```
**********************************************************
    8722 Sun Sep  9 18:34:26 2012
new/usr/src/cmd/localedef/ctype.c
3154 Nonconforming tolower and toupper with UTF-8 locales
**********************************************************
______unchanged_portion_omitted_

 205 void
 206 dump_ctype(void)
 207 {
 208         FILE            *f;
 209         _FileRuneLocale rl;
 210         ctype_node_t    *ctn, *last_ct, *last_lo, *last_up;
 211         _FileRuneEntry  *ct = NULL;
 212         _FileRuneEntry  *lo = NULL;
 213         _FileRuneEntry  *up = NULL;
 214         wchar_t         wc;
 215 #endif /* ! codereview */

 217         (void) memset(&rl, 0, sizeof (rl));
 218         last_ct = NULL;
 219         last_lo = NULL;
 220         last_up = NULL;

 222         if ((f = open_category()) == NULL)
 223                 return;

 225         (void) memcpy(rl.magic, _FILE_RUNE_MAGIC_1, 8);
 226         (void) strncpy(rl.encoding, get_wide_encoding(), sizeof (rl.encoding));

 228         /*
 229          * Preinit the identity map.
 230          */
 231         for (wc = 0; (unsigned)wc < _CACHED_RUNES; wc++) {
 232                 rl.maplower[wc] = wc;
 233                 rl.mapupper[wc] = wc;
 234         }

 236 #endif /* ! codereview */
 237         for (ctn = avl_first(&ctypes); ctn; ctn = AVL_NEXT(&ctypes, ctn)) {
 238                 int conflict = 0;
 239 #endif /* ! codereview */

 241                 wc = ctn->wc;
 214                 wchar_t wc = ctn->wc;
 215                 int conflict = 0;

 243                 /*
 244                  * POSIX requires certain portable characters have
 245                  * certain types.  Add them if they are missing.
 246                  */
 247                 if ((wc >= 1) && (wc <= 127)) {
 248                         if ((wc >= 'A') && (wc <= 'Z'))
 249                                 ctn->ctype |= _ISUPPER;
 250                         if ((wc >= 'a') && (wc <= 'z'))
 251                                 ctn->ctype |= _ISLOWER;
 252                         if ((wc >= '0') && (wc <= '9'))
 253                                 ctn->ctype |= _ISDIGIT;
 254                         if (strchr(" \f\n\r\t\v", (char)wc) != NULL)
 255                                 ctn->ctype |= _ISSPACE;
 256                         if (strchr("0123456789ABCDEFabcdef", (char)wc) != NULL)
 257                                 ctn->ctype |= _ISXDIGIT;
 258                         if (strchr(" \t", (char)wc))
 259                                 ctn->ctype |= _ISBLANK;

 261                         /*
```



```
 262                          * Technically these settings are only
 263                          * required for the C locale.  However, it
 264                          * turns out that because of the historical
 265                          * version of isprint(), we need them for all
 266                          * locales as well.  Note that these are not
 267                          * necessarily valid punctation characters in
 268                          * the current language, but ispunct() needs
 269                          * to return TRUE for them.
 270                          */
 271                         if (strchr("!\"'#$%&()*+,-./:;<=>?@[\\]^_`{|}~",
 272                             (char)wc))
 273                                 ctn->ctype |= _ISPUNCT;
 274                 }

 276                 /*
 277                  * POSIX also requires that certain types imply
 278                  * others.  Add any inferred types here.
 279                  */
 280                 if (ctn->ctype & (_ISUPPER |_ISLOWER))
 281                         ctn->ctype |= _ISALPHA;
 282                 if (ctn->ctype & _ISDIGIT)
 283                         ctn->ctype |= _ISXDIGIT;
 284                 if (ctn->ctype & _ISBLANK)
 285                         ctn->ctype |= _ISSPACE;
 286                 if (ctn->ctype & (_ISALPHA|_ISDIGIT|_ISXDIGIT))
 287                         ctn->ctype |= _ISGRAPH;
 288                 if (ctn->ctype & _ISGRAPH)
 289                         ctn->ctype |= _ISPRINT;

 291                 /*
 292                  * Finally, POSIX requires that certain combinations
 293                  * are invalid.  We don't flag this as a fatal error,
 294                  * but we will warn about.
 295                  */
 296                 if ((ctn->ctype & _ISALPHA) &&
 297                     (ctn->ctype & (_ISPUNCT|_ISDIGIT)))
 298                         conflict++;
 299                 if ((ctn->ctype & _ISPUNCT) &
 300                     (ctn->ctype & (_ISDIGIT|_ISALPHA|_ISXDIGIT)))
 301                         conflict++;
 302                 if ((ctn->ctype & _ISSPACE) && (ctn->ctype & _ISGRAPH))
 303                         conflict++;
 304                 if ((ctn->ctype & _ISCNTRL) & _ISPRINT)
 305                         conflict++;
 306                 if ((wc == ' ') && (ctn->ctype & (_ISPUNCT|_ISGRAPH)))
 307                         conflict++;

 309                 if (conflict) {
 310                         warn("conflicting classes for character 0x%x (%x)",
 311                             wc, ctn->ctype);
 312                 }
 313                 /*
 314                  * Handle the lower 256 characters using the simple
 315                  * optimization.  Note that if we have not defined the
 316                  * upper/lower case, then we identity map it.
 317                  */
 318                 if ((unsigned)wc < _CACHED_RUNES) {
 319                         rl.runetype[wc] = ctn->ctype;
 320                         if (ctn->tolower)
 321                                 rl.maplower[wc] = ctn->tolower;
 322                         if (ctn->toupper)
 323                                 rl.mapupper[wc] = ctn->toupper;
 294                         rl.maplower[wc] = ctn->tolower ? ctn->tolower : wc;
 295                         rl.mapupper[wc] = ctn->toupper ? ctn->toupper : wc;
 324                         continue;
 325                 }
```

```
327                 if ((last_ct != NULL) && (last_ct->ctype == ctn->ctype)) {
328                         ct[rl.runetype_ext_nranges-1].max = wc;
329                         last_ct = ctn;
330                 } else {
331                         rl.runetype_ext_nranges++;
332                         ct = realloc(ct,
333                             sizeof (*ct) * rl.runetype_ext_nranges);
334                         ct[rl.runetype_ext_nranges - 1].min = wc;
335                         ct[rl.runetype_ext_nranges - 1].max = wc;
336                         ct[rl.runetype_ext_nranges - 1].map = ctn->ctype;
337                         last_ct = ctn;
338                 }
339                 if (ctn->tolower == 0) {
340                         last_lo = NULL;
341                 } else if ((last_lo != NULL) &&
342                     (last_lo->tolower + 1 == ctn->tolower)) {
343                         lo[rl.maplower_ext_nranges-1].max = wc;
344                         last_lo = ctn;
345                 } else {
346                         rl.maplower_ext_nranges++;
347                         lo = realloc(lo,
348                             sizeof (*lo) * rl.maplower_ext_nranges);
349                         lo[rl.maplower_ext_nranges - 1].min = wc;
350                         lo[rl.maplower_ext_nranges - 1].max = wc;
351                         lo[rl.maplower_ext_nranges - 1].map = ctn->tolower;
352                         last_lo = ctn;
353                 }

355                 if (ctn->toupper == 0) {
356                         last_up = NULL;
357                 } else if ((last_up != NULL) &&
358                     (last_up->toupper + 1 == ctn->toupper)) {
359                         up[rl.mapupper_ext_nranges-1].max = wc;
360                         last_up = ctn;
361                 } else {
362                         rl.mapupper_ext_nranges++;
363                         up = realloc(up,
364                             sizeof (*up) * rl.mapupper_ext_nranges);
365                         up[rl.mapupper_ext_nranges - 1].min = wc;
366                         up[rl.mapupper_ext_nranges - 1].max = wc;
367                         up[rl.mapupper_ext_nranges - 1].map = ctn->toupper;
368                         last_up = ctn;
369                 }
370         }

372         if ((wr_category(&rl, sizeof (rl), f) < 0) ||
373             (wr_category(ct, sizeof (*ct) * rl.runetype_ext_nranges, f) < 0) ||
374             (wr_category(lo, sizeof (*lo) * rl.maplower_ext_nranges, f) < 0) ||
375             (wr_category(up, sizeof (*up) * rl.mapupper_ext_nranges, f) < 0)) {
376                 return;
377         }

379         close_category(f);
380 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   14341 Sun Sep  9 18:34:27 2012
new/usr/src/cmd/localedef/wide.c
3154 Nonconforming tolower and toupper with UTF-8 locales
**********************************************************
______unchanged_portion_omitted_

 161 /*
 162  * This is used for 8-bit encodings.
 163  */
 164 /* ARGSUSED */
 165 #endif /* ! codereview */
 166 int
 167 towide_none(wchar_t *c, const char *mb, unsigned n)
 168 {
 169         _NOTE(ARGUNUSED(n));

 171         if (mb_cur_max != 1) {
 172                 werr("invalid or unsupported multibyte locale");
 173                 return (-1);
 174         }
 175         *c = (uint8_t)*mb;
 176         return (1);
 177 }

 179 int
 180 tomb_none(char *mb, wchar_t wc)
 181 {
 182         if (mb_cur_max != 1) {
 183                 werr("invalid or unsupported multibyte locale");
 184                 return (-1);
 185         }
 186         *(uint8_t *)mb = (wc & 0xff);
 187         mb[1] = 0;
 188         return (1);
 189 }

 191 /*
 192  * UTF-8 stores wide characters in UTF-32 form.
 193  */
 194 int
 195 towide_utf8(wchar_t *wc, const char *mb, unsigned n)
 196 {
 197         wchar_t c;
 198         int     nb;
 199         int     lv;     /* lowest legal value */
 200         int     i;
 201         const uint8_t *s = (const uint8_t *)mb;

 203         c = *s;

 205         if ((c & 0x80) == 0) {
 206                 /* 7-bit ASCII */
 207                 *wc = c;
 208                 return (1);
 209         } else if ((c & 0xe0) == 0xc0) {
 210                 /* u80-u7ff - two bytes encoded */
 211                 nb = 2;
 212                 lv = 0x80;
 213                 c &= ~0xe0;
 214         } else if ((c & 0xf0) == 0xe0) {
 215                 /* u800-uffff - three bytes encoded */
 216                 nb = 3;
 217                 lv = 0x800;
 218                 c &= ~0xf0;
 219         } else if ((c & 0xf8) == 0xf0) {
```



```
 220                 /* u1000-u1fffff - four bytes encoded */
 221                 nb = 4;
 222                 lv = 0x1000;
 223                 c &= ~0xf8;
 224         } else {
 225                 /* 5 and 6 byte encodings are not legal unicode */
 226                 werr("utf8 encoding too large (%s)", show_mb(mb));
 227                 return (-1);
 228         }
 229         if (nb > n) {
 230                 werr("incomplete utf8 sequence (%s)", show_mb(mb));
 231                 return (-1);
 232         }

 234         for (i = 1; i < nb; i++) {
 235                 if (((s[i]) & 0xc0) != 0x80) {
 236                         werr("illegal utf8 byte (%x)", s[i]);
 237                         return (-1);
 238                 }
 239                 c <<= 6;
 240                 c |= (s[i] & 0x3f);
 241         }

 243         if (c < lv) {
 244                 werr("illegal redundant utf8 encoding (%s)", show_mb(mb));
 245                 return (-1);
 246         }
 247         *wc = c;
 248         return (nb);
 249 }

 251 int
 252 tomb_utf8(char *mb, wchar_t wc)
 253 {
 254         uint8_t *s = (uint8_t *)mb;
 255         uint8_t msk;
 256         int cnt;
 257         int i;

 259         if (wc <= 0x7f) {
 260                 s[0] = wc & 0x7f;
 261                 s[1] = 0;
 262                 return (1);
 263         }
 264         if (wc <= 0x7ff) {
 265                 cnt = 2;
 266                 msk = 0xc0;
 267         } else if (wc <= 0xffff) {
 268                 cnt = 3;
 269                 msk = 0xe0;
 270         } else if (wc <= 0x1fffff) {
 271                 cnt = 4;
 272                 msk = 0xf0;
 273         } else {
 274                 werr("illegal uf8 char (%x)", wc);
 275                 return (-1);
 276         }
 277         for (i = cnt - 1; i; i--) {
 278                 s[i] = (wc & 0x3f) | 0x80;
 279                 wc >>= 6;
 280         }
 281         s[0] = (msk) | wc;
 282         s[cnt] = 0;
 283         return (cnt);
 284 }
```

```
286 /*
287  * Several encodings share a simplistic dual byte encoding.  In these
288  * forms, they all indicate that a two byte sequence is to be used if
289  * the first byte has its high bit set.  They all store this simple
290  * encoding as a 16-bit value, although a great many of the possible
291  * code points are not used in most character sets.  This gives a possible
292  * set of just over 32,000 valid code points.
293  *
294  * 0x00 - 0x7f          - 1 byte encoding
295  * 0x80 - 0x7fff        - illegal
296  * 0x8000 - 0xffff      - 2 byte encoding
297  */
298 static int
299 towide_dbcs(wchar_t *wc, const char *mb, unsigned n)
300 {
301         wchar_t c;

303         c = *(uint8_t *)mb;

305         if ((c & 0x80) == 0) {
306                 /* 7-bit */
307                 *wc = c;
308                 return (1);
309         }
310         if (n < 2) {
311                 werr("incomplete character sequence (%s)", show_mb(mb));
312                 return (-1);
313         }

315         /* Store both bytes as a single 16-bit wide. */
316         c <<= 8;
317         c |= (uint8_t)(mb[1]);
318         *wc = c;
319         return (2);
320 }

322 /*
323  * Most multibyte locales just convert the wide character to the multibyte
324  * form by stripping leading null bytes, and writing the 32-bit quantity
325  * in big-endian order.
326  */
327 int
328 tomb_mbs(char *mb, wchar_t wc)
329 {
330         uint8_t *s = (uint8_t *)mb;
331         int     n = 0, c;

333         if ((wc & 0xff000000U) != 0) {
334                 n = 4;
335         } else if ((wc & 0x00ff0000U) != 0) {
336                 n = 3;
337         } else if ((wc & 0x0000ff00U) != 0) {
338                 n = 2;
339         } else {
340                 n = 1;
341         }
342         c = n;
343         while (n) {
344                 n--;
345                 s[n] = wc & 0xff;
346                 wc >>= 8;
347         }
348         /* ensure null termination */
349         s[c] = 0;
350         return (c);
351 }
```



```
354 /*
355  * big5 is a simple dual byte character set.
356  */
357 int
358 towide_big5(wchar_t *wc, const char *mb, unsigned n)
359 {
360         return (towide_dbcs(wc, mb, n));
361 }

363 /*
364  * GBK encodes wides in the same way that big5 does, the high order
365  * bit of the first byte indicates a double byte character.
366  */
367 int
368 towide_gbk(wchar_t *wc, const char *mb, unsigned n)
369 {
370         return (towide_dbcs(wc, mb, n));
371 }

373 /*
374  * GB2312 is another DBCS.  Its cleaner than others in that the second
375  * byte does not encode ASCII, but it supports characters.
376  */
377 int
378 towide_gb2312(wchar_t *wc, const char *mb, unsigned n)
379 {
380         return (towide_dbcs(wc, mb, n));
381 }

383 /*
384  * GB18030.  This encodes as 8, 16, or 32-bits.
385  * 7-bit values are in 1 byte,  4 byte sequences are used when
386  * the second byte encodes 0x30-39 and all other sequences are 2 bytes.
387  */
388 int
389 towide_gb18030(wchar_t *wc, const char *mb, unsigned n)
390 {
391         wchar_t c;

393         c = *(uint8_t *)mb;

395         if ((c & 0x80) == 0) {
396                 /* 7-bit */
397                 *wc = c;
398                 return (1);
399         }
400         if (n < 2) {
401                 werr("incomplete character sequence (%s)", show_mb(mb));
402                 return (-1);
403         }

405         /* pull in the second byte */
406         c <<= 8;
407         c |= (uint8_t)(mb[1]);

409         if (((c & 0xff) >= 0x30) && ((c & 0xff) <= 0x39)) {
410                 if (n < 4) {
411                         werr("incomplete 4-byte character sequence (%s)",
412                             show_mb(mb));
413                         return (-1);
414                 }
415                 c <<= 8;
416                 c |= (uint8_t)(mb[2]);
417                 c <<= 8;
```

```
418                 c |= (uint8_t)(mb[3]);
419                 *wc = c;
420                 return (4);
421         }

423         *wc = c;
424         return (2);
425 }

427 /*
428  * MS-Kanji (aka SJIS) is almost a clean DBCS like the others, but it
429  * also has a range of single byte characters above 0x80.  (0xa1-0xdf).
430  */
431 int
432 towide_mskanji(wchar_t *wc, const char *mb, unsigned n)
433 {
434         wchar_t c;

436         c = *(uint8_t *)mb;

438         if ((c < 0x80) || ((c > 0xa0) && (c < 0xe0))) {
439                 /* 7-bit */
440                 *wc = c;
441                 return (1);
442         }

444         if (n < 2) {
445                 werr("incomplete character sequence (%s)", show_mb(mb));
446                 return (-1);
447         }

449         /* Store both bytes as a single 16-bit wide. */
450         c <<= 8;
451         c |= (uint8_t)(mb[1]);
452         *wc = c;
453         return (2);
454 }

456 /*
457  * EUC forms.  EUC encodings are "variable".  FreeBSD carries some additional
458  * variable data to encode these, but we're going to treat each as independent
459  * instead.  Its the only way we can sensibly move forward.
460  *
461  * Note that the way in which the different EUC forms vary is how wide
462  * CS2 and CS3 are and what the first byte of them is.
463  */
464 static int
465 towide_euc_impl(wchar_t *wc, const char *mb, unsigned n,
466     uint8_t cs2, uint8_t cs2width, uint8_t cs3, uint8_t cs3width)
467 {
468         int i;
469         int width;
470         wchar_t c;

472         c = *(uint8_t *)mb;

474         /*
475          * All variations of EUC encode 7-bit ASCII as one byte, and use
476          * additional bytes for more than that.
477          */
478         if ((c & 0x80) == 0) {
479                 /* 7-bit */
480                 *wc = c;
481                 return (1);
482         }
```



```
484         /*
485          * All EUC variants reserve 0xa1-0xff to identify CS1, which
486          * is always two bytes wide.  Note that unused CS will be zero,
487          * and that cannot be true because we know that the high order
488          * bit must be set.
489          */
490         if (c >= 0xa1) {
491                 width = 2;
492         } else if (c == cs2) {
493                 width = cs2width;
494         } else if (c == cs3) {
495                 width = cs3width;
496         }

498         if (n < width) {
499                 werr("incomplete character sequence (%s)", show_mb(mb));
500                 return (-1);
501         }

503         for (i = 1; i < width; i++) {
504                 /* pull in the next byte */
505                 c <<= 8;
506                 c |= (uint8_t)(mb[i]);
507         }

509         *wc = c;
510         return (width);
511 }

513 /*
514  * EUC-CN encodes as follows:
515  *
516  * Code set 0 (ASCII):                          0x21-0x7E
517  * Code set 1 (CNS 11643-1992 Plane 1):         0xA1A1-0xFEFE
518  * Code set 2:                                  unused
519  * Code set 3:                                  unused
520  */
521 int
522 towide_euccn(wchar_t *wc, const char *mb, unsigned n)
523 {
524         return (towide_euc_impl(wc, mb, n, 0x8e, 4, 0, 0));
525 }

527 /*
528  * EUC-JP encodes as follows:
529  *
530  * Code set 0 (ASCII or JIS X 0201-1976 Roman): 0x21-0x7E
531  * Code set 1 (JIS X 0208):                     0xA1A1-0xFEFE
532  * Code set 2 (half-width katakana):            0x8EA1-0x8EDF
533  * Code set 3 (JIS X 0212-1990):                0x8FA1A1-0x8FFEFE
534  */
535 int
536 towide_eucjp(wchar_t *wc, const char *mb, unsigned n)
537 {
538         return (towide_euc_impl(wc, mb, n, 0x8e, 2, 0x8f, 3));
539 }

541 /*
542  * EUC-KR encodes as follows:
543  *
544  * Code set 0 (ASCII or KS C 5636-1993):        0x21-0x7E
545  * Code set 1 (KS C 5601-1992):                 0xA1A1-0xFEFE
546  * Code set 2:                                  unused
547  * Code set 3:                                  unused
548  */
549 int
```

```
towide_euckr(wchar_t *wc, const char *mb, unsigned n)
{
        return (towide_euc_impl(wc, mb, n, 0, 0, 0, 0));
}

/*
 * EUC-TW encodes as follows:
 *
 * Code set 0 (ASCII):                          0x21-0x7E
 * Code set 1 (CNS 11643-1992 Plane 1):         0xA1A1-0xFEFE
 * Code set 2 (CNS 11643-1992 Planes 1-16):     0x8EA1A1A1-0x8EB0FEFE
 * Code set 3:                                  unused
 */
int
towide_euctw(wchar_t *wc, const char *mb, unsigned n)
{
        return (towide_euc_impl(wc, mb, n, 0x8e, 4, 0, 0));
}

/*
 * Public entry points.
 */

int
to_wide(wchar_t *wc, const char *mb)
{
        /* this won't fail hard */
        return (_towide(wc, mb, strlen(mb)));
}

int
to_mb(char *mb, wchar_t wc)
{
        int     rv;

        if ((rv = _tomb(mb, wc)) < 0) {
                errf(widemsg);
                free(widemsg);
                widemsg = NULL;
        }
        return (rv);
}

char *
to_mb_string(const wchar_t *wcs)
{
        char    *mbs;
        char    *ptr;
        int     len;

        mbs = malloc((wcslen(wcs) * mb_cur_max) + 1);
        if (mbs == NULL) {
                errf("out of memory");
                return (NULL);
        }
        ptr = mbs;
        while (*wcs) {
                if ((len = to_mb(ptr, *wcs)) < 0) {
                        INTERR;
                        free(mbs);
                        return (NULL);
                }
                wcs++;
                ptr += len;
        }
        *ptr = 0;
```



```
        return (mbs);
}

void
set_wide_encoding(const char *encoding)
{
        int i;

        _towide = towide_none;
        _tomb = tomb_none;
        _encoding = "NONE";
        _nbits = 8;

        for (i = 0; mb_encodings[i].name; i++) {
                if (strcasecmp(encoding, mb_encodings[i].name) == 0) {
                        _towide = mb_encodings[i].towide;
                        _tomb = mb_encodings[i].tomb;
                        _encoding = mb_encodings[i].cname;
                        _nbits = mb_encodings[i].nbits;
                        break;
                }
        }
}

const char *
get_wide_encoding(void)
{
        return (_encoding);
}

int
max_wide(void)
{
        return ((int)((1U << _nbits) - 1));
}
```