jeanlf fixed #2106 History

4 contributors

770 lines (703 sloc) | 20.4 KB

```
1  /*
2   *                     GPAC - Multimedia Framework C SDK
3   *
4   *                     Authors: Jean Le Feuvre
5   *                     Copyright (c) Telecom ParisTech 2007-2012
6   *                                     All rights reserved
7   *
8   *  This file is part of GPAC / common tools sub-project
9   *
10  *  GPAC is free software; you can redistribute it and/or modify
11  *  it under the terms of the GNU Lesser General Public License as published by
12  *  the Free Software Foundation; either version 2, or (at your option)
13  *  any later version.
14  *
15  *  GPAC is distributed in the hope that it will be useful,
16  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
17  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18  *  GNU Lesser General Public License for more details.
19  *
20  *  You should have received a copy of the GNU Lesser General Public
21  *  License along with this library; see the file COPYING.  If not, write to
22  *  the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
23  *
24  */
25
26 #ifndef GPAC_DISABLE_CORE_TOOLS
27
28 #include <gpac/utf.h>
29
```

```
#if 1


/*
 * Copyright 2001-2004 Unicode, Inc.
 *
 * Disclaimer
 *
 * This source code is provided as is by Unicode, Inc. No claims are
 * made as to fitness for any particular purpose. No warranties of any
 * kind are expressed or implied. The recipient agrees to determine
 * applicability of information provided. If this file has been
 * purchased on magnetic or optical media from Unicode, Inc., the
 * sole remedy for any claim will be exchange of defective media
 * within 90 days of receipt.
 *
 * Limitations on Rights to Redistribute This Code
 *
 * Unicode, Inc. hereby grants the right to freely use the information
 * supplied in this file in the creation of products supporting the
 * Unicode Standard, and to make copies of this file in any form
 * for internal or external distribution as long as this notice
 * remains attached.
 */

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

    Conversions between UTF32, UTF-16, and UTF-8. Source code file.
    Author: Mark E. Davis, 1994.
    Rev History: Rick McGowan, fixes & updates May 2001.
    Sept 2001: fixed const & error conditions per
        mods suggested by S. Parent & A. Lillich.
    June 2002: Tim Dodd added detection and handling of incomplete
        source sequences, enhanced error detection, added casts
        to eliminate compiler warnings.
    July 2003: slight mods to back out aggressive FFFE detection.
    Jan 2004: updated switches in from-UTF8 conversions.
    Oct 2004: updated to use UNI_MAX_LEGAL_UTF32 in UTF-32 conversions.

    See the header file "ConvertUTF.h" for complete documentation.

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

typedef u32 UTF32;      /* at least 32 bits */
typedef u16 UTF16;      /* at least 16 bits */
typedef u8 UTF8;        /* typically 8 bits */
typedef u8 Boolean; /* 0 or 1 */

```

```c
/* Some fundamental constants */
#define UNI_REPLACEMENT_CHAR (UTF32)0x0000FFFD
#define UNI_MAX_BMP (UTF32)0x0000FFFF
#define UNI_MAX_UTF16 (UTF32)0x0010FFFF
#define UNI_MAX_UTF32 (UTF32)0x7FFFFFFF
#define UNI_MAX_LEGAL_UTF32 (UTF32)0x0010FFFF

typedef enum {
        conversionOK,           /* conversion successful */
        sourceExhausted,        /* partial character in source, but hit end */
        targetExhausted,        /* insuff. room in target for conversion */
        sourceIllegal           /* source sequence is illegal/malformed */
} ConversionResult;

typedef enum {
        strictConversion = 0,
        lenientConversion
} ConversionFlags;

static const int halfShift  = 10; /* used for shifting by 10 bits */

static const UTF32 halfBase = 0x0010000UL;
static const UTF32 halfMask = 0x3FFUL;

#define UNI_SUR_HIGH_START  (UTF32)0xD800
#define UNI_SUR_HIGH_END    (UTF32)0xDBFF
#define UNI_SUR_LOW_START   (UTF32)0xDC00
#define UNI_SUR_LOW_END     (UTF32)0xDFFF
#define false      0
#define true        1

/*
 * Index into the table below with the first byte of a UTF-8 sequence to
 * get the number of trailing bytes that are supposed to follow it.
 * Note that *legal* UTF-8 values can't have 4 or 5-bytes. The table is
 * left as-is for anyone who may want to do such conversion, which was
 * allowed in earlier algorithms.
 */
static const char trailingBytesForUTF8[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
};

```

```
/*
 * Magic values subtracted from a buffer value during UTF8 conversion.
 * This table contains as many values as there might be trailing bytes
 * in a UTF-8 sequence.
 */
static const UTF32 offsetsFromUTF8[6] = { 0x00000000UL, 0x00003080UL, 0x000E2080UL,
                                          0x03C82080UL, 0xFA082080UL, 0x82082080UL
                                        };

/*
 * Once the bits are split out into bytes of UTF-8, this is a mask OR-ed
 * into the first byte, depending on how many bytes follow.  There are
 * as many entries in this table as there are UTF-8 sequence types.
 * (I.e., one byte sequence, two byte... etc.). Remember that sequencs
 * for *legal* UTF-8 will be 4 or fewer bytes total.
 */
static const UTF8 firstByteMark[7] = { 0x00, 0x00, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC };

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

/* 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 switches.
 * (See "Note A" at the bottom of the file for equivalent code.)
 * If your compiler supports it, the "isLegalUTF8" call can be turned
 * into an inline function.
 */

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

ConversionResult ConvertUTF16toUTF8 (
    const UTF16** sourceStart, const UTF16* sourceEnd,
    UTF8** targetStart, UTF8* targetEnd, ConversionFlags flags) {
        ConversionResult result = conversionOK;
        const UTF16* source = *sourceStart;
        UTF8* target = *targetStart;
        while (source < sourceEnd) {
                UTF32 ch;
                unsigned short bytesToWrite = 0;
                const UTF32 byteMask = 0xBF;
                const UTF32 byteMark = 0x80;
                const UTF16* oldSource = source; /* In case we have to back up because of target o
                ch = *source++;
                /* If we have a surrogate pair, convert to UTF32 first. */
                if (ch >= UNI_SUR_HIGH_START && ch <= UNI_SUR_HIGH_END) {
                        /* If the 16 bits following the high surrogate are in the source buffer...
                        if (source < sourceEnd) {
                                UTF32 ch2 = *source;
                                /* If it's a low surrogate, convert to UTF32. */
```

```
177                             if (ch2 >= UNI_SUR_LOW_START && ch2 <= UNI_SUR_LOW_END) {
178                                     ch = ((ch - UNI_SUR_HIGH_START) << halfShift)
179                                          + (ch2 - UNI_SUR_LOW_START) + halfBase;
180                                     ++source;
181                             } else if (flags == strictConversion) { /* it's an unpaired high s
182                                     --source; /* return to the illegal value itself */
183                                     result = sourceIllegal;
184                                     break;
185                             }
186                     } else { /* We don't have the 16 bits following the high surrogate. */
187                             --source; /* return to the high surrogate */
188                             result = sourceExhausted;
189                             break;
190                     }
191             } else if (flags == strictConversion) {
192                     /* UTF-16 surrogate values are illegal in UTF-32 */
193                     if (ch >= UNI_SUR_LOW_START && ch <= UNI_SUR_LOW_END) {
194                             --source; /* return to the illegal value itself */
195                             result = sourceIllegal;
196                             break;
197                     }
198             }
199             /* Figure out how many bytes the result will require */
200             if (ch < (UTF32)0x80) {
201                     bytesToWrite = 1;
202             } else if (ch < (UTF32)0x800) {
203                     bytesToWrite = 2;
204             } else if (ch < (UTF32)0x10000) {
205                     bytesToWrite = 3;
206             } else if (ch < (UTF32)0x110000) {
207                     bytesToWrite = 4;
208             } else {
209                     bytesToWrite = 3;
210                     ch = UNI_REPLACEMENT_CHAR;
211             }
212
213             target += bytesToWrite;
214             if (target > targetEnd) {
215                     source = oldSource; /* Back up source pointer! */
216                     target -= bytesToWrite;
217                     result = targetExhausted;
218                     break;
219             }
220             switch (bytesToWrite) { /* note: everything falls through. */
221             case 4:
222                     *--target = (UTF8)((ch | byteMark) & byteMask);
223                     ch >>= 6;
224             case 3:
225                     *--target = (UTF8)((ch | byteMark) & byteMask);
```

```
                        ch >>= 6;
                case 2:
                        *--target = (UTF8)((ch | byteMark) & byteMask);
                        ch >>= 6;
                case 1:
                        *--target =  (UTF8)(ch | firstByteMark[bytesToWrite]);
                }
                target += bytesToWrite;
        }
        *sourceStart = source;
        *targetStart = target;
        return result;
}

/*
 * Utility routine to tell whether a sequence of bytes is legal UTF-8.
 * This must be called with the length pre-determined by the first byte.
 * If not calling this from ConvertUTF8to*, then the length can be set by:
 *  length = trailingBytesForUTF8[*source]+1;
 * and the sequence is illegal right away if there aren't that many bytes
 * available.
 * If presented with a length > 4, this returns false.  The Unicode
 * definition of UTF-8 goes up to 4-byte sequences.
 */

Boolean isLegalUTF8(const UTF8 *source, int length) {
        UTF8 a;
        const UTF8 *srcptr = source+length;
        switch (length) {
        default:
                return false;
        /* Everything else falls through when "true"... */
        case 4:
                if ((a = (*--srcptr)) < 0x80 || a > 0xBF) return false;
        case 3:
                if ((a = (*--srcptr)) < 0x80 || a > 0xBF) return false;
        case 2:
                if ((a = (*--srcptr)) > 0xBF) return false;

                switch (*source) {
                /* no fall-through in this inner switch */
                case 0xE0:
                        if (a < 0xA0) return false;
                        break;
                case 0xED:
                        if (a > 0x9F) return false;
                        break;
                case 0xF0:
                        if (a < 0x90) return false;
```

```
                        break;
                case 0xF4:
                        if (a > 0x8F) return false;
                        break;
                default:
                        if (a < 0x80) return false;
                }

        case 1:
                if (*source >= 0x80 && *source < 0xC2) return false;
        }
        if (*source > 0xF4) return false;
        return true;
}

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

ConversionResult ConvertUTF8toUTF16 (
    const UTF8** sourceStart, const UTF8* sourceEnd,
    UTF16** targetStart, UTF16* targetEnd, ConversionFlags flags) {
        ConversionResult result = conversionOK;
        const UTF8* source = *sourceStart;
        UTF16* target = *targetStart;
        while (source < sourceEnd) {
                UTF32 ch = 0;
                unsigned short extraBytesToRead = trailingBytesForUTF8[*source];
                if (source + extraBytesToRead >= sourceEnd) {
                        result = sourceExhausted;
                        break;
                }
                /* Do this check whether lenient or strict */
                if (! isLegalUTF8(source, extraBytesToRead+1)) {
                        result = sourceIllegal;
                        break;
                }
                /*
                 * The cases all fall through. See "Note A" below.
                 */
                switch (extraBytesToRead) {
                case 5:
                        ch += *source++;
                        ch <<= 6; /* remember, illegal UTF-8 */
                case 4:
                        ch += *source++;
                        ch <<= 6; /* remember, illegal UTF-8 */
                case 3:
                        ch += *source++;
                        ch <<= 6;
                case 2:
```

```
                        ch += *source++;
                        ch <<= 6;
                case 1:
                        ch += *source++;
                        ch <<= 6;
                case 0:
                        ch += *source++;
                }
                ch -= offsetsFromUTF8[extraBytesToRead];

                if (target >= targetEnd) {
                        source -= (extraBytesToRead+1); /* Back up source pointer! */
                        result = targetExhausted;
                        break;
                }
                if (ch <= UNI_MAX_BMP) { /* Target is a character <= 0xFFFF */
                        /* UTF-16 surrogate values are illegal in UTF-32 */
                        if (ch >= UNI_SUR_HIGH_START && ch <= UNI_SUR_LOW_END) {
                                if (flags == strictConversion) {
                                        source -= (extraBytesToRead+1); /* return to the illegal v
                                        result = sourceIllegal;
                                        break;
                                } else {
                                        *target++ = UNI_REPLACEMENT_CHAR;
                                }
                        } else {
                                *target++ = (UTF16)ch; /* normal case */
                        }
                } else if (ch > UNI_MAX_UTF16) {
                        if (flags == strictConversion) {
                                result = sourceIllegal;
                                source -= (extraBytesToRead+1); /* return to the start */
                                break; /* Bail out; shouldn't continue */
                        } else {
                                *target++ = UNI_REPLACEMENT_CHAR;
                        }
                } else {
                        /* target is a character in range 0xFFFF - 0x10FFFF. */
                        if (target + 1 >= targetEnd) {
                                source -= (extraBytesToRead+1); /* Back up source pointer! */
                                result = targetExhausted;
                                break;
                        }
                        ch -= halfBase;
                        *target++ = (UTF16)((ch >> halfShift) + UNI_SUR_HIGH_START);
                        *target++ = (UTF16)((ch & halfMask) + UNI_SUR_LOW_START);
                }
        }
        *sourceStart = source;
```

```
373         *targetStart = target;
374         return result;
375 }
376
377
378 GF_EXPORT
379 Bool gf_utf8_is_legal(const u8 *data, u32 length)
380 {
381         //we simply run ConvertUTF8toUTF16 without target
382         const UTF8** sourceStart = (const UTF8**) &data;
383         const UTF8* sourceEnd = (const UTF8*) ( data + length );
384         ConversionResult result = conversionOK;
385         const UTF8* source = *sourceStart;
386
387         while (source < sourceEnd) {
388                 UTF32 ch = 0;
389                 unsigned short extraBytesToRead = trailingBytesForUTF8[*source];
390                 if (source + extraBytesToRead >= sourceEnd) {
391                         result = sourceExhausted;
392                         break;
393                 }
394                 /* Do this check whether lenient or strict */
395                 if (! isLegalUTF8(source, extraBytesToRead+1)) {
396                         result = sourceIllegal;
397                         break;
398                 }
399                 /*
400                  * The cases all fall through. See "Note A" below.
401                  */
402                 switch (extraBytesToRead) {
403                 case 5:
404                         ch += *source++;
405                         ch <<= 6; /* remember, illegal UTF-8 */
406                 case 4:
407                         ch += *source++;
408                         ch <<= 6; /* remember, illegal UTF-8 */
409                 case 3:
410                         ch += *source++;
411                         ch <<= 6;
412                 case 2:
413                         ch += *source++;
414                         ch <<= 6;
415                 case 1:
416                         ch += *source++;
417                         ch <<= 6;
418                 case 0:
419                         ch += *source++;
420                 }
421                 ch -= offsetsFromUTF8[extraBytesToRead];
```

```
422
423                 if (ch <= UNI_MAX_BMP) { /* Target is a character <= 0xFFFF */
424                         /* UTF-16 surrogate values are illegal in UTF-32 */
425                         if (ch >= UNI_SUR_HIGH_START && ch <= UNI_SUR_LOW_END) {
426                                 result = sourceIllegal;
427                                 break;
428                         }
429                 } else if (ch > UNI_MAX_UTF16) {
430                         result = sourceIllegal;
431                         break; /* Bail out; shouldn't continue */
432                 }
433         }
434         return (result==conversionOK) ? GF_TRUE : GF_FALSE;
435 }
436
437 GF_EXPORT
438 u32 gf_utf8_wcslen (const unsigned short *s)
439 {
440         const unsigned short* ptr;
441         if (!s) return 0;
442         for (ptr = s; *ptr != (unsigned short)'\0'; ptr++) {
443         }
444         return (u32) ( ptr - s );
445 }
446
447 GF_EXPORT
448 u32 gf_utf8_wcstombs(char* dest, size_t len, const unsigned short** srcp)
449 {
450         if (!srcp || !*srcp)
451                 return 0;
452         else {
453                 const UTF16** sourceStart = srcp;
454                 const UTF16* sourceEnd = *srcp + gf_utf8_wcslen(*srcp);
455                 UTF8* targetStart = (UTF8*) dest;
456                 UTF8* targetEnd = (UTF8*) dest + len;
457                 ConversionFlags flags = strictConversion;
458
459                 ConversionResult res = ConvertUTF16toUTF8(sourceStart, sourceEnd, &targetStart, ta
460                 if (res != conversionOK) return GF_UTF8_FAIL;
461                 *targetStart = 0;
462                 *srcp=NULL;
463                 return (u32) strlen(dest);
464         }
465 }
466
467 GF_EXPORT
468 u32 gf_utf8_mbstowcs(unsigned short* dest, size_t len, const char** srcp)
469 {
470         if (!srcp || !*srcp)
```

```
471                 return 0;
472         else {
473                 const UTF8** sourceStart = (const UTF8**) srcp;
474                 const UTF8* sourceEnd = (const UTF8*) ( *srcp + strlen( *srcp) );
475                 UTF16* targetStart = (UTF16* ) dest;
476                 UTF16* targetEnd = (UTF16* ) (dest + len);
477                 ConversionFlags flags = strictConversion;
478                 ConversionResult res = ConvertUTF8toUTF16(sourceStart, sourceEnd, &targetStart, ta
479                 if (res != conversionOK) return GF_UTF8_FAIL;
480                 *targetStart = 0;
481                 *srcp=NULL;
482                 return gf_utf8_wcslen(dest);
483         }
484 }
485
486
487 #else
488
489 GF_EXPORT
490 u32 gf_utf8_wcslen (const unsigned short *s)
491 {
492         const unsigned short* ptr;
493         for (ptr = s; *ptr != (unsigned short)'\0'; ptr++) {
494         }
495         return (u32) (ptr - s);
496 }
497
498 GF_EXPORT
499 u32 gf_utf8_wcstombs(char* dest, size_t len, const unsigned short** srcp)
500 {
501         /*
502         * Original code from the GNU UTF-8 Library
503         */
504         size_t count;
505         const unsigned short * src = *srcp;
506
507         if (dest != NULL) {
508                 char* destptr = dest;
509                 for (;; src++) {
510                         unsigned char c;
511                         unsigned short wc = *src;
512                         if (wc < 0x80) {
513                                 if (wc == (wchar_t)'\0') {
514                                         if (len == 0) {
515                                                 *srcp = src;
516                                                 break;
517                                         }
518                                         *destptr = '\0';
519                                         *srcp = NULL;
```

```
                                        break;
                                }
                                count = 0;
                                c = (unsigned char) wc;
                        } else if (wc < 0x800) {
                                count = 1;
                                c = (unsigned char) ((wc >> 6) | 0xC0);
                        } else {
                                count = 2;
                                c = (unsigned char) ((wc >> 12) | 0xE0);
                        }
                        if (len <= count) {
                                *srcp = src;
                                break;
                        }
                        len -= count+1;
                        *destptr++ = c;
                        if (count > 0)
                                do {
                                        *destptr++ = (unsigned char)(((wc >> (6 * --count)) & 0x3F
                                } while (count > 0);
                }
                return (u32) (destptr - dest);
        } else {
                /* Ignore dest and len. */
                size_t totalcount = 0;
                for (;; src++) {
                        unsigned short wc = *src;
                        size_t count;
                        if (wc < 0x80) {
                                if (wc == (wchar_t)'\0') {
                                        *srcp = NULL;
                                        break;
                                }
                                count = 1;
                        } else if (wc < 0x800) {
                                count = 2;
                        } else {
                                count = 3;
                        }
                        totalcount += count;
                }
                return (u32) totalcount;
        }
}


typedef struct
{
```

```
        u32 count : 16;   /* number of bytes remaining to be processed */
        u32 value : 16;   /* if count > 0: partial wide character */
        /*
           If WCHAR_T_BITS == 16, need 2 bits for count,
           12 bits for value (10 for mbstowcs direction, 12 for wcstombs direction).
        */
} gf_utf8_mbstate_t;

static gf_utf8_mbstate_t internal;

GF_EXPORT
u32 gf_utf8_mbstowcs(unsigned short* dest, size_t len, const char** srcp)
{
        gf_utf8_mbstate_t* ps = &internal;
        const char *src = *srcp;

        unsigned short* destptr = dest;
        for (; len > 0; destptr++, len--) {
                const char* backup_src = src;
                unsigned char c;
                unsigned short wc;
                size_t count;
                if (ps->count == 0) {
                        c = (unsigned char) *src;
                        if (c < 0x80) {
                                *destptr = (wchar_t) c;
                                if (c == 0) {
                                        src = NULL;
                                        break;
                                }
                                src++;
                                continue;
                        } else if (c < 0xC0) {
                                /* Spurious 10XXXXXX byte is invalid. */
                                goto bad_input;
                        }
                        if (c < 0xE0) {
                                wc = (wchar_t)(c & 0x1F) << 6;
                                count = 1;
                                if (c < 0xC2) goto bad_input;
                        } else if (c < 0xF0) {
                                wc = (wchar_t)(c & 0x0F) << 12;
                                count = 2;
                        }
                        else goto bad_input;
                        src++;
                } else {
                        wc = ps->value << 6;
                        count = ps->count;
```

```
                }
                for (;;) {
                        c = (unsigned char) *src++ ^ 0x80;
                        if (!(c < 0x40)) goto bad_input_backup;
                        wc |= (unsigned short) c << (6 * --count);
                        if (count == 0)
                                break;
                        /* The following test is only necessary once for every character,
                        but it would be too complicated to perform it once only, on
                        the first pass through this loop. */
                        if ((unsigned short) wc < ((unsigned short) 1 << (5 * count + 6)))
                                goto bad_input_backup;
                }
                *destptr = wc;
                ps->count = 0;
                continue;

bad_input_backup:
                src = backup_src;
                goto bad_input;
        }
        *srcp = src;
        return (u32) (destptr - dest);

bad_input:
        *srcp = src;
        return GF_UTF8_FAIL;
}


#endif


GF_EXPORT
GF_Err gf_utf_get_utf8_string_from_bom(const u8 *data, u32 size, char **out_ptr, char **result)
{
        u32 unicode_type = 0;
        if (!out_ptr || !result || !data) return GF_BAD_PARAM;
        *out_ptr = NULL;
        *result = (char *) data;

        if (size>=5) {
                /*0: no unicode, 1: UTF-16BE, 2: UTF-16LE*/
                if ((data[0]==0xFF) && (data[1]==0xFE)) {
                        if (!data[2] && !data[3]) {
                                return GF_OK;
                        } else {
                                unicode_type = 2;
                        }
```

```
667                 } else if ((data[0]==0xFE) && (data[1]==0xFF)) {
668                         if (!data[2] && !data[3]) {
669                                 return GF_OK;
670                         } else {
671                                 unicode_type = 1;
672                         }
673                 } else if ((data[0]==0xEF) && (data[1]==0xBB) && (data[2]==0xBF)) {
674                         *result = (char *) (data+4);
675                         return GF_OK;
676                 }
677         }
678
679         if (!unicode_type) {
680                 *result = (char *) data;
681                 return GF_OK;
682         }
683
684         if (size%2) size--;
685         u16 *str_wc = gf_malloc(size+2);
686         if (!str_wc) return GF_OUT_OF_MEM;
687         u16 *srcwc;
688         char *dst = gf_malloc(size+2);
689         if (!dst) {
690                 gf_free(str_wc);
691                 return GF_OUT_OF_MEM;
692         }
693         *out_ptr = dst;
694         u32 i;
695         for (i=0; i<size; i+=2) {
696                 u16 wchar=0;
697                 u8 c1 = data[i];
698                 u8 c2 = data[i+1];
699
700                 /*Little-endian order*/
701                 if (unicode_type==2) {
702                         if (c2) {
703                                 wchar = c2;
704                                 wchar <<=8;
705                                 wchar |= c1;
706                         }
707                         else wchar = c1;
708                 } else {
709                         wchar = c1;
710                         if (c2) {
711                                 wchar <<= 8;
712                                 wchar |= c2;
713                         }
714                 }
715                 str_wc[i/2] = wchar;
```

```
716         }
717         str_wc[i/2] = 0;
718         srcwc = str_wc;
719         u32 res = gf_utf8_wcstombs(dst, size, (const unsigned short **) &srcwc);
720         gf_free(str_wc);
721         if (res==GF_UTF8_FAIL) {
722                 gf_free(dst);
723                 *out_ptr = NULL;
724                 return GF_IO_ERR;
725         }
726         *result = dst;
727         return GF_OK;
728 }
729
730
731 #if defined(WIN32)
732
733 GF_EXPORT
734 wchar_t* gf_utf8_to_wcs(const char* str)
735 {
736         size_t source_len;
737         wchar_t* result;
738         if (str == 0) return 0;
739         source_len = strlen(str);
740         result = gf_calloc(source_len + 1, sizeof(wchar_t));
741         if (!result)
742                 return 0;
743         if (gf_utf8_mbstowcs(result, source_len, &str) == GF_UTF8_FAIL) {
744                 gf_free(result);
745                 return 0;
746         }
747         return result;
748 }
749
750 GF_EXPORT
751 char* gf_wcs_to_utf8(const wchar_t* str)
752 {
753         size_t source_len;
754         char* result;
755         if (str == 0) return 0;
756         source_len = wcslen(str);
757         result = gf_calloc(source_len + 1, UTF8_MAX_BYTES_PER_CHAR);
758         if (!result)
759                 return 0;
760         if (gf_utf8_wcstombs(result, source_len * UTF8_MAX_BYTES_PER_CHAR, &str) == GF_UTF8_FAIL)
761                 gf_free(result);
762                 return 0;
763         }
764         return result;
```

```
765 }
766 #endif
767
768 #endif /* GPAC_DISABLE_CORE_TOOLS */
769
770
```