1. Introduction.

QPRINT

Encode or decode file as MIME Quoted-Printable (RFC 1521)

by John Walker

This program is a filter which encodes and decodes files in the "Quoted-Printable" form as defined in RFC 1521. This is a MIME content encoding intended primarily for text whose content consists primarily of ASCII printable characters. This encoding distinguishes white space and end of line sequences from other binary codes which don't correspond to ASCII printable characters. It's possible to encode a binary file in this form by specifying the `-b` or `--binary` and, when appropriate the `-p` or `--paranoid` options, but it's a pretty dopey thing to do; `base64` encoding is far better when the input data are known to be binary.

```
#define REVDATE "16th␣December␣2014"
```

2. Program global context. Let's start by declaring global definitions and program-wide variables and including system interface definitions.

```
#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 4 ⟩
  ⟨ Windows-specific include files 5 ⟩
  ⟨ Global variables 6 ⟩
  ⟨ Forward function definitions 34 ⟩
```

3. Because we may be built on an EBCDIC system, we can't assume that quoted characters generate the ASCII character codes we require for output. The following definitions provide the ASCII codes for characters we need in the program.

```
#define ASCII_HORIZONTAL_TAB 9 /* Horizontal tab */
#define ASCII_LINE_FEED 10 /* Line feed */
#define ASCII_CARRIAGE_RETURN 13 /* Carriage return */
#define ASCII_SPACE 32 /* Space */
#define ASCII_0 48 /* Digit 0 */
#define ASCII_EQUAL_SIGN 61 /* Equal sign */
#define ASCII_A 65 /* Letter A */
#define ASCII_LOWER_CASE_A 97 /* Letter a */
```

4. 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 4 ⟩ ≡

```
#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.

5. The following include files are needed in Win32 builds to permit setting already-open I/O streams to binary mode.

⟨ Windows-specific include files 5 ⟩ ≡
```
#ifdef _WIN32
#define FORCE_BINARY_IO
#include <io.h>
#include <fcntl.h>
#endif
```
This code is used in section 2.

6. These variables are global to all procedures; many are used as "hidden arguments" to functions in order to simplify calling sequences. We'll declare additional global variables as we need them in successive sections.

⟨ Global variables 6 ⟩ ≡
```
typedef unsigned char byte;      /* Byte type */
static FILE *fi;      /* Input file */
static FILE *fo;      /* Output file */
```
See also sections 13, 22, 23, 32, 33, and 44.

This code is used in section 2.

7. Encoding.
The following sections handle encoding the input stream into a Quoted-Printable output stream.

8. Procedure *output_line_break* outputs the standard RFC 822 line break sequence of carriage-return, line-feed and resets the current output line length to zero.

```
static void output_line_break(void)
{
  static char line_break[3] = {ASCII_CARRIAGE_RETURN, ASCII_LINE_FEED, 0};
  fputs(line_break, fo);
  current_line_length = 0;
}
```

9. Procedure *check_line_length* determines whether *chars_required* will fit in the current line. If not, a "soft line break" consisting of a trailing ASCII equal sign and the end of line sequence must be appended. Note that since the final "`=`" in a soft line break counts against the maximum line length (`LINELEN`), we must break a line one character early so as to leave room for a subsequent soft line break. The carriage-return / line-feed at the end of the line *does not* count against the maximum line length.

```
static void check_line_length(int chars_required)
{
  if ((current_line_length + chars_required) ≥ (LINELEN − 1)) {
    putc(ASCII_EQUAL_SIGN, fo);
    output_line_break( );
  }
  current_line_length += chars_required;
}
```

10. Procedure *emit_literally* outputs a non white space character which doesn't need encoding to the output stream.

```
static void emit_literally(int ch)
{
  check_line_length(1);
  putc(ch, fo);
}
```

11. Procedure *emit_hex_encoded* outputs character *ch* encoded as an equal sign followed by two ASCII characters encoded as hexadecimal.

```
static void emit_hex_encoded(int ch)
{
  static char hex[16] = {ASCII_0, ASCII_0 + 1, ASCII_0 + 2, ASCII_0 + 3, ASCII_0 + 4, ASCII_0 + 5,
      ASCII_0 + 6, ASCII_0 + 7, ASCII_0 + 8, ASCII_0 + 9, ASCII_A, ASCII_A + 1, ASCII_A + 2,
      ASCII_A + 3, ASCII_A + 4, ASCII_A + 5};
  check_line_length(3);
  putc(ASCII_EQUAL_SIGN, fo);
  putc(hex[(ch ≫ 4) & #F], fo);
  putc(hex[ch & #F], fo);
}
```

12. Procedure *encode* encodes the file opened as *fi* into Quoted-Printable, writing the output to *fo*. This simply reads the input file character by character and calls *emit_encoded_character* to write encoded characters to the output stream. This isn't entirely squeaky-clean in that if the character we pass to *emit_encoded_character* is the first character of an end of line sequence, we may look ahead to see if it's a CR/LF or LF/CR. But since the code which makes this check pushes back characters not part of a two byte end of line sequence, there's no need to worry about such detail at this level.

```
static void encode(void)
{
  int i, ch;
  ⟨ Initialise character class table 14 ⟩;
  while ((ch = getc(fi)) ≠ EOF) {
    ⟨ Emit encoded character 20 ⟩;
  }
  ⟨ Flush pending white space 24 ⟩;
  ⟨ Flush non-terminated last line 26 ⟩;
}
```

13. The *character_class* indicates which rule in the RFC (with some extensions) governs given octet codes being encoded as Quoted-Printable.

```
⟨ Global variables 6 ⟩ +≡
  typedef enum {
    Rule_1, Rule_2, Rule_3, Rule_4, Rule_EBCDIC
  } character_encoding_rule;
  static character_encoding_rule character_class[256];      /* Character class (by rule in RFC) */
```

14. Fill the *character_class* array with the classification of characters in terms which rule in the RFC definition of Quoted-Printable encoding governs their handling. Note that in all code which initialises this table we must specify ASCII codes numerically rather than as quoted **char** constants, which will be incorrrect when the program is built on an EBCDIC system.

```
⟨ Initialise character class table 14 ⟩ ≡
  ⟨ Initialise Rule 1 characters 15 ⟩;
  ⟨ Initialise Rule 2 characters 16 ⟩;
  ⟨ Initialise Rule 3 characters 17 ⟩;
  ⟨ Initialise Rule 4 characters 18 ⟩;
  ⟨ Initialise EBCDIC Rule characters 19 ⟩;
```

This code is used in section 12.

15. Initially set the *character_class* of all characters to Rule 1 (General 8-bit representation). This is the default for characters not otherwise specified.

```
⟨ Initialise Rule 1 characters 15 ⟩ ≡
  for (i = 0; i ≤ 255; i++) {
    character_class[i] = Rule_1;
  }
```

This code is used in section 14.

16. Rule 2 governs "Literal representation"–characters with code it's safe to represent in ASCII.

```
⟨ Initialise Rule 2 characters 16 ⟩ ≡
  for (i = 33; i ≤ 60; i++) {
    character_class[i] = Rule_2;
  }
  for (i = 62; i ≤ 126; i++) {
    character_class[i] = Rule_2;
  }
```

This code is used in section 14.

17. Rule 3 governs handling of the "white space" character codes for horizontal tab (HT) and space.

```
⟨ Initialise Rule 3 characters 17 ⟩ ≡
  character_class[ASCII_HORIZONTAL_TAB] = Rule_3;      /* Horizontal tab */
  character_class[ASCII_SPACE] = Rule_3;      /* Space */
```

This code is used in section 14.

18. Rule 4 applies to end of line sequences in the input file, depend upon the host system's end of line convention. When encoding pure binary data, these characters *must* be encoded in general 8-bit representation according to Rule 1.

```
⟨ Initialise Rule 4 characters 18 ⟩ ≡
  character_class[ASCII_LINE_FEED] = Rule_4;      /* Line feed */
  character_class[ASCII_CARRIAGE_RETURN] = Rule_4;      /* Carriage return */
```

This code is used in section 14.

19. ASCII characters with no EBCDIC equivalent or whose EBCDIC code differs from that in ASCII must be quoted according to Rule 1 for maximal compatibility with EBCDIC systems. We flag these characters (which would otherwise fall under Rule 2) to permit optional encoding for EBCDIC destination systems.

```
⟨ Initialise EBCDIC Rule characters 19 ⟩ ≡
  character_class[33] =      /* '!' */
  character_class[34] =      /* '"' */
  character_class[35] =      /* '#' */
  character_class[36] =      /* '$' */
  character_class[64] =      /* '@' */
  character_class[91] =      /* '[' */
  character_class[92] =      /* '\\' */
  character_class[93] =      /* ']' */
  character_class[94] =      /* '^' */
  character_class[96] =      /* '\`' */
  character_class[123] =      /* '{' */
  character_class[124] =      /* '|' */
  character_class[125] =      /* '}' */
  character_class[126] = Rule_EBCDIC;      /* '~' */
```

This code is used in section 14.

20. Output character *ch* to the output stream, encoded as required. If *paranoid* is set, we encode all printable ASCII character as hexadecimal escapes. If *EBCDIC_out* is set, we quote ASCII characters which aren't present in EBCDIC. If *binary_input* is set, end of line sequences are also quoted.

```
⟨ Emit encoded character 20 ⟩ ≡
  switch (character_class[ch]) {
  case Rule_1:      /* General 8-bit representation: encode as =XX */
    ⟨ Flush pending white space 24 ⟩;
    emit_hex_encoded(ch);
    break;
  case Rule_2:      /* Literal representation: character doesn't need encoding */
    ⟨ Flush pending white space 24 ⟩;
    if (paranoid) {
      emit_hex_encoded(ch);
    }
    else {
      emit_literally(ch);
    }
    break;
  case Rule_3:      /* White space: may not occur at end of line */
    if (paranoid) {
      emit_hex_encoded(ch);
    }
    else {
      ⟨ Flush pending white space 24 ⟩;      /* Flush already-pending white space */
      pending_white_space = ch;      /* Set this white space as pending */
    }
    break;
  case Rule_4:      /* Line break sequence */
    if (binary_input) {      /* If we're treating the input as a pure binary file, we must encode end of line
          characters as hexadecimal rather than converting them to the canonical end of line sequence. */
      ⟨ Flush pending white space 24 ⟩;
      emit_hex_encoded(ch);
    }
    else {
      ⟨ Digest line break sequence 25 ⟩;      /* We mustn't end a line with white space. If there is pending
          white space at the end of line, emit it hex encoded before the end of line sequence. */
      if (pending_white_space ≠ 0) {
        emit_hex_encoded(pending_white_space);
        pending_white_space = 0;
      }
      output_line_break( );
    }
    break;
  case Rule_EBCDIC: ⟨ Flush pending white space 24 ⟩;      /* If we're generating EBCDIC-compatible
        output, quote the ASCII characters which differ in EBCDIC. */
    if (EBCDIC_out ∨ paranoid) {
      emit_hex_encoded(ch);
    }
    else {
      emit_literally(ch);
    }
    break;
```

```
}
```

This code is used in section 12.

21. Procedure *is_end_of_line_sequence* tests whether the character *ch* is the first character of an end of line sequence and, if so, digests any subsequent characters also part of the end of line sequence. Returns `TRUE` if an end of line sequence is present and `FALSE` otherwise.

```
static int is_end_of_line_sequence(int ch)
{
  if ((ch ≡ ASCII_CARRIAGE_RETURN) ∨ (ch ≡ ASCII_LINE_FEED)) {
    ⟨ Digest line break sequence 25 ⟩;
    return TRUE;
  }
  return FALSE;
}
```

22. To comply with Rule 5 (Soft Line Breaks), we need to keep track of the length of output lines as we assemble them and break them so they don't exceed `LINELEN` characters.

```
⟨ Global variables 6 ⟩ +≡
  static int current_line_length = 0;      /* Length of current line */
```

23. In the interest of readability, we want to encode white space (*Rule_3*) characters: spaces and horizontal tabs as themselves wherever possible, but since we must cope with transfer agents which add and delete trailing white space at will, we must ensure that the last character of each encoded line is never significant white space. We accomplish this by deferring output of white space by storing its character code in *pending_white_space* and emitting it unencoded only upon discovering that there's a subsequent non white space character. If, at end of line, we discover there's white space pending, we must encode it as Hex with *emit_hex_encoded* according to Rule 1.

```
⟨ Global variables 6 ⟩ +≡
  static int pending_white_space = 0;      /* Pending white space character if nonzero */
```

24. Before emitting a non-end-of-line character, regardless of how encoded, we must check for pending white space and, if present, flush it to the output stream. Since we're guaranteed at this point that it isn't at the end of line, there's no need to encode it.

```
⟨ Flush pending white space 24 ⟩ ≡
  if (pending_white_space ≠ 0) {
    emit_literally(pending_white_space);
    pending_white_space = 0;
  }
```

This code is used in sections 12 and 20.

25. We must cope with all the end of line sequences which may be used by various systems. We apply the following rule: an end of line sequence begins with either a carriage return or line feed, optionally followed by a the other of the potential end of line characters. Any other character (including a duplicate of the character which introduced the sequence) is pushed back onto the input stream for subsequent processing. In this code *ch* is the first character of the end of line sequence.

```
⟨ Digest line break sequence 25 ⟩ ≡
  {
    int chn = getc(fi);
    if (chn ≠ EOF) {
      if ((chn ≡ ASCII_LINE_FEED) ∨ (chn ≡ ASCII_CARRIAGE_RETURN)) {
        if (chn ≡ ch) {
          ungetc(chn, fi);
        }
      }
      else {
        ungetc(chn, fi);
      }
    }
  }
```

This code is used in sections 20, 21, and 28.

26. If the file being encoded doesn't end with an end of line sequence, we must emit a soft line break followed by the canonical end of line sequence to guarantee the last encoded output line *is* properly terminated.

```
⟨ Flush non-terminated last line 26 ⟩ ≡
  if (current_line_length > 0) {
    putc(ASCII_EQUAL_SIGN, fo);
    output_line_break( );
  }
```

This code is used in section 12.

27. Decoding.
The following sections handle decoding of a Quoted-Printable input stream into a binary output stream.

28. Procedure *decode* decodes a Quoted-Printable encoded stream from *fi* and emits the binary result on *fo*.

```
static void decode(void)
{
  int ch, ch1, ch2;
  while ((ch = read_decode_character()) ≠ EOF) {
    switch (ch) {
    case ASCII_EQUAL_SIGN:      /* '=': Encoded character or soft end of line */
      ⟨ Decode equal sign escape 29 ⟩;
      if (ch ≠ EOF) {
        putc(ch, fo);
      }
      break;
    case ASCII_CARRIAGE_RETURN:      /* CR: End of line sequence */
      case ASCII_LINE_FEED:      /* LF: End of line sequence */
      ⟨ Digest line break sequence 25 ⟩;
      putc('\n', fo);      /* Output end of line in system EOL idiom */
      break;
    default:      /* Character not requiring encoding */
      putc(ch, fo);
      break;
    }
  }
}
```

29. When we encounter an equal sign in the input stream there are two possibilities: it may introduce two characters of ASCII representing an 8-bit octet in hexadecimal or, if followed by an end of line sequence, it's a "soft end-of-line" introduced to avoid emitting a long longer than the number of chracters prescribed by the `LINELEN` constraint. We look forward in the input stream and return EOF if this equal sign denotes a soft end-of-line or the character code given by the two subsequent hexadecimal digits. While the RFC prescribes that all letters representing hexadecimal digits be upper case, conforming to the recommendation for "robust implementations", we accept lower case letters in their stead.

```
⟨ Decode equal sign escape 29 ⟩ ≡
  ch1 = read_decode_character( );
  ⟨ Ignore white space after soft line break 31 ⟩;
  if (ch1 ≡ EOF) {
    fprintf(stderr, "Error: unexpected end of file after soft line break sequence at byte %"
        FILE_ADDRESS_FORMAT_LENGTH
        "u (0x%"
        FILE_ADDRESS_FORMAT_LENGTH
        "X) of input.\n", decode_input_stream_position − 1, decode_input_stream_position − 1);
    decode_errors ++;
  }
  if (is_end_of_line_sequence(ch1) ∨ (ch1 ≡ EOF)) {
    ch = EOF;
  }
  else {
    int n1, n2;
    n1 = hex_to_nybble(ch1);
    ch2 = read_decode_character( );
    n2 = hex_to_nybble(ch2);
    if (n1 ≡ EOF ∨ n2 ≡ EOF) {
      ⟨ Handle erroneous escape sequences 38 ⟩;
      decode_errors ++;
    }
    ch = (n1 ≪ 4) | n2;
  }
```

This code is used in section 28.

30. There are lots of ways of defining "ASCII white space," but RFC 1521 explicitly states that only ASCII space and horizontal tab characters are deemed white space for the purposes of Quoted-Printable encoding.

```
⟨ Character is white space 30 ⟩ ≡
  ((ch1 ≡ ASCII_SPACE) ∨ (ch1 ≡ ASCII_HORIZONTAL_TAB))
```

This code is used in section 31.

31. Some systems pad text lines with white space (ASCII blank or horizontal tab characters). This may result in a line encoded with a "soft line break" at the end appearing, when decoded, with white space between the supposedly-trailing equal sign and the end of line sequence. If white space follows an equal sign escape, we ignore it up to the beginning of an end of line sequence. Non-white space appearing before we sense the end of line is an error; these erroneous characters are ignored.

```
⟨ Ignore white space after soft line break 31 ⟩ ≡
  while (⟨ Character is white space 30 ⟩) {
    ch1 = read_decode_character( );
    if (is_end_of_line_sequence(ch1)) {
      break;
    }
    if (¬⟨ Character is white space 30 ⟩) {
      if (ch1 ≡ EOF) {
        break;
      }
      ⟨ Report invalid character after soft line break 39 ⟩;
      decode_errors ++;
      ch1 = ASCII_SPACE;      /* Fake a space and soldier on */
    }
  }
```

This code is used in section 29.

32. On systems which support 64-bit file I/O, we want to be able to issue error messages with addresses that aren't truncated at 32 bits, but we may find ourselves confronted with a compiler which doesn't support **unsigned long long** 64-bit integers, or on a system such as the Alpha where **unsigned long** is itself 64 bits in length. The `configure` script determines the length of the **unsigned long long** and **unsigned long** types, setting the length of **unsigned long long** to 0 if the compiler does not support it. Based on the results of these tests, we define the type to be used for file addresses and which format to use when printing them.

```
⟨ Global variables 6 ⟩ +≡
#if (SIZEOF_UNSIGNED_LONG ≡ 8) ∨ (SIZEOF_UNSIGNED_LONG_LONG ≡ 0)
    /* unsigned long on this machine is 64 bits or the compiler doesn't support unsigned long long.
      In either of these rather different cases we want to use unsigned long for file addresses. */
  typedef unsigned long file_address_type;
#define FILE_ADDRESS_FORMAT_LENGTH  "l"
#else     /* Compiler supports unsigned long long and unsigned long is not 64 bits. Use unsigned
      long long for file addresses. */
  typedef unsigned long long file_address_type;
#define FILE_ADDRESS_FORMAT_LENGTH  "ll"
#endif
```

33. Error messages during the decoding process are much more useful if they identify the position in the stream where the error was identified. We keep track of the position in the stream in *decode_input_stream_position*.

We use the variable *decode_errors* to keep track of the number of errors in the decoding process. Even if the user has suppressed error messages, this permits us to return a status indicating that one or more decoding errors occurred.

```
⟨ Global variables 6 ⟩ +≡
  static file_address_type decode_input_stream_position = 0;
  static long decode_errors = 0;
```

34. We need to pre-declare the function *read_decode_character* for those who call it before we introduce it in the source code.

⟨ Forward function definitions 34 ⟩ ≡
```
static int read_decode_character(void);
```
See also section 36.

This code is used in section 2.

35. Procedure *read_decode_character* reads the next character from the input stream and advances the position counter in the stream, *decode_input_stream_position*.

```
static int read_decode_character(void)
{
  int ch;
  ch = getc(fi);
  if (ch ≠ EOF) {
    decode_input_stream_position++;
  }
  return ch;
}
```

36. We also must pre-declare *hex_to_nybble* for the same reasons.

⟨ Forward function definitions 34 ⟩ +≡
```
static int hex_to_nybble(int ch);
```

37. Procedure *hex_to_nybble* converts an ASCII hexadecimal digit character to its binary 4 bit value. An argument which cannot be converted returns `EOF`.

```
static int hex_to_nybble(int ch)
{
  if ((ch ≥ ASCII_0) ∧ (ch ≤ (ASCII_0 + 9))) {
    return ch - '0';
  }
  else if ((ch ≥ ASCII_A) ∧ (ch ≤ (ASCII_A + 5))) {
    return 10 + (ch - ASCII_A);
  }
  else if ((ch ≥ ASCII_LOWER_CASE_A) ∧ (ch ≤ (ASCII_LOWER_CASE_A + 5))) {
    return 10 + (ch - ASCII_LOWER_CASE_A);
  }
  return EOF;
}
```

38. If we encounter an equal sign that isn't either at the end of a line (denoting a "soft line break") or followed by two hexadecimal digits, we increment the number of decoding errors detected and, unless suppressed by the `-n` option, as indicated by the variable *errcheck*, issue an error message on standard output. We print the escape sequence as ASCII characters if possible, but if we're running on a non-ASCII system or one or more of the characters following the equal sign isn't printable, we show the hexadecimal values of the characters.

```
⟨ Handle erroneous escape sequences 38 ⟩ ≡
  if (errcheck) {
    if (⟨ System character code is ASCII 41 ⟩ ∧ Character_is_printable_ISO_8859(ch1) ∧
          Character_is_printable_ISO_8859(ch2)) {
      fprintf(stderr, "Error:␣bad␣equal␣sign␣escape␣\"=%c%c\"␣at␣byte␣%"
          FILE_ADDRESS_FORMAT_LENGTH
          "u␣(0x%"
          FILE_ADDRESS_FORMAT_LENGTH
          "X)␣of␣input.\n", ch1, ch2, decode_input_stream_position − 3, decode_input_stream_position − 3);
    }
    else {      /* Characters after the equal sign are not printable. Display them in hexadecimal. */
      fprintf(stderr, "Error:␣bad␣equal␣sign␣escape␣\"=␣0x%02X␣0x%02X\"␣at␣byte␣%"
          FILE_ADDRESS_FORMAT_LENGTH
          "u␣(0x%"
          FILE_ADDRESS_FORMAT_LENGTH
          "X)␣of␣input.\n", ch1, ch2, decode_input_stream_position − 3, decode_input_stream_position − 3);
    }
  }
```

This code is used in section 29.

39. Another possible decoding error is the presence of a non white space character between the equal sign introducing a soft line break and the end of line sequence which follows it. In order to cope with systems which may pad text lines with white space, white space is permitted between the trailing equal sign and end of line, but once we've seen one white space character following an equal sign, every subsequent character up to the end of line must also be white space. In the following code *ch1* is the invalid character detected in the soft line break.

```
⟨ Report invalid character after soft line break 39 ⟩ ≡
  if (errcheck) {
    if (⟨ System character code is ASCII 41 ⟩ ∧ Character_is_printable_ISO_8859(ch1)) {
      fprintf(stderr, "Error:␣invalid␣character␣\"%c\"␣in␣soft␣line␣break␣sequence␣at␣byte␣%"
          FILE_ADDRESS_FORMAT_LENGTH
          "u␣(0x%"
          FILE_ADDRESS_FORMAT_LENGTH
          "X)␣of␣input.\n", ch1, decode_input_stream_position − 1, decode_input_stream_position − 1);
    }
    else {      /* Invalid character is not not printable. Display it in hexadecimal. */
      fprintf(stderr,
          "Error:␣invalid␣character␣\"0x%02X\"␣in␣soft␣line␣break␣sequence␣at␣byte␣%"
          FILE_ADDRESS_FORMAT_LENGTH
          "u␣(0x%"
          FILE_ADDRESS_FORMAT_LENGTH
          "X)␣of␣input.\n", ch1, decode_input_stream_position − 1, decode_input_stream_position − 1);
    }
  }
```

This code is used in section 31.

40. Utilities.

41. The vast majority of users will run this program on ASCII-based systems, but we must also cope with EBCDIC systems. When issuing error messages, we'd like to be able to include ASCII characters from the input stream in certain cases, but we can't do this on EBCDIC systems without including a necessarily incomplete conversion table which would be absurdly excess baggage for a little program like this. We compromise by falling back to hexadecimal display when running on non-ASCII systems. But how do we discern this? The following expression tests a compiler-generated character for equality with its character code in ASCII. This will fail on EBCDIC systems, permitting us to generate the variant messages.

⟨ System character code is ASCII 41 ⟩ ≡
  (`'a'` ≡ `#61`)

This code is cited in section 42.

This code is used in sections 38 and 39.

42. Even on an ASCII-based system we mustn't include non-printing characters in error messages. Once we've established the system is ASCII using ⟨ System character code is ASCII 41 ⟩ we must further test that the character falls within the printable range for ISO 8859 Latin-1.

#define *Character_is_printable_ISO_8859*(*c*) (((((*c*) ≥ `#20`) ∧ ((*c*) ≤ `#7E`)) ∨ ((*c*) ≥ `#A1`)))

43. Command line parsing.

44. The following global variables represent command-line options.

```
⟨ Global variables 6 ⟩ +≡
  static int decoding = FALSE;      /* Decoding (TRUE) or encoding (FALSE) */
  static int encoding = FALSE;      /* Encoding (TRUE) or decoding (FALSE) */
  static int binary_input = FALSE;      /* Treat input as a binary file ? */
  static int errcheck = TRUE;      /* Check decode input for errors ? */
  static int EBCDIC_out = FALSE;      /* Generate EBCDIC-compatible output */
  static int paranoid = FALSE;      /* Paranoid output: quote everything */
```

45. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax. We support GNU-style "`--`" extended options which aren't directly supported by *getopt* through the following subterfuge: if the main option letter is "`--`", we convert the following letter to upper case, which permits us to discriminate it in the option processing **case** statement, which in many cases will simply be a fall-through into the same code we use for the regular option beginning with a single minus sign. If we need to further disambiguate extended options, this must be done in the case processing the extended option.

```
⟨ Process command-line options 45 ⟩ ≡
  for ( ; ; ) {
    opt = getopt(argc, argv, "bdeinpu-:");
    if (opt ≡ −1) {
      break;
    }
    if (opt ≡ '-') {
        /* If this is an extended "--" option, take the first letter (if it so be) after the second dash and
           translate it to upper case so we can distinguish it in the case statement which follows. */
      if (islower(optarg[0])) {
        opt = toupper(optarg[0]);
      }
    }
    switch (opt) {
    case 'b':     /* -b --binary Binary input file */
    case 'B':
      binary_input = TRUE;
      break;
    case 'C':     /* --copyright */
      printf("This␣program␣is␣in␣the␣public␣domain.\n");
      return 0;
    case 'd':     /* -d --decode Decode */
    case 'D':
      decoding = TRUE;
      break;
    case 'e':     /* -e Encode */
      encoding = TRUE;
      break;
    case 'E':     /* --encode or --ebcdic */
      ⟨ Process extended "−−e" options 46 ⟩;
      break;
    case 'H':     /* --help */
      usage( );
      return 0;
    case 'i':     /* -i EBCDIC-compatible output */
      EBCDIC_out = TRUE;
      break;
    case 'n':     /* -n --noerrcheck Suppress error checking */
    case 'N':
      errcheck = FALSE;
      break;
    case 'p':     /* -p --paranoid Paranoid: quote even printable characters */
    case 'P':
      paranoid = TRUE;
      break;
```

```
    case 'u':    /* -u Print how-to-call information */
    case '?':
      usage( );
      return 0;
    case 'V':    /* --version */
      ⟨ Show program version information 50 ⟩;
      return 0;
    default:    /* Invalid extended option */
      fprintf (stderr, "Invalid␣option:␣--%s\n", optarg);
      return 2;
    }
  }
```

This code is used in section 51.

46. There are two extended options which begin with "`--e`": `--ebcdic` and `--encode`. We must distinguish them by looking at the second letter of the option.

```
⟨ Process extended "−−e" options 46 ⟩ ≡
  switch (optarg[1]) {
  case 'b':    /* --ebcdic */
    EBCDIC_out = TRUE;
    break;
  case 'n':    /* --encode */
    encoding = TRUE;
    break;
  default: fprintf (stderr, "Invalid␣option:␣--%s\n", optarg);
    return 2;
  }
```

This code is used in section 45.

47. After processing the command-line options, we need to check them for consistency, for example, that the user hasn't simultaneously asked us to encode and decode a file.

```
⟨ Check options for consistency 47 ⟩ ≡
  if (encoding ∧ decoding) {
    fprintf (stderr, "Cannot␣simultaneously␣encode␣and␣decode.\n");
    return 2;
  }
  if (¬(encoding ∨ decoding)) {
    fprintf (stderr, "Please␣specify␣--encode␣(-e)␣or␣--decode␣(-d).\n");
    return 2;
  }
```

This code is used in section 51.

48. 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. The first two arguments specify the input and output file. If either argument is omitted or "`-`", standard input or output is used.

On systems which distinguish text and binary I/O (for end of line translation), we always open the input file in binary mode. The output file is opened in binary mode when encoding (since the standard requires RFC 822 CR/LF end of line convention regardless of that used by the host system), but text mode while decoding, since output should conform to the system's end of line convention.

```
⟨ Process command-line file name arguments 48 ⟩ ≡
  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
        "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 ∧ (¬binary_input)) ? "w" : "wb"
#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 51.

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

```
static void usage(void)
{
  printf("%s␣␣--␣␣Encode/decode␣file␣as␣Quoted-Printable.␣␣Call:\n", PRODUCT);
  printf("␣␣␣␣␣␣␣␣␣␣␣␣%s␣[-e␣/␣-d]␣[options]␣[infile]␣[outfile]\n", PRODUCT);
  printf("\n");
  printf("Options:\n");
  printf("␣␣␣␣␣␣␣␣␣␣-b,␣--binary␣␣␣␣␣␣Treat␣input␣as␣pure␣binary␣file\n");
  printf("␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣Print␣copyright␣information\n");
  printf("␣␣␣␣␣␣␣␣␣␣-d,␣--decode␣␣␣␣␣␣Decode␣Quoted-Printable␣encoded␣file\n");
  printf("␣␣␣␣␣␣␣␣␣␣-e,␣--encode␣␣␣␣␣␣Encode␣file␣into␣Quoted-Printable\n");
  printf("␣␣␣␣␣␣␣␣␣␣-i,␣--ebcdic␣␣␣␣␣␣EBCDIC-compatible␣encoding␣output\n");
  printf("␣␣␣␣␣␣␣␣␣␣-n,␣--noerrcheck␣␣Ignore␣errors␣when␣decoding\n");
  printf("␣␣␣␣␣␣␣␣␣␣-p,␣--paranoid␣␣␣␣Paranoid:␣quote␣even␣printable␣characters\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");
}
```

50. Show program version information in response to the `--version` option.

⟨ Show program version information 50 ⟩ ≡

```
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/qprint/\n");
```

This code is used in section 45.

51. Main program.
The exit status returned by the main program is 0 for normal completion, 1 if an error occurred in decoding, and 2 for invalid options or file name arguments.

```
int main(int argc, char *argv[])
{
  extern char *optarg;      /* Imported from getopt */
  extern int optind;
  int f, opt;
#ifdef FORCE_BINARY_IO
  int in_std = TRUE, out_std = TRUE;
#endif
  char *cp;      /* Some C compilers don't allow initialisation of static variables such as fi and fo with
      their library's definitions of stdin and stdout, so we initialise them at runtime. */
  fi = stdin;
  fo = stdout;
  ⟨ Process command-line options 45 ⟩;
  ⟨ Check options for consistency 47 ⟩;
  ⟨ Process command-line file name arguments 48 ⟩;
  ⟨ Force binary I/O where required 52 ⟩;
  if (decoding) {
    decode( );
  }
  else {
    encode( );
  }
  return decode_errors ? 1 : 0;
}
```

52. On Win32, if a binary stream is the default of *stdin* or *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 52 ⟩ ≡

```
#ifdef FORCE_BINARY_IO
  if (in_std) {
#ifdef _WIN32
    _setmode(_fileno(fi), O_BINARY);
#endif
  }
  if (((¬decoding) ∨ binary_input) ∧ out_std) {
#ifdef _WIN32
    _setmode(_fileno(fo), O_BINARY);
#endif
  }
#endif
```

This code is used in section 51.

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

⟨ Character is white space 30 ⟩ Used in section 31.
⟨ Check options for consistency 47 ⟩ Used in section 51.
⟨ Decode equal sign escape 29 ⟩ Used in section 28.
⟨ Digest line break sequence 25 ⟩ Used in sections 20, 21, and 28.
⟨ Emit encoded character 20 ⟩ Used in section 12.
⟨ Flush non-terminated last line 26 ⟩ Used in section 12.
⟨ Flush pending white space 24 ⟩ Used in sections 12 and 20.
⟨ Force binary I/O where required 52 ⟩ Used in section 51.
⟨ Forward function definitions 34, 36 ⟩ Used in section 2.
⟨ Global variables 6, 13, 22, 23, 32, 33, 44 ⟩ Used in section 2.
⟨ Handle erroneous escape sequences 38 ⟩ Used in section 29.
⟨ Ignore white space after soft line break 31 ⟩ Used in section 29.
⟨ Initialise EBCDIC Rule characters 19 ⟩ Used in section 14.
⟨ Initialise Rule 1 characters 15 ⟩ Used in section 14.
⟨ Initialise Rule 2 characters 16 ⟩ Used in section 14.
⟨ Initialise Rule 3 characters 17 ⟩ Used in section 14.
⟨ Initialise Rule 4 characters 18 ⟩ Used in section 14.
⟨ Initialise character class table 14 ⟩ Used in section 12.
⟨ Process command-line file name arguments 48 ⟩ Used in section 51.
⟨ Process command-line options 45 ⟩ Used in section 51.
⟨ Process extended "$--e$" options 46 ⟩ Used in section 45.
⟨ Report invalid character after soft line break 39 ⟩ Used in section 31.
⟨ Show program version information 50 ⟩ Used in section 45.
⟨ System character code is ASCII 41 ⟩ Cited in section 42. Used in sections 38 and 39.
⟨ System include files 4 ⟩ Used in section 2.
⟨ Windows-specific include files 5 ⟩ Used in section 2.

QPRINT