International Character Code Standard for the BE2

June 18, 1987

Tomas Centerlind

Information Technology Center (ITC)
Carnegie Mellon University

1. Major problems with foreign languages

All European languages have a set of unique characters, even Great Britain with their Pound sign. Most of these characters are static and do not change if they are in the end or in the middle of the word. The Greek sigma sign however is an example of a character that changes look depending on the position. If we move on to the non-Roman alphabets like Arabic, they have a more complex structure. A basic rule is that certain of the characters are written together if they follow each other but not otherwise. This complicates representation and requires look ahead. In addition to this many of the languages have leftwards or downwards writing. All together these properties makes it very difficult to integrate them with Roman languages.

Lots of guidelines have to be established to solve these problems, and before any coding can be done a set of standards must be selected or defined. In this paper I intend to gather all that I can think of that must be considered before selecting standards. Only the basic level of the implementation will be designed, so therefore routines that for example display complex languages like Arabic must be written by the user. A basic method that dislpays Roman script correctly will be supported.

1. Standards

1.1 Existing standards

Following is a list of currently existing and used standards.

1.1.1 ASCII, ISO 646

The ASCII standard, that is in use almost anywhere, will probably have to remain as a basic part of the system, and without any doubt it must be possible to read and write ASCII coded documents in the foreseeable future.

1.1.2 Extended ASCII

A standard that has been adopted by some micro computer companies is the Extended-ASCII, which among others Apple use in the Macintosh. In contrast to ASCII they use the eighth "parity" bit and therefore gain another 128 characters. This will handle all European languages but not non-Roman alphabets. All additional characters are placed in the upper 128 characters so the lower segment is equivalent with standard ASCII. None of the international standardization institutes have defined such a standard, so the layout of the character codes is often company specific.

1.1.3 ISO 2022

ISO 2022 is a standard that also can encode all European languages. It can either be a 7- or a 8-bit code. It is a extension of ASCII, and provides the possibility to exchange information between 7-bit and 8-bit environments.

If **7-bit** code is used, the SI and SO control characters are used to invoke an additional set of 94 graphics. The control characters DEL and SPACE are not affected by SI and SO. More than two graphic sets may be reached by using an escape sequence to select another set of graphics is then invoked with SO. This leads to a virtually unlimited number of graphic sets that may be used. With an escape sequence you may also change the control character representation.

The **8-bit** representation uses the high-order 128 combinations as well as the low-order. This gives the following advantages:

1) a set of 32 additional control characters

2) a set of 94 additional graphic characters

3) no need to use SO and SI, both sets are available all the time

Escape sequences are used to change graphic sets in the same manner as for 7-bit codes. To be able to know if 7- or 8-bit coded data is coming, there are special escape sequences that should precede a character stream. This tell the software how to interpret the characters.

This seem to be quite easily installed into existing software, but it has some major effects. First of all it will be difficult to sort text that is stored with this standard, secondly it will be hard to produce the output if you don't have stored somewhere the look of each character possible to generate. The advantage is that if it is displayed on a computer that only can generate normal ASCII, the output will still be readable. There also exist standardized routines to convert between 7- and 8-bit codes. ISO 2022 does not deal with what characters should be placed in the new areas.

1.1.4 Xerox XC1-2-2-0

Xerox has made further extensions to ISO 2022 and is now using it throughout their line of work stations. This standard is spanning the following other international standards:

- All ISO 646 IRV graphic characters
- All ISO 5426 graphic characters
- All ISO 5428 graphic characters
- All ISO 6937 graphic characters
- All ANSI 7-bit ASCII graphic characters
- All CCITT 8-bit Teletex "G0" & "G2" graphic characters
- All Xerox 860 graphic characters
- All EBCDIC 8-bit graphic characters
- All JIS C 6226 graphic characters (including 6,249 most-frequent Japanese *kanji*)
- All characters required to write the following languages:
  English, Russian, German, French, Spanish, Italian, Portuguese, Dutch, Swedish, Norwegian, Danish, Japanese, Malay/Indonesian, Greek, Ukrainan, Polish, and other languages.
- All standard office typewriter keyboard characters for the European languages above
- The most commonly-used office, technical, and general symbols

Xerox uses 8-bit encoding for characters which eliminates the need to use a shift-code to reach the high order 128 characters. Like ISO 2022 they store the diacritics before the character they should affect. The problem of what an accented character should look like has been solved through a *Rendering Set* where each legal combination of diacritics and letters is assigned a unique number. The *Rendering Set* uses most of the 8-bit codes. This implies that two additional character sets are needed, one for the 128 unshifted characters and one for the shifted.

To allocate space for the font as well they have chosen to use 16-bit encoding for each character when processing and a 8-bit coding when on mass storage. This will give a coding that is easy to manipulate and display even though it is slightly more memory consuming when not on mass storage. The high-order byte is the character-set code and the low-order byte is the ISO character code. For example character-set 0_8 is the Latin alphabet, and 45_8 is Japanese hiragana. The rendering character codes are stored in character sets 360_8 and 361_8.

This standard does not define the lexigraphical order of the characters within a specific alphabet. This has to be done within the user application.

1.1.5 Xerox 2

This is not a standard but merely a proposal of a way to store multilingual text. It assumes that it is possible to entirely change the internal representation of characters. Basically you use a 8-bit code where 1111 1111 is reserved to indicate that the next byte should be treated as a character set code. This mean that when you encounter this byte a new character set will be loaded and all

output will be interpreted for this character set in a appropriate way. If two 1111 1111 bytes follow each other this indicate that 16-bit words will be used instead. This gives enough space for languages like Chinese and Japanese.

This scheme makes it possible to have up to 255 different scripts at the same time in a text. Since there is one character set for each alphabet, the lexigraphical order of the letters is easily maintained. It will also be quite easy to generate the output. However a large disadvantage is that a specific font table is needed for each font and script.

1.1.6 Apple Script Manager

Apple's *Script manager* extends the standard character set to provide up to 64 different scripts at the same time (all Roman scripts are already gathered in *Apple Extended ASCII* so they represent one script). Each language is given a number through which it is called. No further details about the internal representation are given in the document I have received. However it seems that they use 16-bit words to store each character, and different scripts have received different areas of the large space of 65,536 symbols. They support leftwards writing and combinations of left and right writing languages on the same line are allowed.

1.2 Requirements

Following is a wish list of features that should be available in BE2.

1.2.1 Character Set requirements

There are basically six aspects that must be fulfilled to have a powerful character code standard:

- We need an efficient and general way to store the characters. Here three characteristics have to be balanced: space efficient storing, ease of processing the characters without a lot of overhead due to decoding, and finally should the encoding cover as many characters as possible.

- Routines that can display the entire character set in an appropriate way should be possible to write.

- The lexigraphical order for each alphabet must be easily accessible.

- The font of each character must be prestored somewhere to allow fast displaying.

- Compability with other Character Standards must exist for interactive reasons.

- An easy way to insert the characters through the keyboard.

1.2.2 Other requirements for a full fledged text insertion environment

It must be possible to detect every keypress to gain full control over the keyboard, including all function keys, the shift keys, the alternate keys etc. This should be possible at the programmer's level.

2 . A 7-bit coded extended character set in BE2 using ISO 2022

This proposal is based on the fact that the following conditions must be met:

- Only 7-bit codes can be used, mainly because of communication problems and the mail interface.

- Within the BE2 text data object each character displayed on the screen may only take up one byte in memory. Other information must be stored elsewhere. Otherwise large structural changes have to be made to the text drawing routines.

2.1 ISO 2022 and Xerox Character Code Standard

The mapping of graphics from Xerox Character Code Standard seems to be a good standard to use. It consists of 256 tables of 256 characters of which about half is used. This will at least cover the following languages:

- All Latin languages.
- Arabic, Hebrean, Greek, Korean, Russian.
- Japanese: JIS, Kanji, Gaiji.
- Chinese.
- A set of general and technical symbols.

In addition to this there is a lot of space left for new languages and additional graphics. So far Xerox is the only Standard that I have seen which at least uses parts of ISO 2022 to encode this many different languages. The *ASCII/ISO/CCITT Roman alphabet with extension for punctuation* only defines one of the 126 possible character sets.

The following scheme should according to ISO 2022 be used to change character set. Since 7-bit coding will be used, only half of the 256 graphics each character set consists of may be accessed at the same time. Therefore the SO and SI control characters will be used to switch between the G0 and G1 character sets (where G0 is the lower 128 characters and G1 is the 128 higher) in the following way:

SO	Change to G1 character set
SI	Change to G0 character set

To change to another character set, escape sequences have to be used. The ISO 2022 ones only address 126 (2*63) different sets which are invoked by the following sequences:

ESC 2/8 4/1	to ESC 2/8 7/14	Change character set G0
ESC 2/12 4/1	to ESC 2/12 7/14	Change character set G0
ESC 2/9 4/1	to ESC 2/9 7/14	Change character set G1
ESC 2/13 4/1	to ESC 2/13 7/14	Change character set G1

This will not do if you want to encode 256 different character sets as Xerox has done. Instead I propose that the following sequence be used:

ESC 2/6 3/1 4/1 to ESC 2/6 3/15 4/15

This will give 16*16 different character sets. I assume that both the G0 and G1 character sets are loaded at the same time. The ESC 2/6 code is not used by ISO 2022. The two last characters will point out one of the 256 character sets.

So for example to print the *clubs* sign which is in character set 357_8, first an escape followed by character 2/8 should appear. This will imply that the next two characters indicate what character set should be selected. These characters should be 3/14 and 3/15. Then a SO must follow so the G1 character set is selected. Finally in the line comes 116_8, which is the code of the *clubs* sign within character set 357_8.

For most of the character sets only a few of different faces are needed such as different sizes and bold. Stylistic aspects such as italics and various looks do not apply to character sets like the *general and technical symbols*. What the situation is for Chinese and Japanese is not clear, but I guess you can be pretty glad if you manage to design a fairly small font that looks good.

2.2 Text input

There are basically two different ways to insert the characters. Either you use some sort of escape or control sequences, or you totally remap the keyboard. For smaller amounts of input a small

keyboard chart on the screen may be used.

What is to be preferred is basically a question of the kind of usage. If Andrew is going to be used successful in a non-English country a possibility to remap the keyboard is absolutely a requirement. Usually a different set of key tops are present on the keyboard, and these must be scanned correctly. In the case of adapting ASCII to foreign characters simply a few of them like [, { and | are replaced with the national ones. This makes it easy to do this kind of modification. With ISO 2022 the entire character set has to be changed in order to reach non-ASCII characters. This must be done at a higher level in the software. It is therefore desirable to make it possible to change the entire mapping of the keyboard easily.

This could lead to some nice side effects. A library of different keyboard mappings could be built, where the standardized mapping for each country can be found and displayed on the screen. A mapping should be easy to edit, and it should be possible to store personal ones. Ways to change the keyboard via control sequences also have to considered. Each window on the screen should be allowed to have its own mapping of the keyboard.

2.3 Design

Following is a proposal of a design that would make it possible to use a wide variety of languages without losing the support for all existing ASCII dependent software:

Use ISO 2022 7-bit coding.

Exploit Xerox Character Set encoding standard.

Make it possible to reconfigure the keyboard so that different languages can be typed easily.

BE2 routines:

Display routines. (*modifiy*)

Extended font support. (*modifiy*)

Facility in the software to turn input of the extended character set on and off. (*new*)

Facility to restrict the output to ASCII characters. (*new*)

Routine to remove all non ASCII characters form a string. (*new*)

Routines to convert a ASCII string written in a certain language into a ISO 2022 string. (*new*)

Routines to convert between 7- and 8-bit coded ISO 2022 strings. (*new*)

Sorting routines for each language. (*new*)

Routines to load and alter keyboard mappings. (*new*)

Posibility to display the current keyboard mapping, and do input from it by pointing with the mouse. (*new*)

2.3.1 Modifications to text routines

There will be two tables maintained within BE2. One will hold all character set tables and is going to be used to make up the keyboard mapping tables. The other one will specify what each character should be converted to in order to transfer it into ISO 2022. There will be three new features added into the text machinery:

1) A keyboard remapping utility that will take the actual characters generated from the keyboard and turn them into any other character. Eg. if the user presses the character ";" on a keyboard with Swedish mapping, this character will be transfered to an "a" with a circle above, since this is the standardized location for that character in Sweden. If however other hardware is used, the character on the right side of the "l" key may be something other than semicolon. In this case it is easy to modify the tables to fit this new hardware. This gives total freedom to place any character anywhere on the keyboard. Even the function keys may be used for infrequent characters.

I suggest that one keyboard mapping be present for each language, and starting from this each user may design any personal mapping he desires. If no mapping is specified the default ASCII mapping is used. The mapping should be possible to change from the pop-up menu, where either the system-supported mapping or the user-defined one can be selected. It should also be possible to bring up the mapping on the screen, and enter characters by pointing on the keyboard with the mouse.

These tables are stored in separate files with the extension ".keys" and are designed in a special editor. Since most mappings will be dedicated for a special language the name of the language will also be stored in the file. The format of this file can be found in Appendix B, and the character set tables in Appendix C.

2) Secondly, the routine that inserts the character into the text data object has to be modified. It must take care of the following two new issues:

When you switch to a non ASCII keyboard all text that is to be written will be surrounded with a *style* holding the name of that language. This will in no way affect the drawing of the character, it is rather a way to provide this information to software that would need it.

The converted character will have three properties: first, the value of the character within the character set table, secondly the number of the character set table where it is to be found, and finally whether the lower or the upper 128 characters are to be addressed. If the character is not a member of the ASCII character table, a style is added to specify this information and the corresponding ASCII character is inserted into the text data.

3) Third, the routine that displays this information must load the appropriate font table before displaying a non ASCII character. This information is easily retrieved from the style information.

To describe the usage of these tables, one character will be followed from the keyboard to the data stream. Below is a sketch to display the flow of the character from the keyboard into the text data structure:

Fig 1: Character flow from keyboard to text object

2.3.2 Modifications to the font manager

I don't see any reasons to modify the font manager. New font tables have to be added since the number of characters will increase significantly. To support all Latin languages it will be enough to add another four font tables with 128 characters in each, for every font, size and shape. These tables are 000_8 u (upper), 043_8 u, 361_8 l (lower) and 361_8 u. Then languages like Greek 046_8 and Cyrillic 047_8 have their own character set tables, and they probably just have one font with different sizes and shapes.

To know what font to get, an extended font naming convention must be used. An example would be:

< font name > < table # > < l/u > < font size > < shape > .fwm

2.4 Interface between style format and ISO 2022

Since the foreign characters are stored as styles within BE2 text objects, a conversion routine must be called before the text is sent to a computer not running Andrew. To do this conversion tables are used. This implies that it will be fairly easy to convert into any existing standard. If there is a country style wrapped around a chunk of text, more intelligent rendering could take place according to the character coding standard in that country.

2.5 Disadvantages with this scheme

Output routines other then the BE2 ones, like the ones in X-window, that do not now about styles will not be able to use the extended character set. To people in other countries it seems foolish to rely on 7-bit codes. All new programming environments are at least using the eighth bit to extend the character set. It would be strange to build a modern advanced object oriented environment, and only use 7-bit ASCII as the basic character code level. Also it must be preferred to support the character codes on a lower level that is suggested here. When searching in this kind of text there will be quite a lot of overhead to check what style each character has since the character in the text object doesn't tell what style it has, so this has to be looked up every time the style change. No general lexigraphical order is maintained within the standard.

2.6 Advantages with this scheme

It will fit into the existing system with a minimum of changes. It will tell what language a text is written in, which is neccesery to do intelligent spelling checking. The text may easily be converted to any standard. It can be expanded into a large number of characters which will be needed to type Chinese and Japanese.

2.7 Software that will be affected

Use of ISO 2022 will require all modifying software that is sensitive to control characters or foreign escape sequences. Further software that ignores them may also cause problems. This is because the syntactical order may be lost. An example of this may be a C-program where the user in a comment types *left quote, vertical arrow,* in sequence. If SO and SI are not interpreted these characters will be rendered as */, end of comment, against the the users intent. As a consequence the rest of the characters will be treated as a part of the program which probably will lead to a compilation error.

Mail programs and other software that communicates with an ASCII environment must prevent the user from inserting undesirable characters until it is known if the remote host can understand them. Further all software that generates output to the printers must be enhanced so a proper output is generated.

The compiler problem may easily be fixed by creating an alias that preprocesses the input and removes all non-ASCII characters in a comment and causes a compilation error if there are non-ASCII characters elsewhere, and then pipes the output to the compiler. Of course this will mean that there will be one preprocessor for each type of programming language.

If sorting has to be performed on the exteded character set new routines must be added.

- For example, an RT keyboard in not(USA) countries will generate 8-bit codes. How will these be handled with 7-bit character codes?

3. A 8-bit extended character set in BE2

Most computer manufacturers tend to use 8-bit extended ASCII in their new generation of computers, with or without penalty for more elaborate character coding schemes. It looks like there is much to be gained by using 8-bit codes.

A reasonable way to go is to use 8-bit coding within BE-2 and 7-bit format outside this environment. This would speed up the processing and still support compability towards the existing environment.

On disc the *in-line style* representation will be used, internaly the *environment styles* will be utilized. The transformation will be done thru cross reference tables. The increased number of characters will be accessed via multiple keyboard mappings kept in hardware dependent tables. A 32-bit *longChar* variable have to be used in order to store the nev character format outside a TextView.

3.1 Character set

The *ASCII/ISO/CCITT Roman Alphabet and Punctuation* will character mapping scheme will be used. This almost the only standard that exists in this field and is widely accepted among computer manufacturers. Minor differences between the mapping exist, but all the important characters is usually the same. Then there is space left for another 255 tables of up to 182 character each.

3.2 Text input

The characters will be inserted by using the keyboard or by using the mouse. A sketch of the current keyboard will be possible to bring up on the screen. Thru a mouse orientated editor the mapping of the keys can be redesigned according to personal taste.

3.2.1 Keyboard input

The overall basic keyboard will have the Standard ASCII mapping. This keyboard will have no penalty for foreign characters and the user will not notice any difference compared to the current keyboard. To be able to use foreign characters the user must change to a keyboard that include the ones needed.

The system supported keymappings will be based on each countries standard. Separate keyboards for symbols and other utility characters will also be included. Personal mappings will be easily designed and stored in the users home directory. It must be noticed that a mapping on a IBM keyboard may not be valid for a SUN, so the user have to design a separate mapping for every different terminal type he uses.

The function keys can be used since each mapping is for a separate machine, eventhough they not will be utilized on the system supported ones.

The default keyboard can be changed by a switch in the users *preferences file*.

3.2.2 Mouse input

If the user desires, the pointing device can be used to insert characters by requesting a map of the keyboard to be drawn on the screen. This will probably be prefered if just a few characters is going to be typed from a seldom used keyboard. This map will be attached to the bottom of the current window.

3.3 Internal representation

Within the memory, all characters will be stored as 8-bit characters (se Appendix D for a table). For non ASCII characters, four bytes are required to represent each. Out from the Key-Map package will a four byte longword be received. This one will contain sufficient information to identify any Latin character (for techincal details, see Appendix F). For these a *control code* will be inserted into the text stream instead. To this control code a style will point in which the multi byte character is stored. There will with other words be one style for each multi byte character. If the user desires so, another style telling which language a pice of text is written in will surround this area. This language information will be transparant and not affect the text. However if it is important to know what language a piece of text is written in, it can easily be retrieved.

Fig: The internal representation

3.3 Mass media representation

On mass storage another way to represent the data will be adopted. Essentially the extended characters is stored as a style is stored at the precent. This is, braces preceeded by an identifying string are used to quote all extended characters. The identifying string is the actual name of the character. Some examples follow below (a full list is found in Appendix E):

String	Comment
\.Copyright{}	A copyright sign
\.Diaeresis{a}	The Swedish a whith two dots above.
\.UStrokeH{}	Uppercase Maltesian character H with a stroke.

\.Acute{e}	An e whith a acute accent above.
\.Grave{a}	An a whith a grave accent above.
\.Cedilla{C}	A cedilla under a C.

A piece of text might look like this:

Je suis \.Acute{e}tudiante \.Grave{a} l'universit\.Acute{e}.
\.Cedilla{C}a me fait plaisir.

3.4 Text insertion routines

All keyboard input will be passed thru a filter. This filter consist of a table in which every key, and what output this key should produces, is listed. This makes it possible to remap the keyboard in any way. An editor will be provided to do this remapping so it will be possible for each user to customize his own keyboard layout and store it locally. The standardized keyboard layout for each language possible to write will be supported and a sketch of each will be possible to bring up on the screen.The user can select the different keyboard in two ways, either all text written with the keyboard will be surrounded by a language style. This is good if you want different programs to know what language the context has. The other vay is to insert the foreign characters you want but not care add any language to it. This is important if you just want to add a few symbols from another script without changing the language in the middle of a text.

3.4 Dispaly routines

The display

Appendix E

Table of character names

The character codes within **Character Set 0_8** :

40_8	Space
41_8	ExclamationPoint
42_8	NeutralDoubleQuote
43_8	NumberSign
44_8	CurrencySymbol
45_8	Percent
46_8	Ampersand
47_8	Apostrophe
50_8	OpeningParenthesis
51_8	ClosingParenthesis
52_8	Asterisk
53_8	Plus
54_8	Comma
55_8	NeutralDash
56_8	Period
57_8	Slash
60_8	0
61_8	1
62_8	2
63_8	3
64_8	4
65_8	5
66_8	6
67_8	7
70_8	8
71_8	9
72_8	Colon
73_8	SemiColon
74_8	LessThan
75_8	Equals
76_8	GreaterThan
77_8	QuestionMark
100_8	CommercialAt
101_8	A
102_8	B
103_8	C
104_8	D
105_8	E
106_8	F
107_8	G
110_8	H

111_8	I
112_8	J
113_8	K
114_8	L
115_8	M
116_8	N
117_8	O
120_8	P
121_8	Q
122_8	R
123_8	S
124_8	T
125_8	U
126_8	V
127_8	W
130_8	X
131_8	Y
132_8	Z
133_8	OpeningBracket
134_8	BackSlash
135_8	ClosingBracket
136_8	CircumflexSpacing
137_8	LowBar
140_8	Grave
141_8	a
142_8	b
143_8	c
144_8	d
145_8	e
146_8	f
147_8	g
150_8	h
151_8	i
152_8	j
153_8	k
154_8	l
155_8	m
156_8	n
157_8	o
160_8	p
161_8	q
162_8	r
163_8	s
164_8	t
165_8	u
166_8	v
167_8	w

170_8	x
171_8	y
172_8	z
173_8	OpeningBrace
174_8	VerticalBar
175_8	ClosingBrace
176_8	TildeSpacing
241_8	InvertedExclamationPoint
242_8	Cent
243_8	Pound
244_8	Dollar
245_8	Yen
247_8	Section
251_8	LeftSingleQuote
252_8	LetfDoubleQuote
253_8	LeftDoubleGuillemet
254_8	LeftArrow
255_8	UpArrow
256_8	RightArrow
257_8	DownArrow
260_8	Degree
261_8	Plus/Minus
262_8	SuperTwo
263_8	SuperThree
264_8	Multiply
265_8	Micro
266_8	Paragraph
267_8	CenteredDot
270_8	Divide
271_8	RightSingleQuote
272_8	RightDoubleQuote
273_8	RightDoubleGuillemet
274_8	OneQuarter
275_8	OneHalf
276_8	ThreeQuarters
277_8	InvertedQuestionMark
301_8	Grave
302_8	Acute
303_8	Circumflex
304_8	Tilde
305_8	Macron
306_8	Breve
307_8	OverDot
310_8	Diaeresis
312_8	OverRing
313_8	Cedilla

314_8	Underline
315_8	DoubleAcute
316_8	Ogonek
317_8	Hachek
320_8	HorizontalBar
321_8	SuperOne
322_8	Registered
323_8	Copyright
324_8	Trademark
325_8	MusicNote
334_8	OneEighth
335_8	ThreeEighths
336_8	FiveEighths
337_8	SevenEighths
340_8	Ohm
341_8	UDigraphAE
342_8	UStrokeD
343_8	OrdinalA
344_8	UStrokeH
345_8	LDotlessJ
346_8	UDigraphIJ
347_8	UMiddleDotL
350_8	UStrokeL
351_8	USlashO
352_8	UDigraphOE
353_8	OrdinalO
354_8	UThorn
355_8	UStrokeT
356_8	UEngma
357_8	LApostropheN
360_8	LGreenlandicK
361_8	LDigraphAE
362_8	LStrokeD
363_8	LEth
364_8	LStrokeH
365_8	LDotlessI
366_8	LDigraphIJ
367_8	LDotL
370_8	LStrokeL
371_8	LSlashO
372_8	LDigraphOE
373_8	DoubleS
374_8	LThorn
375_8	LStrokeT
376_8	LEngma

Appendix F

Definitions for the 8-bit solution

Character Longword, 4-byte: #define longchar longint

< Character Table 1 > < Character Table 0 > < Character Code 1 > < Character Code 0 >

Character Table 1 (byte 3), is the table where an eventual diacritic will be found.
Character Code 1 (byte 1), is the character code of an eventual diacritic.
Character Table 0 (byte 2), is the table where the character will be found.
Character Code 0 (byte 0), is the character code.

This longword will hold any character. The character codes and the number of the table where they belong may be found in Appendix C. This representation implies that if the value is below 128, the character belong to the ASCII character set. If it is between 128 and 255 it is a extended ASCII character. Between 256 and 65535 it is a character with a diacritic, and above 65535 the character belongs in another character table.

Keyboard Table

The Keyboard Table is machine dependent and contains information about the keytop layout and what output each key produces. This table is used to draw the keyboard sketch on the screen. The table is stored as a formatted text file. Each key can have any rectangular shape with an arbitrary text inside. The format is as follow:

< number of keys > , < font > , Comment
< x > , < y > , < xl > , < yl > , < a/o > , < Lin > , < Uin > , < Ltext > , < Utext > , < Lx > , < Ly > , < Ux > , < Uy > , < Lsize > , < Usize > , < config > ,

< x > Upper left corner of key rectangle.
< y > Upper left corner of key rectangle.
< xl > Length of key rectangle.
< yl > Length of key rectangle.
< a/o > a if Lin, Uin, Ltext and Utext should be entered in alpha format, o if in octal format. If octal they should be null terminated, and no comma between them, see example.
< Lin > Lowercase input string or char.
< Uin > Uppercase input string or char.
< Ltext > Lowercase text on keytop.
< Utext > Uppercase text on keytop.
< Lx > Location of lowercase text.
< Ly > Location of lowercase text.
< Ux > Location of uppercase text.
< Uy > Location of uppercase text.
< Lsize > Size of lowercase text.
< Usize > Size of uppercase text.

< config > Reconfigurable key (y/n).

Example:

```
101,times,              IBM-RT keyboard mapping (87/06/23)  V01
20,20,20,20,o,033 000 000 105 163 143 000 000,3,12,0,0,8,8,n,          Esc
64,20,20,20,a,,,F1,,3,12,0,0,8,8,y,
86,20,20,20,a,,,F2,,3,12,0,0,8,8,y,
108,20,20,20,a,,,F3,,3,12,0,0,8,8,y,
.
.
20,76,31,20,o,011 000 011 000 124 141 142 000 000,3,12,0,0,8,8,n,      Tab
.
.
502,120,20,42,o,015 000 000 105 156 164 000 000,2,23,0,0,8,8,n,        Ent = Cr
436,142,42,20,a,,,0,,3,9,0,0,8,8,n,             0 on the numerical part
480,142,20,20,a,,,.,,3,9,0,0,8,8,n,             . on the numerical part
```

This is the configuration file for IBM-RT. Leading spaces are not allowed between the parameters. No blank lines should exist. After the last comma in a definition can a comment appear.

Environment Style Transfer Table

This table contain enough information to be able to transfer the information between internal format and external format. Externally each non ASCII character is stored by it's name. Internally it is stored as 377_8 in the text object together with a Environment Style pointing to this character. The Environment Style will hold information about the character together with pointers to rendering procedures.

There is actually two tables with the same contents but sorted on different arguments to speed up the translation. The format of the tables are as follow:

< name > < key code > < table >

< name >	Character name.
< key code >	Code of character.
< table >	Number of table character belongs to.

Example, Character Set 000_8 :

Space	40_8	000_8

ExclamationPoint	41_8	000_8	
NeutralDoubleQuote	42_8	000_8	
.		.	.
.		.	.
LStrokeT		375_8	000_8
LEngma		376_8	000_8

This table is sorted on character codes for fast conversion to external format. The other one is sorted on character names and is used when an external file is read. Theese tables are stored within the procedures that uses them.

Keyboard Remapping Table

When a character is typed on the keyboard a code is generated and sent to the UNIX-kernel. The kernel parses this code and sends it further to BE2 as a ASCII character. The first thing that BE2 does is to check this character towards the *Keyboard Remapping Table* table. The table contain information about all modifications to the keyboard layout. This mean that you could place any character on any key by defining the relocation in this table. If the character is found there it will be converted into another character accordingly to the table. For instance, if you want to place the German DoubleS on the BackSlash key this should be inserted to the table.

This table is stored as a binary file either in a system directory for system supported ones, or in an user directory for an user defined one. The format is as follow:

< name >
< from char > < to char >

< name > The first string in the file is the name of the keyboard. This is probably a language.

< from char > Character to convert from. This is actually a null terminated ASCII string.

< to char > Character to convert to. This is a *longchar*.

Example:

From Char	To Char				Comment
swedish\0					
"\0	000_8	000_8	310_8	141_8	Swedish Diaeresis a.
"\0	000_8	000_8	310_8	101_8	Swedish Diaeresis A.
[\0	000_8	000_8	310_8	157_8	Swedish Diaeresis o.

{\0 000 8 000 8 310 8 117 8 Swedish Diaeresis O.

.
.
.

Only the keys that are about to be remapped has to be listed, for all others the default will be used. Default will be the US ASCII keyboard mapping.

Appendix G

Required changes to BE2 for the 8-bit solution

Changes has to be done in many different places within BE2. Below is a compilation of the required ones:

Global Defines

Defines that have to be supported to the user.

```
#define longchar longint

#define NUMOFFOREIGNCHARS  179
```

Local Defines to BE2

Internally BE2 have to maintain a number of variables that holds information about current keyboard name etc.

```
char *currentKeyboardPath                    Path to current keyboard remapping file.
0 if default.

char *currentKeyboardName                    Name of the current keyboard.

struct keyboardRemapStruct {                 This is the Keyboard Remapping Table
        char *fromChar;
        longchar toChar;
} keyboardRemapTable[];

struct codeToName                            This structure holds the names of
{                                                    the characters and the
code of
        char *charName;                              them. It is sorted on names.
        char charCode;
        char tableNumber;
} codeToNameTable[] ;

struct nameToCode                            This structure holds the names of
{                                                    the characters and the
code of
        char *charName;                              them. It is sorted on codes.
        char charCode;
        char tableNumber;
} nameToCodeTable[] ;
```

IM

New routies

Within IM the keyboard remapping has to be done. The function *RemapKey* will use the current table to remap the character sent to it. It returns a longchar holding the character (see Appendix F for the format of the table). This function will be used internally only. If no table is loaded it will use the default. This function will be supplied to the users.

```
longchar im__RemapKey(c)
char *c;
```

To load an alternate *Keyboard Remapping Table* the function *LoadAlternateKeyboard* is provided. This function takes a full path to a *Keyboard Remapping Table* or looks in the system directory for the file holding the information. If found it will load it and set it as the current one. It will return 1 if ok, 0 if there is a problem with the file. A null string will load the default one. The default one will also be loaded if an error occur. This function will be supplied to the users.

In the preferences file the user may define the path to the directory where it first should look for *Keyboard Remapping Tables.*

```
boolean im__LoadAlternateKeyboard(path)
char *path;
```

The associated function *StoreAlternateKeyboard* will store it on a file. Returns 1 if ok, 0 if it couldn't store. The full path has to be supported. This function will be supplied to the users.

```
boolean im__StoreAlternateKeyboard(path)
char *path;
```

To get information about the current keyboard the *GetKeyboardInfo* procedure is supported. This one will return information about the path to the keyboard and the name of it. This function will be supplied to the users.

```
void im__GetKeyboardInfo(path, name)
char *path;
char *name;
```

To alter the keyboard mapping three routines are used, *GetKeyRemapping*, *AddKeyRemapping* and *DeleteKeyRemapping*. The *GetKeyRemapping* function will return the remapping for the passed string. This function will be supplied to the users.

```
longchar im__GetKeyRemapping(c)
```

```
char *c;
```

The *AddKeyRemapping* procedure will let the applications program to alter the remapping of a key. If there already is a entry for this one, the old one will be replaced. The arguments are: char from keyboard, char to remap into. This function will be supplied to the users.

```
void im__AddKeyRemapping(c, newc)
char *c;
longchar newc
```

The *DeleteKeyRemapping* procedure will let the applications program to remove the current remapping for that key and revert to default. If there isn't a remapping for this key, nothing will happen. This function will be supplied to the users.

```
void im__DeleteKeyRemapping(c)
char *c;
```

The sketch of the keyboard is brought up on the screen by the *DisplayKeyboardLayout* procedure. This will be done in a window hooked onto the bottom of the current one. From this window it is possible to redesign the keyboard or load prestored keyboard mappings. This function will be supplied to the users.

```
void im__DisplayKeyboardLayout()
```

TEXT

New routies

Two routines will handle conversion between character code and character name. *FindCharacterCode* will convert form character code to character name. It will return a pointer to a record containing the name. If the character number not is found a NULL pointer is returned. This function will be supplied to the users.

```
struct codeToName *text__FindCharacterCode(code)
longchar          code;
```

The other one, *FindCharacterName* will take a pointer to a string and try to find a character with that name. It will return a pointer to a record containing the character. If the character name not is found a NULL pointer is returned. This function will be supplied to the users.

```
struct nameToCode *text__FindCharacterName(c)
char *c;
```

Routines to change

The *GetChar* function have to change in order to pull out character information from the Environment Style.

```
longchar text__GetChar(txt, pos)
struct text *txt;
long pos;
```

The *ReadSubString* function have to change so it calls FindCharacterName and adds Environment Styles for non ASCII characters. There will be no change in calling syntax.

The *WriteSubString* procedure have to change so it calls FindCharacterCode and quotes non ASCII characters. There will be no change in calling syntax.

TEXT VIEW

Routines to change

The procedure *FullUpdate* must recognize non ASCII charactrers. There will be no change in calling syntax.

The procedure *Update* must recognize non ASCII charactrers. There will be no change in calling syntax.

The procedure *KeyIn* must recognize non ASCII charactrers. It should return a longchar.

```
longchar textview__KeyIn(ip, ch)
struct view *ip;
int ch;
```

KEY MAP

Routines to change

The *BindToKey* function should store the keys as longchars. The char *keys argument must be changed to an array of longchars.

```
boolean keymap__BindToKey(self, functionName, module, keys)
struct keymap *self;
char *functionName;
char *module;
longchar keys[];
```

The Lookup function should lookup a longchar instead.

```
enum keymap__Lookup(self, key, object)
struct keymap *self;
longchar key;
char **object;
```

KEY STATE

Routines to change

The *ApplyKeyValues* function should store the keys as longchars. The char *keys argument must be changed to an array of longchars.

```
enum keystate__ApplyKeyValues(self, key, ppe, pobject)
struct keystate *self;
longchar key;
struct proctable_Entry **ppe;
struct basicobject **pobject;
```

FONT

Routines to change

FLIP

a Foreign Language Interface Package for BE2

Current Components

87-07-28
Tomas Centerlind

Information Technology Center
Carnegie-Mellon Univeristy
Pittsburgh, PA 152 13

At the present Andrew and related software lack the possibility to easily handle foreign characters. A few new routines within BE2 will add this capability. This paper will describe the current components of the *Foreign Language Interface Package, FLIP*. The package consist of a set of additional files and modifications to BE2. This set of new software will make up the basic layer to support foreign languages as well as additional symbols used in various areas.

The files needed are found in seven directories under user "tc8y", namely:

doc	Where all the documentation and files containing modification code are located.
mods	Files containing modifications to BE2 source code.
fonts	Where complete font files for character set 000 and 361 are located. Fonts available are andy8, andy10 and andy12 (andy12 is not complete at this time).
keyboard	Directory holding keyboard layout files and system supported keyboard remapping files.
ct/keyboard	All files for the keyboard table editor.
ct/basics	Here kbremap.c and kbremap.H are located.
ct/t	A test application.

A full explanation of how to install this in your own BE2 tree is found in Appendix H of the major report. It is currently installed properly in the ~tc8y/ct BE2 subtree even though it certainly will not work in a few weeks from the current date.

Below will be a list of files in each directory and a short explanation of its contents:

doc/	p.d	General paper.
	aa.d	Appendix A, references.
	ab.d	Appendix B, old stuff, not of interest.
	ac.d	Appendix C, first suggestion for solution.
	ad.d	Appendix D, character sets this is only a page header.
	ae.d	Appendix E, table of character names.
	af.d	Appendix F, definitions and tables.
	ag.d	Appendix G, new classes.
	ah.d	Appendix H, installation of FLIP in BE2.
	ai.d	Appendix I, Source code listing.

mods/	basics.Makefile.mods	
	graphic.mods.c	Modifications to basics/graphics.c
	graphic.mods.H	Modifications to basics/graphics.H
	fontdesc.mods.c	Modifications to basics/fontdesc.c
	fontdesc.mods.H	Modifications to basics/fontdesc.H
	im.mods.c	Modifications to basics/im.c
	im.mods.H	Modifications to basics/im.H
	wmgraphic.mods.c	Modifications to basics/wmgraphic.c
	wmgraphic.mods.H	Modifications to basics/wmgraphic.H
	sizes.mods.h	Modifications to basics/sizes.h
fonts/	andy8.fwm ASCII	Andy8 font file.
	andyaaal8.fwm	ASCII Andy8 font file.
	andyaaau8.fwm	Upper 128 characters in ASCII table for Andy8 font.
	andydgbl8.fwm	Rendering characters for Andy8.
	andydgbu8.fwm	Rendering characters for Andy8.
	andy10.fwm	ASCII Andy10 font file.
	andyaaal10.fwm	ASCII Andy10 font file.
	andyaaau10.fwm	Upper 128 characters in ASCII table for Andy8 font.
	andydgbl10.fwm	Rendering characters for Andy10.
	andydgbu10.fwm	Rendering characters for Andy10.
	andy12.fwm	ASCII Andy12 font file.
	andyaaal12.fwm	ASCII Andy12 font file.
	andyaaau12.fwm	Upper 128 characters in ASCII table for Andy8 font.
	andydgbl12.fwm	Rendering characters for Andy12.
	andydgbu12.fwm	Rendering characters for Andy12.
keyboard/	ibm032	Directory holding keyboard remapping files
	ibm032.klayout	File containing keyboard layout information for IBM-RT
	sun2	Directory holding keyboard remapping files
	sun2.klayout	File containing keyboard layout information for SUN-2, currently empty.
	sun3	Directory holding keyboard remapping files
	sun3.klayout	File containing keyboard layout information for VAX.
	vax	Directory holding keyboard remapping files
	vax.klayout	File containing keyboard layout information for VAX, currently empty.
ct/keyboard/	Makefile	Plain Makefile
	kbmgr.c	Main file for the keyboard manager.
	kbmgr.H	Ditto.
	kbpanel.c	Panel view for kbmgr.c.
	kbpanel.H	Ditto.
	kbtable.c	Key code table for kbmgr.c.
	kbtable.H	Ditto.
ct/basics/	kbremap.c	File containing procedures used by im.
	kbremap.H	Ditto.

ct/t/	Makefile	Makefile for the test application
	baseclass.c	Test program.
	baseclass.H	Ditto.
	t.c	File containing main unit.

Appendix A

References

[1] International Organization for Standardization, 7-bit Coded Character Set for Information Processing Exchange. ISO 646-1973.

[2] International Organization for Standardization, Code Extension Techniques for use with the ISO 7-bit Coded Character Set. ISO 2022-1972

[3] Xerox System Integration Standard, Character Code Standard, XNSS 058405 May 1986.

[4] Apple Corporation, The Script Manager, Engeneering Draft 9/24/86.

[5] International Telecommunication Union, CCITT, Recommendation T.61, October 1984.

[6] International Organization for Standardization, Coded character sets for text communication - Part 1 and 2. ISO 6937/1, ISO 6937/2, 1983.

Appendix E

Table of character names

The character codes within **Character Set 0_8**:

40_8	Space
41_8	ExclamationPoint
42_8	NeutralDoubleQuote
43_8	NumberSign
44_8	CurrencySymbol
45_8	Percent
46_8	Ampersand
47_8	Apostrophe
50_8	OpeningParenthesis
51_8	ClosingParenthesis
52_8	Asterisk
53_8	Plus
54_8	Comma
55_8	NeutralDash
56_8	Period
57_8	Slash
60_8	0
61_8	1
62_8	2
63_8	3
64_8	4
65_8	5
66_8	6
67_8	7
70_8	8
71_8	9
72_8	Colon
73_8	SemiColon
74_8	LessThan
75_8	Equals
76_8	GreaterThan
77_8	QuestionMark
100_8	CommercialAt
101_8	A
102_8	B
103_8	C
104_8	D
105_8	E
106_8	F
107_8	G
110_8	H
111_8	I
112_8	J
113_8	K
114_8	L

115_8	M
116_8	N
117_8	O
120_8	P
121_8	Q
122_8	R
123_8	S
124_8	T
125_8	U
126_8	V
127_8	W
130_8	X
131_8	Y
132_8	Z
133_8	OpeningBracket
134_8	BackSlash
135_8	ClosingBracket
136_8	CircumflexSpacing
137_8	LowBar
140_8	GraveSpacing
141_8	a
142_8	b
143_8	c
144_8	d
145_8	e
146_8	f
147_8	g
150_8	h
151_8	i
152_8	j
153_8	k
154_8	l
155_8	m
156_8	n
157_8	o
160_8	p
161_8	q
162_8	r
163_8	s
164_8	t
165_8	u
166_8	v
167_8	w
170_8	x
171_8	y
172_8	z
173_8	OpeningBrace
174_8	VerticalBar
175_8	ClosingBrace
176_8	TildeSpacing
241_8	InvertedExclamationPoint
242_8	Cent
243_8	Pound

244_8	Dollar
245_8	Yen
247_8	Section
251_8	LeftSingleQuote
252_8	LetfDoubleQuote
253_8	LeftDoubleGuillemet
254_8	LeftArrow
255_8	UpArrow
256_8	RightArrow
257_8	DownArrow
260_8	Degree
261_8	Plus/Minus
262_8	SuperTwo
263_8	SuperThree
264_8	Multiply
265_8	Micro
266_8	Paragraph
267_8	CenteredDot
270_8	Divide
271_8	RightSingleQuote
272_8	RightDoubleQuote
273_8	RightDoubleGuillemet
274_8	OneQuarter
275_8	OneHalf
276_8	ThreeQuarters
277_8	InvertedQuestionMark
301_8	Grave
302_8	Acute
303_8	Circumflex
304_8	Tilde
305_8	Macron
306_8	Breve
307_8	OverDot
310_8	Diaeresis
312_8	OverRing
313_8	Cedilla
314_8	Underline
315_8	DoubleAcute
316_8	Ogonek
317_8	Hachek
320_8	HorizontalBar
321_8	SuperOne
322_8	Registered
323_8	Copyright
324_8	Trademark
325_8	MusicNote
334_8	OneEighth
335_8	ThreeEighths
336_8	FiveEighths
337_8	SevenEighths
340_8	Ohm
341_8	UDigraphAE
342_8	UStrokeD
343_8	OrdinalA

344_8	UStrokeH
345_8	LDotlessJ
346_8	UDigraphIJ
347_8	UMiddleDotL
350_8	UStrokeL
351_8	USlashO
352_8	UDigraphOE
353_8	OrdinalO
354_8	UThorn
355_8	UStrokeT
356_8	UEngma
357_8	LApostropheN
360_8	LGreenlandicK
361_8	LDigraphAE
362_8	LStrokeD
363_8	LEth
364_8	LStrokeH
365_8	LDotlessI
366_8	LDigraphIJ
367_8	LDotL
370_8	LStrokeL
371_8	LSlashO
372_8	LDigraphOE
373_8	DoubleS
374_8	LThorn
375_8	LStrokeT
376_8	LEngma

Appendix F

Definitions for the 8-bit solution

Character Longword, 4-byte: typedef long longchar

<Character Table 1><Character Table 0><Character Code 1><Character Code 0>

Character Table 1 (byte 3), is the table where an eventual diacritic will be found.
Character Code 1 (byte 1), is the character code of an eventual diacritic.
Character Table 0 (byte 2), is the table where the character will be found.
Character Code 0 (byte 0), is the character code.

This longword will hold any character. Each symbol is 16-bits, but a 32-bit word is used to accommodate a diacritic associated with a character. This can be used in a more general way, so that any two symbols can be used to form a new one. The character codes and the number of the table where they belong may be found in Appendix C. This representation implies that if the value is below 128, the character belong to the ASCII character set. If it is between 128 and 255 it is an extended ASCII character. Between 256 and 65535 it is a character with a diacritic, and above 65535 the character belongs in another character table.

Keyboard Table

The Keyboard Table is machine dependent and contains information about the keytop layout and what output each key produces. This table is used to draw the keyboard sketch on the screen. The table is stored as a formatted text file. Each key can have any rectangular shape with an arbitrary text inside. The format is as follow:

<number of keys>,<font>, Comment
<x>,<y>,<xl>,<yl>,<a/o>,<Lin>,<Uin>,<Ltext>,<Utext>,<Lx>,<Ly>,<Ux>,<Uy>,<Lsize>,<Usize>,<config>,

<x>	Upper left corner of key rectangle.
<y>	Upper left corner of key rectangle.
<xl>	Length of key rectangle.
<yl>	Length of key rectangle.
<a/o>	a if Lin, Uin, Ltext and Utext should be entered in alpha format, o if in octal format. If octal they should be null terminated, and no comma between them, see example.
<Lin>	Lowercase input string or char.
<Uin>	Uppercase input string or char.
<Ltext>	Lowercase text on keytop.
<Utext>	Uppercase text on keytop.
<Lx>	Location of lowercase text.
<Ly>	Location of lowercase text.
<Ux>	Location of uppercase text.
<Uy>	Location of uppercase text.
<Lsize>	Size of lowercase text.
<Usize>	Size of uppercase text.
<config>	Reconfigurable key (y/n).
<special>	Special key, 0=normal key, 1=control, 2=shift, 3=caps

Example:

```
101,times,                IBM-RT keyboard mapping (87/06/23)  V01
20,20,20,20,o,033 000 000 105 163 143 000 000,3,12,0,0,8,8,n,0,        Esc
64,20,20,20,a,,,F1,,3,12,0,0,8,8,y,0,
86,20,20,20,a,,,F2,,3,12,0,0,8,8,y,0,
108,20,20,20,a,,,F3,,3,12,0,0,8,8,y,0,
.
.
20,76,31,20,o,011 000 011 000 124 141 142 000 000,3,12,0,0,8,8,n,0,    Tab
.
.
502,120,20,42,o,015 000 000 105 156 164 000 000,2,23,0,0,8,8,n,0,      Ent = Cr
436,142,42,20,a,,,0,,3,9,0,0,8,8,n,0,          0 on the numerical part
480,142,20,20,a,,,,,,3,9,0,0,8,8,n,0,          . on the numerical part
```

This is the configuration file for IBM-RT. Leading spaces are not allowed between the parameters. No blank lines should exist. After the last comma in a definition a comment can appear.

Environment Style Transfer Table

This table contain enough information to be able to transfer the information between internal format and external format. Externally each non ASCII character is stored by its name. Internally it is stored as 377_8 in the text object together with an Environment Style pointing to this character. The Environment Style will hold information about the character together with pointers to rendering procedures.

There are actually two tables with the same contents but sorted on different arguments to speed up the translation. The format of the tables is as follow:

```
<name> <longchar>

<name>      Character name.
<longchar>  longchar corresponding to the character name.
```

Example, Character Set 000_8:

Space	40_8	000_8
ExclamationPoint	41_8	000_8
NeutralDoubleQuote	42_8	000_8
.	.	.
.	.	.
LStrokeT	375_8	000_8
LEngma	376_8	000_8

This table is sorted on character codes for fast conversion to external format. The other one is sorted on character names and is used when an external file is read. Theese tables are stored within the procedures that uses them.

Keyboard Remapping Table

When a character is typed on the keyboard a code is generated and sent to the UNIX-kernel. The kernel parses this code and sends it further to BE2 as an ASCII character. The programmer receives this key thru the *key binding* mechanism. The first thing that then should be done if FLIP is used is to check this

character against the *Keyboard Remapping Table*. This is done thru the **im_RemapKey** IM method. The table contains information about all modifications to the keyboard layout. This means that you could place any character on any key by defining the relocation in this table. If the character is found there it will be converted into another character according to the table. For instance, if you want to place the German DoubleS on the BackSlash key this should be inserted to the table.

This table is stored as a binary file either in a system directory for system supported ones, or in a user directory for a user defined one. The format is as follows:

<from char><to char>

<from char>	Character to convert from. This is actually a null terminated ASCII string.
<to longchar>	Character to convert into. This is a longchar.

Example:

From Char	To Char	Comment
\0	000₈ 000₈ 310₈ 141₈	Swedish Diaeresis a.
"\0	000₈ 000₈ 310₈ 101₈	Swedish Diaeresis A.
[\0	000₈ 000₈ 310₈ 157₈	Swedish Diaeresis o.
\0	000₈ 000₈ 310₈ 117₈	Swedish Diaeresis O.
.		
.		
.		

Only the keys that are about to be remapped has to be listed, for all others the default will be used. Default will be the US ASCII keyboard mapping.

Font Name Extension

The font names must be extended since there can be up to 256 font files associated with each font within a family. A restriction within WM limits the symbols that are allowed to appear in a font name to letters. This has reduced the possibility to construct an easy to read format. The extension to the font name should look like this:

<font family><table><l/u><size><face>.fwm

<font family>	Name of the family this font belongs to.
<table>	Number of the table character belongs to. The number is a three-letter octal string where the letter "a" is 0, "b" is 1 and so on up to "h". All lower case.
<l/u>	"l" if char belongs to the lower 128 character, "u" if the upper 128.
<size>	Size of the font.
<face>	Bold, Italics etc.

Example:

timesaaal10b.fwm	Is a font from family andy, table 000 and the lower 128 characters.

Rendering Table

The rendering table is used to enhance the appearance for characters with diacritics. It consists of a list of characters, and what character in a *Rendering Table* they correspond to. The table is stored within the routine that uses it. There is currently one *Rendering Table* for Latin characters with diacriatics. This one is located in *Character Set* 3618 (see Appendix E). The format is as follow:

<from char> <to char>

<from char>	longchar containing character with diacritic.
<to char>	longchar containing character with rendering character.

Appendix G

Description of new BE2 classes and variables.

Changes have to be made in a few places within the basic classes of BE2. For an exact description on how to install the software, see appendix H. Below is a summary of the required changes:

Global Defines

The new variable type longchar must be included in the file <sizes.h>. This file usually doesn't have to be included, but sometimes in conjuction with the class command it might be necessary.

```
typedef  long longchar;
```

Local Defines to BE2

Internally BE2 has to maintain a number of variables which hold information about current keyboard name etc.

```
char *kbremap_Path;                 Path to current keyboard remapping file. 0 if default.

char *currentKeyboardName           Name of the current keyboard.

struct keyboardRemapStruct {        This is the Keyboard Remapping Table
    char *fromChar;
    longchar toChar;
} keyboardRemapTable[];
```

```
struct codeToName                   This structure holds the names of
{                                   the characters and their codes. It
char *charName;                     is sorted by names.
    char charCode;
    char tableNumber;
} codeToNameTable[] ;
```

```
struct nameToCode                   This structure holds the names of
{                                   the characters and their code. It
    char *charName;                 is sorted by codes.
    char charCode;
    char tableNumber;
} nameToCodeTable[] ;
```

IM

New routines

Within IM the keyboard remapping has to be done. The function *RemapKey* will use the current table to remap the character sent to it. It returns a longchar holding the character (see Appendix F for the format of the table). This function must be called if FLIP is used to extend the character set. If no table is loaded it

will use the default. Default is precently an empty table. If no remapping for the passed character is found, NULL will be returned. This function will be supplied to the users.

```
longchar im__RemapKey(c)
char *c;
```

To load an alternate *Keyboard Remapping Table* the function *LoadAlternateKeyboard* is provided. This function takes a full path to a *Keyboard Remapping Table*, or a file name and looks in the directories specified in the users *preferences* for the file holding the information. If found it will load it and set it as the current one. It will return TRUE if ok, FALSE if a problem occurs while loading the *Keyboard Remapping Table*. A null string will load the default one. Nothing will happen if an error occurs while trying to get the keyboard remapping. The default table is currently an empty table. This function will be supplied to the users.

In the preferences file the user may define the path to the directory where it should look for *Keyboard Remapping Tables.*

```
boolean im__LoadAlternateKeyboard(path)
char *path;
```

The associated function *StoreAlternateKeyboard* will store it into a file. Returns TRUE if ok, FALSE if it couldn't store. The full path has to be sent as an argument to the function. This function will be supplied to the users.

```
boolean im__StoreAlternateKeyboard(path)
char *path;
```

To get information about the current keyboard the *GetKeyboardInfo* procedure is supported. This procedure will return information about the path to the keyboard and its name. This function will be supplied to the users.

```
void im__GetKeyboardInfo(path, name)
char *path;
char *name;
```

The *AddKeyRemapping* procedure will let the applications program alter the remapping of a key. If there already is a entry for this one, the old one will be replaced. The arguments are: a null terminated string of characters from the keyboard, longchar to remap into. This function will be supplied to the users.

```
void im__AddKeyRemapping(c, newc)
char *c;
longchar newc;
```

The DeleteKeyRemapping procedure will let the applications program remove the current remapping for that key and revert to default. If there isn't a remapping for this key, nothing will happen. This function will be supplied to the users.

```
void im__DeleteKeyRemapping(c)
char *c;
```

The information needed for these routines isstored in the IM object. Here are the variables which have been added:

```
struct kbremap_Struct {          Structure for remapping entities on the heap.
    char *fromChar;              Pointer to string to remap from.
    long toChar;                 longchar to remap to.
};


char  *kbremap_Path;             Path to current Keyboard Remapping File.
struct kbremap_Struct *kbremap_Table;  Array of remapping entries.
int   kbremap_TableLength;       Length of remapping table in number of entries.
int   kbremap_SpaceAlloc;        Space allocated for remapping table.
```

GRAPHIC

New routines

A new *DrawStringLongChar* routine must be added to print strings composed of longchars. This procedure take the same arguments as the DrawString procedure, except that the String argument must be an array or longchars.

```
void graphic__DrawStringLongChar(self, string, operation)
struct graphic *self;
longchar string[];
long operation;
```

A new *DrawTextLongChar* routine must be added to print strings composed of longchars. This procedure take the same arguments as the *DrawText* procedure, except that the String argument must be an array or longchars.

```
void graphic__DrawTextLongChar(self, string, stringLength, operation)
struct graphic *self;
longchar string[];
long stringLength;
long operation;
```

TEXT

New routines

Two routines will handle conversion between character code and character name. *FindCharacterCode* will convert from character code to character name. It will return a pointer to a record containing the name. If the character number is not found a NULL pointer is returned. This function will be supplied to the users.

```
struct codeToName *text__FindCharacterCode(code)
longchar    code;
```

The other one, *FindCharacterName* will take a pointer to a string and try to find a character with that name. It will return a pointer to a record containing the character. If the character name is not found a NULL pointer is returned. This function will be supplied to the users.

```
struct nameToCode *text__FindCharacterName(c)
```

```
char *c;
```

The *GetLongChar* function has to be added in order to pull out character information from the *Environment Style*. It is equivalent to the *GetChar* function except that it returns a longchar instead.

```
longchar text__GetLongChar(txt, pos)
struct text *txt;
long pos;
```

Routines to change

The *ReadSubString* function has to change so it calls *FindCharacterName* and adds *Environment Styles* for non-ASCII characters. There will be no change in calling syntax.

The *WriteSubString* procedure has to change so it calls *FindCharacterCode* and quotes non-ASCII characters. There will be no change in calling syntax.

TEXT VIEW

Routines to change

The procedure *FullUpdate* must recognize non-ASCII characters. There will be no change in calling syntax.

The procedure *Update* must recognize non-ASCII characters. There will be no change in calling syntax.

The procedure *KeyIn* must recognize non-ASCII characters. It should return a longchar.

```
longchar textview__KeyIn(ip, ch)
struct view *ip;
int ch;
```

FONT DESC

New Routines

If longchars are used, the *TextSize* function has to be replaced with another function that recognizes longchars. The *TextSizeLongChar* serves this purpose. To this function an array of longchars is passed. The new format should look like this:

```
long fontdesc__TextSizeLongChar(fontdesc, self, text, TextLength, XWidth, YWidth)
struct fontdesc *fontdesc;
struct graphic *self;
longchar text[];
long TextLength;
long *XWidth;
long *YWidth;
```

If longchars are used, the *StringSize* function has to be replaced with another function that recognizes longchars. The *StringSizeLongChar* serves this purpose. To this function an array of longchars is passed. The new format should look like this:

```
long fontdesc__StringSizeLongChar(fontdesc, self, text, XWidth, YWidth)
struct fontdesc *fontdesc;
struct graphic *self;
longchar text[];
long *XWidth;
long *YWidth;
```

Lookup for characters with diacritics in the *Rendering Set* will be done by the *FindRenderingCharacter* function. The argument is a longchar with the character and diacritic. The function will return a longchar containing the number of the Character Set where the rendering character will be found and the character within the set. If no rendering character is found NULL is returned. The function finds out the rendering character by looking it up in a table (see Appendix F) kept in the function. This function will be supplied to the users.

```
longchar fontdesc__FindRenderingCharacter(c)
longchar c;
```

KBMGR

New Routines

An editor for the Keyboard Remapping Tables is placed in an object named *kbmgr*. This dynamically loaded object will export two class procedures, one to open the keyboard window and one to close it. To use the keyboard remap editor, the user must add an option to the pop-up menu that will display the mock-up keyboard on the screen. The two procedures that should be used for this are OpenKeyboardWindow and *CloseKeyboardWindow*.

The sketch of the keyboard is brought up on the screen by the *OpenKeyboardWindow* procedure. This will be done in a window hooked onto the bottom of the current one. From this window it is possible to redesign the keyboard or load a prestored keyboard mapping. The pointer to the view where the keyboard should be inserted must be supported. The first boolean argument (edit) specifies whether the user may edit a Keyboard Remapping Table. The second one determines if another keyboard remapping may be selected. This function will be supplied to the users.

```
void kbmgr__OpenKeyboardWindow(self, edit, select)
struct view *self;
boolean edit;
boolean select;
```

The sketch of the keyboard is hidden by using the *CloseKeyboardWindow* procedure. All space used on the heap is deallocated. The pointer to the view where the keyboard window is located must be supported. This function will be supplied to the users.

```
void kbmgr__CloseKeyboardWindow(self)
struct view *self;
```

KBREMAP

New Routines

This new object should be totally transparent to the user. All access to it is done thru IM. The reason for this is to avoid loading the program code when not needed. For a full description of arguments and usage, see under IM. The names of the functions within kbremap are:

longchar **kbremap__RemapKey**(c)

boolean **kbremap__LoadAlternateKeyboard**(path)

boolean **kbremap__StoreAlternateKeyboard**(path)

void **kbremap__GetKeyboardInfo**(path, name)

void **kbremap__AddKeyRemapping**(c, newc)

void **kbremap__DeleteKeyRemapping**(c)

Miscellaneous Changes

The Window Initiallization routine must look in the preferences file for the search paths to the keyboard remapping tables. The entry should look like this:

*.KeyboardRemappingPath:/cmu/itc/tc8y/keyremap:/usr/andy/keyremap

The Window Initialization routine must look in the preferences file for the default keyboard remapping, and load that one. It will look for a file with the extension ".kremap. The entry should look like this:

wm.KeyboardRemapping:swedish will load the swedish keyboard remapping.

Appendix H

Installation of FLIP set in BE2

Changes have to be made in many different source files within BE2. The actual code that should be inserted may be found in Appendix I. Below is a summary of the required changes:

basics/sizes.h Add a typedef to the list of types, see **sizes.mods.h.**

basics/im.c In IM indirect calls to kbremap have to be added, the procedures are:

im__RemapKey
im__LoadAlternateKeyboard
im__StoreAlternateKeyboard
im__GetKeyboardInfo
im__AddKeyRemapping
im__DeleteKeyRemapping

Place these last in the file. In addition to these procedures a few other modifications should be added, see **im.mods.c**.

basics/im.H Defines, structs and methods in file **im.mods.H** should be added.

basics/graphic.c Procedures in file **graphic.mods.c** should be added. The procedures are:

graphic_DrawCharsLongChar
graphic__DrawStringLongChar
graphic__DrawTextLongChar

They must come in this order and be located at the end of the file.

basics/graphic.H Defines and methods in file **graphic.mods.H** should be added.

basics/wmgraphic.c Procedures in file **wmgraphic.mods.c** should be added. The procedures are:

wmgraphic_DrawCharsLongChar
wmgraphic__DrawStringLongChar
wmgraphic__DrawTextLongChar

They must come in this order and be located at the end of the file.

basics/wmgraphic.H Defines and methods in file **wmgraphic.mods.H** should be added.

basics/fontdesc.c Functions in file **fontdesc.mods.c** should be added. The functions are:

fontdesc__FindRenteringCharacter
fontdesc__TextSizeLongChar
fontdesc__StringSizeLongChar

They must come in this order and be located at the end of the file.

basics/fontdesc.H Methods and classprocedures in file **fontdesc.mods.H** should be added.

basics/kbremap.c This file should be installed in the directory.

basics/kbremap.H This file should be installed in the directory.

basics/Makefile The makefile must be updated to reflect the changes to the files. See: **basics.Makefile.mods.**

A new directory named **keyboard** must be added to the base of the tree. To this directory the following files should be added:

keyboard/kbmgr.c This file should be installed in the directory.

keyboard/kbmgr.H This file should be installed in the directory.

keyboard/kbtable.c This file should be installed in the directory.

keyboard/kbtable.H This file should be installed in the directory.

keyboard/kbpanel.c This file should be installed in the directory.

keyboard/kbpanel.H This file should be installed in the directory.

keyboard/Makefile The makefile must be added to reflect these additional files. See: **keyboard.Makefile.mods.**

The global Makefile must be altered so that files in directory **keyboard** will be compiled. See: **Makefile.mods.**

Appendix I

Source code listing

This appendix contains source code for all new files and all that are modified.

New files:

- basics/kbremap.c
- basics/kbremap.H
- keyboard/kbmgr.c
- keyboard/kbmgr.H
- keyboard/kbpanel.c
- keyboard/kbpanel.H
- keyboard/kbtable.c
- keyboard/kbtable.H
- keyboard/Makefile

Files modified:

- basics/fontdesc.c
- basics/fontdesc.H
- basics/graphic.c
- basics/graphic.H
- basics/im.c
- basics/im.H
- basics/Makefile
- basics/sizes.h
- basics/wmgraphic.c
- basics/wmgraphic.H
- Makefile