Appendix A: EUC and Non-EUC Codesets

A.1. Extended UNIX Code (EUC)

Solaris EUC has adopted from USL's Multi-National Language Supplement (MNLS) and it follows the pattern of ISO 2022 standard, a.k.a., EUC multiple-width representation. EUC is used primarily for storing data in files.

EUC is composed of one primary codeset and three supplementary codesets. The primary codeset, codeset 0, is used for ASCII. The three supplementary codesets (codesets 1, 2, and 3) can be assigned to different character sets by the locale administrator.

The primary codeset is defined to use single bytes with the most significant bit (MSB) set to zero. The supplementary codesets can use multiple bytes, and the MSB of each byte is set to one. Codesets 2 and 3 have a preceding single-shift character, known as SS2 (0x8E) in codeset 2 and SS3 (0x8F) in codeset 3.

Differentiating between codesets is done as follows: If the MSB is 0, the code is one-byte ASCII. If the MSB is 1, the byte is checked (SS2 or SS3) to determine codeset. The length (in bytes) of characters from that codeset is retrieved from the LC_CTYPE locale database of the current locale.

Table 1: EUC codeset representations

Codeset	**EUC Representation**
codeset 0	0xxxxxxx
codeset 1	1xxxxxxxx -or- 1xxxxxxxx 1xxxxxxxx -or- 1xxxxxxxx 1xxxxxxxx 1xxxxxxxx
codeset 2	SS2 1xxxxxxx -or- SS2 1xxxxxxx 1xxxxxxx -or- SS2 1xxxxxxx 1xxxxxxx 1xxxxxxx
codeset 3	SS3 1xxxxxxx -or- SS3 1xxxxxxx 1xxxxxxx -or- SS3 1xxxxxxx 1xxxxxxx 1xxxxxxx

In accord with ISO 2022 and ISO 6937, EUC devides the codeset space into graphic and control characters. Graphic characters are those that can be displayed. Special characters include control characters, unassigned characters, and the space and delete characters. Control characters are characters other than graphic characters, whose occurrence may

initiate, modify, or stop a control operation.

Table 2: Single-byte special characters

Special Characters	EUC Representations
Space	00100000 (0x20)
Delete	01111111 (0x7F)
Control codes (Primary)	000xxxxx (0x00 ~ 0x1F)
Control codes (Supplementary)	100xxxxx (0x80 ~ 0x9F)

EUC is defined primarily for external data storage, and its encoding schemes provide reasonably compact representations for data storage. However, EUC is not very convenient for internal processing—its variable length nature complicates constructing homogeneous character arrays, for example. To assist convenient internal processing, Solaris provides a wide character format, plus a collection of library functions for operating on wide characters. Brief information on those functions and some additional functions for conversion between multi-byte characters and wide characters will be presented at section 4.

As specified at section 2, a wide character in Solaris is a four bytes entity and following table shows the relationship between EUC and wide character representations:

Table 3: An Example—EUC and Wide character representation

Codeset	EUC Representation	Wide Character representation
0	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
1	1xxxxxxx 1yyyyyyy	00110000 00000000 00xxxxxx xyyyyyyy
2	SS2 1xxxxxxx 1yyyyyyy 1zzzzzzz	00010000 000xxxxx xxyyyyyy yzzzzzzz
3	SS3 1xxxxxxx 1yyyyyyy 1zzzzzzz	00100000 000xxxxx xxyyyyyy yzzzzzzz

Far East Asian languages' EUC representations and encoding specifications in table form are like below:

Table 4: Japanese Solaris EUC representation and code ranges

Codeset	ECU Representation	Encoding
Codeset 0 (ASCII / JIS X 0201-1976 / JIS-Roman)	0xxxxxxx	Byte range: 0x21 ~ 0x7e
Codeset 1 (JIS X 0208-1990)	1xxxxxxx 1yyyyyyy	First byte: 0xA1 ~ 0xFE Second byte: 0xA1 ~ 0xFE

Table 4: Japanese Solaris EUC representation and code ranges

Codeset	ECU Representation	Encoding
Codeset 2 (JIS X 0201-1976 half-width Katakana)	SS2 1xxxxxxx	First byte: 0x8E Second byte: 0xA1 ~ 0xDF
Codeset 3 (JIS X 0212-1990)	SS3 1xxxxxxx 1yyyyyyy	First byte: 0x8F Second byte: 0xA1 ~ 0xFE Third byte: 0xA1 ~ 0xFE

JIS X 0201-1976 character set is consists of 94 JIS-Roman characters and 63 half-width Katakana characters. JIS-Roman is almost identical to that of USASCII except three character replacements: \ (backslash) to ¥ (yen sign), ~ (tilde) to ¯ (overbar), and ¦ (broken bar) to | (bar).

JIS X 0208-1990 character set is consists of 6,879 characters like miscellaneous symbols, Arabic numerals and Roman characters in full-width, Hiragana, Katakana, Greek and Cyrillic characters, line-drawing elements, 2,965 JIS level 1 Kanji, 3,384 JIS level 2 Kanji, and six additional Kanji characters.

JIS X 0212-1990 character set is a supplemental character set that contains 5,801 supplemental Kanji, diacritics and another set of miscellaneous symbols, Greek characters with diacritics, Eastern European characters, and miscellaneous alphabetic characters.

Both JIS X 0208-1990 and JIS 0212-1990 are composed of a 94x94 character space, respectively.

Table 5: Chinese / PRC Solaris EUC representation and code ranges

Codeset	EUC Representation	Encoding
Codeset 0 (ASCII / GB 1988-89)	0xxxxxxx	Byte range: 0x21 ~ 0x7F
Codeset 1 (GB 2312-80)	1xxxxxxx 1yyyyyyy	First byte: 0xA1 ~ 0xFE Second byte: 0xA1 ~ 0xFE
Codeset 2 - not used	-	-
Codeset 3 - not used	-	-

GB 1988-89 is the Chinese / PRC equivalent of USASCII except one character replacement: $ (dollar sign) to either Chinese / PRC currency symbol (¥) or international currency symbol (¤) depending on where it is used.

GB 2312-80 character set consists of total 7,445 characters like miscellaneous symbols,

Arabic and Greek numerals, Roman characters, Greek and Cyrillic characters, Hanzi Radicals, 3,755 level 1 Hanzi, and 3,008 level 2 Hanzi characters. This character set also using a 94x94 character space.

Table 6: Chinese / Taiwan Solaris EUC representation and code ranges

Codeset	**EUC Representation**	**Encoding**
Codeset 0 (ASCII / CNS 11643-0)	0xxxxxxx	Byte range: 0x21 ~ 0x7F
Codeset 1 (CNS 11643-1992 plane 1)	1xxxxxxx 1yyyyyyy	First byte: 0xA1 ~ 0xFE Second byte: 0xA1 ~ 0xFE
Codeset 2 (CNS 11643-1992 plane 2 ~ 7, 15, 16)	SS2 1xxxxxxx 1yyyyyyy 1zzzzzzz	First byte: 0x8E Second byte: 0xA2 ~ 0xB0 Third byte: 0xA1 ~ 0xFE Fourth byte: 0xA1 ~ 0xFE
Codeset 3 - not used	-	-

Currently Chinese / Taiwan Solaris is defining CNS 11643-0 as a USASCII-equivalent character set for codeset 0 and there seems no difference between USASCII and CNS 11643-0.

CNS 11643-1992 defines total 16 planes as like below and the second byte of codeset 2 specifies plane number of a code, i.e., plane number = the second byte - 0xA0:

- Plane 1:
  Miscellaneous symbols, Hanzi radicals, and Roman and Greek alphabets, total 684 symbol characters in range of 0x2121 to 0x427E and 5,401 most commonly used Hanzi characters in range of 0x4421 to 0x7D4B.
- Plane 2:
  7,650 secondary commonly used Hanzi characters in range of 0x2121 to 0x7244.
- Plane 3:
  Total 6,148 characters of other Hanzi characters including user defined characters from some of original plane 14 characters and some of Republic of China (ROC) Department of Education defined different shape characters (range: 0x2121 ~ 0x6246).
- Plane 4:
  This plane has total 7,298 characters including some of ISO / IEC 10646 defined CJK Unified Han characters. (range: 0x2121 ~ 0x6E5C).
- Plane 5:
  This plane has total 8,603 characters that ROC Department of Education defined as currently used characters but not included in plane 1 to 4 (range: 0x2121 ~ 0x7C51).
- Plane 6:
  This plane has total 6,388 characters that ROC Department of Education defined

as different shape characters but not included in plane 1 to 5 (range: 0x2121 ~ 0x647A).

- Plane 7:
  This plane has total 6,539 characters that ROC Department of Education defined as different shape characters but not included in plane 1 to 6 (range: 0x2121 ~ 0x6655).
- Plane 8 to plane 11: These planes are not yet defined.
- Plane 12 to plane 16: These planes are for user defined characters.

Each of above planes in CNS 11643-1992 character set is using a 94x94 character space.

Table 7: Korean Solaris EUC representations and code ranges

Codeset	**EUC Representation**	**Encoding**
Codeset 0 (ASCII / KS C 5636)	0xxxxxxx	Byte range: 0x21 ~ 0xFE
Codeset 1 (KS C 5601-1992)	1xxxxxxx 1yyyyyyy	First byte range: 0xA1 ~ 0xFE Second byte range: 0xA1 ~ 0xFE
Codeset 2 - not used	-	-
Codeset 3 - not used	-	-

KS C 5636 character set is a Korean version of ASCII that only difference is there is no backslash (\) but Korean currency sign, won (₩).

KS C 5601-1992 character set is using 94x94 character space as like other Asian character sets and has following characters: 432 miscellaneous symbols, 30 Arabic and Roman numerals, 94 Korean Jamo (alphabet) characters, Roman, Greek, Latin, Japanese (both Hiragana and Katakana), and Cyrillic characters (total 362 characters), 68 line drawing characters, pre-composed 2350 Hangul characters, more correctly, syllables, and 4,888 Hanja characters.

A.2. Non-EUC Codesets

To fulfill their own needs, besides EUC-based codesets, Far East Asian countries devised more than one codesets which are not based on EUC encoding. Among them, following codesets are the ones that sometimes more dominant than EUC-based codesets:

- Japan: Shift-JIS
- Korea: Johap
- Taiwan: Big5
- Universal Multiple-Octet Coded Character Set (UCS): UCS-2 / 4 and UTF-8

In this subsection, we will describe above codesets and their representations in Solaris.

Shift-JIS, a.k.a., MS-Kanji, is a name of encoding developed by Microsoft Corp. and is widely implemented and used by various platforms and customers in Japan.

Originally, Japanese PC users used only half-width Katakana and later there were needs for using not only half-width Katakana characters but also Kanji characters and others. To support those needs and not break backward compatibility, Microsoft devised Shift-JIS (SJIS) by defining those additional (double-byte) characters not using the original half-width Katakana characters' code range, i.e., 0xA1 ~ 0xDF, but *shifted* around the code range like table at below:

Table 8: SJIS code ranges including ASCII

Characters	**Range**
ASCII / JIS-Roman	0x21 ~ 0x7E
Half-width Katakana	0xA1 ~ 0xDF
Double-byte characters: First byte: Second byte: Double-byte user-defined characters: First byte: Second byte:	 0x81 ~ 0x9F, 0xE0 ~ 0xEF 0x40 ~ 0x7E, 0x80 ~ 0xFC 0xF0 ~ 0xFC 0x40 ~ 0x7E, 0x80 ~ 0xFC

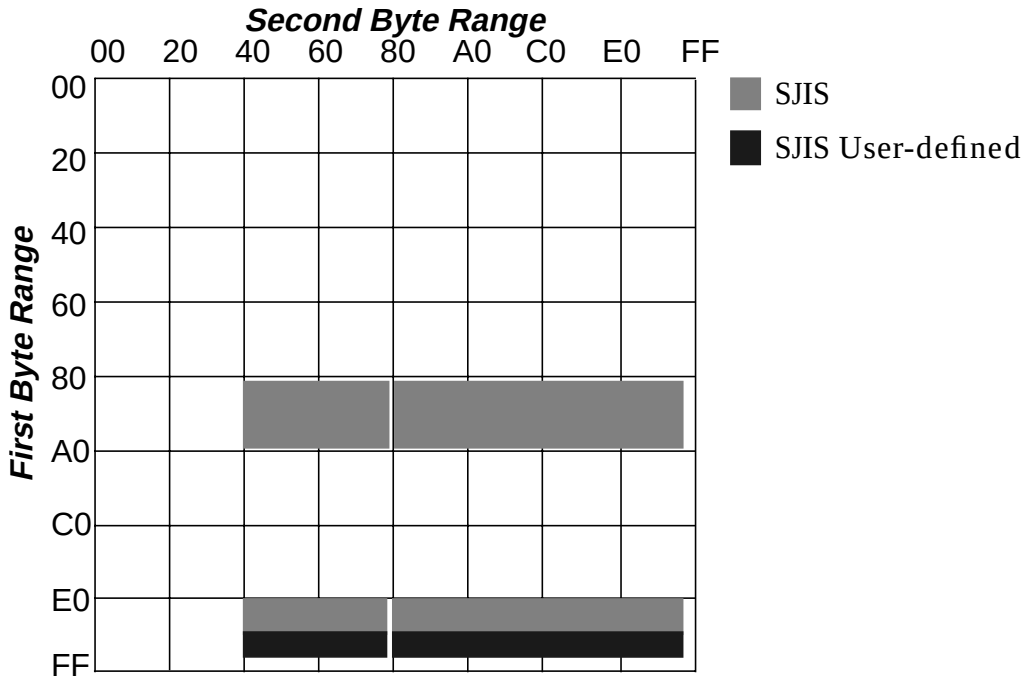

Figure A.1: Double-byte SJIS code range

† These are locale-specific representation translation APIs between multi-byte and wide character. These APIs are defined at *mb.so* described at section 3.

This layout can be depicted as like Figure A.1, i.e., double-byte characters' areas. And if we look at the double-byte code ranges, there is no guarantee that the MSB of the second byte of a double-byte code will be one which means we cannot directly use SJIS in our EUC-based platform. However, the first byte of the double-byte code will always have MSB set to one.

Japanese Solaris 2.5 CSI_EA (Code Set Independence Early Access) version defines SJIS codesets like below (this is subject to change at next release):

Table 9: Japanese Solaris / CSI SJIS Representation

Codeset	**SJIS Representation**	**Wide Character Representation**
Codeset 0 (ASCII / JIS-Roman)	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
Codeset 1 (Double-byte characters)	1xxxxxxx yyyyyyyy	00110000 00000000 00*xxxxxx xyyyyyyy*
Codeset 2 (Half-width Katakana)	1xxxxxxx	00010000 00000000 00000000 0xxxxxxx
Codeset 3 - not used	-	-

Internally, the wide character representations of SJIS codesets are identical to that of Japanese EUC's wide character representations. The codeset 0 and codeset 2 are identical to that of EUC codeset 0 and codeset 2, respectively, in terms of both multi-byte representation and wide character representation. And, in case of codeset 1, there are internal code conversions (which is very light-weight computation) like following figure:

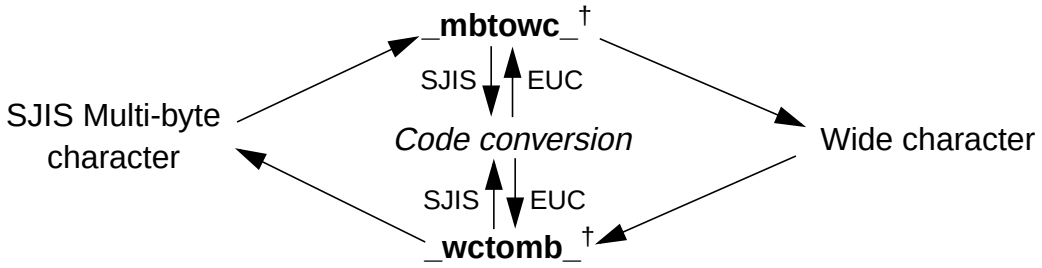

Figure A.2: SJIS multi-byte representation and wide character representation

so that wide character representation, i.e., 00110000 00000000 00*xxxxxx xyyyyyyy,* will be identical to that of EUC. The reason why the conversion is possible is because SJIS character sets is a subset of EUC character sets.

Differentiating between codesets is done by comparing each byte values: if the byte has MSB set to zero, it belongs to codeset 0. If not and if it is in range of from 0xA1 to 0xDF, it

belongs to codeset 2. Otherwise, it is a double-byte character and belongs to codeset 1.

SJIS is protecting null byte and ASCII slash (/, 0x2F) characters and hence it is UNIX file system safe.

Johap, a.k.a., Combination, is a name of encoding which is widely used in Korean PC environment and is developed by a few Korean computer vendors initially and now it is included in KS C 5601-1992 as a supplementary code (KS C 5601-1992 Annex 3).

Unlike KS C 5601-1992 pre-composed Hangul, Johap code allows user composing each syllable dynamically by dividing a two byte length character code into three bit fields for different possible components, i.e., an initial consonant, a vowel, and an optional ending consonant, of a Korean syllable like Figure A.3 at below:

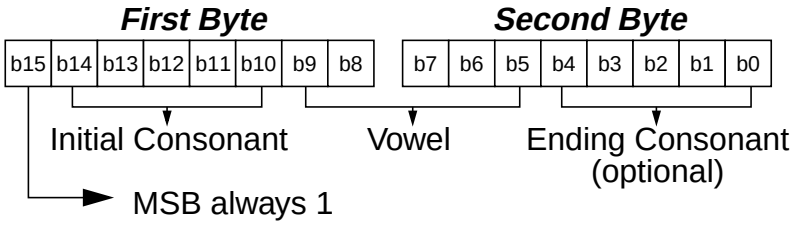

Figure A.3: Johap multi-byte representation—Hangul syllable portion

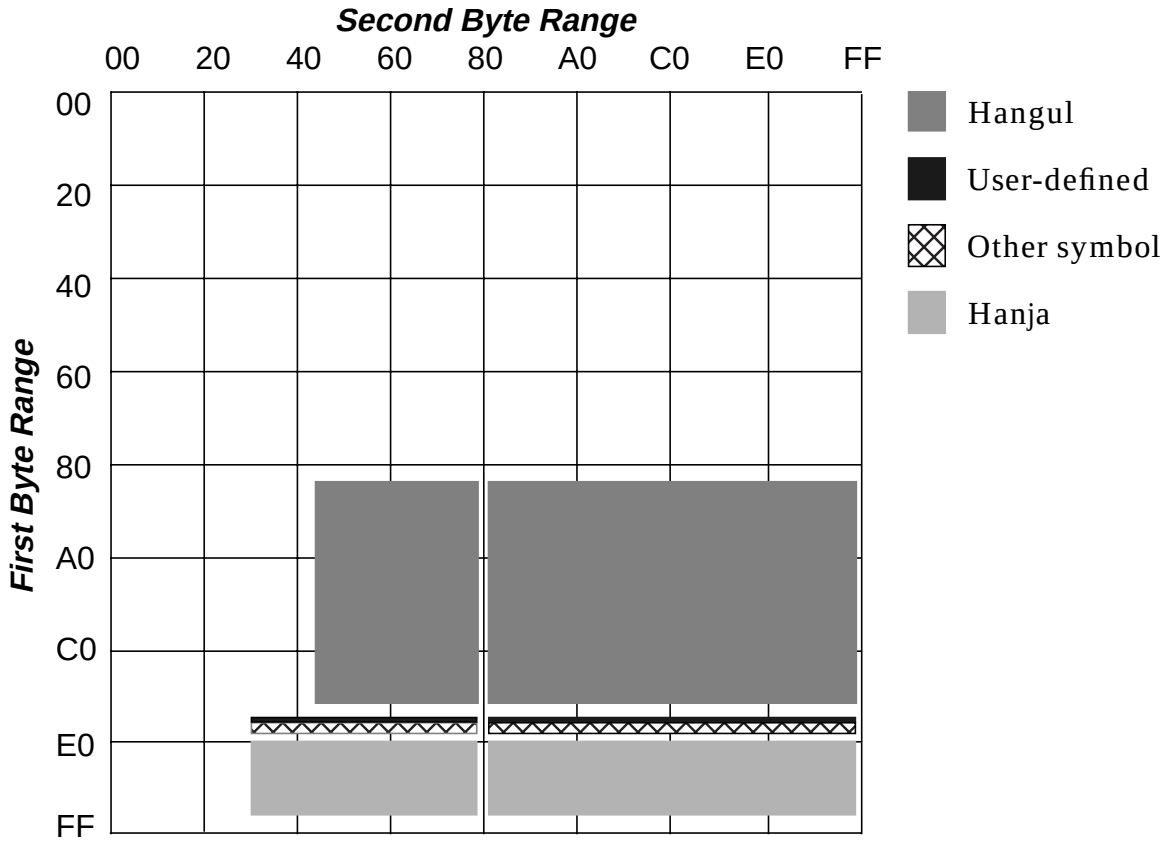

Figure A.4: Two-byte Johap code range

Since the maximum number of possible Modern Hangul initial consonants, vowels, and optional ending consonants are 19, 21, and 27, respectively, such assignment of five bits per each component (Jamo in Korean) is big enough as a result to represent all of Modern Hangul syllables (11,172 syllables).

In addition to the Hangul syllables, there are areas that other symbol and Hanja characters can be represented in this code like following table:

Table 10: Johap code ranges including ASCII

Characters	**Range**
ASCII / KS C 5636:	0x21 ~ 0x7F
Hangul: First byte: Second byte:	 0x84 ~ 0xD3 0x41 ~ 0x7E, 0x81 ~ 0xFE
User defined: First byte: Second byte:	 0xD8 0x31 ~ 0x7E, 0x91 ~ 0xFE
Miscellaneous symbol: First byte: Second byte:	 0xD9 ~ 0xDE 0x31 ~ 0x7E, 0x91 ~ 0xFE
Hanja: First byte: Second byte:	 0xE0 ~ 0xF9 0x31 ~ 0x7E, 0x91 ~ 0xFE

This layout also can be depicted as like Figure A.4, i.e., two-byte characters' areas. As like SJIS, Johap also doesn't guarantee that the MSB of the second byte of a two-byte code will be one which again means we cannot directly assign Johap codes into EUC codesets. However, the MSB of the first byte will always be one. Since every bit of a character code is significant, we define Johap multi-byte character and its corresponding wide character representation as like table at below:

Table 11: Korean Johap multi-byte and wide character representations

Codeset	**Johap Representation**	**Wide Character Representation**
Codeset 0 (ASCII / KS C 5636)	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
Codeset 1 (Two-byte characters)	xxxxxxxx yyyyyyyy	00110000 00000000 xxxxxxxx yyyyyyyy
Codeset 2 - not used	-	-
Codeset 3 - not used	-	-

The main reason why not saving one bit is for a slightly better translation performance, i.e., having one more if expression in a loop or not.

Differentiating between codeset 0 and codeset 1 is done by initially checking the MSB of a byte. If the byte has MSB set to zero, it belongs to codeset 0. If not, it belongs to codeset 1.

Since Johap has all the characters of EUC character sets and has more Hangul syllables, i.e., 8,642 more syllables, it is a superset. Johap also protecting null and ASCII slash (/) characters and hence it is UNIX file system safe.

Big5 is a name of encoding which is widely used in Taiwan and it has named since it has defined by five major Taiwan computer vendors in May 1984 including Institute of Information Industry (III). Even though it is not a Chinese National Standard (CNS) but a de facto standard as like SJIS of Japan and Johap of Korea, it is used much more widely than the CNS 11643-1992.

The total number of characters defined in Big5 is 13,523 characters and depending on an implementation, the number can be either increased or decreased.

The code ranges of Big5 is like below table:

Table 12: Big5 code ranges including ASCII

Characters	**Range**
ASCII	0x21 ~ 0x7F
Miscellaneous symbol: First byte: Second byte:	 0xA1 ~ 0xA3 0x40 ~ 0x7E, 0xA1 ~ 0xFE
Level 1 Hanzi: First byte: Second byte:	 0xA4 ~ 0xC2 0x40 ~ 0x7E, 0xA1 ~ 0xFE
Level 2 Hanzi: First byte: Second byte:	 0xC9 ~ 0xF9 0x40 ~ 0x7E, 0xA1 ~ 0xFE

The Table 12 can be depicted as like following figure, i.e., the areas of two-byte code

areas:

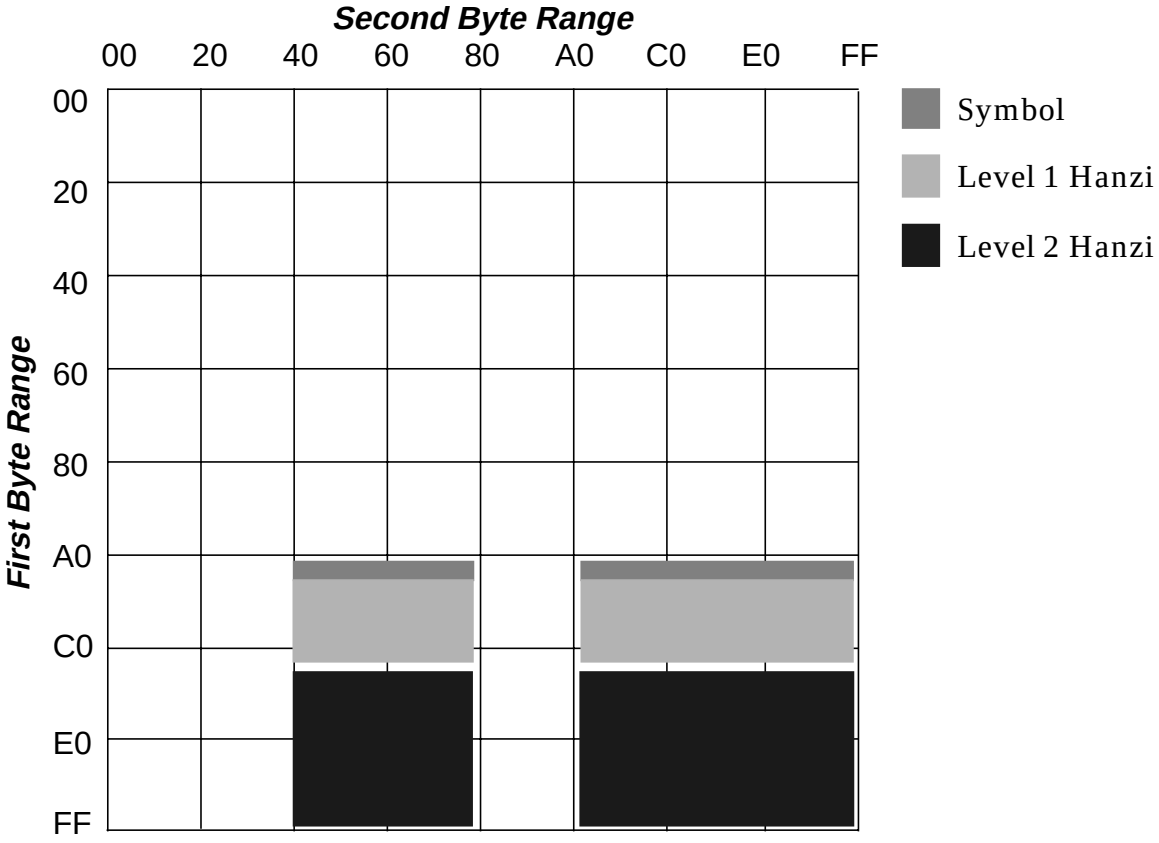

Figure A.5: Two-byte Big5 code range

As like other non-EUC encodings, Big5 doesn't guarantee the MSB of the second byte se to one. However, MSB of the first byte is always one.

Since every bit of a code is significant and for the performance reason as like Johap, we could define Big5 wide character representation as like below table:

Table 13: Taiwan Big5 multi-byte and wide character representations

Codeset	**Big5 Representation**	**Wide Character Representation**
Codeset 0 (ASCII)	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
Codeset 1 (Two-byte characters)	xxxxxxxx yyyyyyyy	00110000 00000000 xxxxxxxx yyyyyyyy
Codeset 2 - not used	-	-
Codeset 3 - not used	-	-

As like Johap multi-byte representation, differentiating between codeset 0 and codeset 1 is done by checking the MSB of a byte. If the byte has MSB set to zero, it belongs to

codeset 0 and if not, it belongs to codeset 1 and the next byte will be a part of a code.

Even though Big5 has more than thirteen thousand characters, it is still a subset of CNS 11643-1992 character set. In real implementation, since Big5 character set is a subset of CNS 11643-1992, it could realized as like SJIS, i.e., have code conversions before the wide character translation of multi-byte character and after the reverse-translation from wide character.

Big5 also protecting null and ASCII slash (/) characters and hence it is UNIX file system safe.

ISO/IEC 10646-1:1993 (**10646**) defines **UCS** and the goal in creating was to include all characters from all significant languages: to be a universal multi-octet coded character set. People often use "10646" and "Unicode" interchangeably, although there are differences between the two sets.

10646 differs in some ways from codesets described in this section since, unlike others, 10646 doesn't include *portable characters* as single-octet entities and that is because 10646 is encoded in multiple octets. Code space is organized into 128 groups of 256 planes each.

10646 allows two basic forms for characters:

- Universal Coded Character Set-2 (UCS-2):
  Also known as Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell).
- Universal Coded Character Set-4 (UCS-4):
  Characters are encoded in the full four octets.

In addition to the UCS-2 and UCS-4 forms, 10646 also includes an encoding technique in which multiple characters can be combined to form *composite sequences*. These are already present in other standards, e.g., ISO 6937, Thai standard TIS 620, Indian standard 10194, Arabic National Standard ASMO 449+, and are designed to allow a nearly infinite variety of character combinations. For instance, á (lower case a with acute accent) can be encoded as the plain **a** followed by ´ (an acute accent) even though 10646 has the letter-with-diacric:

Character:	a	´	á
UCS-2 code value:	0x00 0x61	0x03 0x01	0x00 0xE1

Figure A.7: Two **á** encodings

The resulting composite sequence consumes four octets; that is, two for the **a** and two for the ´. In 10646, certain characters are defined as *combining diacritical marks*, and it is permissible to combine these marks with any non-combining character and also any number of combining marks can follow a base character. There are also some languages

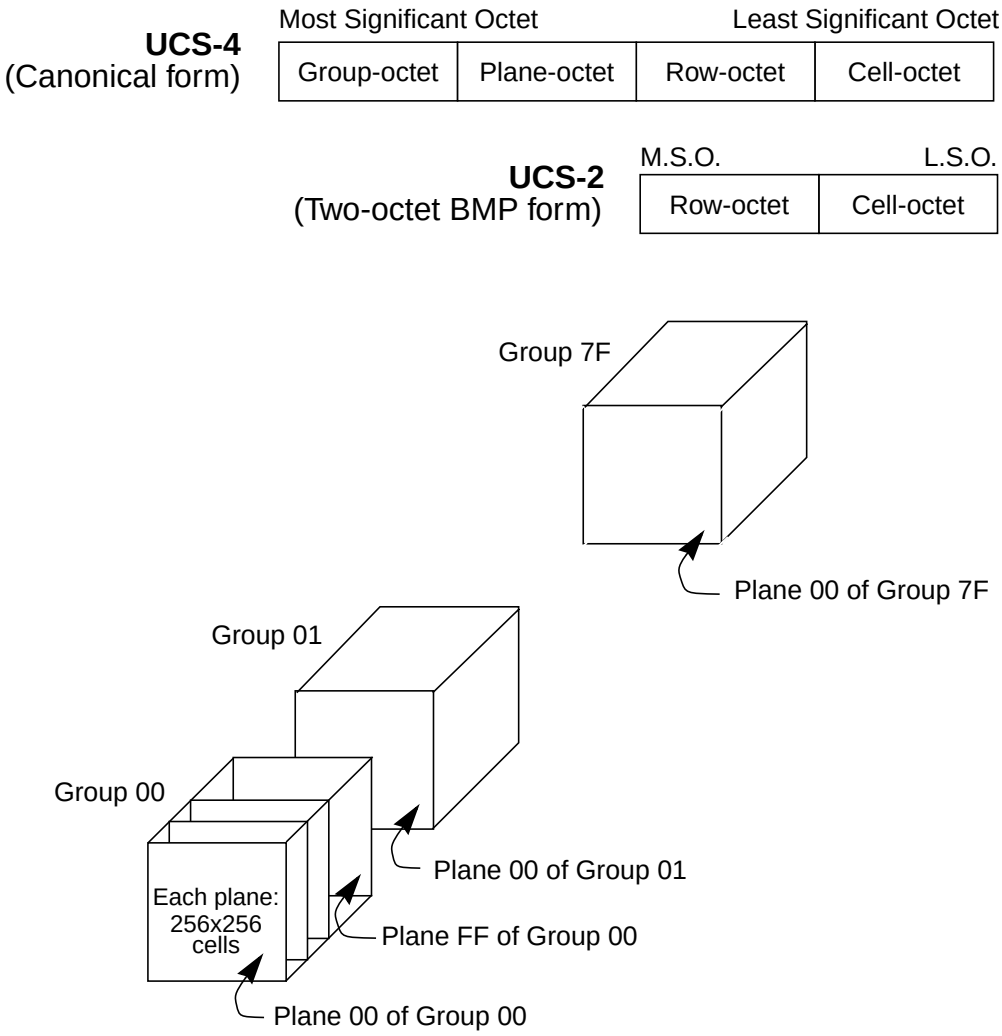

Figure A.6: UCS-2, UCS-4, and entire coding space

which are fully(?) supported by 10646 through the use of such combining characters, e.g., Thai, Arabic, and Hangul.

Although combining characters give 10646 great deal of flexibility, they also create programming challenges that do not exist in many commonly used codesets. Because not all suppliers want to revise software to handle composite character sequences, 10646 has three conformance levels:

- Level 1: Combining characters are not allowed.
- Level 2: Combining characters are allowed for Arabic, Hebrew, Indic, and Thai scripts only.
- Level 3: Combining characters are allowed with no restrictions.

Thus, with 10646, it is possible for an implementation to support one or more of the

following:

- UCS-2, Level 1: Two-octet form, no combining characters
- UCS-2, Level 2: Two-octet form, combining characters allowed with restrictions
- UCS-2, Level 3: Two-octet form, combining characters allowed with no restrictions.
- UCS-4, Level 1: Four-octet form, no combining characters
- UCS-4, Level 2: Four-octet form, combining characters allowed with restrictions
- UCS-4, Level 3: Four-octet form, combining characters allowed with no restrictions.

Unicode 1.1 is only equivalent to UCS-2, Level 3.

In addition to the normative forms of 10646, one of the standard's informative annexes defines another form called UTF (UCS Transformation Format), i.e., UTF-1 and UTF-8 at Annex G and P, respectively.

UTF-1 is defined so that the form does not use octet values specified in ISO 2022 as coded representations of C0, SPACE, DEL, or C1 characters and can thus be used for transmitting text data through communication systems that are sensitive to these octet values. However, since UTF-1 does not restrict octets from having the same value of ASCII slash (/, code value of 0x2F) character, the UTF-1 cannot directly usable as an encoding for file names on most of UNIX systems including current Solaris kernel, i.e., not file system safe.

Because of this and other limitations with UTF-1, XoJIG WG created a second transformation format called UTF-8, i.e., also formerly known as UTF-2 and FSS-UTF (File System Safe-UTF). In this version, the MSB of an ISO / IEC 646 IRV character is 0; the MSB for all octets of all other character is 1 as like following table:

Table 14: Mapping between UCS-* and UTF-8

Bits	Hex Min	Hex Max	UTF-8 Binary Encoding
7	00000000	0000007F	0xxxxxxx
11	00000080	000007FF	110xxxxx 10xxxxxx
16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
21	00010000	001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
26	00200000	03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
31	04000000	7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Although above mapping shows UTF-8 characters being up to six octets long, the current 10646 only defines BMP which is only exploring two octets and so the maximum number of UTF-8 binary encoding will be three octets. XoJIG I18N WG and Unicode, Inc. defined UCS and UTF-8 as wide character and multi-byte character encodings, respectively, and Sun's prototype of UTF locale will, most likely, base on the definitions

and there will be a single or two codesets.

Another aspect of 10646 is that the standard allows *subsetting* of coded graphic characters: *limited subset* and *selected subset*. An adopted subset may comprise either of them or a combination of the two. The limited subset consists of a list of graphic characters in the specified subset and a claim of conformance referring to a limited subset shall list the graphic characters in the subset by the names of graphic characters or code positions as defined in 10646. A selected subset consists of a list of collections of graphic characters as defined in annex A of 10646. A selected subset shall always automatically include the cells 0x00000020 to 0x0000007E. A claim of conformance referring to a selected subset shall list the collections chosen as defined in 10646.

The UCS-2, UCS-4, and UTF-1 are not UNIX file system safe since they do not protect null bytes and / or the ASCII slash characters.

Appendix B: Worldwide Portability Interfaces (WPI)

ISO C defines a minimal set of interfaces for manipulating wide-character codes, basically, allowing conversion to and from multi-byte character sequences, i.e., mbchar(3C) APIs. XPG4 extends this set of interfaces to allow wide characters to be used wherever single-byte characters can be used in the standard C model. The additional interfaces are known as WPI, i.e., functions for wide-character string manipulation, collation, conversion, printing, scanning, I/O operations, and so on. Also, some I/O functions like scanf() and printf() has been extended to handle wide-characters and wide-character strings in their argument lists.

By using these WPI, following advantages can be achieved:

- Efficiency
  Internal processing codes can be manipulated as integral values of type **wchar_t**.
- Portability
  Applications/utilities written by using and exploiting wide-character representation are more portable across platforms and locales.
- Codeset independence
  SunSoft's CSI relies on application/utilities using wide-character representation internally not multi-byte character representation.

XPG4 and hence Solaris 2.4 and after-defined WPI are like below:

- getwc(3I):
  getwc(), getwchar(), fgetwc()
- getws(3I):
  getws(), fgetws()
- putwc(3I):
  putwc(), putwchar(), fputwc()
- putws(3I):
  putws(), fputws()
- iswalpha(3I):
  iswalpha(), iswupper(), iswlower(), iswdigit(), iswxdigit(), iswalnum(), iswspace(), iswpunct(), iswprint(), iswcntrl(), iswascii(), iswgraph()
  (Followings are defined by Solaris only in this group: isphonogram(), isideogram(), isenglish(), isnumber(), isspecial().)
- iswctype(3I):
  iswctype()
- mbchar(3C):
  mbtowc(), mblen(), wctomb()
- mbstring(3C):
  mbstowcs(), wcstombs()
- wconv(3I):
  towlower(), towupper()
- ungetwc(3I):
  ungetwc()
- wcstring(3I):

wcscat(), wscat(), wcsncat(), wsncat(), wcscmp(), wscmp(), wcsncmp(), wsncmp(), wcscpy(), wscpy(), wcsncpy(), wsncpy(), wcslen(), wslen(), wcwidth(), wcswidth(), wcschr(), wschr(), wcsrchr(), wsrchr(), windex(), wrindex(), wcspbrk(), wspbrk(), wcswcs(), wcsspn(), wsspn(), wcscspn(), wscspn(), wcstok(), wstok()

- wctype(3I):
  wctype()
- wcscoll(3I):
  wcsoll(), wscoll()
- wcsxfrm(3I):
  wcsxfrm(), wsxfrm()
- wcsftime(3I):
  wcsftime()
- wcstol(3I): (Solaris only functions.)
  wcstol(), wstol(), watol(), watoll(), watoi()
- wcstod(3I): (Solaris only functions.)
  wcstod(), wstod(), watof()

All the interfaces defined in man section 3I are defined at libw and required to link with the library. The interfaces in iswalpha(3I), mbchar(3C), mbstring(3C), and wconv(3I) are MT-safe as long as setlocale(3C) is not being called to change the locale. Others are MT-safe with no exception.