Appendix A

Transformation Formats

Existing software and practices in information technology frequently depend on character data being represented as a sequence of bytes. Older, but still prevalent practices often assume that only 7 bits of each byte are significant for the purpose of interchanging character data. To make use of Unicode character data in such systems, it is necessary to transform individual Unicode character values and pairs of surrogates into a sequence of one or more bytes that represent the same information, but which are restricted in their numerical range, so they can be interchanged with or transmitted through such systems. Typically, a transformation format allows a certain number of code values in the ASCII range to be transmitted as-is (except for truncation of the leading zero byte), a property known as *transparency*—while other code values are represented through an escape mechanism. This approach typically makes use of a variable-length encoding to achieve greater efficiency when invoking the escape mechanism.

Two transformation formats have been developed to meet the needs just described. There are two transformation formats instead of just one due to differences in the transparency requirements of the expected transmission channels. These are known as *UTFs* (Universal Character Set Transformation Formats); specifically UTF-7 and UTF-8. Both UTF-7 and UTF-8 can be used with MIME; UTF-8 is often used as a file code in X/Open environments. UTF-8 is included as Amendment Number 2 of ISO/IEC 10646. UTF-7 is not part of ISO/IEC 10646 as of this writing.

A.1 UTF-7

The term UTF-7 stands for UCS Transformation Format, 7-bit form.

Many existing character transmission media support only 7 bits of significant data within individual bytes; furthermore, in the majority of these cases, only a subset of the integral values 0...127 may be interchanged transparently. Typically, C0 (0...31) and DEL (127) and certain other values are not transparent in these media. Examples of such media include the most common mail transport agents employed in the Internet, particularly those based on the Simple Mail Transport Protocol (SMTP).

To address the needs of such transmission media and, in particular, to address the needs of the developing Multimedia Internet Mail Extensions (MIME) standard, a transformation format was devised which supports the necessary transparency to facilitate effective interchange of Unicode (UCS-2) character data in such environments. This format is known as UTF-7. UTF-7 is described in Internet Network Working Group RFC-1642 and is available in electronic form via the `unicode.org` World Wide Web Page and from the `unicode.org` FTP archive (see *Section 1.6, Resources*, for addresses).

The following discussion is a summary of the RFC. Character set UTF-7 is safe for Internet mail transmission and therefore may be used with any content transfer encoding in MIME (except where line length and line break restrictions are violated). Specifically, the 7-bit encoding for bodies and the Q encoding for headers are both acceptable. The MIME character set identifier is UNICODE-1-1-UTF-7.

Specification of UTF-7 depends on some definitions of US-ASCII character subsets.

Set D (directly encoded characters, derived from RFC 1521, Appendix B) consists of the upper and lower case letters A through Z and a through z, the 10 digits 0-9, and nine special characters listed in Table A-1 (note that "+" and "=" are omitted).

Table A-1. UTF-7 Set D Special Characters

Character	ASCII & Unicode Value (decimal)
'	39
(	40
)	41
,	44
-	45
.	46
/	47
:	58
?	63

Set O (optional direct characters) consists of the characters listed in Table A-2 (note that "\" and "~" are omitted because they are often redefined in variants of ASCII).

Table A-2. UTF-7 Set O

Character	ASCII & Unicode Value (decimal)
!	33
"	34
#	35
$	36
%	37
&	38
*	42
;	59
<	60
=	61
>	62
@	64
[	91
]	93
^	94
_	95
`	96
{	123
|	124
}	125

Set B (Modified Base 64) is the set of characters in the Base64 alphabet defined in Internet RFC 1521, excluding the pad character "=" (decimal value 61). The pad character "=" is excluded because UTF-7 is designed for use within header fields as set forth in RFC 1522. Since the only readable encoding in RFC 1522 is "Q" (based on RFC 1521's Quoted-Printable), the "=" character is not available for use (without a lot of escape sequences).

A UTF-7 stream represents 16-bit Unicode characters in 7-bit US-ASCII as follows.

Rule 1: Direct Encoding

Unicode characters in Set D may be encoded directly as their ASCII equivalents. Unicode characters in Set O may optionally be encoded directly as their ASCII equivalents; bear in mind that many of these characters are illegal in header fields, or may not pass correctly through some mail gateways.

Rule 2: Unicode Shifted Encoding

Any Unicode character sequence may be encoded using a sequence of characters in set B, when preceded by the shift character "+" (US-ASCII character value decimal 43). The "+" signals that subsequent bytes are to be interpreted as elements of the Modified Base64 alphabet until a character not in that alphabet is encountered. Such characters include control characters such as carriage returns and line feeds; thus, a Unicode shifted sequence always terminates at the end of a line. As a special case, if the sequence terminates with the character "-" (US-ASCII decimal 45) then that character is absorbed; other terminating characters are not absorbed and are processed normally. A terminating character is necessary when the next character after the Modified Base64 sequence is part of character set B. The sequence "+-" may be used to encode the character "+". A "+" character followed immediately by any character other than members of set B or "-" is an ill-formed sequence.

Unicode data is encoded using Modified Base64 by first converting Unicode 16-bit quantities to a byte stream (with the most significant byte first). Text with an odd number of bytes is ill-formed.

The byte stream is then encoded by applying the Base64 content transfer encoding algorithm as defined in RFC 1521, modified to omit the "=" pad character. Instead, when encoding, zero bits are added to pad to a Base64 character boundary. When decoding, any bits at the end of the Modified Base64 sequence that do not constitute a complete 16-bit Unicode character are discarded. If such discarded bits are non-zero the sequence is ill-formed. The pad character "=" is not used when encoding Modified Base64 because that conflicts with its use as an escape character for the Q content transfer encoding in RFC 1522 header fields.

Rule 3: ASCII Equivalents

The space (decimal 32), tab (decimal 9), carriage return (decimal 13), and line feed (decimal 10) characters may be directly represented by their ASCII equivalents. However, note that MIME content transfer encodings have rules concerning the use of such characters. Usage that does not conform to the restrictions of RFC 822, for example, would have to be encoded using MIME content transfer encodings other than 7-bit or 8-bit, such as quoted-printable, binary, or base64.

Given this set of rules, Unicode characters that may be encoded via rules 1 or 3 take one byte per character, and other Unicode characters are encoded on average with 2 2/3 bytes per character plus one byte to switch into Modified Base64 and an optional byte to switch out.

Sample Implementation of the UTF-7 Conversions

```
/* ========================================================== */
/*      The following definitions are compiler-specific.
    I would use wchar_t for UniChar, except that the C standard
    does not guarantee that it has at least 16 bits, so wchar_t is
    no more portable than unsigned short!
*/

typedef unsigned short  UniChar;
```

```
/* ================================================================== */
/*      Each of these routines converts the text between *sourceStart and
sourceEnd, putting the result into the buffer between *targetStart and
targetEnd. Note: the end pointers are *after* the last item: e.g.
*(sourceEnd - 1) is the last item.

    The return result indicates whether the conversion was successful,
and if not, whether the problem was in the source or target buffers.

    After the conversion, *sourceStart and *targetStart are both
updated to point to the end of last text successfully converted in
the respective buffers.

        In ConvertUniChartoUTF7, optional indicates whether UTF-7 optional
characters should be directly encoded, and verbose controls whether the
shift-out character, "-", is always emitted at the end of a shifted
sequence.
*/

typedef enum {
    ok,                             /* conversion successful */
    sourceCorrupt,              /* source contains invalid UTF-7 */
    targetExhausted             /* insuff. room in target for conversion */
} ConversionResult;

extern ConversionResult        ConvertUniChartoUTF7 (
        UniChar** sourceStart, UniChar* sourceEnd,
        char** targetStart, char* targetEnd,
        int optional, int verbose);

extern ConversionResult        ConvertUTF7toUniChar (
        char** sourceStart, char* sourceEnd,
        UniChar** targetStart, UniChar* targetEnd);

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


#include "ConvertUTF7.h"

static char base64[]
    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
static short invbase64[128];

static char direct[] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'(),-./:?";
static char optional[] = "!\"#$%&*;<=>@[]^_`{|}";
static char spaces[] = " \011\015\012";/* space, tab, return, line feed */
static char mustshiftsafe[128];
static char mustshiftopt[128];

static int needtables = 1;

#define SHIFT_IN '+'
#define SHIFT_OUT '-'

static void
tabinit()
{
    int i, limit;

    for (i = 0; i < 128; ++i)
    {
        mustshiftopt[i] = mustshiftsafe[i] = 1;
        invbase64[i] = -1;
    }
    limit = strlen(direct);
    for (i = 0; i < limit; ++i)
        mustshiftopt[direct[i]] = mustshiftsafe[direct[i]] = 0;
    limit = strlen(spaces);
    for (i = 0; i < limit; ++i)
        mustshiftopt[spaces[i]] = mustshiftsafe[spaces[i]] = 0;
    limit = strlen(optional);
    for (i = 0; i < limit; ++i)
        mustshiftopt[optional[i]] = 0;
    limit = strlen(base64);
    for (i = 0; i < limit; ++i)
        invbase64[base64[i]] = i;

    needtables = 0;
}

#define DECLARE_BIT_BUFFER \
    register unsigned long BITbuffer = 0, buffertemp = 0; int bufferbits = 0
#define BITS_IN_BUFFER bufferbits
#define WRITE_N_BITS(x, n) \
    ((BITbuffer |= ( ((x) & ~(-1L<<(n))) << (32-(n)-bufferbits) ) ), \
    bufferbits += (n) )
#define READ_N_BITS(n) \
    ((buffertemp = (BITbuffer >> (32-(n)))), \
        (BITbuffer <<= (n)), (bufferbits -= (n)), buffertemp)
#define TARGETCHECK  {if (target >= targetEnd) {result = targetExhausted; break;}}
```

```
ConversionResult ConvertUniChartoUTF7(
        UniChar** sourceStart, UniChar* sourceEnd,
        char** targetStart, char* targetEnd,
        int optional, int verbose)
{
    ConversionResult result = ok;
    DECLARE_BIT_BUFFER;
    int shifted = 0, needshift = 0, done = 0;
    register UniChar *source = *sourceStart;
    register char *target = *targetStart;
    char *mustshift;

    if (needtables)
        tabinit();

    if (optional)
        mustshift = mustshiftopt;
    else
        mustshift = mustshiftsafe;

    do
    {
        register UniChar r;

        if (!(done = (source >= sourceEnd)))
            r = *source++;
        needshift = (!done && ((r > 0x7f) || mustshift[r]));

        if (needshift && !shifted)
        {
            TARGETCHECK;
            *target++ = SHIFT_IN;
            /* Special case handling of the SHIFT_IN character */
            if (r == (UniChar)SHIFT_IN) {
                TARGETCHECK;
                *target++ = SHIFT_OUT;
            }
            else
                shifted = 1;
        }

        if (shifted)
        {
            /* Either write the character to the bit buffer, or pad
               the bit buffer out to a full base64 character.
               */
            if (needshift)
                WRITE_N_BITS(r, 16);
            else
                WRITE_N_BITS(0, (6 - (BITS_IN_BUFFER % 6))%6);

            /* Flush out as many full base64 characters as possible
               from the bit buffer.
               */
            while ((target < targetEnd) && BITS_IN_BUFFER >= 6)
            {
                *target++ = base64[READ_N_BITS(6)];
            }

            if (BITS_IN_BUFFER >= 6)
                TARGETCHECK;

            if (!needshift)
            {
                /* Write the explicit shift out character if
                    1) The caller has requested we always do it, or
                    2) The directly encoded character is in the
                    base64 set.
                    */
                if (verbose || ((!done) && invbase64[r] >= 0))
                {
                    TARGETCHECK;
                    *target++ = SHIFT_OUT;
                }
                shifted = 0;
            }
        }

        /* The character can be directly encoded as ASCII. */
        if (!needshift && !done)
        {
            TARGETCHECK;
            *target++ = (char) r;
        }

    }
    while (!done);

    *sourceStart = source;
    *targetStart = target;
    return result;
}
```

```
ConversionResult ConvertUTF7toUniChar(
        char** sourceStart, char* sourceEnd,
        UniChar** targetStart, UniChar* targetEnd)
{
    ConversionResult result = ok;
    DECLARE_BIT_BUFFER;
    int shifted = 0, first = 0, wroteone = 0, base64EOF, base64value, done;
    unsigned int c, prevc;
    unsigned long junk;
    register char *source = *sourceStart;
    register UniChar *target = *targetStart;

    if (needtables)
        tabinit();

    do
    {
        /* read an ASCII character c */
        if (!(done = (source >= sourceEnd)))
            c = *source++;
        if (shifted)
        {
            /* We're done with a base64 string if we hit EOF, it's not a valid
               ASCII character, or it's not in the base64 set.
               */
            base64EOF = done || (c > 0x7f) || (base64value = invbase64[c]) < 0;
            if (base64EOF)
            {
                shifted = 0;
                /* If the character causing us to drop out was SHIFT_IN or
                   SHIFT_OUT, it may be a special escape for SHIFT_IN. The
                   test for SHIFT_IN is not necessary, but allows an alternate
                   form of UTF-7 where SHIFT_IN is escaped by SHIFT_IN. This
                   only works for some values of SHIFT_IN.
                   */
                if (!done && (c == SHIFT_IN || c == SHIFT_OUT))
                {
                    /* get another character c */
                    prevc = c;
                    if (!(done = (source >= sourceEnd)))
                        c = *source++;
                    /* If no base64 characters were encountered, and the
                       character terminating the shift sequence was
                       SHIFT_OUT, then it's a special escape for SHIFT_IN.
                       */
                    if (first && prevc == SHIFT_OUT)
                    {
                        /* write SHIFT_IN unicode */
                        TARGETCHECK;
                        *target++ = (UniChar)SHIFT_IN;
                    }
                    else if (!wroteone)
                    {
                        result = sourceCorrupt;
                    }
                }
                else if (!wroteone)
                {
                    result = sourceCorrupt;
                }
            }
            else
            {
                /* Add another 6 bits of base64 to the bit buffer. */
                WRITE_N_BITS(base64value, 6);
                first = 0;
            }

            /* Extract as many full 16 bit characters as possible from the
               bit buffer.
               */
            while (BITS_IN_BUFFER >= 16 && (target < targetEnd))
            {
                /* write a unicode */
                *target++ = READ_N_BITS(16);
                wroteone = 1;
            }

            if (BITS_IN_BUFFER >= 16)
                TARGETCHECK;

            if (base64EOF)
            {
                junk = READ_N_BITS(BITS_IN_BUFFER);
                if (junk)
                {
                    result = sourceCorrupt;
                }
            }
        }
```

```
            if (!shifted && !done)
            {
                if (c == SHIFT_IN)
                {
                    shifted = 1;
                    first = 1;
                    wroteone = 0;
                }
                else
                {
                    /* It must be a directly encoded character. */
                    if (c > 0x7f)
                    {
                        result = sourceCorrupt;
                    }
                    /* write a unicode */
                    TARGETCHECK;
                    *target++ = c;
                }
            }
        }
        while (!done);

        *sourceStart = source;
        *targetStart = target;
        return result;
    }
```

A.2 UTF-8

The term UTF-8 stands for UCS Transformation Format, 8-bit form.

To address the use of Unicode character data in 8-bit UNIX environments, X/Open developed and promulgated a transformation format known as File System Safe UTF (FSS-UTF, also known as UTF-2). Since that time, this UTF has been accepted as a normative addendum to ISO/IEC 10646 and has been renamed UTF-8, for UCS Transformation Format, 8-bit form.UTF-8 is described in ISO/IEC 10646 AM1. It is available in electronic form via the `unicode.org` World Wide Web Page and the `unicode.org` FTP archive (see *Section 1.6, Resources,* for addresses).

The UTF-8 transformation form maintains transparency for *all* of the ASCII code values (0...127). Furthermore, the values 0...127 do not appear in any byte of a transformed result except as the direct representation of these ASCII values. Each code value (non-surrogates) is represented in UTF-8 by 1, 2, or 3 bytes, depending on the code value. Pairs of surrogates take 4 bytes.

UTF-8 is a variable length encoding of the Unicode Standard using 8-bit sequences, where the high bits indicate which part of the sequence a byte belongs to. Table A-3 shows how the bits in a Unicode value (or surrogate pair) are distributed among the bytes in the UTF-8 encoding.

Table A-3. UTF-8 Bit Distribution

Unicode value	1st Byte	2nd Byte	3rd Byte	4th Byte
000000000xxxxxxx	0xxxxxxx			
00000yyyyyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
110110wwwwzzzzyy+ 110111yyyyxxxxxx	11110uuu [a]	10uuzzzz	10yyyyyy	10xxxxxx

a. where uuuuu = wwww + 1
(to account for addition of 10000_{16} as in *Section 3.7, Surrogates*)

Thus the ASCII range (U+0000 → U+007F) can be expressed as single bytes; most non-ideographics (U+0080 → U+07FF) can be expressed as 2 bytes; the remaining Unicode values can be expressed as 3 bytes, and surrogate pairs can be expressed as 4 bytes.

When converting Unicode values to UTF-8, always use the shortest form that can represent those values. This preserves uniqueness of encoding. For example, the Unicode value <0000000000000001> is encoded as <00000001>, not as <11000000 10000001>. The latter is an example of an unused UTF-8 byte sequence. *Do not make use of these unused byte sequences for encoding any other information.*

When converting from UTF-8 to Unicode values, however, implementations do not need to check that the shortest encoding is being used, which simplifies the conversion algorithm.

Some of the important characteristics of UTF-8 are

- Unicode characters from U+0000 to U+007E (ASCII repertoire) map to UTF-8 bytes 00 to 7E (ASCII values).
- ASCII values do *not* otherwise occur in a UTF-8 transformation. This provides compatibility with historical file systems and other systems which parse for ASCII bytes.
- It is very simple and efficient to convert to and from Unicode text.
- The first byte indicates the number of bytes to follow in a multi-byte sequence. This allows for efficient forward parsing.
- It is efficient to find the start of a character starting from an arbitrary location in an byte stream. You need to search at most four bytes backwards, and it is simple to recognize an initial byte. For example, in C
  `isInitialByte = ((byte & 0xC0) != 0x80);`
- UTF-8 is reasonably compact in terms of number of bytes used for encoding.

Sample Implementation of the UTF-8 Conversions

The following is provided as a sample implementation. For brevity, it does not deal with all possible error conditions.

```
/* ============================================================ */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* ============================================================ */
/*  The following 4 definitions are compiler-specific.
*/

typedef unsigned longUCS4;
typedef unsigned shortSimpleUniChar;
typedef unsigned shortUniChar;
typedef unsigned charUTF8;

typedef enum {false, true} Boolean;


const UCS4 kReplacementCharacter =0x0000FFFDUL;
const UCS4 kMaximumSimpleUniChar =0x0000FFFFUL;
const UCS4 kMaximumUniChar =0x0010FFFFUL;
const UCS4 kMaximumUCS4 =0x7FFFFFFFUL;

/* ============================================================ */
/*  Each of these routines converts the text between *sourceStart and
sourceEnd, putting the result into the buffer between *targetStart and
targetEnd. Note: the end pointers are *after* the last item: e.g.
*(sourceEnd - 1) is the last item.

    The return result indicates whether the conversion was successful,
and if not, whether the problem was in the source or target buffers.
```

```
    After the conversion, *sourceStart and *targetStart are both
updated to point to the end of last text successfully converted in
the respective buffers.
*/

typedef enum {
    ok,             /* conversion successful */
    sourceExhausted,/* partial character in source, but hit end */
    targetExhausted/* insuff. room in target for conversion */
} ConversionResult;

ConversionResultConvertUCS4toUniChar (
        UCS4** sourceStart, const UCS4* sourceEnd,
        UniChar** targetStart, const UniChar* targetEnd);

ConversionResultConvertUniChartoUCS4 (
        UniChar** sourceStart, UniChar* sourceEnd,
        UCS4** targetStart, const UCS4* targetEnd);

ConversionResultConvertUniChartoUTF8 (
        UniChar** sourceStart, const UniChar* sourceEnd,
        UTF8** targetStart, const UTF8* targetEnd);

ConversionResultConvertUTF8toUniChar (
        UTF8** sourceStart, UTF8* sourceEnd,
        UniChar** targetStart, const UniChar* targetEnd);

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

#include "ConvertUTF.h"

const int halfShift= 10;
const UCS4 halfBase= 0x0010000UL;
const UCS4 halfMask= 0x3FFUL;
const UCS4 kSurrogateHighStart= 0xD800UL;
const UCS4 kSurrogateHighEnd= 0xDBFFUL;
const UCS4 kSurrogateLowStart= 0xDC00UL;
const UCS4 kSurrogateLowEnd= 0xDFFFUL;

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

ConversionResultConvertUCS4toUniChar (
        UCS4** sourceStart, const UCS4* sourceEnd,
        UniChar** targetStart, const UniChar* targetEnd) {
    ConversionResult result = ok;
    register UCS4* source = *sourceStart;
    register UniChar* target = *targetStart;
    while (source < sourceEnd) {
        register UCS4 ch;
        if (target >= targetEnd) {
            result = targetExhausted; break;
        };
        ch = *source++;
        if (ch <= kMaximumSimpleUniChar) {
            *target++ = ch;
        } else if (ch > kMaximumUniChar) {
            *target++ = kReplacementCharacter;
        } else {
            if (target + 1 >= targetEnd) {
                result = targetExhausted; break;
            };
            ch -= halfBase;
            *target++ = (ch >> halfShift) + kSurrogateHighStart;
            *target++ = (ch & halfMask) + kSurrogateLowStart;
        };
    };
    *sourceStart = source;
    *targetStart = target;
    return result;
};

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

ConversionResultConvertUniChartoUCS4 (
        UniChar** sourceStart, UniChar* sourceEnd,
        UCS4** targetStart, const UCS4* targetEnd) {
    ConversionResult result = ok;
    register UniChar* source = *sourceStart;
    register UCS4* target = *targetStart;
    while (source < sourceEnd) {
        register UCS4 ch;
        ch = *source++;
        if (ch >= kSurrogateHighStart && ch <= kSurrogateHighEnd
                && source < sourceEnd) {
            register UCS4 ch2 = *source;
            if (ch2 >= kSurrogateLowStart && ch2 <= kSurrogateLowEnd) {
                ch = ((ch - kSurrogateHighStart) << halfShift)
                    + (ch2 - kSurrogateLowStart) + halfBase;
                ++source;
            };
        };
        if (target >= targetEnd) {
```

```
            result = targetExhausted; break;
        };
        *target++ = ch;
    };
    *sourceStart = source;
    *targetStart = target;
    return result;
};

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

UCS4 offsetsFromUTF8[6] ={0x00000000UL, 0x00003080UL, 0x000E2080UL,
                         0x03C82080UL, 0xFA082080UL, 0x82082080UL};
char bytesFromUTF8[256] = {
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2, 3,3,3,3,3,3,3,3,4,4,4,4,5,5,5,5};

UTF8 firstByteMark[7] = {0x00, 0x00, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC};

/* ================================================================ */
/*  This code is similar in effect to making successive calls on the
mbtowc and wctomb routines in FSS-UTF. However, it is considerably
different in code:
* it is adapted to be consistent with Unicode characters with surrogates,
* the interface converts a whole buffer to avoid function-call overhead
* constants have been gathered.
* loops & conditionals have been removed as much as possible for
efficiency, in favor of drop-through switch statements.
*/

/* ================================================================ */
ConversionResultConvertUniChartoUTF8 (
        UniChar** sourceStart, const UniChar* sourceEnd,
        UTF8** targetStart, const UTF8* targetEnd)
{
    ConversionResult result = ok;
    register UniChar* source = *sourceStart;
    register UTF8* target = *targetStart;
    while (source < sourceEnd) {
        register UCS4 ch;
        register unsigned short bytesToWrite = 0;
        register const UCS4 byteMask = 0xBF;
        register const UCS4 byteMark = 0x80;
        ch = *source++;
        if (ch >= kSurrogateHighStart && ch <= kSurrogateHighEnd
                && source < sourceEnd) {
            register UCS4 ch2 = *source;
            if (ch2 >= kSurrogateLowStart && ch2 <= kSurrogateLowEnd) {
                ch = ((ch - kSurrogateHighStart) << halfShift)
                    + (ch2 - kSurrogateLowStart) + halfBase;
                ++source;
            };
        };
        if (ch < 0x80) { bytesToWrite = 1;
        } else if (ch < 0x800) {bytesToWrite = 2;
        } else if (ch < 0x10000) {bytesToWrite = 3;
        } else if (ch < 0x200000) {bytesToWrite = 4;
        } else if (ch < 0x4000000) {bytesToWrite = 5;
        } else if (ch <= kMaximumUCS4){bytesToWrite = 6;
        } else {                bytesToWrite = 2;
                                        ch = kReplacementCharacter;
        }; /* I wish there were a smart way to avoid this conditional */

        target += bytesToWrite;
        if (target > targetEnd) {
            target -= bytesToWrite; result = targetExhausted; break;
        };
        switch (bytesToWrite) {/* note: code falls through cases! */
            case 6:*--target = (ch | byteMark) & byteMask; ch >>= 6;
            case 5:*--target = (ch | byteMark) & byteMask; ch >>= 6;
            case 4:*--target = (ch | byteMark) & byteMask; ch >>= 6;
            case 3:*--target = (ch | byteMark) & byteMask; ch >>= 6;
            case 2:*--target = (ch | byteMark) & byteMask; ch >>= 6;
            case 1:*--target =  ch | firstByteMark[bytesToWrite];
        };
        target += bytesToWrite;
    };
    *sourceStart = source;
    *targetStart = target;
    return result;
};

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

ConversionResultConvertUTF8toUniChar (
        UTF8** sourceStart, UTF8* sourceEnd,
```

```
        UniChar** targetStart, const UniChar* targetEnd)
{
    ConversionResult result = ok;
    register UTF8* source = *sourceStart;
    register UniChar* target = *targetStart;
    while (source < sourceEnd) {
        register UCS4 ch = 0;
        register unsigned short extraBytesToWrite = bytesFromUTF8[*source];
        if (source + extraBytesToWrite > sourceEnd) {
            result = sourceExhausted; break;
        };
        switch(extraBytesToWrite) {/* note: code falls through cases! */
            case 5:ch += *source++; ch <<= 6;
            case 4:ch += *source++; ch <<= 6;
            case 3:ch += *source++; ch <<= 6;
            case 2:ch += *source++; ch <<= 6;
            case 1:ch += *source++; ch <<= 6;
            case 0:ch += *source++;
        };
        ch -= offsetsFromUTF8[extraBytesToWrite];

        if (target >= targetEnd) {
            result = targetExhausted; break;
        };
        if (ch <= kMaximumSimpleUniChar) {
            *target++ = ch;
        } else if (ch > kMaximumUniChar) {
            *target++ = kReplacementCharacter;
        } else {
            if (target + 1 >= targetEnd) {
                result = targetExhausted; break;
            };
            ch -= halfBase;
            *target++ = (ch >> halfShift) + kSurrogateHighStart;
            *target++ = (ch & halfMask) + kSurrogateLowStart;
        };
    };
    *sourceStart = source;
    *targetStart = target;
    return result;
};
```