A fast, constant-order, symbol ranking text compressor

Peter Fenwick

Technical Report 145
ISSN 1173-3500
April 16, 1997

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
peter-f@cs.auckland.ac.nz

Abstract

Recent work on "symbol ranking" text compressors included a version which combines moderate compression with high speed and has been described in a poster at the 1997 Data Compression Conference. That work is extended here, especially to produce a compressor capable of efficient hardware implementation.

The compressor is based on a conventional set-associative cache, with LRU update. To process a symbol, a context of the three preceding symbols is hashed to access a particular cache line. If the access "hits", the LRU index is emitted as the recoded symbol, or otherwise the symbol is emitted and the LRU status updated. Several versions are described, with some improving the performance by maintaining a Move-To-Front list of symbols and emitting some symbols as "short" 5-bit indices into this table, or by encoding runs of correctly-predicted symbols.

The final performance is in the range of 3.5 – 4 bits per byte over the Calgary Corpus, with software speeds of up to 1 Mbyte/s. The best of the hardware designs should run at 100Mbyte/s with discrete components, or 200–300 Mbyte/s with a completely integrated design.

1. Introduction

In earlier work the author investigated a class of "symbol ranking" lossless text compressors, leading to one paper already published[2] and another presented as a poster at the 1997 Data Compression Conference[3] and also submitted for journal publication. This last work includes a novel lossless compressor which is extended within this Report.

Symbol ranking compression is based on techniques first described by Shannon[4] when investigating the entropy of printed English. He postulated a "predictor" which, examining the preceding text or context, prepared a list of possible symbols in ranked order of likelihood in that context.

- The encoder recodes each symbol into its position or "rank" in the list for the current context; experience shows that the recoded symbol distribution is highly skewed and may be considerably compacted by an entropy encoder.
- The decoder prepares a similar list of symbols for the current context, recovers the incoming rank by an entropy decoder, and then delivers the appropriate symbol from the ranked list.

Most of the earlier work concentrated on compressors which gave good compression, usually at the cost of compression speed. In general they predicted symbols first at a high-order context, dropping down to shorter contexts as needed. Here we emphasise speed, while achieving only moderate compression, and use constant-order, or fixed length, contexts.

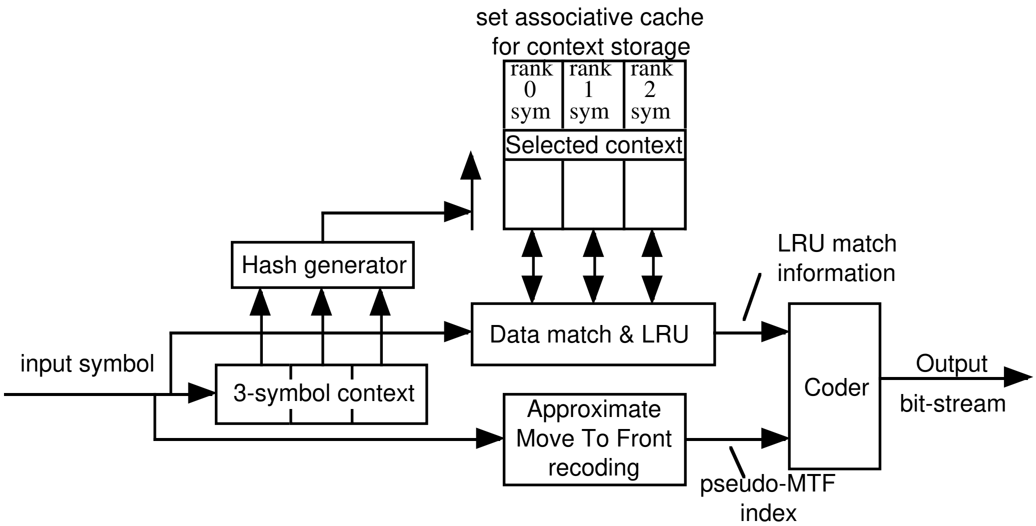

Figure 1. A hardware constant-order symbol ranking compressor

2. Constant order Symbol Ranking Compression

The heart of a symbol-ranking compressor is the management of the lists of ranked symbols for each context. The mechanism here is precisely that of a standard set-associative cache with LRU (Least Recently Used) updating, but used in an unconventional manner. In a normal cache we are concerned only with hits and misses; the LRU mechanism is completely private to the cache and is irrelevant to its external operation. Here we are definitely concerned with the LRU functions, with the position of a hit within the LRU list used to encode a symbol. Figure 1 shows the compressor, as it might be implemented in hardware.

The three previous symbols are used as an order-3 context, with the 6 low bits of each symbol concatenated as an index to access one line of the cache[1] . The input symbol is matched against the LRU-ordered symbols, the match position is presented to the coder and the LRU status updated appropriately, using any standard method such as a permutation vector or physical data shuffling. Earlier versions used 4-way set-association, the 4 bytes fitting conveniently into a 32-bit word, but the 3-way version shown gives very similar performance and is slightly simpler.

The final coding stage accepts the LRU match information and emits it with a defined variable-length coding, or the symbol itself if there is no LRU match. The coding is essentially unary, with a 0 for a correct prediction and a 1 for a failure and step to the next rank. An incoming symbol which matches the most recent member of the LRU set is emitted as a "rank–0" code. A match to the middle member is emitted as "rank–1", and a match to the oldest of the three members of the set as "rank–2".

Two additional techniques improve compression by about 15%

1. A Move-To-Front recoding of the symbol allows a shorter code for the more frequent symbols. While full MTF recoding is far too expensive in data movement, a similar effect can be achieved by exchanging the symbol with one halfway to the front of the MTF list [1]. A "short literal" of 5 bits (actually a 5 bit index into the MTF table) gives the best performance. Other literals are transmitted as the full 8-bit index. Dithering the last 3 bits of the exchanged position gives an improvement of about 5%. (Thomborson and Wei[5] discuss a systolic implementation of a compressor with full MTF recoding.)
2. Runs of more than about 16 0's are transmitted as an actual run length. The number of bits in the run length is sent as a "long literal" but with a value in the range of a short literal; no true long literal indices have these values.
3. If the empty contexts are initialised to more-probable symbols, some symbols from "virgin" contexts may be emitted as ranks rather than literals. For binary files the symbols are probably of little importance, except that a NUL (`0x00`) should be present. For text files a sensible choice is {' ', 'e' , 't'}. As a compromise, all contexts are initialised to `0x00652000` or {'e', ' ', 'NUL'}, (rank–0 is least-significant byte).

The compressor therefore uses two similar but quite distinct data movement operations

1. Each context has its own LRU list of the most recent symbols in that context. This list is shuffled or otherwise updated during each cache reference.
2. The Move-To-Front mechanism processes all of the symbols so that more-frequent symbols can be emitted as "short" literals for better compression.

[1] This approach gives about 10% better compression than a "good" hash function. It seems that it helps to retain some of input structure when forming the hash index.

The term “LRU” will *always* refer to the first usage, in conjunction with the cache and individual contexts, and “MTF” to the second, for handling literals.

The full coding of this compressor is shown in Table 1, together with the code frequencies for compressing PAPER1 with 64K contexts.

code		usage	Paper1 freq
0		Rank–0 match	45%
10	*xxxxx*	short literal (index < 32)	28%
110		Rank–1 match	15%
1110		Rank–2 match	7%
1111	*xxxxxxxx*	Long literal; also long run of Rank–0	4%

Table 1. Output coding for constant-order compressor, 3 ranks

Compression for the whole Calgary Corpus is shown in Table 2.

Bib	*Book1*	*Book2*	*Geo*	*News*	*Obj1*	*Obj2*	*Paper1*	*Paper2*	*Pic*	*ProgC*	*ProgL*	*ProgP*	*Trans*	*AVG*
3.70	3.94	3.41	6.28	3.86	4.76	3.69	3.63	3.61	1.15	3.60	2.65	2.79	2.76	**3.56**

Table 2. Constant-order symbol ranking compressor – 64K contexts, 3 ranks

Increasing the maximum rank to 3 (LRU list length to 4, or 4-way set association) improves the compression of text files by about 2%, but degrades the compression of binary files, leaving a similar overall result. Less-compressible files , especially GEO and OBJ1, improve as the maximum rank is reduced. Coding only ranks 0 and 1 with 64K contexts (and adjusting the coding to handle only these ranks) gives 6.10 bit/byte for GEO and 4.66 with OBJ1, improvements of 6% over coding up to rank = 3.

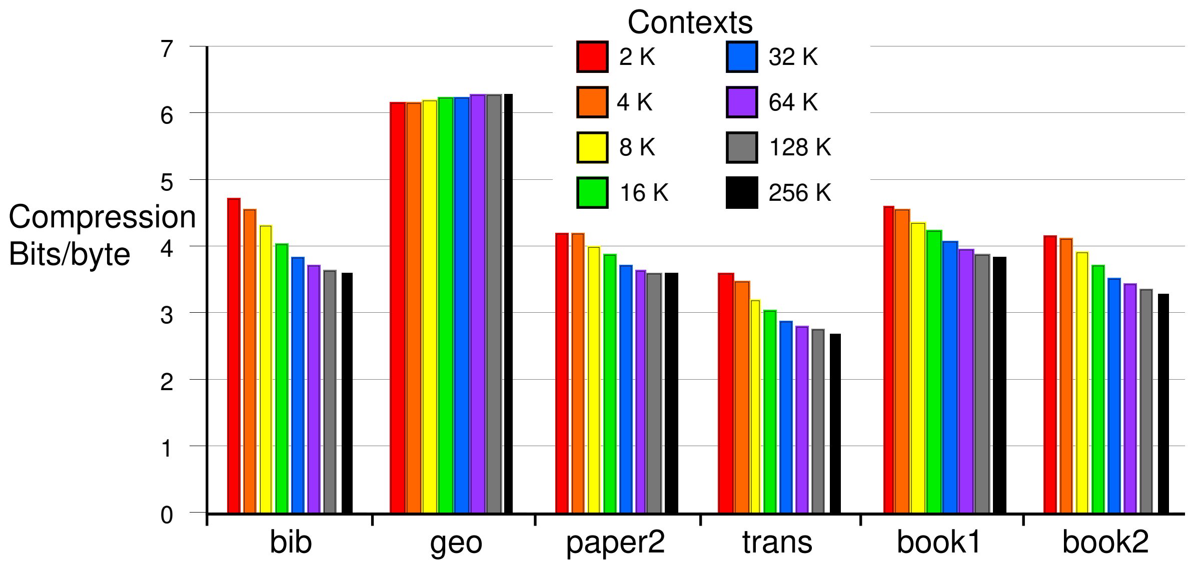

Figure 2. Constant-order compressor, compression vs number of contexts

Results for representative files of the Calgary corpus are shown in Figure 2. Useful compression requires only a few thousand contexts for most files The performance for varying order with 32K contexts is shown for a few files in Figure 3. Again, GEO and other binary files are quite anomalous, with the compression being better with fewer contexts.

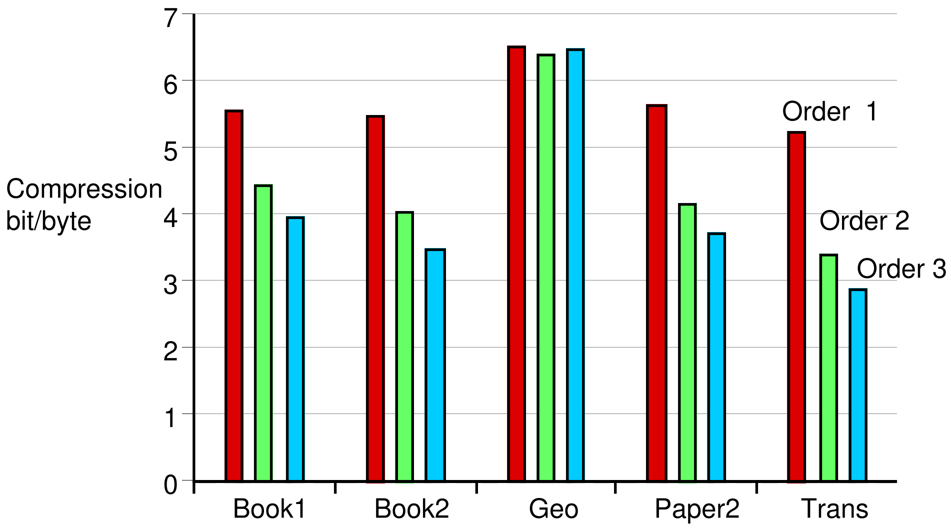

Figure 3. Constant-order symbol ranking compressor – 32K contexts, compression vs context order

3. Hardware Implementation

The compressor is based on a conventional cache and should be suitable for direct hardware implementation. A preliminary examination shows that the major bottleneck is the pseudo Move-To-Front operation for processing literals, even in its simplified form. Each symbol needs 6 references into the MTF tables; overlapping the references can reduce this to 3 cycle times, but no further. Further reduction may be possible with multi-port memory but that is more expensive and needs extra complication to resolve problems from collisions.

Accordingly, we first remove all of the optimisations given in the previous sections. Literals are now always encoded as 8 bits and there is no encoding of rank–0 runs. End-of-File is signalled by emitting as a literal the rank–0 symbol of the current context.

With these simplifications, and coding shown in Table 3, the compression for the Calgary Corpus is shown in Table 4.

code		meaning	Paper1 freq
0		Rank-0 match	45%
10	*xxxxxxxx*	Literal	33%
110		Rank–1 match	15%
111		Rank–2 match	7%

Table 3. Output coding for constant-order compressor (hardware)

Bib	*Book1*	*Book2*	*Geo*	*News*	*Obj1*	*Obj2*	*Paper1*	*Paper2*	*Pic*	*ProgC*	*ProgL*	*ProgP*	*Trans*	*AVG*

4.23	4.92	4.18	6.10	4.62	5.21	3.84	4.43	4.48	1.91	4.26	3.18	3.24	3.20	**4.13**

Table 4. Constant-order symbol ranking compressor (HW) – 64K contexts,maximum rank = 2

As a further improvement which is compatible with a fast hardware implementation we restore the encoding of runs of rank–0 codes. A run is now signalled by a literal matching the rank–1 symbol[2]; the "literal" is followed by 1 or more bytes, each encoding 7 bits of the run length, least significant byte first. Each byte except the last has its high-order bit set to indicate "more to come". The results of Table 5 indicate a small improvement over Table 4 for PIC and, to a lesser extent, OBJ1.

Bib	*Book1*	*Book2*	*Geo*	*News*	*Obj1*	*Obj2*	*Paper1*	*Paper2*	*Pic*	*ProgC*	*ProgL*	*ProgP*	*Trans*	*AVG*
4.04	4.74	4.00	6.15	4.43	5.02	3.80	4.25	4.32	1.34	4.08	3.03	3.13	3.00	**3.95**

Table 5. Constant-order symbol ranking compressor, with run-encoding –

64K contexts,maximum rank = 2

An alternative approach, following from the author's experience with the Burrows-Wheeler block sorting compressor, might be to detect runs in the *raw data* rather than in the compressor output. Briefly, any sequence of *N* identical symbols is always followed by a count of the number of following similar symbols; the count is treated as symbols to be compressed.

4. Conclusions

Results from the earlier tables are collected in Table 6, together with results from the compressors "compact", "compress" and "gzip" as reference. The final column shows the degradation of the "hardware" run-encoded, relative to the "full coding" version (with MTF and short literals). Prose files are about 20% poorer, other text files 10–15% poorer and some binary files about the same.

[2] This encoding requires special care —

- The rank-1 byte of the first symbol of the run must be saved, to be emitted later as the "literal" to signal the run.
- The contexts array *must* be initialised with different rank–0 and rank–1 symbols in all contexts (the value used is 0x00652000).

	UNIX compact	UNIX compress	gzip level-1	gzip level-9	full coding 64K contexts	HW cache 32k contexts	HW cache 64K contexts	HW cache 64k ctxs runs	Degradation of "HW runs" rel to "full coding"
Bib	5.24	3.89	3.15	2.51	3.70	4.23	4.11	4.04	9%
Book1	4.56	4.06	3.80	3.25	3.94	4.92	4.76	4.74	20%
Book2	4.83	4.25	3.26	2.70	3.41	4.18	4.04	4.00	17%
Geo	5.70	6.10	5.45	5.34	6.28	6.10	6.15	6.15	-2%
News	5.23	4.90	3.48	3.06	3.86	4.62	4.51	4.43	15%
Obj1	6.06	6.15	3.98	3.84	4.76	5.21	5.23	5.02	6%
Obj2	6.30	5.19	3.04	2.63	3.69	3.84	3.83	3.80	3%
Paper1	5.04	4.43	3.25	2.79	3.63	4.43	4.39	4.25	17%
Paper2	4.65	3.98	3.41	2.89	3.61	4.48	4.42	4.32	20%
Pic	1.66	0.99	1.02	0.82	1.14	1.91	1.91	1.34	17%
ProgC	5.26	4.41	3.12	2.68	3.59	4.26	4.30	4.08	14%
ProgL	4.81	3.57	2.24	1.80	2.64	3.18	3.17	3.03	15%
ProgP	4.92	3.72	2.17	1.81	2.79	3.24	3.24	3.13	12%
Trans	5.58	3.94	2.05	1.61	2.76	3.20	3.11	3.00	9%
AVG	**4.99**	**4.26**	**3.10**	**2.70**	**3.56**	**4.13**	**4.08**	**3.95**	
		performance loss, relative to full coding				16.02%	14.75%	11.06%	

Table 6. Complete results

The main feature of this compressor is its speed. On a 275MHz DEC Alpha, a software implementation compresses at about 1 Mbyte/s. A hardware implementation uses well-proven cache techniques and with discrete components such as field-programmable logic arrays and fast static RAM a speed of 100 Mbyte/s should be possible with little trouble. A completely integrated design should be able to run at several times that speed.

References

1. Fenwick, P.M. , "A New Technique for Self Organising List Searches", *Computer Journal*, pp 450–454, Oct. 1991.
2. Fenwick, P.M. "Symbol Ranking Text Compression with Shannon Recodings", *J.UCS*, February 1997.
3. Fenwick, P.M. "Symbol Ranking text compressors", *Data Compression Conference DCC–97*, Snowbird Utah, March 1997, p 436.
4. Shannon, C.E. "Prediction and Entropy of Printed English", *Bell System Technical Journal*, Vol 30, pp 50–64, Jan 1951.
5. Thomborson, C.D. and Wei, B. W-Y., "Systolic Implementations of a Move-To-Front Text Compressor", *Proc AM SPAA (Special Purpose Algorithms and Architectures).* also *Comp Arch News,* Vol 19, No 1, pp53–60 March 1991.
6. Arnold, R., Bell, T., "A corpus for the evaluation of lossless compression algorithms", *Data Compression Conference DCC–97*, Snowbird Utah, March 1997, pp 201–210.

Appendix A. Results on the "Canterbury Corpus"

A recent development in text compression is the announcement by Arnold and Bell[6] of a new corpus of test files, as an alternative to the established "Calgary Corpus". In general, their work in establishing the new corpus has validated the value of the Calgary Corpus, even though that first one was assembled in a rather ad hoc manner.

Figure A1 contains some selected results from compressing the Canterbury Corpus[3], to which have been added results from the symbol ranking compressor (3 ranks, 64K contexts).

Test_file	text	fax	Csrc	Excl	SPRC	tech	poem	html	lisp	man	play	**AVG**
gzip level 9	2.85	0.82	2.24	1.63	2.67	2.71	3.23	2.59	2.65	3.31	3.12	**2.53**
gzip level 1	3.43	1.02	2.63	1.90	2.96	3.26	3.80	2.94	2.89	3.53	3.63	**2.91**
srank 64K	3.57	1.14	3.15	2.02	3.66	3.38	3.91	3.60	3.43	4.19	3.83	**3.14**
compress	3.27	0.97	3.56	2.41	4.21	3.06	3.38	3.68	3.90	4.43	3.51	**3.31**
pack (or compact)	4.62	1.66	5.12	3.60	5.42	4.70	4.58	5.30	4.87	5.10	4.85	**4.53**

Figure A1. Comparative results on the "Canterbury Corpus".

[3] Information on the Canterbury Corpus is available from `http://www.cosc.canterbury.ac.nz/~tim/corpus`

Appendix B. Source Code.

Included here is a complete listing of the “full coding” compressor, to run as a stand-alone Unix program. This source code is available from `ftp.cs.auckland.ac.nz/out/peter-f/srank.c`.

```
/*..:....|....:....|....:....|....:....|....:....|....:....|....:....|...*/

/*
  Symbol ranking text compressor P M Fenwick 5 September 1996.

  V 0.2  9 Sep 1996
         * uses 32-bit words as context,
         * move most globals into code/decode procedures to aid
            register allocation
         * adds variable coding order (with transmission to decoder)
         * run without parameters displays help messages.

         -S reports symbol-rank counts and percentages
         -F reports symbol-index counts from MTF
         -D specifies number of MTF dither bits

  V 0.5  uses 6-bit ASCII subsets to generate contexts.
         Unary coding for short runs

  V 1.0  Removes rank=3 symbol to allow shorter codes - negligible change.
    1.1  Changes initialisation of Contexts array (10 Apr 97)

  The basis of symbol-ranking compression is that we have a list of symbols
  ordered according to their likelihood in the current context.
  On compression, each input symbol is translated into its index in that
  list, with the most likely symbol 'Rank-0', the next 'Rank-1', and so on.
  On expansion, the received index is used to get the symbol from the list.

  This version is intended for high speed rather than good compression.

  It is based on what is really a 4-way set associative cache, with LRU
  update.  The preceding 3-symbol context is hashed and used to select
  a context, which has 3 symbols in LRU order. Each context has
  its own "cache line" -- usually 16--64K contexts are held.

  Each word in the "Contexts" array holds the three ranked symbols, from
    Rank-2 (Most Sig) to Rank-0 (least Sig)

  For a cache "hit" we encode the LRU position -- rank=0 for most recent
  through to rank=2 for least recent.

  For a miss we could encode the actual symbol, but rather do an approximate
  MTF to favour more frequent symbols.

  After coding, each symbol is moved to the front of the LRU list for its context

  The complete coding is --

      0               rank-0
      10 xxxxx        literal, MTF index < 32
      110             rank-1
      1110            rank-2
      1111 xxxxxxxx   literal, index >= 32 (also EOF, with code = 0)

Short runs of rank-0 are emitted "as is".
Longer runs are emitted as a run-length bit count emitted as a long literal
followed by the length. (Such values cannot occur in a proper literal.)

The "pseudo MTF" mechanism has a table of symbols in approximate MTF order.
When one is referenced it is exchanged with one half-way to the front of
the table. A mapping table is included to accelerate the operation, giving
a constant 6 memory references for any symbol.
A small dither is added to the exchange-position to improve performance.

"Stats" should be 0 for normal use, but 1 to enable extended statistics reports.
Enabling statistics slows execution slightly, but no other effect
*/

#define Stats 0         /* ########### 0 or 1 to control extended statistics */
```

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>
#include <string.h>
#include <signal.h>

#define srName      "srank"

#define srVersion   "1"
#define srLevel     "0"
#define srSuffix    ".sr"

#define outMode "wb"            /* may have to be "w" on some computers */

#define usChar  unsigned char

#if (INT_MAX > 2000000L)        /* an integer of 32 bits */
  #define int32 int
  #define us32  unsigned int
#else
  #define int32 long
  #define us32  unsigned long
#endif

#define runThresh    16         /* encode 0 runs longer than this */

FILE    *inFile,                /* the main input file */
        *outFile;               /* the main output file */

time_t sttTime, endTime;        /* to time the operation */

us32    *Contexts,              /* the table of contexts */
        mask1, mask23;
int     ChToIx[260],            /* for pseudo MTF */
        IxToCh[260];            /*  "   "   "   "  */

int     report = 1,             /* enable reporting */
        Order = 3,                  /* encoding order */
        error = 0;              /* error indicator */
#if Stats
   int  dithMask =  7;          /* mask to dither the MTF index */
#else
  #define dithMask 7
#endif
#define errFinish  1            /* just quietly finish */
#define errRemove  2            /* must remove output file */

#if Stats
  #define logSize    300
  int   Trigger,                /* log from this position */
        logCnt;                 /* symbols left to log */
#endif

int32   ctxParam = 15,          /* initially 2^15 contexts */
        currContexts = -1,      /* number of contexts */
        ctxMask,                /* get context index from hash */
        grpBits = 3,            /* sets contexts in group */
        grpMask,
        compBytes, dataBytes,   /* data counters */
        runLength;              /* rank-0 length */

char    Header[20],             /* the file header */
        outSuffix[8],           /* output suffix */
        inName[80], outName[80];        /* file names */

/* ---------- statistics counters ---------- */

#if Stats
  long R0, R1, R2, shortLit, longLit, Runs,
       Ranks[260],
       Lengths[20];
#endif
```

```
/* ---------- prototype declarations --------- */

void startoutputtingbits(void);
void doneoutputtingbits(void);
void startinputtingbits(void);

int  makeHeader();
void emitHeader(int V);
int  checkHeader();
void Initialize();
void emitRun();
void decodeFile();
void encodeFile();
void writeReport();
void printHelp();
char convByte(int byte);
int prog(int argc, char *argv[]); /* used if program is run as a subroutine */

/* ===== Convert a symbol to a printable code (Macintosh specific) ===== */

#if Stats
  char convByte(int byte)
    {
    byte &= 0xff;
    if (byte >= ' ') return byte;      /* normal character */
    if (byte == '\n') return '©';      /* new-line   LF    */
    if (byte ==   0 ) return '°';      /* zero       NUL   */
    if (byte == '\r') return '®';      /* return     CR    */
    if (byte == '\t') return 'Δ';      /* hor tab    HT    */
    return '•';                        /* something < space */
    }
#endif

/* ========== input output routines ========== */

us32    BitBuffer,   /* the bit output buffer */
        BitsInBuf,   /* bits in the buffer */
        theBits,     /* return value from getBits and lookBits */
        Masks[17] = {0, 0x1,    0x3,    0x7,    0xF,
                        0x1F,   0x3F,   0x7F,   0xFF,
                        0x1FF,  0x3FF,  0x7FF,  0xFFF,
                        0x1FFF, 0x3FFF, 0x7FFF, 0xFFFF};

void startoutputtingbits(void)
    {
    BitBuffer = 0;      /* clear the buffer */
    BitsInBuf = 0;      /* and the bit count */
    }

#define putBits(Bits, N)                                            \
    {                                                               \
    BitBuffer = (BitBuffer << N) | Bits;                            \
    BitsInBuf += N;                                                 \
    while (BitsInBuf >= 8)      /* write buffer to file */          \
      {                                                             \
      putc((BitBuffer >> (BitsInBuf-8)), outFile);                  \
      BitsInBuf -= 8;                                               \
      BitBuffer &= Masks[BitsInBuf];    /* remaining bits */        \
      compBytes ++;                                                 \
      }
\
    }

void doneoutputtingbits(void)
    {
    while (BitsInBuf > 8)       /* process high-order bytes */
      {
      putc(BitBuffer >> (BitsInBuf - 8), outFile);
      BitsInBuf -= 8;
      compBytes ++;
      }
    if (BitsInBuf > 0)  /* and the low-order bits */
      {
      putc(BitBuffer << (8-BitsInBuf), outFile);
      compBytes ++;
```

```
      }
    }

void startinputtingbits(void)
    {
    BitBuffer = 0;      /* clear the buffer */
    BitsInBuf = 0;      /* and the bit count */
    }

/* ========== read N bits from the input buffer ========== */
/* the value is returned in the global variable 'theBits' */

#define getBits(N)                                              \
    {                                                           \
    while (BitsInBuf < (N))     /* ensure enough bits */        \
      {                                                         \
      BitBuffer = (BitBuffer << 8) | getc(inFile);              \
      BitsInBuf += 8;           /* another byte in buff */      \
      compBytes ++;                                             \
      }                                                         \
    theBits = (BitBuffer >> (BitsInBuf - N));  /* align */      \
    BitsInBuf -= N;                     /* N bits taken */      \
    BitBuffer &= Masks[BitsInBuf];      /* select bits */       \
    }

/* ========== inspect N bits from the input buffer ========== */

#define lookBits(N)                                             \
    {                                                           \
    while (BitsInBuf < (N))                                     \
      {                                                         \
      BitBuffer = (BitBuffer << 8) | getc(inFile);              \
      BitsInBuf += 8;                                           \
      compBytes ++;                                             \
      }                                                         \
    theBits = BitBuffer >> (BitsInBuf - N);                     \
    }

/* ========== process the file header ========== */
/*
   It has the form "srzp#VLnm", where "srzp" are those letters,
   V is a version number, L is a level number,
   n is a value indicating the context size, and
   m is the compression order (1, 2 or 3)
*/

int makeHeader()        /* make header, return its length */
    {
    strcpy(Header, srName);
    strcat(Header, "#");
    strcat(Header, srVersion);
    strcat(Header, srLevel);
    return strlen(Header);
    }

void emitHeader(int V1)                 /* emit the header bytes */
    {
    int L, i;

    L = makeHeader();
    for (i = 0; i < L; i ++)
      putBits(Header[i], 8);
    putBits(V1 & 0xFF, 8);              /* context size */
    putBits(Order, 8);                  /* compression order */
    }

int checkHeader()                       /* check the header */
    {
    int i, L, V1;

    L = makeHeader();
    for (i = 0; i < L; i++)
      {
      getBits(8);
      if (theBits != Header[i])
        {
```

```
        error = errRemove;
        return -1;                      /* error value */
        }
      }
    getBits(8);                         /* the parameter */
    V1 = theBits;
    getBits(8);
    Order = theBits;                    /* set decompression order */
    return V1;                          /* return context size info */
    }

/* ====== initialise the coding models etc, and emit values ====== */

void Initialize(int log2Ctx)
    {
    int i, ctxSize;

    ctxSize = 1 << log2Ctx;
    if (ctxSize != currContexts)        /* check for changed context size */
      {
      if (Contexts != NULL)
        free(Contexts);                 /* release old space */
      Contexts = (us32 *) calloc(ctxSize+8, sizeof(us32));
      if (Contexts == NULL)             /* failure */
        {
        fprintf(stderr, "Contexts allocation failure - %ld contexts\n",
                       ctxSize);
        exit(1);
        }
      currContexts = ctxSize;           /* remember number of contexts */
      }

    for (i = 0; i < ctxSize; i++)
      Contexts[i] = 0x00652000;         /* initialise to  {NUL, ' ', 'e'} */

    ctxMask = (ctxSize-1);    /* get valid context index */

    compBytes = dataBytes = runLength = 0;

    for (i = 0; i < 256; i++)
      ChToIx[i] = IxToCh[i] = i;        /* set up pseudo MTF table */

#if Stats
    R0 = R1 = R2 = shortLit = longLit = Runs = 0;
    for (i = 0; i < 260; i++)
      Ranks[1] = 0;
    for (i = 0; i < 20; i++)
      Lengths[i] = 0;
#endif
    }

/* ========== ========== */

/* maintain a sort of MTF list by exchanging a character with one
   half-way to the start of the list.
*/

#define pseudoMTF(currCh, Dither)
\
    {
\
    int currIx, exchCh, exchIx;
\

\
    currIx = ChToIx[currCh];    /* position of current character */     \
    exchIx = currIx >> 1;       /* new posn of current character */     \
    exchIx ^= ((Dither) & dithMask);                                    \
    exchCh = IxToCh[exchIx];    /* character to be exchanged */         \
                                                                        \
    ChToIx[currCh] = exchIx;    /* new position of current char */      \
    IxToCh[exchIx] = currCh;                                            \
    ChToIx[exchCh] = currIx;    /* new posn of exchange char */         \
    IxToCh[currIx] = exchCh;                                            \
    }
```

```
/* ===== code for ISO Fletcher checksum ===== */

#define addCheckSum(C)                                          \
    {                                                           \
    check1 += C;               /* add character into check1 */ \
    if (check1 >= 255)         /* ... modulo 255 */            \
      check1 -= 255;                                            \
    check2 += check1;          /* add check1 into check2 */    \
    if (check2 >= 255)         /* ... modulo 255 */            \
      check2 -= 255;                                            \
    }

/* ========== report on compression performance ========== */

void writeReport()
    {
    float procTime, Rate;
    char self[8], vers[8];
    time_t theTime;
#if Stats
    int i, f;
#endif

    procTime = (1.0*(endTime-sttTime))/CLOCKS_PER_SEC;
    Rate = (1.0e-6*dataBytes)/procTime;

    if (report > 1)
      {
      theTime = time(NULL);

      strcpy(self, srName);
      strcpy(vers, srVersion);
      strcat(vers, ".");
      strcat(vers, srLevel);
      fprintf(stderr, "'%s V%s' file '%s', %ld contexts, order %d, %s",
              self, vers, inName, currContexts, Order, ctime(&theTime));
      }

    fprintf(stderr, "%ld data bytes, %ld comp. bytes, %5.3f bit/byte",
             dataBytes, compBytes,(8.0*compBytes)/dataBytes);
    fprintf(stderr, "%7.3f sec, %5.3f Mbyte/s\n", procTime, Rate);

#if Stats
    if (report > 2)                            /* report rank statistics */
      {
      fprintf(stderr, "%6ld R0, %ld R1, %ld R2, %ld short, %ld long, %ld runs\n",
                       R0, R1, R2, shortLit, longLit, Runs);
      f = dataBytes/100;
      fprintf(stderr, "%5ld%% R0, %3ld%% R1, %2ld%% R2, %3ld%% short, %2ld%% long\n",
                       R0/f, R1/f, R2/f, shortLit/f, longLit/f);
      fprintf(stderr, "Lengths ");
      for (i = 1; i < 11; i++)
        fprintf(stderr, " %d:%ld,", i, Lengths[i]);
      fprintf(stderr, "\n");

      if (report > 3)                         /* report on symbol index frequencies */
        for (i = 0; i < 256; i++)
          fprintf(stderr, "%ld\n", Ranks[i]);
      }
#endif

    if (report > 1)
      fprintf(stderr, "\n");
    }

/* ========== expand a file ========== */

void decodeFile()
    {
    int headVal,               /* value from header record */
        rxCheck1=0, rxCheck2=0; /* checksums from trailer */

    int SymIx,                 /* index in pseudo MTF table */
        symbol,                /* THE decoded symbol */
        check1, check2;        /* ISO checksums */
#if Stats
```

```
   int  code;
   char rank;
#endif

   us32 W,                      /* the selected context */
        prev1, prev2, prev3,    /* hash codes of prev symbols */
        runLength,
        Hx;                     /* and its index */

    prev1 = prev2 = prev3 = runLength = 0;
    check1 = check2 = 0;

    sttTime = clock();
    startinputtingbits();       /* start input */
    headVal = checkHeader();    /* check header */
    if (headVal < 0)
      {
      fprintf(stderr, "Invalid file header\n");
      exit(1);
      }
    Initialize(headVal);

    for (;;)                    /* start of main decoding loop */
      {
#if Stats
     if (dataBytes == Trigger)
       logCnt = logSize;                /* start logging */
     SymIx = -1;
#endif

      Hx = (prev1 | prev2 | prev3) & mask23 & ctxMask;
      W = Contexts[Hx];         /* select context  */

      if (runLength > 0)
        {
        symbol = W & 0xFF;       /* rank-0 symbol, in run */
        runLength --;
#if Stats
        R0++;
        rank = '0';
        code = 0;
#endif
        }
      else
        {
        lookBits(4);                    /* get code into 'theBits' */
        if ((theBits & 8) == 0)         /* 0xxx = Rank-0 */
          {
          getBits(1);
          symbol = W & 0xFF;            /* rank-0 symbol */
#if Stats
          R0++;
          rank = '0';
          code = 0;
#endif
          }
        else if ((theBits & 12) == 8)   /* 10xx = Short literal */
          {
          getBits(7);                   /* 2 prefix & 5 data */
          SymIx  = theBits & 0x1F;      /* remove prefix */
          symbol = IxToCh[SymIx];       /* get symbol from index */
          Contexts[Hx] = (W << 8) | symbol; /* force as most recent symbol */
#if Stats
          shortLit++;
          Ranks[SymIx] ++;
          code = 10;
          rank = 'S';
#endif
          }
        else if ((theBits & 14) == 12)  /* 110x = Rank-1 */
          {
          getBits(3);
          symbol = (W >> 8) & 0xFF;     /* fetch the rank-1 symbol */
          Contexts[Hx] =  (W & 0xFFFF0000) |
                         ((W & 0xFF) << 8) |
                         symbol;
```

```
#if Stats
          R1++;
          rank = '1';
          code = 110;
#endif
          }
        else if ((theBits & 15) == 14)  /* 1110 = Rank-2 */
          {
          getBits(4);
          symbol = (W >> 16) & 0xFF;     /* fetch the rank-2 symbol */
          Contexts[Hx] =  (W & 0xFF000000) |
                         ((W & 0xFFFF) << 8) |
                         symbol;
#if Stats
          R2++;
          rank = '2';
          code = 1110;
#endif
          }
        else if (theBits == 15)         /* 1111 = long Literal */
          {
          getBits(12);                  /* get code & literal */
          SymIx = theBits & 0xFF;       /* remove prefix code */
          if (SymIx == 0)               /* end of file */
            {
            getBits(8);                 /* get the checksums */
            rxCheck1 = theBits;
            getBits(8);
            rxCheck2 = theBits;
            if ((check1 != rxCheck1) || (check2 != rxCheck2))   /* final check */
              {
              fprintf(stderr, "File '%s' - invalid checksum\n", inName);
              error = errRemove;
              }
            break;
            }
          else if (SymIx < 32)          /* encoded run */
            {
            getBits(SymIx);             /* SymIx has bit-length of run-length */
            runLength = theBits + runThresh - 1; /* get length & adjust */
            symbol = W & 0xff;          /* first symbol of run */
#if Stats
            Runs++;
            if (logCnt > 0)
              fprintf(stderr, "    Run %ld\n", runLength + 1);
#endif
            }
          else                          /* long literal */
            {
            symbol = IxToCh[SymIx];
            Contexts[Hx] = (W << 8) | symbol;
#if Stats
            longLit++;
            rank = 'L';
            Ranks[SymIx] ++;
            code = 11111;
#endif
            }
          }
        else
          {
          fprintf(stderr, "File '%s' -- decode failure - code = %d, dataBytes = %ld\n",
                inName, theBits, dataBytes);
          error = errRemove;
          break;
          }
        } /* end of main decode */

#if Stats
      if (logCnt-- > 0)
        {
        fprintf(stderr, "%7ld < %-7ld '%c' %02X  %c\"%c%c%c\"  rank %c, code %-6d",
                       dataBytes, compBytes, convByte(symbol), symbol,
                       convByte(W>>24),
                       convByte(W), convByte(W>>8), convByte(W>>16),
                       rank, code);
```

```
        if (SymIx >= 0)
          fprintf(stderr, " %3d", SymIx);
        fprintf(stderr, "\n");
        }
#endif

      dataBytes++;
      pseudoMTF(symbol, dataBytes);     /* update pseudo MTF table */
      prev3 = (prev2 << 6);             /* move along hash codes */
      prev2 = ((prev1 & 0x3F) << 6);
      prev1 = symbol & mask1;           /* and get new one */
      addCheckSum(symbol);              /* build check sum */
      putc(symbol,outFile);             /* write symbol to file */
      } /* end of decompressing a symbol */

    endTime = clock();
    fclose(outFile);
    if ((error & errRemove) != 0)
      remove(outName);
    else if (report > 0)
      writeReport();
    }

/* ========== emit the length of a run ========== */

void emitRun()
    {
    int L;
    int32 mask;

#if Stats
    if (runLength < 20)
      Lengths[runLength] ++;
#endif

    if (runLength <= runThresh)
      putBits(0, runLength)    /* unary code for shorter runs */
    else
      {
#if Stats
      Runs++;
      if (logCnt > 0)
        fprintf(stderr, "    Run %ld\n", runLength);
#endif
      runLength -= runThresh;
      for (L = 0, mask = 1; mask <= runLength; L ++)
        mask <<= 1;
      putBits(15, 4);           /* long Lit = 1111 */
      putBits(L, 8);            /* bits to encode the run length */
      putBits(runLength, L);    /* and the run length itself */
      }
    runLength = 0;              /* clear the length */
    }

/* ========== accept input symbols, handling run-encoding ========== */

void encodeFile()
   {
   us32 W;                            /* the selected context */

   int  SymIx,                        /* symbol index in MTF table */
        symbol,                       /* the actual symbol */
        check1, check2;               /* ISO checksums */
#if Stats
   int  code;
   char rank;
#endif

   us32 prev1, prev2, prev3,          /* hash codes of prev symbols */
        Hx;                           /* and its index */

   sttTime = clock();
   startoutputtingbits();
   emitHeader(ctxParam);
   Initialize(ctxParam);
```

```
   prev1 = prev2 = prev3 = 0;
   check1 = check2 = 0;

   for (;;)
     {
     symbol = getc(inFile);              /* get next symbol */
     if (symbol == EOF)
        break;                           /* exit if EOF */
     dataBytes ++;
     addCheckSum(symbol);
#if Stats
     if (dataBytes == Trigger)
       logCnt = logSize;                 /* start logging */
     SymIx = -1;
#endif

     Hx = (prev1 | prev2 | prev3) & mask23 & ctxMask;
     W = Contexts[Hx];          /* select context  */

     if((W & 0xFF) == symbol)            /* rank-0 match */
       {
       runLength ++;
#if Stats
       R0++;
       rank = '0';
       code = 0;
#endif
       }
     else if (((W >> 8) & 0xFF) == symbol) /* rank-1 match */
       {
       if (runLength > 0)
         emitRun();                      /* emit any pending run */
       putBits(6, 3);                    /* emit '110' */
       Contexts[Hx] =  (W & 0xFFFF0000) | ((W & 0xFF) << 8) | symbol;
#if Stats
       R1++;
       rank = '1';
       code = 110;
#endif
       }
     else if (((W >> 16) & 0xFF) == symbol)  /* rank-2 match */
       {
       if (runLength > 0)
         emitRun();                      /* emit any pending run */
       putBits(14, 4);                   /* emit '1110' */
       Contexts[Hx] =  (W & 0xFF000000) | ((W & 0xFFFF) << 8) | symbol;
#if Stats
       R2++;
       rank = '2';
       code = 1110;
#endif
       }
     else                                /* no match */
       {
       if (runLength > 0)
         emitRun();                      /* emit any pending run */
       SymIx = ChToIx[symbol];           /* get index in MTF table */
#if Stats
       Ranks[SymIx] ++;
#endif
       if (SymIx < 32)                   /* near symbol */
         {
         putBits(2, 2);                  /* emit '10' short Lit */
         putBits(SymIx, 5);              /* emit index */
#if Stats
         shortLit++;
         code = 10;
         rank = 'S';
#endif
         }
       else                              /* remote symbol */
         {
         putBits(15, 4);                 /* emit '1111' */
         putBits(SymIx, 8);              /* and the index */
#if Stats
         longLit++;
```

```
         code = 1111;
         rank = 'L';
#endif
         }
       Contexts[Hx] =  (W & 0xFF000000) | ((W & 0xFFFF) << 8) | symbol;
       }

#if Stats
     if (logCnt-- > 0)
       {
       fprintf(stderr, "%7ld > %-7ld '%c' %02X  %c\"%c%c%c\"  rank %c, code %-6d",
                        dataBytes, compBytes, convByte(symbol), symbol,
                        convByte(W>>24),
                        convByte(W), convByte(W>>8), convByte(W>>16),
                        rank, code);
       if (SymIx >= 0)
         fprintf(stderr, " %3d", SymIx);
       fprintf(stderr, "\n");
       }
#endif

     pseudoMTF(symbol, dataBytes & 0xFF);
     prev3 = (prev2 << 6);             /* move along hash codes */
     prev2 = ((prev1 & 0x3F) << 6);
     prev1 = symbol & mask1;           /* and get new one */
     }  /* end of compression loop */

   if (runLength > 0)
     emitRun();                         /* emit any pending run */
   putBits(15, 4);
   putBits( 0, 8);                      /* the EOF symbol */
   putBits(check1, 8);                  /* and the checksum */
   putBits(check2, 8);
   doneoutputtingbits();

   endTime = clock();
   fclose(inFile);

   fclose(outFile);
   if (report > 0)
     writeReport();
   }

/* =================================================== */

void printHelp()
    {
    char suff[20], self[8], vers[8];

    strcpy(self, srName);
    strcpy(vers, srVersion);
    strcat(vers, ".");
    strcat(vers, srLevel);
    strcpy(suff, srSuffix);

#if Stats
    fprintf(stderr, "\nFile compressor %s V%s - max rank = 3 (Statistics enabled)\n",
                      self, vers);
#else
    fprintf(stderr, "\nFile compressor %s V%s - max rank = 3\n", self, vers);
#endif
    fprintf(stderr, "Run as per standard Unix conventions --\n");
    fprintf(stderr, "  %s [ options ] file [options ] file...\n", self);
    fprintf(stderr, "Options and file names may be intermixed, with options \n");
    fprintf(stderr, "  applying to all later files until overwritten.\n");
    fprintf(stderr, "Options are preceded by a '-', with any number as a group\n");
    fprintf(stderr, " -q     disable summary report\n");
    fprintf(stderr, " -v     enable one line report (default)\n");
    fprintf(stderr, " -V     enable two line report\n");
#if Stats
    fprintf(stderr, " -S     report rank statistics\n");
    fprintf(stderr, " -F     report symbol index statistics, after MTF\n");
#endif
    fprintf(stderr, " -Cn    set number of compression contexts. 'n' is a digit (1 <= n <=
8)\n");
    fprintf(stderr, "         The number of contexts is 2^(10+n), so that the option\n");
```

```
    fprintf(stderr, "          -C5 sets 2^15 = 32768 contexts (the default).\n");
    fprintf(stderr, " -oX    expands files 'file%s' to 'file.X'\n", suff);
    fprintf(stderr, "          (This is useful for testing.) X = '.' implies no suffix\n");
    fprintf(stderr, " -On    set compression order - 1, 2 or 3\n");
#if Stats
    fprintf(stderr, " -Dn    set 'dither' mask to n bits - usually 3 bits\n");
    fprintf(stderr, " -Lnnnn start logging from symbol nnnn (nnnn is a decimal number)\n");
#endif
    fprintf(stderr, "\n");
    fprintf(stderr, "Compressed files have the suffix '%s' added to the file name.\n",
                 suff);
    fprintf(stderr, "Files with the suffix %s are automatically expanded\n", suff);
    fprintf(stderr, "The input file is NOT deleted\n\n");

    exit(0);
    }

/* int prog(int argc, char *argv[])        /*  */
int main(int argc, char *argv[])        /*  */
    {
    int n, i, L, Ls, v, compress, errval;
    char *P, c,
        suffix[8],
        wMode[8];

    strcpy(outSuffix, ".n");            /* default output suffix */
    error = 0;
    Order = 3;
    mask1  = 0x3F;                      /* order 3 -- 18 bits */
    mask23 = 0x3FFFF;
#if Stats
    Trigger = -1;
    dithMask = 7;                       /* 3 bits */
#endif
    if (argc == 1)
      printHelp();
    for (n = 1; n < argc; n++)
      {
      P = argv[n];                      /* the n-th parameter string */
      if (P[0] == '-')                  /* it looks like an option */
        {
        for (i = 1; P[i] != 0; i++)     /* scan the option list */
          switch (P[i])
            {
 case 'q' : report = 0;                 /* no summary */
            break;
 case 'v' : report = 1;                 /* brief report */
            break;
 case 'V' : report = 2;                 /* complete report */
            break;
#if Stats
 case 'S' : report = 3;                 /* statistics report */
            break;
 case 'F' : report = 4;                 /* MTF ranks report */
            break;
 case 'L' : Trigger = 0;                /* start logging */
            while (((c = P[++i]) >= '0') && (c <= '9'))
              Trigger = Trigger*10 + (c - '0');
            i--;
            break;
#else
 case 'S' :
 case 'F' :
 case 'L' : fprintf(stderr, "Option '%c' requires compilation with \"Stats = 1\"\n",
                 P[i]);
            break;
#endif
 case 'o' : c = P[++i];                 /* set recovered suffix */
            if (c == '.')
              outSuffix[0] = 0;         /* no suffix at all */
            else
              {
              outSuffix[0] = '.';       /* build '.X' string */
              outSuffix[1] = c;
              outSuffix[2] = 0;
              }
```

```
            break;
 case 'C' : c = P[++i];
            if ((c > '0') && (c < '9'))
              ctxParam = c - '0' + 10;  /* set context size */
            else
              {
              fprintf(stderr, "C must be followed by digit 1 - 8\n");
              error = errFinish;
              }
            break;
 case 'O' : c = P[++i];
            if (( c >= '1') && (c <= '4'))
              Order = c - '0';
            else
              {
              fprintf(stderr, "O must be followed by digit 1 - 4\n");
              error = errFinish;
              }
            if (Order == 3)
              mask23 = 0x3FFFF;
            else if (Order == 2)
              mask23 = 0xFFF;
            else if (Order == 1)
              mask23 = 0x3F;
            break;
#if Stats
 case 'D' : c = P[++i];
            if (( c >= '0') && (c <= '9'))
              dithMask = (1 << (c - '0')) -1;
            else
              {
              fprintf(stderr, "D must be followed by digit 0 - 9\n");
              error = errFinish;
              }
            break;
#endif
  default : fprintf(stderr, "Invalid parameter '%c'\n", P[i]);
            error = errFinish;
            }
        }
      else if (P[0] == '?')
        printHelp();
      else                              /* it looks like a file name */
        {
        if (error != 0)
          break;
        strcpy(inName, P);              /* set up the file names */
        strcpy(outName, P);
        L = strlen(inName);             /* and the name lengths */

        strcpy(suffix, srSuffix);       /* suffix for compressed files */
        Ls = strlen(suffix);

        v = strcmp(&inName[L-Ls], suffix);      /* is there a suffix? */
        if (v == 0)
          {                             /* suffix matches -- must expand file */
          outName[L-Ls] = 0;            /* delete old suffix */
          strcat(outName, outSuffix);   /* append new suffix, if any */
          compress = 0;
          strcpy(wMode, outMode);       /* recover as plain file */
          }
        else
          {                             /* no match - must compress */
          strcat(outName, srSuffix);    /* extend to compressed name */
          compress = 1;
          strcpy(wMode, "wb");          /* write compressed as binary */
          }

        inFile = fopen(inName, "rb");   /* try to open the input file */
        errval = errno;
        if (inFile == NULL)             /* open error */
          {
          fprintf(stderr, "Error in opening input file '%s', error code = %d\n",
                                                inName, errval);
          error = errFinish;
          }
```

```
        outFile = fopen(outName, wMode); /* try to open the output file */
        errval = errno;
        if (outFile == NULL)            /* open error */
          {
          fprintf(stderr, "Error in opening output file '%s', error code = %d\n",
                                              outName, errval);
          error = errFinish;
          }

        if (error != 0)
          break;
        else if (compress)              /* now process the file */
          encodeFile();
        else
          decodeFile();
#if Stats
        logCnt = 0;
        Trigger = -1;                   /* reset the logging threshold */
#endif
        } /* end of handling file name */
      } /* end of scanning parameters */

    return 0;
    } /* main */
```