1. Introduction.

BASE64

Encode or decode file as MIME base64 (RFC 1341)

by John Walker
http://www.fourmilab.ch/

EBCDIC support courtesy of Christian.Ferrari@fccrt.it, 2000-12-20.

```
#define REVDATE "10th␣June␣2007"
```

2. Program global context.

```
#define TRUE 1
#define FALSE 0
#define LINELEN 72 /* Encoded line length (max 76) */
#define MAXINLINE 256 /* Maximum input line length */
#include "config.h" /* System-dependent configuration */
  ⟨ Preprocessor definitions ⟩
  ⟨ System include files 3 ⟩
  ⟨ Windows-specific include files 4 ⟩
  ⟨ Global variables 5 ⟩
```

3. We include the following POSIX-standard C library files. Conditionals based on a probe of the system by the `configure` program allow us to cope with the peculiarities of specific systems.

```
⟨ System include files 3 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef HAVE_STRING_H
#include <string.h>
#else
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
#endif
#ifdef HAVE_GETOPT
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#else
#include "getopt.h" /* No system getopt–use our own */
#endif
```

This code is used in section 2.

4. The following include files are needed in WIN32 builds to permit setting already-open I/O streams to binary mode.

```
⟨ Windows-specific include files 4 ⟩ ≡
#ifdef _WIN32
#define FORCE_BINARY_IO
#include <io.h>
#include <fcntl.h>
#endif
```

This code is used in section 2.

5. These variables are global to all procedures; many are used as "hidden arguments" to functions in order to simplify calling sequences.

⟨ Global variables 5 ⟩ ≡

```
  typedef unsigned char byte;      /* Byte type */
  static FILE *fi;      /* Input file */
  static FILE *fo;      /* Output file */
  static byte iobuf[MAXINLINE];      /* I/O buffer */
  static int iolen = 0;      /* Bytes left in I/O buffer */
  static int iocp = MAXINLINE;      /* Character removal pointer */
  static int ateof = FALSE;      /* EOF encountered */
  static byte dtable[256];      /* Encode / decode table */
  static int linelength = 0;      /* Length of encoded output line */
  static char eol[] =      /* End of line sequence */
#ifdef FORCE_BINARY_IO
  "\n"
#else
  "\r\n"
#endif
  ;
  static int errcheck = TRUE;      /* Check decode input for errors ? */
```

This code is used in section 2.

6. Input/output functions.

7. Procedure *inbuf* fills the input buffer with data from the input stream *fi*.

```
static int inbuf(void)
{
  int l;
  if (ateof) {
    return FALSE;
  }
  l = fread(iobuf, 1, MAXINLINE, fi);      /* Read input buffer */
  if (l ≤ 0) {
    if (ferror(fi)) {
      exit(1);
    }
    ateof = TRUE;
    return FALSE;
  }
  iolen = l;
  iocp = 0;
  return TRUE;
}
```

8. Procedure *inchar* returns the next character from the input line. At end of line, it calls *inbuf* to read the next line, returning `EOF` at end of file.

```
static int inchar(void)
{
  if (iocp ≥ iolen) {
    if (¬inbuf()) {
      return EOF;
    }
  }
  return iobuf[iocp++];
}
```

9. Procedure *insig* returns the next significant input character, ignoring white space and control characters. This procedure uses *inchar* to read the input stream and returns `EOF` when the end of the input file is reached.

```
static int insig(void)
{
  int c;
  while (TRUE) {
    c = inchar();
    if (c ≡ EOF ∨ (c > ' ')) {
      return c;
    }
  }
}
```

10. Procedure *ochar* outputs an encoded character, inserting line breaks as required so that no line exceeds `LINELEN` characters.

```
static void ochar(int c)
{
  if (linelength ≥ LINELEN) {
    if (fputs(eol, fo) ≡ EOF) {
      exit(1);
    }
    linelength = 0;
  }
  if (putc(((byte) c), fo) ≡ EOF) {
    exit(1);
  }
  linelength++;
}
```

11. Encoding.

Procedure *encode* encodes the binary file opened as *fi* into base64, writing the output to *fo*.

```
static void encode(void)
{
  int i, hiteof = FALSE;
  ⟨ initialise encoding table 12 ⟩;
  while (¬hiteof) {
    byte igroup[3], ogroup[4];
    int c, n;
    igroup[0] = igroup[1] = igroup[2] = 0;
    for (n = 0; n < 3; n++) {
      c = inchar( );
      if (c ≡ EOF) {
        hiteof = TRUE;
        break;
      }
      igroup[n] = (byte) c;
    }
    if (n > 0) {
      ogroup[0] = dtable[igroup[0] ≫ 2];
      ogroup[1] = dtable[((igroup[0] & 3) ≪ 4) | (igroup[1] ≫ 4)];
      ogroup[2] = dtable[((igroup[1] & #F) ≪ 2) | (igroup[2] ≫ 6)];
      ogroup[3] = dtable[igroup[2] & #3F];      /* Replace characters in output stream with "=" pad
          characters if fewer than three characters were read from the end of the input stream. */
      if (n < 3) {
        ogroup[3] = '=';
        if (n < 2) {
          ogroup[2] = '=';
        }
      }
      for (i = 0; i < 4; i++) {
        ochar(ogroup[i]);
      }
    }
  }
  if (fputs(eol, fo) ≡ EOF) {
    exit(1);
  }
}
```

12. Procedure *initialise_encoding_table* fills the binary encoding table with the characters the 6 bit values are mapped into. The curious and disparate sequences used to fill this table permit this code to work both on ASCII and EBCDIC systems, the latter thanks to Ch.F.

In EBCDIC systems character codes for letters are not consecutive; the initialisation must be split to accommodate the EBCDIC consecutive letters:

A–I J–R S–Z a–i j–r s–z

This code works on ASCII as well as EBCDIC systems.

⟨ initialise encoding table 12 ⟩ ≡

```
for (i = 0; i < 9; i++) {
  dtable[i] = 'A' + i;
  dtable[i + 9] = 'J' + i;
  dtable[26 + i] = 'a' + i;
  dtable[26 + i + 9] = 'j' + i;
}
for (i = 0; i < 8; i++) {
  dtable[i + 18] = 'S' + i;
  dtable[26 + i + 18] = 's' + i;
}
for (i = 0; i < 10; i++) {
  dtable[52 + i] = '0' + i;
}
dtable[62] = '+';
dtable[63] = '/';
```

This code is used in section 11.

13. Decoding.

Procedure *decode* decodes a base64 encoded stream from *fi* and emits the binary result on *fo*.

```
static void decode(void)
{
  int i;
  ⟨ Initialise decode table 14 ⟩;
  while (TRUE) {
    byte a[4], b[4], o[3];
    for (i = 0; i < 4; i++) {
      int c = insig( );
      if (c ≡ EOF) {
        if (errcheck ∧ (i > 0)) {
          fprintf(stderr, "Input␣file␣incomplete.\n");
          exit(1);
        }
        return;
      }
      if (dtable[c] & #80) {
        if (errcheck) {
          fprintf(stderr, "Illegal␣character␣'%c'␣in␣input␣file.\n", c);
          exit(1);
        }       /* Ignoring errors: discard invalid character. */
        i−−;
        continue;
      }
      a[i] = (byte) c;
      b[i] = (byte) dtable[c];
    }
    o[0] = (b[0] ≪ 2) | (b[1] ≫ 4);
    o[1] = (b[1] ≪ 4) | (b[2] ≫ 2);
    o[2] = (b[2] ≪ 6) | b[3];
    i = a[2] ≡ '=' ? 1 : (a[3] ≡ '=' ? 2 : 3);
    if (fwrite(o, i, 1, fo) ≡ EOF) {
      exit(1);
    }
    if (i < 3) {
      return;
    }
  }
}
```

14. Procedure *initialise decode table* creates the lookup table used to map base64 characters into their binary values from 0 to 63. The table is built in this rather curious way in order to be properly initialised for both ASCII-based systems and those using EBCDIC, where the letters are not contiguous. (EBCDIC fixes courtesy of Ch.F.)

In EBCDIC systems character codes for letters are not consecutive; the initialisation must be split to accommodate the EBCDIC consecutive letters:

A–I J–R S–Z a–i j–r s–z

This code works on ASCII as well as EBCDIC systems.

```
⟨ Initialise decode table 14 ⟩ ≡
  for (i = 0; i < 255; i++) {
    dtable[i] = #80;
  }
  for (i = 'A'; i ≤ 'I'; i++) {
    dtable[i] = 0 + (i − 'A');
  }
  for (i = 'J'; i ≤ 'R'; i++) {
    dtable[i] = 9 + (i − 'J');
  }
  for (i = 'S'; i ≤ 'Z'; i++) {
    dtable[i] = 18 + (i − 'S');
  }
  for (i = 'a'; i ≤ 'i'; i++) {
    dtable[i] = 26 + (i − 'a');
  }
  for (i = 'j'; i ≤ 'r'; i++) {
    dtable[i] = 35 + (i − 'j');
  }
  for (i = 's'; i ≤ 'z'; i++) {
    dtable[i] = 44 + (i − 's');
  }
  for (i = '0'; i ≤ '9'; i++) {
    dtable[i] = 52 + (i − '0');
  }
  dtable['+'] = 62;
  dtable['/'] = 63;
  dtable['='] = 0;
```

This code is used in section 13.

15. Utility functions.

16. Procedure *usage* prints how-to-call information.

```
static void usage(void)
{
  printf("%s␣␣--␣␣Encode/decode␣file␣as␣base64.␣␣Call:\n",PRODUCT);
  printf("␣␣␣␣␣␣␣␣␣␣␣␣%s␣[-e␣/␣-d]␣[options]␣[infile]␣[outfile]\n",PRODUCT);
  printf("\n");
  printf("Options:\n");
  printf("␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣Print␣copyright␣information\n");
  printf("␣␣␣␣␣␣␣␣␣␣-d,␣--decode␣␣␣␣␣␣Decode␣base64␣encoded␣file\n");
  printf("␣␣␣␣␣␣␣␣␣␣-e,␣--encode␣␣␣␣␣␣Encode␣file␣into␣base64\n");
  printf("␣␣␣␣␣␣␣␣␣␣-n,␣--noerrcheck␣␣Ignore␣errors␣when␣decoding\n");
  printf("␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣Print␣this␣message\n");
  printf("␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣Print␣version␣number\n");
  printf("\n");
  printf("by␣John␣Walker\n");
  printf("http://www.fourmilab.ch/\n");
}
```

17. Main program.

```
int main(int argc, char *argv[])
{
  extern char *optarg;      /* Imported from getopt */
  extern int optind;
  int f, decoding = FALSE, opt;
#ifdef FORCE_BINARY_IO
  int in_std = TRUE, out_std = TRUE;
#endif
  char *cp;      /* 2000-12-20 Ch.F. UNIX/390 C compiler (cc) does not allow initialisation of static
      variables with non static right-value during variable declaration; it was moved from declaration to
      main function start. */
  fi = stdin;
  fo = stdout;
  ⟨ Process command-line options 18 ⟩;
  ⟨ Process command-line arguments 19 ⟩;
  ⟨ Force binary I/O where required 20 ⟩;
  if (decoding) {
    decode( );
  }
  else {
    encode( );
  }
  return 0;
}
```

18. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax.

```
⟨ Process command-line options 18 ⟩ ≡
  while ((opt = getopt(argc, argv, "denu-:")) ≠ −1) {
    switch (opt) {
    case 'd':    /* -d Decode */
      decoding = TRUE;
      break;
    case 'e':    /* -e Encode */
      decoding = FALSE;
      break;
    case 'n':    /* -n Suppress error checking */
      errcheck = FALSE;
      break;
    case 'u':    /* -u Print how-to-call information */
      case '?': usage();
      return 0;
    case '-':    /* – Extended options */
      switch (optarg[0]) {
      case 'c':    /* –copyright */
        printf("This␣program␣is␣in␣the␣public␣domain.\n");
        return 0;
      case 'd':    /* –decode */
        decoding = TRUE;
        break;
      case 'e':    /* -encode */
        decoding = FALSE;
        break;
      case 'h':    /* –help */
        usage();
        return 0;
      case 'n':    /* –noerrcheck */
        errcheck = FALSE;
        break;
      case 'v':    /* –version */
        printf("%s␣%s\n", PRODUCT, VERSION);
        printf("Last␣revised:␣%s\n", REVDATE);
        printf("The␣latest␣version␣is␣always␣available\n");
        printf("at␣http://www.fourmilab.ch/webtools/base64\n");
        return 0;
      }
    }
  }
```

This code is used in section 17.

19. This code is executed after *getopt* has completed parsing command line options. At this point the external variable *optind* in *getopt* contains the index of the first argument in the *argv*[ ] array.

```
⟨ Process command-line arguments 19 ⟩ ≡
  f = 0;
  for ( ; optind < argc; optind++) {
    cp = argv[optind];
    switch (f) {      /* Warning! On systems which distinguish text mode and binary I/O (MS-DOS,
          Macintosh, etc.) the modes in these open statements will have to be made conditional based
          upon whether an encode or decode is being done, which will have to be specified earlier. But it's
          worse: if input or output is from standard input or output, the mode will have to be changed on
          the fly, which is generally system and compiler dependent. 'Twasn't me who couldn't conform
          to Unix CR/LF convention, so don't ask me to write the code to work around Apple and
          Microsoft's incompatible standards. */
    case 0:
      if (strcmp(cp, "-") ≠ 0) {
        if ((fi = fopen(cp,
#ifdef FORCE_BINARY_IO
          decoding ? "r" : "rb"
#else
          "r"
#endif
          )) ≡ Λ) {
          fprintf(stderr, "Cannot␣open␣input␣file␣%s\n", cp);
          return 2;
          }
#ifdef FORCE_BINARY_IO
          in_std = FALSE;
#endif
        }
        f++;
        break;
    case 1:
      if (strcmp(cp, "-") ≠ 0) {
        if ((fo = fopen(cp,
#ifdef FORCE_BINARY_IO
          decoding ? "wb" : "w"
#else
          "w"
#endif
          )) ≡ Λ) {
          fprintf(stderr, "Cannot␣open␣output␣file␣%s\n", cp);
          return 2;
          }
#ifdef FORCE_BINARY_IO
          out_std = FALSE;
#endif
        }
        f++;
        break;
    default: fprintf(stderr, "Too␣many␣file␣names␣specified.\n");
      usage( );
      return 2;
```

```
    }
  }
```

This code is used in section 17.

20. On WIN32, if the binary stream is the default of `stdin`/`stdout`, we must place this stream, opened in text mode (translation of CR to CR/LF) by default, into binary mode (no EOL translation). If you port this code to other platforms which distinguish between text and binary file I/O (for example, the Macintosh), you'll need to add equivalent code here.

The following code sets the already-open standard stream to binary mode on Microsoft Visual C 5.0 (Monkey C). If you're using a different version or compiler, you may need some other incantation to cancel the text translation spell.

```
⟨ Force binary I/O where required 20 ⟩ ≡
#ifdef FORCE_BINARY_IO
  if ((decoding ∧ out_std) ∨ ((¬decoding) ∧ in_std)) {
#ifdef _WIN32
    _setmode(_fileno(decoding ? fo : fi), O_BINARY);
#endif
  }
#endif
```

This code is used in section 17.

21. Index. The following is a cross-reference table for `base64`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

⟨ Force binary I/O where required 20 ⟩ Used in section 17.
⟨ Global variables 5 ⟩ Used in section 2.
⟨ Initialise decode table 14 ⟩ Used in section 13.
⟨ Process command-line arguments 19 ⟩ Used in section 17.
⟨ Process command-line options 18 ⟩ Used in section 17.
⟨ System include files 3 ⟩ Used in section 2.
⟨ Windows-specific include files 4 ⟩ Used in section 2.
⟨ initialise encoding table 12 ⟩ Used in section 11.

BASE64