            break;

        case DIFFERENCE_BIT_STEP:
            token_difference_bit = c;
            /* The actual length needs to be added up. */
            token_length = 0;
            break;

        case LENGTH_STEP:
            token_length += c;
            /* The modified_length is the last input length segment. */
            modified_length = c;
            break;

        case CHARACTER_STEP:
            token_character = c;
            put_decoded_token();
            break;
    }
}


/*****************************************************************
    Prepare to decode the next part of the token.  If the token is
    complete, adjust the model.
*****************************************************************/
void adjust_decoder_model( s )
SYMBOL *s;
{
    switch ( current_step )
    {
        case INDEX_STEP:
            totals = difference_bit_totals;
            number_of_symbols = 2;
            current_step = DIFFERENCE_BIT_STEP;
            break;
```

```
        case DIFFERENCE_BIT_STEP:
            totals = length_totals;
            number_of_symbols = ( LENGTH_BREAKPOINT + 1 );
            current_step = LENGTH_STEP;
            break;

        case LENGTH_STEP:
            /* Proceed only if all length segments are done. */
            if ( modified_length < LENGTH_BREAKPOINT )
            {
                totals = character_totals;
                number_of_symbols = 257;
                current_step = CHARACTER_STEP;
            }
            break;

        case CHARACTER_STEP:
            if ( token_character == END_OF_STREAM )
            {
                decoding_complete = 1;
            }
            else
            {
                adjust_totals();
                current_step = INDEX_STEP;

                /* Make sure there is something to decode. */
                if ( current_associative_list_length == 0 )
                {
                    totals = nothing_to_encode_totals;
                    number_of_symbols = NOTHING_TO_ENCODE_LENGTH;
                }
                else
                {
                    totals = index_totals;
                    number_of_symbols =
                        current_associative_list_length;
                }
            }
            break;
    }
}

/************************** End of ARITH-GM.C ************************/
```

```
/************************* Start of ACB.C ****************************/
/*
 * ACB Compressor
 * Sean Lambert
 * 3/15/99
 *
 * This is a data modler by Sean Lambert which plugs into a generic
 * arithmetic coder adapted from code by Mark Nelson.
 *
 * This portion of the modler contains the main modelling routines.
 *
 * Compile with BITIO.C, ERRHAND.C, ARITH-GE.C, ARITH-GM.C and either
 * MAIN-C.C or MAIN-E.C
 */
/*********************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include "acb.h"


/* local constants */
#define DICTIONARY_LENGTH    ( 1024 )


/* local function prototypes */
#ifdef __STDC__

void remove_character_from_dictionary( void );
void add_character_to_dictionary( unsigned char new_character );
void build_associative_list( void );
void add_index_to_associative_list( short int new_index,
    short int new_match_length );
void build_totals_from_counts( short int *current_totals,
    unsigned long int *current_counts, short int current_length );
short int find_dictionary_position( short int reference );
short int find_associative_list_position( short int reference );
short int context_match_length( short int current, short int match );
short int content_match_length( short int current, short int match );
short int dictionary_index( short int reference, short int offset );
void print_statistics( void );

#else

void remove_character_from_dictionary();
void add_character_to_dictionary();
void build_associative_list();
void add_index_to_associative_list();
void build_totals_from_counts();
short int find_dictionary_position();
short int find_associative_list_position();
short int context_match_length();
short int content_match_length();
short int dictionary_index();
void print_statistics();

#endif
```

```
/* externed variables */
FILE *input_file;
FILE *output_file;

int token_index = 0;
int token_difference_bit = 0;
int token_length = 0;
int token_character = 0;

int modified_length = 0;

int current_associative_list_length = 0;

short int index_totals[ ASSOCIATIVE_LIST_LENGTH + 1 ];
short int difference_bit_totals[ 3 ];
short int length_totals[ LENGTH_BREAKPOINT + 2 ];
short int character_totals[ 258 ];
short int nothing_to_encode_totals[ NOTHING_TO_ENCODE_LENGTH + 1 ];


/* local global variables */
static short int dictionary_start = 0;
static short int dictionary_end = 0;
static short int sort_start = 0;
static short int sort_end = 0;
short int current_dictionary_length = 0;

static unsigned char dictionary[ DICTIONARY_LENGTH ];
static short int next_index[ DICTIONARY_LENGTH ];
static short int previous_index[ DICTIONARY_LENGTH ];

static short int associative_list[ ASSOCIATIVE_LIST_LENGTH ];

static unsigned long int index_counts[ ASSOCIATIVE_LIST_LENGTH + 1 ];
static unsigned long int length_counts[ LENGTH_BREAKPOINT + 2 ];
static unsigned long int difference_bit_counts[ 3 ];


/**************************************************************************
    Create a new, empty dictionary.
**************************************************************************/
void initialize_dictionary()
{
    dictionary_start = 0;
    dictionary_end = 0;
    sort_start = 0;
    sort_end = 0;
    current_dictionary_length = 0;
}


/**************************************************************************
    Initialize the totals used for arithmetic compression.
**************************************************************************/
void initialize_totals()
{
    short int i;
```

```
    /* The index i will be encoded according to weights determined */
    /* by context match length. */
    /* The first token does not actually need an index. */
    nothing_to_encode_totals[ 0 ] = 0;
    nothing_to_encode_totals[ 1 ] = 1;

    /* The difference bit d will be adaptively encoded. */
    for ( i = 0; i < 2; i++ )
    {
        difference_bit_counts[ i ] = 1;
        difference_bit_totals[ i ] = i;
    }
    difference_bit_counts[ 2 ] = 2;
    difference_bit_totals[ 2 ] = 2;

    /* The length l will be adaptively encoded. */
    for ( i = 0; i <= LENGTH_BREAKPOINT; i++ )
    {
        length_counts[ i ] = 1;
        length_totals[ i ] = i;
    }
    length_counts[ LENGTH_BREAKPOINT + 1 ] = LENGTH_BREAKPOINT + 1;
    length_totals[ LENGTH_BREAKPOINT + 1 ] = LENGTH_BREAKPOINT + 1;

    /* The character c is considered to be evenly distributed, with */
    /* The exception of the EOF character, which is less likely. */
    /* The numbers are scaled as close to 16383 as possible. */
    for ( i = 0; i < 257; i++ )
    {
        character_totals[ i ] = i * 63;
    }
    character_totals[ 257 ] = character_totals[ 256 ] + 1;
}


/*************************************************************************
    Find the next token to encode.  This is the heart of the model
    and is the most complicated part.
*************************************************************************/
void find_next_token()
{
    int i;
    short int bottom;
    short int top;
    int next_character = 0;
    short int copy_list[ ASSOCIATIVE_LIST_LENGTH ];
    short int temp_index;
    int temp_length;
    unsigned char dictionary_update[ DICTIONARY_LENGTH ];


    /* Special case: associative list is empty, no index to encode. */
    if ( current_associative_list_length == 0 )
    {
        /* The nothing to encode table requires an index of 0. */
        token_index = 0;
        token_length = 0;
```

```
    /* Input a character from the file. */
    next_character = getc( input_file );
    if ( next_character == EOF )
    {
        next_character = END_OF_STREAM;
    }

    /* Determine the difference bit */
    if ( next_character == 1 )
    {
        token_difference_bit = 0;
    }
    else
    {
        token_difference_bit = 1;
    }

    /* Find length, matching input with all 0s or all 1s. */
    while ( ( ( token_difference_bit == 1 ) &&
              ( next_character == 0 ) ) ||
        ( ( token_difference_bit == 0 ) &&
            ( next_character == 255 ) ) )
    {
        token_length++;

        /* Input a character from the file. */
        next_character = getc( input_file );
        if ( next_character == EOF )
        {
            next_character = END_OF_STREAM;
        }
    }

    /* A character was found which did not match. */
    token_character = next_character;

    /* Now that the token has been found, update the dictionary. */
    i = token_length;
    while ( i > 0 )
    {
        if ( token_difference_bit == 1 )
        {
            add_character_to_dictionary( 0 );
        }
        else
        {
            add_character_to_dictionary( 255 );
        }
        i--;
    }
    if ( token_character != END_OF_STREAM )
    {
        add_character_to_dictionary( token_character );
    }

    /* Store total length for output. */
    temp_length = token_length;
}
```

```
    else
    {

/*--------------------------------------------------------------------*/

        /* Search the associative list for the best match. */
        token_index = current_associative_list_length;
        token_length = 0;

        /* Start at the list ends and squeeze inward. */
        bottom = 0;
        top = current_associative_list_length - 1;
        for ( i = bottom; i <= top; i++ )
        {
            copy_list[ i ] = associative_list[ i ];
        }

        /* Keep looking until the correct index is found. */
        while ( token_index == current_associative_list_length )
        {
            /* Update copy list to current match length. */
            for ( i = bottom; i <= top; i++ )
            {
                copy_list[ i ] =
                    dictionary_index( copy_list[ i ], 1 ) ;
            }

            /* Input a character from the file. */
            next_character = getc( input_file );
            if ( next_character == EOF )
            {
                next_character = END_OF_STREAM;
            }

            /* Squeeze the boundaries if possible. */
            while ( ( top > bottom ) &&
                ( next_character <
                    (int)dictionary[ copy_list[ top ] ] ) )
            {
                top--;
            }
            while ( ( top > bottom ) &&
                ( next_character >
                    (int)dictionary[ copy_list[ bottom ] ] ) )
            {
                bottom++;
            }

            /* See if the index has been determined. */
            if ( top == bottom )
            {
                token_index = top;
            }
            else
            {
                token_length++;
            }
        }
```

```
        /* Index has been found, now find length. */
        while ( next_character ==
            (int)dictionary[ copy_list[ token_index ] ] )
        {
            token_length++;

            /* Update copy list to current match length. */
            copy_list[ token_index ] =
                dictionary_index( copy_list[ token_index ], 1 ) ;

            /* Input a character from the file. */
            next_character = getc( input_file );
            if ( next_character == EOF )
            {
                next_character = END_OF_STREAM;
            }
        }

        /* A character was found which did not match. */
        token_character = next_character;

        /* Determine the difference bit. */
        if ( next_character <
            (int)dictionary[ copy_list[ token_index ] ] )
        {
            token_difference_bit = 0;
        }
        else
        {
            token_difference_bit = 1;
        }

/*-----------------------------------------------------------------------*/

        /* Store total length for dictionary update. */
        temp_length = token_length;

        /* Shorten the length if possible. */
        if ( ( token_difference_bit == 0 ) && ( token_index == 0 ) )
        {
            temp_index =
                dictionary_index( associative_list[ token_index ], 1 );
            while ( dictionary[ temp_index ] == 0 )
            {
                /* Find the next character. */
                temp_index = dictionary_index( temp_index, 1 );
                token_length--;
            }
        }
        else if ( ( token_difference_bit == 1 ) &&
            ( token_index == current_associative_list_length - 1 ) )
        {
            temp_index =
                dictionary_index( associative_list[ token_index ], 1 );
            while ( dictionary[ temp_index ] == 255 )
            {
                /* Find the next character. */
                temp_index = dictionary_index( temp_index, 1 );
```

```
                token_length--;
            }
        }
        else if ( token_difference_bit == 0 )
        {
            token_length -= content_match_length(
                associative_list[ token_index ],
                associative_list[ token_index - 1 ] );
        }
        else
        {
            token_length -= content_match_length(
                associative_list[ token_index ],
                associative_list[ token_index + 1 ] );
        }

/*--------------------------------------------------------------------*/

        /* Now that the token has been found, update the dictionary. */
        temp_index = associative_list[ token_index ];
        i = temp_length;
        top = 0;
        while ( i > 0 )
        {
            /* Find the next character. */
            temp_index = dictionary_index( temp_index, 1 );
            dictionary_update[ top ] = dictionary[ temp_index ];
            i--;
            top = ( top + 1 ) % DICTIONARY_LENGTH;
        }

        /* The dictionary update is loaded.  Update the dictionary. */
        if ( temp_length < DICTIONARY_LENGTH )
        {
            for ( i = 0; i < temp_length; i++ )
            {
                add_character_to_dictionary(
                    dictionary_update[ i ] );
            }
        }
        else
        {
            /* Add only the last DICTIONARY_LENGTH characters to */
            /* the dictionary. */
            for ( i = 0; i < DICTIONARY_LENGTH; i++ )
            {
                add_character_to_dictionary(
                    dictionary_update[ top ] );
                top = ( top + 1 ) % DICTIONARY_LENGTH;
            }
        }

        /* Don't forget the token character! */
        if ( token_character != END_OF_STREAM )
        {
            add_character_to_dictionary( token_character );
        }
    }
```

```
    /* Output statistics when encoding is done. */
    if ( token_character == END_OF_STREAM )
    {
        print_statistics();
    }
}


/*****************************************************************************
    Decode the token and update the dictionary.  This is somewhat
    easier than finding the token.
*****************************************************************************/
void put_decoded_token()
{
    int i;
    short int top;
    short int temp_index;
    int temp_length;
    unsigned char dictionary_update[ DICTIONARY_LENGTH ];


    /* Store total length for dictionary update. */
    temp_length = token_length;

    /* Special case: associative list is empty, no index to encode. */
    if ( current_associative_list_length == 0 )
    {
        /* Now that the token has been found, update the dictionary. */
        i = token_length;
        while ( i > 0 )
        {
            if ( token_difference_bit == 1 )
            {
                add_character_to_dictionary( 0 );
                putc( (unsigned char)0, output_file );
            }
            else
            {
                add_character_to_dictionary( 255 );
                putc( (unsigned char)255, output_file );
            }
            i--;
        }
        if ( token_character != END_OF_STREAM )
        {
            add_character_to_dictionary( token_character );
            putc( (unsigned char)token_character, output_file );
        }
    }
    else
    {

/*-------------------------------------------------------------------------*/

        /* Expand the length if it was shortened. */
        if ( ( token_difference_bit == 0 ) && ( token_index == 0 ) )
        {
            temp_index =
```

```
                dictionary_index( associative_list[ token_index ], 1 );
            while ( dictionary[ temp_index ] == 0 )
            {
                /* Find the next character. */
                temp_index = dictionary_index( temp_index, 1 );
                temp_length++;
            }
        }
        else if ( ( token_difference_bit == 1 ) &&
            ( token_index == current_associative_list_length - 1 ) )
        {
            temp_index =
                dictionary_index( associative_list[ token_index ], 1 );
            while ( dictionary[ temp_index ] == 255 )
            {
                /* Find the next character. */
                temp_index = dictionary_index( temp_index, 1 );
                temp_length++;
            }
        }
        else if ( token_difference_bit == 0 )
        {
            temp_length += content_match_length(
                associative_list[ token_index ],
                associative_list[ token_index - 1 ] );
        }
        else
        {
            temp_length += content_match_length(
                associative_list[ token_index ],
                associative_list[ token_index + 1 ] );
        }

/*-------------------------------------------------------------------------*/

        /* Now that the token has been found, update the dictionary. */
        temp_index = associative_list[ token_index ];
        i = temp_length;
        top = 0;
        while ( i > 0 )
        {
            /* Find the next character. */
            temp_index = dictionary_index( temp_index, 1 );
            dictionary_update[ top ] = dictionary[ temp_index ];
            putc( dictionary[ temp_index ], output_file );
            i--;
            top = ( top + 1 ) % DICTIONARY_LENGTH;
        }

        /* The dictionary update is loaded.  Update the dictionary. */
        if ( temp_length < DICTIONARY_LENGTH )
        {
            for ( i = 0; i < temp_length; i++ )
            {
                add_character_to_dictionary(
                    dictionary_update[ i ] );
            }
        }
```

```
        else
        {
            /* Add only the last DICTIONARY_LENGTH characters to */
            /* the dictionary. */
            for ( i = 0; i < DICTIONARY_LENGTH; i++ )
            {
                add_character_to_dictionary(
                    dictionary_update[ top ] );
                top = ( top + 1 ) % DICTIONARY_LENGTH;
            }
        }

        /* Don't forget the token character! */
        if ( token_character != END_OF_STREAM )
        {
            add_character_to_dictionary( token_character );
            putc( (unsigned char)token_character, output_file );
        }
    }

    /* Output statistics when encoding is done. */
    if ( token_character == END_OF_STREAM )
    {
        print_statistics();
    }
}


/**************************************************************************
    Rebuild the associative list and update the length totals since
    length is adaptively encoded.
**************************************************************************/
void adjust_totals()
{
    /* Get a new associative list. */
    build_associative_list();
    build_totals_from_counts( index_totals, index_counts,
        current_associative_list_length );

    /* Update the length counts and totals.  Keep in mind the */
    /* unusual method the length is encoded. */
    while ( token_length >= LENGTH_BREAKPOINT )
    {
        token_length -= LENGTH_BREAKPOINT;
        length_counts[ LENGTH_BREAKPOINT ]++;
        length_counts[ LENGTH_BREAKPOINT + 1 ]++;
    }
    length_counts[ token_length ]++;
    length_counts[ LENGTH_BREAKPOINT + 1 ]++;
    build_totals_from_counts( length_totals, length_counts,
        ( LENGTH_BREAKPOINT + 1 ) );

    /* Update the difference bit totals. */
    difference_bit_counts[ token_difference_bit ]++;
    difference_bit_counts[ 2 ]++;
    build_totals_from_counts( difference_bit_totals,
        difference_bit_counts, 2 );
}
```

```
/*************************************************************************
    The functions below are internal to this file.
*************************************************************************/


/*************************************************************************
    Remove the first character in the dictionary (primarily used to
    make space when the dictionary is full.)
*************************************************************************/
void remove_character_from_dictionary()
{
    if ( current_dictionary_length > 1 )
    {
        /* Remove the character from the sorting. */
        if ( sort_start == dictionary_start )
        {
            /* The character is first in the sort. */
            sort_start = next_index[ dictionary_start ];
            previous_index[ sort_start ] = DICTIONARY_LENGTH;
        }
        else if ( sort_end == dictionary_start )
        {
            /* The character is last in the sort. */
            sort_end = previous_index[ dictionary_start ];
            next_index[ sort_end ] = DICTIONARY_LENGTH;
        }
        else
        {
            /* The character is in the middle of the sort. */
            next_index[ previous_index[ dictionary_start ] ] =
                next_index[ dictionary_start ];
            previous_index[ next_index[ dictionary_start ] ] =
                previous_index[ dictionary_start ];
        }

        current_dictionary_length--;
        dictionary_start =
            ( dictionary_start + 1 ) % DICTIONARY_LENGTH;
    }
    else
    {
        /* Dictionary is or becomes empty. */
        initialize_dictionary();
    }
}


/*************************************************************************
    Add the given character to the dictionary and sort it
    appropriately.  The dictionary is sorted using a list of next and
    prevoius indices, and a sort start and a sort end index.  The
    next index for the last character and the previous index for the
    first character will be DICTIONARY_LENGTH, which is an invalid
    index.
*************************************************************************/
void add_character_to_dictionary( new_character )
unsigned char new_character;
```

```
{
    short int position;


    /* Make room if needed. */
    if ( current_dictionary_length == DICTIONARY_LENGTH )
    {
        remove_character_from_dictionary();
    }

    current_dictionary_length++;

    if ( current_dictionary_length == 1 )
    {
        /* First dictionary entry. */
        dictionary_end = dictionary_start;
        sort_start = dictionary_start;
        sort_end = dictionary_start;
        next_index[ dictionary_start ] = DICTIONARY_LENGTH;
        previous_index[ dictionary_start ] = DICTIONARY_LENGTH;
        dictionary[ dictionary_start ] = new_character;
    }
    else
    {
        /* Add the new character. */
        dictionary_end = ( dictionary_end + 1 ) % DICTIONARY_LENGTH;
        dictionary[ dictionary_end ] = new_character;

        /* Move the character into sorted position. */
        position = find_dictionary_position( dictionary_end );

        if ( position == DICTIONARY_LENGTH )
        {
            /* The character should be first. */
            next_index[ dictionary_end ] = sort_start;
            previous_index[ dictionary_end ] = DICTIONARY_LENGTH;
            previous_index[ sort_start ] = dictionary_end;
            sort_start = dictionary_end;
        }
        else if ( position == sort_end )
        {
            /* The character should be last. */
            next_index[ dictionary_end ] = DICTIONARY_LENGTH;
            next_index[ position ] = dictionary_end;
            previous_index[ dictionary_end ] = position;
            sort_end = dictionary_end;
        }
        else
        {
            /* The chararacter is in the middle of the sort. */
            next_index[ dictionary_end ] = next_index[ position ];
            previous_index[ dictionary_end ] = position;
            next_index[ position ] = dictionary_end;
            previous_index[ next_index[ dictionary_end ] ] =
                dictionary_end;
        }
    }
}
```

```
/*************************************************************************
    Build the associative list, which is a list of the dictionary
    indices that have the best context match with the current position
    (the last dictionary entry.)
*************************************************************************/
void build_associative_list()
{
    short int down_index;
    short int up_index;
    short int down_match_length = 0;
    short int up_match_length = 0;


    /* Initialize associative list. */
    current_associative_list_length = 0;

    /* Initialize the local variables. */
    if ( current_dictionary_length > 1 )
    {
        down_index = previous_index[ dictionary_end ];

        if ( down_index != DICTIONARY_LENGTH )
        {
            down_match_length =
                context_match_length( dictionary_end, down_index );
        }

        up_index = next_index[ dictionary_end ];
        if ( up_index != DICTIONARY_LENGTH )
        {
            up_match_length =
                context_match_length( dictionary_end, up_index );
        }
    }
    else
    {
        /* There are no entries to add to the associative list. */
        down_index = DICTIONARY_LENGTH;
        up_index = DICTIONARY_LENGTH;
    }

    /* Fill the associative list as long as there are dictionary */
    /* entries left to add and the associative list is not full. */
    while ( ( up_index != down_index ) &&
        ( current_associative_list_length !=
            ASSOCIATIVE_LIST_LENGTH ) )
    {
        if ( up_index == DICTIONARY_LENGTH )
        {
            /* Top of dictionary has been reached. */
            add_index_to_associative_list( down_index,
                down_match_length );
            down_index = previous_index[ down_index ];
            if ( down_index != DICTIONARY_LENGTH )
            {
                down_match_length =
                    context_match_length( dictionary_end, down_index );
```

```
            }
        }
        else if ( down_index == DICTIONARY_LENGTH )
        {
            /* Bottom of dictionary has been reached. */
            add_index_to_associative_list( up_index, up_match_length );
            up_index = next_index[ up_index ];
            if ( up_index != DICTIONARY_LENGTH )
            {
                up_match_length =
                    context_match_length( dictionary_end, up_index );
            }
        }
        else if ( up_match_length > down_match_length )
        {
            /* The up match is better than the down match. */
            add_index_to_associative_list( up_index, up_match_length );
            up_index = next_index[ up_index ];
            if ( up_index != DICTIONARY_LENGTH )
            {
                up_match_length =
                    context_match_length( dictionary_end, up_index );
            }
        }
        else
        {
            /* The down match is better than the up match, or they */
            /* are the same length. */
            add_index_to_associative_list( down_index,
                down_match_length );
            down_index = previous_index[ down_index ];
            if ( down_index != DICTIONARY_LENGTH )
            {
                down_match_length =
                    context_match_length( dictionary_end, down_index );
            }
        }
    }
}


/*************************************************************************
    Add the new index to the associative list, if there is not already
    a duplicate entry.  Sort the associative list by content.
*************************************************************************/
void add_index_to_associative_list( new_index, new_match_length  )
short int new_index;
short int new_match_length;
{
    short int position;
    short int i;


    /* See if this is the first entry in the associative list. */
    if ( current_associative_list_length == 0 )
    {
        current_associative_list_length++;
        associative_list[ 0 ] = new_index;
```

```
            index_counts[ 0 ] = new_match_length + 1;
            index_counts[ 1 ] = new_match_length + 1;
        }
        else
        {
            position = find_associative_list_position( new_index );

            /* Add the new index to the list if the entry does not */
            /* duplicate an entry already in the list. */
            if ( position <= current_associative_list_length )
            {
                /* Make position the location the new index will occupy. */
                if ( position == current_associative_list_length )
                {
                    position = 0;
                }
                else
                {
                    position++;
                }

                /* Shift the associative list down to make room. */
                index_counts[ current_associative_list_length + 1 ] =
                    index_counts[ current_associative_list_length ];
                for ( i = current_associative_list_length; i > position;
                    i-- )
                {
                    associative_list[ i ] = associative_list[ i - 1 ];
                    index_counts[ i ] = index_counts[ i - 1 ];
                }

                /* Add the new information. */
                current_associative_list_length++;
                associative_list[ position ] = new_index;
                index_counts[ position ] = new_match_length + 1;
                index_counts[ current_associative_list_length ] +=
                    ( new_match_length + 1 );
            }
        }
}


/*****************************************************************************
    Build the totals list from a counts list.  Make sure the final
    total is not above 16383.  Still has potential for very large
    files to overflow the counts, the sum of which cannot be greater
    than ~0UL, usually about 4 billion.
*****************************************************************************/
void build_totals_from_counts( current_totals, current_counts,
    totals_length )
short int *current_totals;
unsigned long int *current_counts;
short int totals_length;
{
    double scale;
    short int i;
```

```
    if ( current_counts[ totals_length ] > 16383 )
    {
        /* Add up the totals and scale them. */
        current_totals[ 0 ] = 0;
        scale = 16383 / (double)( current_counts[ totals_length ] -
            totals_length );

        for ( i = 0; i < totals_length; i++ )
        {
            current_totals[ i + 1 ] = current_totals[ i ] + 1 +
                (short int)( (double)current_counts[ i ] * scale );
        }
    }
    else
    {
        /* Add up the totals. */
        current_totals[ 0 ] = 0;
        for ( i = 0; i < totals_length; i++ )
        {
            current_totals[ i + 1 ] =
                current_totals[ i ] + current_counts[ i ] ;
        }
    }
}


/***************************************************************************
    Find the index of the dictionary entry which would be sorted
    directly before the reference entry.  If the reference entry
    would be sorted first, return DICTIONARY_LENGTH.
***************************************************************************/
short int find_dictionary_position( reference )
short int reference;
{
    short int length;
    short int current_position;
    short int current_index;
    short int match_index;
    unsigned char position_found = 0;


    /* Search bottom up for simplicity. */
    current_position = sort_end;

    while ( ! position_found )
    {
        current_index = current_position;
        match_index = reference;
        length = 0;

        /* Find the end of the current match. */
        while ( ( length < current_dictionary_length ) &&
            ( dictionary[ current_index ] ==
                dictionary[ match_index ] ) )
        {
            length++;
            current_index = dictionary_index( current_index, -1 ) ;
            match_index = dictionary_index( match_index, -1 ) ;
```

```
        }

        if ( dictionary[ current_index ] <= dictionary[ match_index ] )
        {
            /* The character comes after the current position. */
            position_found = 1;
        }
        else if ( current_position == sort_start )
        {
            /* The character comes first. */
            current_position = DICTIONARY_LENGTH;
            position_found = 1;
        }
        else
        {
            /* The correct position has not been found */
            current_position = previous_index[ current_position ];
        }
    }

    return ( current_position );
}


/*************************************************************************
    Find the index of the associative list entry which would be sorted
    directly before the reference entry.  If the reference entry
    would be sorted first, return current_associative_list_length.
    If the reference is a duplicate of an entry already in the
    associative list, return current_associative_list_length + 1.
*************************************************************************/
short int find_associative_list_position( reference )
short int reference;
{
    short int associative_match_length;
    short int i;
    short int current_position;
    short int current_index;
    short int match_index;
    unsigned char position_found = 0;


    /* Search bottom up for simplicity. */
    i = current_associative_list_length - 1;
    current_position = associative_list[ i ];

    while ( ! position_found )
    {
        current_index = current_position;
        match_index = reference;
        associative_match_length = 0;

        /* Content begins after the current character. */
        current_index = dictionary_index( current_index, 1 ) ;
        match_index = dictionary_index( match_index, 1 ) ;

        /* Find the end of the current match. */
        while ( ( associative_match_length <
```

```
                current_dictionary_length ) &&
            ( dictionary[ current_index ] ==
                dictionary[ match_index ] ) )
        {
            associative_match_length++;
            current_index = dictionary_index( current_index, 1 ) ;
            match_index = dictionary_index( match_index, 1 ) ;
        }

        if ( associative_match_length == current_dictionary_length )
        {
            /* The character duplicates another position. */
            i = current_associative_list_length + 1;
            position_found = 1;
        }
        else if ( dictionary[ current_index ] <=
            dictionary[ match_index ] )
        {
            /* The character comes after the current position. */
            position_found = 1;
        }
        else if ( i == 0 )
        {
            /* The character comes first. */
            i = current_associative_list_length;
            position_found = 1;
        }
        else
        {
            /* The correct position has not been found */
            i--;
            current_position = associative_list[ i ];
        }
    }

    return ( i );
}


/*************************************************************************
    Find the length of the context match between two parts of the
    dictionary.  If the entire dictionary is included in the match,
    return the dictionary length.
*************************************************************************/
short int context_match_length( current, match )
short int current;
short int match;
{
    short int length = 0;
    short int current_index;
    short int match_index;


    current_index = current;
    match_index = match;

    /* Match can be 0 to current_dictionary_length. */
    while ( ( length < current_dictionary_length ) &&
```

```
        ( dictionary[ current_index ] == dictionary[ match_index ] ) )
    {
        length++;
        current_index = dictionary_index( current_index, -1 ) ;
        match_index = dictionary_index( match_index, -1 ) ;
    }

    return ( length );
}


/*************************************************************************
    Find the length of the content match between two parts of the
    dictionary.  If the entire dictionary is included in the match,
    return the dictionary length.
*************************************************************************/
short int content_match_length( current, match )
short int current;
short int match;
{
    short int length = 0;
    short int current_index;
    short int match_index;


    current_index = current;
    match_index = match;

    /* Content begins after the current character. */
    current_index = dictionary_index( current_index, 1 ) ;
    match_index = dictionary_index( match_index, 1 ) ;

    /* Match can be 0 to current_dictionary_length. */
    while ( ( length < current_dictionary_length ) &&
        ( dictionary[ current_index ] == dictionary[ match_index ] ) )
    {
        length++;
        current_index = dictionary_index( current_index, 1 ) ;
        match_index = dictionary_index( match_index, 1 ) ;
    }

    return ( length );
}


/*************************************************************************
    Return the index into the dictionary which results from adding the
    offset to the reference index.  If the reference index is not in
    the dictionary then return DICTIONARY_LENGTH.
*************************************************************************/
short int dictionary_index( reference, offset )
short int reference;
short int offset;
{
    short int modified_offset;
    short int index;
```

```
modified_offset = offset;
index = reference;

/* Find out if reference index is in the dictionary. */
if ( ( current_dictionary_length == 0 ) ||
     ( ( dictionary_start <= dictionary_end ) &&
         ( ( index < dictionary_start ) ||
             ( index > dictionary_end ) ) ) ||
     ( ( dictionary_start > dictionary_end ) &&
         ( ( index < dictionary_start ) &&
             ( index > dictionary_end ) ) ) )
{
    index = DICTIONARY_LENGTH;
}
else
{
    /* Move the index one step at a time until done. */
    while ( modified_offset != 0 )
    {
        /* Decide which direction to shift. */
        if ( modified_offset > 0 )
        {
            if ( index == dictionary_end )
            {
                index = dictionary_start;
            }
            else if ( index == ( DICTIONARY_LENGTH - 1 ) )
            {
                index = 0;
            }
            else
            {
                index++;
            }
            modified_offset--;
        }
        else
        {
            if ( index == dictionary_start )
            {
                index = dictionary_end;
            }
            else if ( index == 0 )
            {
                index = DICTIONARY_LENGTH - 1;
            }
            else
            {
                index--;
            }
            modified_offset++;
        }
    }
}

/* Output a warning if this function failed. */
if ( index == DICTIONARY_LENGTH )
{
```

```
            printf("~");
        }

        return ( index );
}


/*************************************************************************
    Print things that would be useful to know.
*************************************************************************/
void print_statistics()
{
    short int i;


    /* Counts for length: */
    printf( "\n\nCounts for length:\n" );
    for ( i = 0; i <= LENGTH_BREAKPOINT + 1; i++ )
    {
        printf( "%2d: %ld\n", i, length_counts[ i ] );
    }

    /* Counts for difference bit: */
    printf( "\nCounts for difference bit:\n" );
    for ( i = 0; i <= 2; i++ )
    {
        printf( "%2d: %ld\n", i, difference_bit_counts[ i ] );
    }

    printf( "\n" );
}


/************************** End of ACB.C ****************************/
```