HANGUL OSD FONTS

IN THE EMBEDDED MICROPROCESSOR

Font Storage

The normal method for storing OSD fonts (glyphs) is to have bitmap for each glyph. In the case of the modern Korean language, there are 11,172 syllables as defined in the latest Unicode standard. These 11,172 syllables can produce all modern Korean text. However, this results in excessive FLASH memory storage requirements. Furthermore, the OSD should also be capable of displaying English ASCII language text. Even Korean language text and phrases necessarily include English letters or words from time to time. And of course, we need the usual numbers and punctuation symbols. This can be done by adding the 95 glyphs from the ASCII table (codes 32 through 126).

In our hardware, one "glyph" is a 16 x 16 pixel array (32 bytes), and a 4-bit width parameter. This takes 33 bytes (actually 32-1/2 bytes) of storage per glyph. This will result in the following FLASH memory storage requirements:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	11,172	33	368,676	
English	95	33	3,135	
			371,811	100%

The first simplification we can make is to throw away the width data. We plan to make all glyphs fixed width, so there is no reason to save this repetitive data. This gives a very small reduction as follows:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	11,172	32	357,504	
English	95	32	3,040	
			360,544	97%

The next possible reduction is to consider the reduced Hangul syllable set from the Korean Graphic Character Set for Information Exchange, KS-X-1001 (formerly KS-C-5601). This standard is also the basis for the ISO-2022 encoding method called EUC-KR (Extended Unix Code), and is registered character set number 149 with ISO on Oct 1988. This standard defines 2350 common Hangul syllables, which are contained in 25 rows of 94-glyphs each (ISO-2022 multi-byte graphics character sets define a possible 94 x 94 array of glyphs). This gives a significant reduction in memory size, as can be seen below:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	2,350	32	75,200	
English	95	32	3,040	
			78,240	21%

However, further reduction is still possible. A study at Dr. Kang Sung-sik's Korean Language Processing and Information Retrieval Laboratory was brought to my attention. This study shows a frequency analysis of the syllables in modern Korean language, to determine which of the 2350 syllables are the most commonly used. I don't have the details of his analysis, for example, what kind of texts were used, but it is easy to analyze his results:

Number of Syllables	Cumulative Percentage	Flash Storage
714	99.0	26,272
859	99.5	30,912
1037	99.8	36,608
1164	99.9	40,672

Even the 99.9 percent point cuts the number of syllables by about half. I'm not exactly sure where it would be best to draw the cut-line. I think it would be best for a native Korean speaker to examine the list and make the cut. But for this study, let's just guess 1000 syllables. Therefore, we now have the following storage:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	1,000	32	32,000	
English	95	32	3,040	
			35,040	9.5%

That's a big improvement. But we can do better, AND get full symbol coverage as well. So let's abandon this above technique and look freshly at a new approach. In my research I learned about a method for building all the possible Hangul syllables by using a reduced set of fonts called "combining letters". This technique was used by an old X11 software package called Hanterm used in the 1990's. In this system, the 19 leading consonants, the 21 vowels, and the 27 trailing consonants are prepared as individual symbols. But instead of simply drawing one symbol per letter, each letter is drawn in all the various ways it might be used, considering all the possible variation of syllables. For example, the leading consonants are drawn in 10 different positions and sizes, resulting in 210 glyphs for the 21 leading consonants.

This font method is appropriately called JOHAB, and a typical Johab font contains 529 different glyphs. This is a further improvement by almost 50% from the previous method.

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	529	32	16,928	
English	95	32	3,040	
			19,968	5.4%

This savings does come at a slight cost. There is added complexity to make each syllable. Instead of just finding the syllable you want in a font table, now each syllable has to be "built" by combining 2 or 3 combining letters to make the syllable. Still, however, this is a worth the effort in order to save the memory. The added logic and processing time are simple.

While this seems to be the smallest set of glyphs, more reduction is possible. If you study the set of Johab glyphs, further simplification is clear. The first thing we see is that there are a lot of symbols included for the display of historical Hangul text.

JOHAB Glyphs	Total	Modern	Historical
Leading Consonants	310	190	120
Vowels	94	70	24
Trailing Consonants	124	108	16
TOTALS	**528**	**368**	**160**

If we restrict the Johab letters to those in modern Korean, we get the following:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	368	32	11,776	
English	95	32	3,040	
			14,816	4.0%

Then on further inspection, I noticed that many of the remaining modern glyphs looked almost identical. I therefore performed a cross-correlation of each letter against all the other letters, searching for duplicate glyphs. For example, I compared all 10 KIYEOK symbols with each other, then next all 10 NIEUN symbols with each other, etc. Here is an example of one symbol, the leading MIEUM. Ten versions of MIEUM are stored in locations 61 to 70 in the standard Johab font tables:

	61	62	63	64	65	66	67	68	69	70
61	-	52	52	30	30	26	54	54	30	30
62	-	-	**0**	26	26	36	26	26	26	26
63	-	-	-	26	26	36	26	26	26	26
64	-	-	-	-	**0**	10	38	38	**0**	**0**
65	-	-	-	-	-	10	38	38	**0**	**0**
66	-	-	-	-	-	-	42	42	10	10
67	-	-	-	-	-	-	-	**0**	38	38
68	-	-	-	-	-	-	-	-	38	38
69	-	-	-	-	-	-	-	-	-	**0**
70	-	-	-	-	-	-	-	-	-	-

In this case, there are 5 duplicate glyphs which can be omitted. Completing the analysis, the results are as follows:

JOHAB Glyphs	Modern	Duplicates	Remaining
Leading Consonants	190	85	105
Vowels	70	18	52
Trailing Consonants	108	22	86
TOTALS	**368**	**125**	**243**

I suspected that this duplication would vary from font-to-font, among the different styles of Johab fonts available. An analysis of 4 different Johab fonts shows this to be true, although the resulting savings is still beneficial in all cases.

	Duplicate Leading	Duplicate Vowels	Duplicate Trailing	Duplicate Total	Remaining Glyphs
Iyagi Bold	85	18	22	125	243
Gothic Medium	59	28	54	141	227
Myeongjo Bold	29	4	23	56	312
Philgi Bold	0	18	27	45	323
AVERAGE					**276**

For this study, let's use the average of these four fonts, realizing that the final choice of font will have a slight impact on the final memory storage. There is one other point to consider with this round of simplification is the algorithm used to combine the glyphs into a syllable. It has to be adjusted to deal with these "holes" in the font map. There may be an elegant solution, but one brute force way to do it is to make a couple of auxiliary lookup tables which point to the location of the glyph. I have studied this way of changing the algorithm, and the "expense" of these additional tables is about 500 bytes.

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	276	32	8,832	
English	95	32	3,040	
Algorithm "cost"			500	
			12,372	3.3%

At this point, I think I have optimized the Hangul as far as practical, and as far as necessary. In my opinion, any further reductions would not be helpful. However, there is one additional reduction possible regarding the English ASCII font table. The original font(s) which I have been using were originally only 8-bits wide (8x16). To use them in the embedded system, I doubled the width of each pixel, artificially making a 16 x 16 font. Now that we will be building glyphs dynamically in real time, there's no reason to waste space to hold this duplicate information. The last memory reduction I propose is to store the ASCII font in the original 8x16 format, and do the pixel doubling in real time. One "brute force" way to implement this is a 256 x 2 byte lookup table (there may also be a more efficient algorithmic way as well). But using the 512 byte lookup table as worst case, here is the resulting memory requirement:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	276	32	8,832	
English	95	16	1,520	
Algorithm "cost"			1,012	
			11,364	3.1%

At this point, let's stop. We began with a normal technique to display Korean and English which required 371,811 bytes. By applying successive techniques and slightly increasing the font processing algorithms, we have succeeded to reduce this FLASH storage requirement to just 11,364 bytes, about 1/32 of the original requirement. Furthermore, the original storage size was too large to fit within the internal FLASH memory of most of the embedded microprocessors under consideration. By implementing this reduction, we will eliminate the need and complexity of adding off-chip FLASH memory storage.

A summary of each step is shown in the below combined table:

	Glyphs	Bytes/Glyph	Total Bytes	Percentage
Korean	11,172	33	368,676	
English	95	33	3,135	
		Original Starting Point	**371,811**	**100%**
Korean	11,172	32	357,504	
English	95	32	3,040	
		Use Fixed Width	**360,544**	**97%**
Korean	2,350	32	75,200	
English	95	32	3,040	
		Use KS-X-1001 Char Set	**78,240**	**21%**
Korean	1,000	32	32,000	
English	95	32	3,040	
		Only include 99.x% of Syllables	**35,040**	**9.5%**
Korean	529	32	16,928	
English	95	32	3,040	
		Change to Hanterm's Johab Method	**19,968**	**5.4%**
Korean	368	32	11,776	
English	95	32	3,040	
		Only use Modern Jamo	**14,816**	**4.0%**
Korean	276	32	8,832	
English	95	32	3,040	
Algorithm "cost"			500	
		Remove Duplicate Glyphs	**12,372**	**3.3%**
Korean	276	32	8,832	
English	95	16	1,520	
Algorithm "cost"			1,012	
		Store ASCII as 8x16	**11,364**	**3.1%**

Fonts

For the actual source of the fonts, I began with the font package which was included with the original development system, from the Wen Quan Yi (Spring of Letters) project. They have compiled a number of open source CJK bitmapped font packages. The package we received from originally was version 1.0. It appears a version 1.1 was released, but before I was able to find that, I found the Unifont collection.

The GNU Unifont bitmap project began with Roman Czyborra, and was carried on by Paul Hardy. I'm not clear on the dates, but apparently the Wen Quan Yi fonts were cooperatively merged into the GNU Unifont package. Studying the information on their sites, I was alerted to the older Johab method of font preparation. They were interested in the Johab method as a way to obtain older (and perhaps better) fonts and add them to the Unifont package. But I quickly realized this technique had advantages to use in the embedded system world as well.

One challenge with fonts for use with OSD is that they be clearly readable. I faced this challenge before when choosing a font for English. I finally found one which is referred to as "high visibility" font. The fonts I've located for Hangul don't appear to have this feature. There is one obvious issue – that is Hangul syllables have more detail, and the lines cannot be too thick. But I think a more thorough search can reveal a better Hangul font for use with OSD.

English

In my previous researches, it was quite difficult to find a suitable font that was mono spaced. Ideally, such a font would be available in italic and bold versions as well. I was never exactly able to find this perfect font, but came pretty close with the High Visibility font mentioned before. Interestingly, my research led me to studies done by organizations for blind people. They have a keen interest in access to clearly visible fonts, and have done a lot of study in this area. The Canadian National Institute for the Blind did a few studies in this area. They recommend the standard fonts include VERDANA, ARIAL, TAHOMA, although these fonts don't come in mono spaced versions. Further digging led to a font especially designed for subtitling by Dr. John Gill for the Royal National Institute for the Blink in the U.K. This font is called TIRESIAS. However, Dr. Gill told me that he didn't make a mono spaced version, but advised me to discuss other options with the folks at Bitstream, the company which maintains and sells this font (and others). Bitstream didn't have an off-the-shelf solution available, but was willing to custom make one for us. They also recommended we consider a font called LETTER GOTHIC. Further research revealed a company called Ascender Corp, offers some fonts suitable for OSD. One is called ANDALE MONO, which is reportedly very similar to the VERDANA , along with several other fonts specially designed for closed caption decoders. Their top recommendation was LUCIDA SANS. All of the fonts described above require various license fees. At the time we originally developed our product, it wasn't clear that a purchased font was necessary, nor would we sell enough units to support the licensing fees.

With that in mind, I searched for some free fonts as well. I found four different mono spaced fonts, but each one of them had some minor problem or another. These fonts are PROFONT by Tobias Jung. VERA is from the Gnome organization. Anonymous Pro by Mark Simonson. And BPtypewriter and BPmono by George Triantafyllakos.

Now that my tools are better, I believe it would be a good exercise to revisit these fonts (especially the free ones) and do some testing on the real hardware.

Hangul

Besides some early testing, I have ignored the "improves" Hangul syllables which were included in the big Unifont package. The reason, Mr. Hardy says that he used a "thin stroke" version of Johab in order to make his glyphs, because they look nicer on the computer screen. This seemed the opposite of what I want – I need thicker strokes for better visibility. Let's come back to this point later.

Instead, I searched his original sources and located the following basic four Johab font packages which are maintained at the online KAIST file archive. These are the four mentioned above:

Iyagi Bold
Gothic Medium
Myeongjo Bold
Philgi Bold

So far, neither of these four fonts seem to be the ideal solution. I am continuing to search for other fonts. I have found this list of candidate fonts, and will continue to experiment to determine the best visible font (in my opinion). These additional fonts include:

Ming-style	**Sans-serif**	**Script**	**Other**
Batang	Apple Gothic	UnYetgul (*)	UnGunSeo (*)
BatangChe	Dotum		Nanum Pen (*)
Gungsuh	DotumChe		Jeubsida Sun-Moon (*)
GungsuChe	Gulim		
UnBatang (*)	GulimChe		
UnGungsuh (*)	Malgum Gothic		
Baekmuk Batang (*)	New Gulim		
Nanum Myeongjo (*)	UnDotum (*)		
Jieubsida Batang (*)	UnShinmum (*)		
	Baekmuk Gulim (*)		
	Nanum Gothic (*)		
	Jeupsida Dotum (*)		

I have found other lists online as well. I will keep searching.

Attributes

Final comment about fonts regards attributes. Ideally character attributes like bold and italics should be built as separate glyphs (underlining is really simple to do dynamically). Since I did the original OSD project, I've learned that there are some simple "poor-man's" methods to making bold and italics. For bold, just take the symbol, and shift it one pixel, and add it back to the original to simulate bold. Similarly, a surprisingly good italic glyph can be made by shifting the character by one pixel at one or two places. And as mentioned, underlining is very simple.

I've done some tests with these techniques, and not surprisingly it works better with English than Hangul. But, the Hangul font I was using was the boldest one I could find. I want to try it some more, using some of these different font styles. We might be able to make a highly visible Hangul font from just "bolding" a thin-stroke font.

Font Processing Tools

I have developed a number of font processing tools and library functions to help the development of OSD projects. All are written in standard C-language, and typically compiled with GNU gcc compiler.

BDF file processing module includes functions for reading and finding glyphs in files using the Adobe Glyph Bitmap Distribution Format. This format was chosen because it was already in use by the existing tools, and is a common standard for bitmapped fonts. Functions include:

```
FONT *init_font_processing( char *fname, FONT *pf, int font_type );
FONT *read_font_properties( FONT *pf );
void rewind_font_file( FONT *pf );
GLYPH *find_glyph( uint32_t code, FONT *pf );
GLYPH *next_glyph( FONT *pf );
```

The ADV processing module converts glyphs into the binary format needed by the ADV on-screen display engine. It can make "font packs" which are identically formatted as the ones generated by the ADI's offline menu generating tool. This is useful, because it makes it very easy to load font and string packs to the chip using existing functions in the ADI library.

```
void mkadv_hfile_font_packs( FILE *fout, int nglyphs, GLYPH *pgl[] );
void mkadv_hfile_string_packs( FILE *fout, int nchars, uchar *wheel );
void mkadv_hfile_string_buffer( FILE *fout, int nchars, uchar *wheel );
```

Program B2A was made to convert a BDF font directly into ADV format. It accepts the following inputs:

```
B2A bdf-font-file font-pack-name beg-code end-code type x-scale spi-flag
```

An example, consider these commands which generate a set of raw (no spi header) font packs for use by our OSD crawl algorithm:

```
b2a iyagi16.bdf JOHAB_IYAGI16 0 528 j 1 0 2>adv_johab.txt >adv_johab.h
b2a hvf-ascii.bdf ASCII_HVF 32 126 a 1 0 2>adv_ascii.txt >adv_ascii.h
```

Another tool is available which prepares font and string packs based on an input text file containing Unicode Hangul (and English) text. This requires a BDF font file containing the full Hangul set (it doesn't use the Johab technique). Based on the text contained in the input file, it chooses the needed glyphs and includes them in the resulting font pack. This program is called K2B, as follows:

```
k2a text-file ascii-bdf-file hangul-bdf-file
```

An example, I prepared a Unicode text file containing a Korean poem by Kim So-wol, "Azaleas".

```
k2a azaleas.txt hvf-ascii.bdf iyagi16.bdf 2>ktest.txt >ktest.h
```

Just as an example, here is a sample of the output from the above program. All the font packs are very similar, so only one example will be shown. The input file is:

```
This is a Korean poem:
나 보기가 역겨워  가실 때에는
말없이 고이 보내 드리우리다
영변에 약산  진달래꽃
아름 따다 가실 길에 뿌리오리다.
가시는 걸음 걸음  놓인 그 꽃을
사뿐히 즈려 밟고 가시옵소서.
나 보기가 역겨워  가실 때에는
죽어도 아니 눈물 흘리오리다.
Maya Fighting.
```

And here are the output h-file font and string packs (omitting some data to make the table smaller):

```
// PACK: PACK_NAME_UNKNOWN
//***************************************************************************
//***************************************************************************
//*****                                                                 *****
//*****                        Font Pack ROM                            *****
//*****                                                                 *****
//***************************************************************************
//***************************************************************************
#define _NUM_GLYPHS_TOT   (79)
#define _BEG_GLYPH_WORD   (20)
#define _END_GLYPH_WORD   (98)
#define ADV_ASCII_SPACE   (20)

//***************************************************************************
//*****          Character ROM - Font Width Section (RAM Left)       *****
//***************************************************************************
UCHAR CONSTANT FONT_L_PACK_NAME_UNKNOWN[] = {
  0x51, 0x00, // Burst Length 81 (0x51)
  0x0d, 0x14, // SPI Command
  // Total of 79 glyphs in this pack
  // Loaded into FONT RAM locations 20 (0x14) to 98 (0x62)
  // Width Data, (Char RAM Left) Character Set: ASCII
  0xf0,  // Index  20[14], Word   0[00], CodePt     32    [ ] space
  0xf0,  // Index  21[15], Word   1[01], CodePt     36    [$] dollar sign
  0xf0,  // Index  22[16], Word   2[02], CodePt     37    [%] percent sign
  0xf0,  // Index  23[17], Word   3[03], CodePt     58    [:] colon
  0xf0,  // Index  24[18], Word   4[04], CodePt     84    [T] uppercase T
  0xf0,  // Index  25[19], Word   5[05], CodePt    104    [h] lowercase h
  0xf0,  // Index  26[1a], Word   6[06], CodePt    105    [i] lowercase i
  0xf0,  // Index  27[1b], Word   7[07], CodePt    115    [s] lowercase s
  0xf0,  // Index  28[1c], Word   8[08], CodePt     97    [a] lowercase a
  0xf0,  // Index  29[1d], Word   9[09], CodePt     75    [K] uppercase K
  0xf0,  // Index  30[1e], Word  10[0a], CodePt    111    [o] lowercase o
  0xf0,  // Index  31[1f], Word  11[0b], CodePt    114    [r] lowercase r
  0xf0,  // Index  32[20], Word  12[0c], CodePt    101    [e] lowercase e
  0xf0,  // Index  33[21], Word  13[0d], CodePt    110    [n] lowercase n
  0xf0,  // Index  34[22], Word  14[0e], CodePt    112    [p] lowercase p
  0xf0,  // Index  35[23], Word  15[0f], CodePt    109    [m] lowercase m
  0xf0,  // Index  36[24], Word  16[10], CodePt  45208        nieun a
  0xf0,  // Index  37[25], Word  17[11], CodePt  48372        pieup o
  0xf0,  // Index  38[26], Word  18[12], CodePt  44592        kiyeok i
  0xf0,  // Index  39[27], Word  19[13], CodePt  44032        kiyeok a
  0xf0,  // Index  40[28], Word  20[14], CodePt  50669        ieung yeo kiyeok
  0xf0,  // Index  41[29], Word  21[15], CodePt  44200        kiyeok yeo
  0xf0,  // Index  42[2a], Word  22[16], CodePt  50892        ieung weo
  0xf0,  // Index  43[2b], Word  23[17], CodePt  49892        sios i rieul
  0xf0,  // Index  44[2c], Word  24[18], CodePt  46412        ssangtikeut ae
  0xf0,  // Index  45[2d], Word  25[19], CodePt  50640        ieung e
  0xf0,  // Index  46[2e], Word  26[1a], CodePt  45716        nieun eu nieun
  0xf0,  // Index  47[2f], Word  27[1b], CodePt  47568        mieum a rieul
  0xf0,  // Index  48[30], Word  28[1c], CodePt  50630        ieung eo pieupsios
  0xf0,  // Index  49[31], Word  29[1d], CodePt  51060        ieung i
  0xf0,  // Index  50[32], Word  30[1e], CodePt  44256        kiyeok o
  0xf0,  // Index  51[33], Word  31[1f], CodePt  45236        nieun ae
  0xf0,  // Index  52[34], Word  32[20], CodePt  46300        tikeut eu
  0xf0,  // Index  53[35], Word  33[21], CodePt  47532        rieul i
  0xf0,  // Index  54[36], Word  34[22], CodePt  50864        ieung u
  0xf0,  // Index  55[37], Word  35[23], CodePt  45796        tikeut a
```

```
  0xf0,  // Index  56[38], Word  36[24], CodePt  50689        ieung yeo ieung
  0xf0,  // Index  57[39], Word  37[25], CodePt  48320        pieup yeo nieun
  0xf0,  // Index  58[3a], Word  38[26], CodePt  50557        ieung ya kiyeok
  0xf0,  // Index  59[3b], Word  39[27], CodePt  49328        sios a nieun
  0xf0,  // Index  60[3c], Word  40[28], CodePt  51652        cieuc i nieun
  0xf0,  // Index  61[3d], Word  41[29], CodePt  45804        tikeut a rieul
  0xf0,  // Index  62[3e], Word  42[2a], CodePt  47000        rieul ae
  0xf0,  // Index  63[3f], Word  43[2b], CodePt  44867        ssangkiyeok o chieuch
  0xf0,  // Index  64[40], Word  44[2c], CodePt  50500        ieung a
  0xf0,  // Index  65[41], Word  45[2d], CodePt  47492        rieul eu mieum
  0xf0,  // Index  66[42], Word  46[2e], CodePt  46384        ssangtikeut a
  0xf0,  // Index  67[43], Word  47[2f], CodePt  44600        kiyeok i rieul
  0xf0,  // Index  68[44], Word  48[30], CodePt  49100        ssangpieup u
  0xf0,  // Index  69[45], Word  49[31], CodePt  50724        ieung o
  0xf0,  // Index  70[46], Word  50[32], CodePt     46    [.] period
  0xf0,  // Index  71[47], Word  51[33], CodePt  49884        sios i
  0xf0,  // Index  72[48], Word  52[34], CodePt  44152        kiyeok eo rieul
  0xf0,  // Index  73[49], Word  53[35], CodePt  51020        ieung eu mieum
  0xf0,  // Index  74[4a], Word  54[36], CodePt  45459        nieun o hieuh
  0xf0,  // Index  75[4b], Word  55[37], CodePt  51064        ieung i nieun
  0xf0,  // Index  76[4c], Word  56[38], CodePt  44536        kiyeok eu
  0xf0,  // Index  77[4d], Word  57[39], CodePt  51012        ieung eu rieul
  0xf0,  // Index  78[4e], Word  58[3a], CodePt  49324        sios a
  0xf0,  // Index  79[4f], Word  59[3b], CodePt  49104        ssangpieup u nieun
  0xf0,  // Index  80[50], Word  60[3c], CodePt  55176        hieuh i
  0xf0,  // Index  81[51], Word  61[3d], CodePt  51592        cieuc eu
  0xf0,  // Index  82[52], Word  62[3e], CodePt  47140        rieul yeo
  0xf0,  // Index  83[53], Word  63[3f], CodePt  48159        pieup a rieulpieup
  0xf0,  // Index  84[54], Word  64[40], CodePt  50741        ieung o pieup
  0xf0,  // Index  85[55], Word  65[41], CodePt  49548        sios o
  0xf0,  // Index  86[56], Word  66[42], CodePt  49436        sios eo
  0xf0,  // Index  87[57], Word  67[43], CodePt  51453        cieuc u kiyeok
  0xf0,  // Index  88[58], Word  68[44], CodePt  50612        ieung eo
  0xf0,  // Index  89[59], Word  69[45], CodePt  46020        tikeut o
  0xf0,  // Index  90[5a], Word  70[46], CodePt  45768        nieun i
  0xf0,  // Index  91[5b], Word  71[47], CodePt  45576        nieun u nieun
  0xf0,  // Index  92[5c], Word  72[48], CodePt  47932        mieum u rieul
  0xf0,  // Index  93[5d], Word  73[49], CodePt  55128        hieuh eu rieul
  0xf0,  // Index  94[5e], Word  74[4a], CodePt     77    [M] uppercase M
  0xf0,  // Index  95[5f], Word  75[4b], CodePt    121    [y] lowercase y
  0xf0,  // Index  96[60], Word  76[4c], CodePt     70    [F] uppercase F
  0xf0,  // Index  97[61], Word  77[4d], CodePt    103    [g] lowercase g
  0xf0,  // Index  98[62], Word  78[4e], CodePt    116    [t] lowercase t
}; // END OF FONT_L_PACK_NAME_UNKNOWN
```

```
//*****************************************************************************
//*****         Character ROM - Font Upper Section (RAM Middle)      *****
//*****************************************************************************
UCHAR CONSTANT FONT_M_PACK_NAME_UNKNOWN[] = {
  0xf2, 0x04, // Burst Length 1266 (0x4f2)
  0x0f, 0x14, // SPI Command
  // Total of 79 glyphs in this pack
  // Loaded into FONT RAM locations 20 (0x14) to 98 (0x62)
  // Pixel Data, Upper Half (Char RAM Middle) Character Set: ASCII
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x03,0x00,0x3f,0xf0,0xff,0xfc,0xf3,0x3c,0x03,0x3c,0x03,0x3c,0x0f,0xf0,0x3f,0xc0,
0x00,0xfc,0x03,0xcf,0xc3,0xcf,0xf0,0xfc,0x3c,0x00,0x0f,0x00,0x03,0xc0,0x00,0xf0,
0x00,0x00,0x00,0x00,0x00,0x00,0x03,0xf0,0x03,0xf0,0x03,0xf0,0x00,0x00,0x00,0x00,
0xff,0xff,0xff,0xff,0x03,0xc0,0x03,0xc0,0x03,0xc0,0x03,0xc0,0x03,0xc0,0x03,0xc0,
0x00,0x3c,0x00,0x3c,0x00,0x3c,0x00,0x3c,0x3f,0xfc,0xff,0xfc,0xf0,0x3c,0xf0,0x3c,
0x0f,0x00,0x0f,0x00,0x00,0x00,0x00,0x00,0x0f,0x00,0x0f,0x00,0x0f,0x00,0x0f,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x3f,0xf0,0xff,0xfc,0xf0,0x3c,0x03,0xfc,
```

(etc…)

```
//*****************************************************************************
//*****************************************************************************
//*****                                                                   *****
//*****               Ordinary Wheel String Buffer                        *****
//*****                                                                   *****
//*****************************************************************************
//*****************************************************************************
#define NWHEEL (157)
UCHAR CONSTANT WHEEL_KOREAN_TEST[] = {
  0x18,0x19,0x1a,0x1b,0x14,0x1a,0x1b,0x14,0x1c,0x14,0x1d,0x1e,0x1f,0x20,0x1c,0x21,0x14,
  0x22,0x1e,0x20,0x23,0x17,0x24,0x14,0x25,0x26,0x27,0x14,0x28,0x29,0x2a,0x27,0x2b,0x14,
  0x2c,0x2d,0x2e,0x2f,0x30,0x31,0x14,0x32,0x31,0x14,0x25,0x33,0x14,0x34,0x35,0x36,0x35,
  0x37,0x38,0x39,0x2d,0x14,0x3a,0x3b,0x3c,0x3d,0x3e,0x3f,0x40,0x41,0x14,0x42,0x37,0x14,
  0x27,0x2b,0x14,0x43,0x2d,0x14,0x44,0x35,0x45,0x35,0x37,0x46,0x27,0x47,0x2e,0x14,0x48,
  0x49,0x14,0x48,0x49,0x4a,0x4b,0x14,0x4c,0x14,0x3f,0x4d,0x4e,0x4f,0x50,0x14,0x51,0x52,
  0x14,0x53,0x32,0x14,0x27,0x47,0x54,0x55,0x56,0x46,0x24,0x14,0x25,0x26,0x27,0x14,0x28,
  0x29,0x2a,0x27,0x2b,0x14,0x2c,0x2d,0x2e,0x57,0x58,0x59,0x14,0x40,0x5a,0x14,0x5b,0x5c,
  0x14,0x5d,0x35,0x45,0x35,0x37,0x46,0x5e,0x1c,0x5f,0x1c,0x14,0x60,0x1a,0x61,0x19,0x62,
  0x1a,0x21,0x61,0x46,
}; // END OF WHEEL_KOREAN_TEST
```

Another tool similar tool which is the same as K2A, but this program demonstrates the Johab construction method of making syllables. This program reads the same Unicode input file, and prepares the font and string packs.

```
j2a text-file ascii-bdf-file jamo-bdf-file
```

There are two tools I wrote to visualize the font and string packs on the desktop, without needing the actual hardware. These are called FONT_DISPLAY and STRING_DISPLAY. They are invoked by re-compiling with the desired font packs, and they generate a series of .PNG image files. Here is the output from the Korean poem test above (shown here zero-based):

Font Pack Display

String Pack Display

Note that these methods, using pre-formed font packs, can (and do) save font memory by taking advantage of repeating syllables within the input text. You can see this above – the string is longer than the font table, because repeating syllables and letters are only stored once.

Miscellaneous functions were prepared as needed. One such example are modules that process UTF-8 and UTF-16 input streams. These might not specifically be needed on the desktop, but are useful in the embedded application.

Another tool I made involves Romanization. In general, I really hate Romanization of Hangul, but in this project I found one useful reason to use it. I don't have my text editors and command windows configured to display Hangul (I briefly tried without success). If I want to visually see Hangul in the source code or debugging notes, I have to use ASCII only. To that end, I made a few tools to help out. One is simple, a table of syllable names (I didn't write these, but downloaded them and converted easily into C data table). For example, here are a few entries from each table:

```
#define MAX_HANGUL (11172)
const char *hangul_name[MAX_HANGUL] = {
    "kiyeok a",
    "kiyeok a kiyeok",
    "kiyeok a ssangkiyeok",
    "kiyeok a kiyeoksios",
    "kiyeok a nieun",
    "kiyeok a nieuncieuc",
    "kiyeok a nieunhieuh",
    "kiyeok a tikeut",
    "kiyeok a rieul",
    "kiyeok a rieulkiyeok",
    "kiyeok a rieulmieum",
    "kiyeok a rieulpieup",
    "kiyeok a rieulsios",
    "kiyeok a rieulthieuth",
    "kiyeok a rieulphieuph",
    "kiyeok a rieulhieuh",

          etc…
```

Similarly, I have a table of ASCII and Johab names:

```
const char *ascii_name[MAX_ASCII] = {
    "NUL null character ",
    "SOH start of header ",
    "STX start of text ",
    "ETX end of text ",
    "EOT end of transmission ",
    "ENQ enquiry ",
    "ACK acknowledge ",
    "BEL bell (ring) ",
    "BS backspace ",
    "HT horizontal tab ",
    "LF line feed ",
    "VT vertical tab ",
    "FF form feed ",
    "CR carriage return ",
    "SO shift out ",
    "SI shift in ",
    "DLE data link escape ",
    "DC1 device control 1 ",
    "DC2 device control 2 ",

          etc…
```

```
#define MAX_JAMO_LEADING (20)
const char *jamo_name_leading
[MAX_JAMO_LEADING] = {
    "blank",            // 0
    "kiyeok",           // 1
    "ssangkiyeok",      // 2
    "nieun",            // 3
    "tikeut",           // 4
    "ssangtikeut",      // 5
    "rieul",            // 6
    "mieum",            // 7
    "pieup",            // 8
    "ssangpieup",       // 9
    "sios",             // 10
    "ssangsios",        // 11
    "ieung",            // 12
    "cieuc",            // 13
    "ssangcieuc",       // 14
    "chieuch",          // 15
    "khieukh",          // 16
    "thieuth",          // 17
    "phieuph",          // 18
    "hieuh",            // 19
};
```

```
#define MAX_JAMO_VOWEL (22)
const char *jamo_name_vowel
[MAX_JAMO_VOWEL] = {
    "blank",            // 0
    "a",                // 1
    "ae",               // 2
    "ya",               // 3
    "yae",              // 4
    "eo",               // 5
    "e",                // 6
    "yeo",              // 7
    "ye",               // 8
    "o",                // 9
    "wa",               // 10
    "wae",              // 11
    "oi",               // 12
    "yo",               // 13
    "u",                // 14
    "wo",               // 15
    "we",               // 16
    "wi",               // 17
    "yu",               // 18
    "eu",               // 19
    "ui",               // 20
    "i",                // 21
};
```

```
#define MAX_JAMO_TRAILING (28)
const char *jamo_name_trailing
[MAX_JAMO_TRAILING] = {
    "blank",            // 0
    "kiyeok",           // 1
    "ssangkiyeok",      // 2
    "kiyeoksios",       // 3
    "nieun",            // 4
    "nieuncieuc",       // 5
    "nieunhieuh",       // 6
    "tikeut",           // 7
    "rieul",            // 8
    "rieulkiyeok",      // 9
    "rieulmieum",       // 10
    "rieulpieup",       // 11
    "rieulsios",        // 12
    "rieulthieuth",     // 13
    "rieulphieuph",     // 14
    "rieulhieuh",       // 15
    "mieum",            // 16
    "pieup",            // 17
    "pieupsios",        // 18
    "sios",             // 19
    "ssangsios",        // 20
    "ieung",            // 21
    "cieuc",            // 22
    "chieuch",          // 23
    "khieukh",          // 24
    "thieuth",          // 25
    "phieuph",          // 26
    "hieuh",            // 27
};
```

One tool I prepared was a function to Romanize random Korean text, using the South Korean Government's Proclamation No. 2000-8 standard "The Revised Romanization of Korean". I didn't implement every single feature, as these weren't made very clear in the standard. But the major points were illustrated in a table, which was fairly easy to put into an algorithm. As an example, consider the above poem sample text. Converting that into Romanization using my algorithm results in the following:

Original Text	My Romanization	Online Romanization Tool
나 보기가 역겨워 가실 때에는 말없이 고이 보내 드리우리다 영변에 약산 진달래꽃 아름 따다 가실 길에 뿌리오리다. 가시는 걸음 걸음 놓인 그 꽃을 사뿐히 즈려 밟고 가시옵소서. 나 보기가 역겨워 가실 때에는 죽어도 아니 눈물 흘리오리다.	na bo gi ga yeok gyeo wo ga sil ttae e neun mar eops i go i bo nae deu ri u ri da yeong byeon e yak san jin dal lae kkoch a reum tta da ga sil gir e ppu ri o ri da. ga si neun geor eum geor eum noh in geu kkoch eul sa ppun hi jeu ryeo balp go ga si op so seo. na bo gi ga yeok gyeo wo ga sil ttae e neun jug eo do a ni nun mul heul li o ri da.	na bo gi ga yeok gyeo wo ga sir ttae e neun mar eops i go i bo nae deu ri u ri da yeong byeon e yak san jin dal lae kkoc a reum tta da ga sir gir e ppu ri o ri da. ga si neun geor eum geor eum noh in geu kkoc eul sa ppun hi jeu ryeo balp go ga si op so seo . na bo gi ga yeok gyeo wo ga sir ttae e neun jug eo do a ni nun mur heul li o ri da.

There are a few differences between my results and the online engine, which I can explain by how we interpreted the rules differently. For example, I convert "가실 때…" as GA SIL TTAE…, but the online engine has GA SIR TTAE…. When a ending ㄹ is followed by a ㄸ, it should change to an R. However, as I understand the rules, this happens only when the two syllables are in the **same word**. In the poem, 가실 is a separate word from 때에는. If my interpretation is wrong, it is a simple matter to correct.

The last tool for font processing is the implementation of the Johab construction algorithm. I found a version of the algorithm in a perl script written by Mr. Shin Jung-shik at Yale University, and modified slightly over the years by Mr. Czyborra and Mr. Hardy. I implemented this algorithm in C-language for use by the embedded microprocessor.

In addition to making the font collection, the GNU Unifont project has introduced a very convenient ASCII format for bitmapped font files, which can be edited, merged, viewed, and converted to BDF formats with a few small utilities. These are `hex2pdf.pl`, `hexdraw.pl`, `hexmerge.pl`, and `bdfimplode.pl`. A simple example will demonstrate the power of these tools. I took Unifont's

Hangul Unicode syllable set of 11,172 glyphs and converted them to this hex format with the following command

```
E:> bdfimplode uni-iyagi16.bdf >iyagi.hex
```

Here a few lines from the hex file, to give an example of how simple the format is:

```
AC00:0000003800183F9801980198031F06180C1830180018001800180018001800100000
AC01:000000383F9801980198031F06180C18301800100FF8001800180018001800100000
AC02:000000383F9801980198031F06180C18301800101FF8031803180318021800100000
AC03:000000383F9801980198031F06180C18301800101F3003300338036C02CC0000
AC04:000000383F9801980198031F06180C1830180010070003000300030001FC0000
AC05:000000383F9801980198031F06180C18301800101CFC0C300C300C7807CC0000
AC06:000000383F9801980198031F06180C18301800101C300CFC0C780CCC07780000
AC07:000000383F9801980198031F06180C18301800100FF806000600060003F80000
AC08:000000383F9801980198031F06180C18301800100FF800180FF80C0007F80000
AC09:000000383F9801980198031F06180C18301800101F7803181F1818180F100000

                                    etc…
```

Say I want to quickly change one syllable for some testing reason. Consider the syllable 한, located at Unicode point U+D55C. I can quickly fetch this syllable as follows:

```
E:> grep “^D55C” iyagi.hex
D55C:00000C387F981E183318331F33181E1800180010070003000300030001FC0000
E:> grep “^D55C” iyagi.hex | hexdraw
D55C:   ----------------
        ----##----###---
        -########--##---
        ---####----##---
        --##--##---##---
        --##--##---#####
        --##--##---##---
        ---####----##---
        -----------##---
        -----------#----
        -----###--------
        ------##--------
        ------##--------
        ------##--------
        -------#######--
        ----------------

E:> grep “^D55C” iyagi.hex | hexdraw > han.txt
```

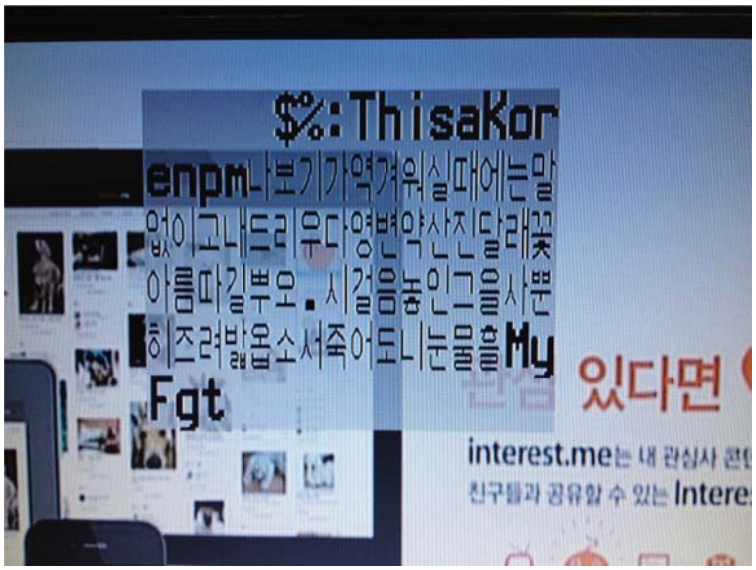
Then suppose I edit the simple text file han.txt, say to make the symbol extra bold, it can be easily put back into the font pack as follows:

```
E:> type han2.txt
D55C:   ----##----------
        -########-####--
        ---####----###--
        --######---###--
        -###--###--#####
        -###--###--#####
        -###--###--###--
        --######---###--
        ---####----###--
        -----------##---
        -######----#----
        --####----------
        --####----------
        --#############--
        ---############--
        ----------------
E:> hexdraw han2.txt > han2.hex            // convert back to hex
E:> type han*.hex                          // observe the difference
D55C:00000C387F981E183318331F33181E18001800100700030003000300030001FC0000
D55C:0C007FBC1E1C3F1C739F739F739C3F1C1E1C00187E103C003C003FFC1FFC0000
E:> grep –v "^D55C" iyagi.hex > old.hex    // remove old symbol
E:> hexmerge han2.hex old.hex > iyagi.hex  // merge back new symbol
E:> hex2bdf iyagi.hex > iyagi.bdf          // make updated bdf file
```

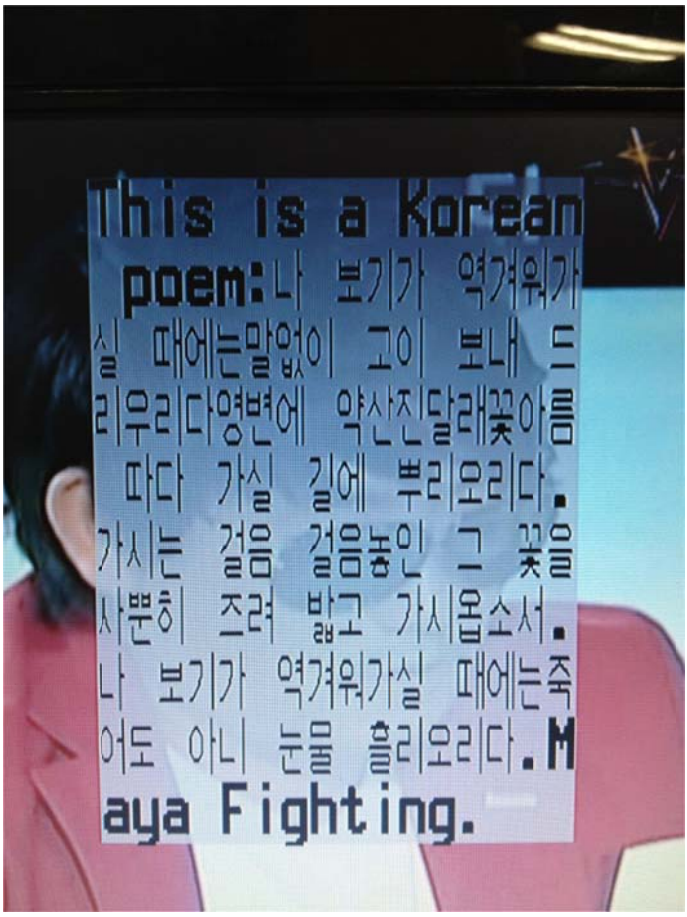
Obviously one wouldn't want to hand edit thousands of syllables like this, but it's a very convenient tool for debugging and testing. Although currently most of my offline font processing is with the BDF format, I will be upgrading add this new, much simpler format. Furthermore, I will definitely use this format to improve the uploading and download of fonts and strings within the embedded micro command-line debugging code.

Example Displays

So, what does it look like? Here are a few photos of the screen using the example of the above Korean poem. Here is the font table:



And here is the "string" containing the poem:



Notes and References

There is a lot of information on the internet, but I found these sites to be the most helpful. I'm sure you can find similar or better material amongst the Korean-language websites.

Wen Quan Yi (Spring of Letters) project
http://wenq.org/enindex.cgi

Original GNU Unicode Font page (by Roman Czyborra)
http://czyborra.com/unifont/

Current GNU Unifont page (by Paul Hardy)
http://unifoundry.com/unifont.html

- Page devoted to Hangul
  http://unifoundry.com/hangul/index.html
- Updated Perl script for building syllables from Johab letters
  http://unifoundry.com/hangul/johab2ucs2

Current Repository of Johab Bitmapped Fonts (at KAIST)
ftp://ftp.kaist.ac.kr/hangul/terminal/hanterm

A very thorough review of Hangul and Unicode (Dr. Gernot Katzer)
http://www.uni-graz.at/~katzer/korean_hangul_unicode.html

Dr. Kang's Korean Language Processing and Information Retrieval Laboratory
http://nlp.kookmin.ac.kr/

Study of Hangul Syllable Frequency
http://nlp.kookmin.ac.kr/data/syldown.html

"Glyph Bitmap Distribution Format(BDF) Specification"
Adobe Systems, Inc., Version 2.2, 22 March 1993
http://partners.adobe.com/public/developer/en/font/5005.BDF_Spec.pdf

These two books were also very helpful to understanding some of the background and history of Asian fonts in general (they are print books, but available online in PDF format)

"CJKV Information Processing, Second Edition", by Ken Lunde
O'Reilly Media, Inc., © 2009 ISBN-13: 978-0-596-51447-1

"Fonts and Encodings" by Yannis Haralambous
O'Reilly Media, Inc., © 2007 ISBN-13: 978-0-596-10242-5