AN1182

Fonts in the Microchip Graphics Library

Author: *Paolo Tamayo*
Pradeep Budagutta
Microchip Technology Inc.

INTRODUCTION

Embedded system application displays vary from complex devices, such as PDAs, cell phones and compact computers, to simple devices, such as home air-conditioner and security controllers, coffee makers and door entry keypads. Most of the simpler devices already have been adapted to use graphical displays previously found only in high-end devices, and with the price of displays continuing to fall, more and more simple devices will be using displays.

A common concern with simple devices is keeping down costs to improve market competitiveness. This requires reducing components, including memory and display sizes. As more functionalities are included in device designs, keeping down costs has become more challenging.

With today's global economy, products increase their sales by being sold in more geographies, but that requires the products to be adapted to other languages. For products with displays, that increases the functional requirements and compounds the problem of keeping costs down.

The character sets that produce the display's font images are central to the cost problem. While English and most other European languages can be handled with a 256-character set, Chinese, Japanese, Korean and other languages require many more characters. In some cases, the character set can be in the thousands which significantly increases a systems' memory requirements.

For simple low-cost devices, market economics make it impossible to provide individual functionality for every character in languages with such large character sets.

The Microchip Graphics Library solves this problem by creating font images that contain only the characters that an application will be using. That significantly reduces the amount of system memory required for fonts.

This application note describes the format of the Microchip Graphics Library's font image. It also tells how to reduce the number of characters in a font and automate the creation of the character arrays referring to an application's strings.

This document's examples are applicable to 16-bit and 32-bit PIC® microcontrollers with peripherals capable of connecting to display devices, such as the Parallel Master Port (PMP) or Graphics Controller (GFX), and which are supported by the Microchip Graphics Library.

For an overview of the library's architecture and uses, refer to the application note *"How to Use Widgets in Microchip Graphics Library" (*AN1136*)*. For more information, see the library's Help documentation that comes with the library software. (The library can be downloaded from www.microchip.com/graphics.)

FONT IMAGE FORMAT

The Microchip Graphics Library uses the image structure for fonts shown in Table 1.

> **Note:** The format shown in Table 1 is compatible with the Microchip Graphics Library Version 2.00 or higher.

TABLE 1: FONT TABLE FORMAT

<table>
<tr><th>Byte Offset</th><th>Name</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="18">Font Header</td></tr>
<tr><td>0x00</td><td></td><td colspan="2">Reserved</td><td colspan="2">Orient</td><td colspan="2">Bits per Pixel</td><td>Reserved</td><td>Extended Glyph</td><td colspan="8">Font ID</td></tr>
<tr><td>0x02</td><td></td><td colspan="16">First Character ID</td></tr>
<tr><td>0x04</td><td></td><td colspan="16">Last Character ID</td></tr>
<tr><td>0x06</td><td></td><td colspan="16">Height[2]</td></tr>
<tr><td colspan="18">Character Table[1]</td></tr>
<tr><td>0x08</td><td colspan="17">1st Character's Glyph Entry</td></tr>
<tr><td>0x08 + W</td><td colspan="17">2nd Character's Glyph Entry</td></tr>
<tr><td>…</td><td>…</td><td colspan="8">…</td><td colspan="8">…</td></tr>
<tr><td>0x08 + mW</td><td colspan="17">Last Character's Glyph Entry</td></tr>
<tr><td colspan="18">Font Bit Maps</td></tr>
<tr><td>n^1</td><td>1st Character</td><td colspan="16">1st Character Bit Map Data</td></tr>
<tr><td>n^2</td><td>2nd Character</td><td colspan="16">2nd Character Bit Map Data</td></tr>
<tr><td>.
.
.</td><td>.
.
.</td><td colspan="16">.
.
.</td></tr>
<tr><td>n^L</td><td>Last Character</td><td colspan="16">Last Character Bit Map Data</td></tr>
</table>

Legend: W = Size of the Glyph Entry (Four bytes for Normal Glyph or 12 bytes for Extended Glyph); m = (Last Character ID – First Character ID); n^1, n^2 ... n^L = The character's byte offsets, from Font Header start to the beginning of the character's bit map data, defined by the offset value in the Glyph Entry for the character found in the Character Table.

Note 1: The Character Table consists of a series of either: Normal Glyph Entries (Table 2) or Extended Glyph Entries (Table 3). The type of Glyph Entry is determined by the "Extended Glyph" bit in the Font Header.

2: In Microchip Graphics Library, versions earlier than v3.04, an 8-bit wide height was used.

© 2012 Microchip Technology Inc.

GLYPH ENTRY

There are two types of Glyph Entry formats. The Normal format, which specifies the offset of the glyph (character's bit map) from the Font Header start. The Extended format, which not only specifies the offset but also the dimension and relative position of the glyph in relation to the previous glyph, as explained below.

In the commonly used character maps, such as Latin (English, German, French, etc.), Japanese, Chinese, Russian, Korean, etc., each character code defines a unique alphabet. Therefore, while rendering a text, each glyph can be placed next to each other; a simple Glyph Entry format is sufficient, as shown in Table 2.

For certain character maps used in Asia, such as Hindi, Thai, etc., multiple character codes may be required to represent an alphabet, as shown in Figure 1. Here, multiple glyphs overlap each other to form a single alphabet and it is evident that the second character needs to be positioned relative to the first character, shown in Figure 1.

FIGURE 1: OVERLAPPING GLYPHS

Adjacent character codes	Normal Glyph Entry	Extended Glyph Entry
त + े	त े	ते

To accommodate this relative positioning information, Glyph Entry format, described in Table 2, is not sufficient and hence, an Extended format is required as shown in Table 3. Note that Extended Glyphs are applicable only for ANSI fonts and not for unicode fonts.

A font can use either Normal Glyphs or Extended Glyphs, but not both. However, an embedded project can have multiple fonts in which some of them use the Extended Glyphs and others use the Normal Glyphs.

Note:	The user can select either Normal Glyphs or Extended Glyphs during the selection of fonts in software tools, such as the Graphics Resource Converter (GRC). Character maps, which need the Extended Glyph, usually require the full range of characters (0-255) to be converted unless the font filtering technique is used (see the **Reducing Font Images** section).

TABLE 2: GLYPH ENTRY– NORMAL FORMAT

Byte Offset	Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	Glyph Entry	Offset <7:0>								Glyph Width							
0x02		Offset <23:16>								Offset <15:8>							

TABLE 3: GLYPH ENTRY – EXTENDED FORMAT

Byte Offset	Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	Glyph Entry Extended	Offset <15:0>															
0x02		Offset <31:16>															
0x04		Cursor Advance <15:0>															
0x06		Glyph Width <15:0>															
0x08		X Adjust <15:0>															
0x0A		Y Adjust <15:0>															

TABLE 4: FONT TABLE FIELDS

Field	Bits	Description
Font Header Block		
Orient	2	Font Orientation: `00` = Normal `01` = Characters rotated 270 degrees clockwise `10` = Characters rotated 180 degrees `11` = Characters rotated 90 degrees clockwise
Bits per Pixel	2	Specifies the Color Depth of Each Pixel in the Character Bit Map: `00` = 1 bit per pixel `01` = 2 bits per pixel (used for anti-aliased fonts, see the **Font Anti-Aliasing** section) `1x` = Reserved
Extended Glyph	1	Specifies the Glyph Entry Format: `0` = Normal Glyph Entry format `1` = Extended Glyph Entry format
Font ID	8	User-defined value.
First Character ID	16	Character ID for the first character in the font table.
Last Character ID	16	Character ID for the last character in the font table.
Height(4)	16	Height of all the characters in the font table in pixels.
Character Table Block		
Glyph Width	8(1)/16(2)	Width of a character in pixels excluding the padding bits.(3)
Offset	24(1)/32(2)	Byte offset from the start of the Font Header to the beginning of the character's bit map.
Cursor Advance	16	The value by which the cursor position has to be advanced after rendering this character (indicates the starting position of the next character). It may have a different value than Glyph Width and is font dependent.
X Adjust	16 (Signed)	The value by which the character's x position has to be adjusted with respect to the current cursor's x position. This is used when the next character overlaps the previous character and is font dependent.
Y Adjust	16 (Signed)	The value by which the character's y position has to be adjusted with respect to the current cursor's y position and is font dependent.

Note 1: For Normal Glyph Entry.
2: For Extended Glyph Entry.
3: In cases where the character width is not equal to a byte boundary, padding bits are required.
4: In Microchip Graphics Library, versions earlier than v3.04, an 8-bit wide height was used.

© 2012 Microchip Technology Inc.

The Font Header block defines the height of all the characters and the range from the first character to the last character entries. Users can use the header's Font ID field to group the fonts into appropriate classifications. IDs could be used to distinguish between bold and normal fonts. They can also be used to differentiate Chinese fonts from Japanese fonts. The Orient field defines the character bit map's orientation when the font table was created.

In the case of Normal Glyphs, the Character Table Block defines an array of character entries, each consisting of two 2-byte words. The second byte and second word of each entry is the 24-bit offset from the beginning of the image to the character bit map. The first byte of each entry is the character width.

For example, to locate the first character bit map of a font image structure, located at address 0x0040:

1. The 24-bit offset is obtained by concatenating the second byte with the third and fourth bytes (second word) of the first character offset entries from the Character Table, as shown:

   Offset<23:0> = Offset<23:16>:Offset<7:0>

2. The obtained, Offset<23:0>, is added to 0x0040, as shown:

   First Bit Map = Offset<23:0> + 0x0040

The first character's bit map will be located at the resulting address.

In the case of Extended Glyphs, the Character Table block defines an array of character entries, each consisting of six 16-bit words. The first two words of each entry are the 32-bit offset, from the beginning of the image to the character bit map. The location of the character bit map can be calculated in the same way as that of the Normal Glyphs, except that the offset is now 32 bits wide.

For example, to locate the first character bit map of a font image structure, located at address 0x0040:

1. The 32-bit offset is obtained by concatenating the two 16-bit words, as shown:

   Offset<31:0> = Offset<31:16>:Offset<15:0>

2. The obtained Offset<31:0> is added to 0x0040, as shown:

   First Bit Map = Offset<31:0> + 0x0040

The first character's bit map will be located at the resulting address.

In both cases, the number of character entries in the font image is calculated as $(\text{LastChar} - \text{FirstChar}) + 1$.

The font bit map block contains the character bit map definitions. Each character is stored as a contiguous set of bytes. Each pixel is represented by 1 bit for normal fonts or 2 bits for anti-aliased fonts (see **Font Anti-Aliasing**). The glyphs are considered as either 1-bit images for normal fonts or 2-bit images for anti-aliased fonts.

The (x, y) position for rendering is calculated as shown in Equation 1 and Equation 2. Where curX is the current X position and curY is the current Y position of the cursor. The (curX, curY) is modified after rendering the character, so that the next character can be rendered with respect to the new (curX, curY) position. Figure 1 shows the effect of Equation 1 and Equation 2 on glyph rendering.

EQUATION 1: RENDERING POSITION FOR NORMAL GLYPH

While rendering:

$x = \text{curX}$
$y = \text{curY}$

The character is rendered starting from this resultant (x, y) position.

After rendering:

$\text{curX} = \text{curX} + \text{Glyph Width}$
$\text{curY} = \text{curY}$

EQUATION 2: RENDERING POSITION FOR EXTENDED GLYPH

While rendering:

$x = \text{curX} + \text{xAdjust}$
$y = \text{curY} + \text{yAdjust}$

The character is rendered starting form this resultant (x, y) position.

After rendering:

$\text{curX} = \text{curX} + \text{Cursor Advance}$
$\text{curY} = \text{curY}$

Font Anti-Aliasing

Anti-aliasing is a technique used to make the edges of text appear smooth. This is useful, especially with characters such as ‘A’, ‘O’, etc., which have slanted or curved lines. Since the pixels of the display are arranged in rectangular fashion, slanted edges cannot be represented smoothly. To make them appear smooth, a pixel adjacent to the slanted pixels is painted with an average of the foreground and background colors, as shown in Figure 2.

FIGURE 2: FONT WITH ANTI-ALIASING

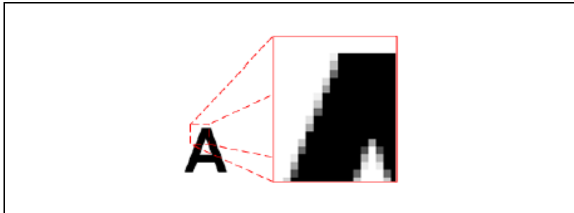
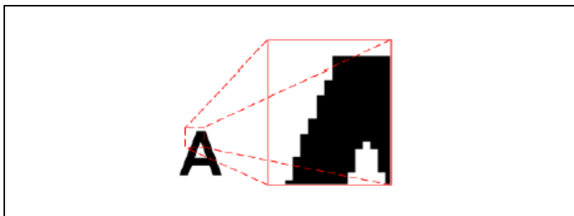


FIGURE 3: FONT WITH NO ANTI-ALIASING



When anti-aliasing is turned off, the pixels abruptly change from background color to foreground color, as shown in Figure 3.

To implement anti-aliasing, adjacent pixels transition from background to foreground color using 25% or 75% mid-color values. This font feature will require roughly twice the size of memory storage required for font glyphs with no anti-aliasing.

Since the average of foreground and background colors needs to be calculated at run-time, the rendering of anti-aliased fonts may take more time than rendering normal fonts. To optimize the rendering speed, the programmer can use the available macro, `GFX_Font_SetAntiAliasType`, where anti-alias type can be set to: `ANTIALIAS_OPAQUE` or `ANTIALIAS_TRANSLUCENT`.

- `ANTIALIAS_OPAQUE` (default after initialization of graphics) – Character pixel color is calculated once while rendering each character, which is ideal for rendering text over a constant background.
- `ANTIALIAS_TRANSLUCENT` – The new pixel color is calculated for every necessary pixel. This feature is useful while rendering text over an image or on a non-constant color background.

As a result, rendering anti-aliased text usually takes longer with the `ANTIALIAS_TRANSLUCENT` type as compared to the `ANTIALIAS_OPAQUE` type.

To use anti-aliasing, enable the compiler switch, `#define USE_ANTIALIASED_FONTS` in the `GraphicsConfig.h` file, and enable anti-aliasing in the Graphics Resource Converter (GRC) tool while selecting the font.

Note:	Even when anti-aliasing is enabled, normal fonts can also be used without the anti-alias effect.

© 2012 Microchip Technology Inc.

GOL Scheme

The library uses the fonts for the control widgets. Control widgets are the Graphical User Interface's (GUI) components that are manipulated with a mouse or keyboard. Each widget can have its own style scheme. The style scheme specifies the colors used to draw the widget, as well as the fonts used to label the widgets.

The font image is a style scheme component assigned to the widgets. A pointer is allocated in the style scheme to point to the font structure where a member of that structure is the pointer to the font image to be used. (In Example 1, the scheme's font structure pointer is shown in bold text.)

For more information on the font structure, refer to the *"Microchip Graphics Library Help"*, which is installed with the library.

This implementation permits the assignment of different fonts to widgets. Each widget could use several different font styles, or a group of widgets could use one style with another group using another style. This provides flexibility in the design of application screens.

This application note shows how different language fonts can be implemented into one application by switching the fonts and text assigned to a widget.

The locations of the fonts are transparent to the library. As long as a location can be addressed, it can be accessed by the graphic library's API. It does not matter if the font image is in the internal Flash, external Flash or RAM.

Throughout this document, the terms, font and font image, are used interchangeably. Both terms will be referring to the font image stored in memory.

EXAMPLE 1: GRAPHICS OBJECT LAYER (GOL) SCHEME STRUCTURE

```
typedef struct {
    WORD EmbossDkColor;          // Emboss dark color used for 3d effect.
    WORD EmbossLtColor;          // Emboss light color used for 3d effect.
    WORD TextColor0;             // Character color 0 used for objects that supports text.
    WORD TextColor1;             // Character color 1 used for objects that supports text.
    WORD TextColorDisabled;      // Character color used when object's state is disabled.
    WORD Color0;                 // Color 0 usually assigned to an Object state.
    WORD Color1;                 // Color 1 usually assigned to an Object state.
    WORD ColorDisabled;          // Color used when an Object is in a disabled state.
    WORD CommonBkColor;          // Background color used to hide Objects.
    void *pFont;                 // Pointer to the font structure assigned to the
                                 // style scheme.
    .......
    .......
} GOL_SCHEME;
```

MEMORY REQUIREMENTS

In most languages, 256 characters are enough to cover the character set of a font. Since the character glyphs are stored as bit maps, the size of the font will depend on the character height selected when the font images are created.

In the English language, usually a code range of 32-127 is enough to cover any application with texts and strings. A font using this range and having a character height of 24 pixels, using Normal Glyphs and without anti-aliasing, can occupy about 3.5 Kbytes of memory. The actual memory requirement will vary depending on the kind of font used, type of glyph table used, whether anti-aliasing is enabled and the height of the character.

In applications using fonts with more than 256 characters, the memory requirement becomes a challenge. Some Asian fonts have character sets numbering in the thousands. The complete character set of Simplified Chinese contains more than 50,000 characters. The approximate memory requirement for this character set, with the same 24-pixel height, is more than 2 Mbytes.

Since PIC® devices have limited internal Flash, sometimes it is not possible to store all fonts in the internal memory.

In some embedded systems, a font engine is used to optimize font operations. “Font engine” is a generic term for software that renders font images from algorithms or vector equations, and draws the lines or curves of the characters.

Storing the equations instead of the images makes the font scalable; it also reduces the memory requirements to tens of Kbytes. The drawback of this solution is that a significant amount of processing power and memory are needed to render the characters from equations; that consumes resources that could be used by the application code.

For this reason, font engines are common in high-end devices, but not in low-cost devices, where cost containment is important.

REDUCING FONT IMAGES

The problem of the font’s memory requirements, in limited resource environments, can be resolved by reducing those requirements. This can be done two ways:

- Reduce the range of characters being stored in memory
- Store only the characters that are going to be used

By defining a font character range, you can reduce the number of characters stored in memory.

The ASCII table defines a character set from 0 to 127, with the extended set ranging from 128 to 255. Instead of storing that complete character set, this approach defines a smaller range by designating a start and end character.

A range could be defined from Character Code 32 to 127 – from the space character to the delete character. The eliminated codes of 0 through 31 are control (non-printable) characters that do not produce symbols.

If the application uses only capital letters and the numbers 0-9, the range could be further reduced. But the numbers of the character codes must be sequential, and those letters and numbers are not contiguous, so unused character glyphs would have to be stored in memory.

As a result, this solution saves memory, but has the inefficiency of storing unused characters.

The second solution, storing only the characters that are going to be used, is more efficient and provides larger reductions in the font’s memory requirements.

In applications that use static strings, only the predefined set of character glyphs need to be stored in the font image. This eliminates all of the unused characters.

This solution, however, requires the character codes to be changed. That is because the graphics library expects the font character codes to be sequential, so a coding change is necessary.

If an application is tasked to display the string “HELLO WORLD!”, the normal C code is as shown in Example 2.

EXAMPLE 2: NORMAL CHARACTER ARRAY DECLARATION

```
char EngStr[] = “HELLO WORLD!”;
```

The C compiler automatically recognizes the string and replaces the characters with the character codes from the standard ASCII Character Table. Using a font that stores the complete character set (Codes 32 to 127), the application correctly displays the `EngStr[]` characters.

If the application uses a reduced character set, storing only the characters in the string, the C code must be changed to what is shown in Example 3.

EXAMPLE 3: MODIFIED CHARACTER ARRAY DECLARATION USING A REDUCED CHARACTER SET

```
char EngStr[] = {      0x24,0x23,0x25,0x25,
                       0x26,0x20,0x28,0x26,
                       0x27,0x25,0x22,0x21,
                       0x00};
```


© 2012 Microchip Technology Inc.

Except for the space and exclamation mark character, all the character codes have changed. The change is due to the reduction of the character set that has converted the old, numerically dispersed character codes into sequential code numbers, assigned according to the alphabetical order of the used letters.

The space character and exclamation mark character codes did not change because they are adjacent to each other in the original ASCII table, and the new, reduced character set has the space character as the first character.

Another reason for starting the character codes from 0x20 (or 32) is to avoid assigning a control character code to any printable character. Since the control character codes are standard, routines in the library may be using these standard control codes to format and display strings and characters.

The NULL string terminator, 0x00, must be included in the array to terminate the string – a task normally done for string literals by the compiler.

The graphics library can use single signed byte, single unsigned byte or two-byte characters. Selecting single byte or two-byte characters is done by defining `XCHAR`, as shown in Example 4. (The `XCHAR` definition is in the library file, `Primitive.h`.)

EXAMPLE 4: XCHAR DEFINITION

```
#if defined (USE_MULTIBYTECHAR)
    #define XCHAR unsigned short        // unsigned 2 bytes, up to 65536 characters
 #elif defined (USE_UNSIGNED_XCHAR)
    #define XCHAR unsigned char         // unsigned byte, up to 256 characters
#else
    define XCHAR   char                 // signed byte, up to 128 characters
#endif
```

For a byte character:

- `XCHAR` is defined as `char` or `unsigned char`.
- The string, “HELLO WORLD!”, must be coded using one of the following ways:
  - If the font used was generated using a character range that did not change the character codes, the string can be coded normally, as shown in Example 2.
  - If the font has changed the character codes, the new character codes for each character must be coded as shown in Example 3, using only `char` as the data type.

For a 2-byte character:

- `XCHAR` is defined as `unsigned short`.
- The string must be coded using one of the following ways:
  - If the font used was generated using a character range that did not change the character codes, the “HELLO WORLD!” string is coded as shown in Example 5.
  - If the font has changed the character codes, the new character codes for each character must be coded as shown in Example 6.

EXAMPLE 5: DECLARING A CHARACTER ARRAY WITH A MULTI-BYTE XCHAR DEFINITION

```
XCHAR EngStr [] = {'H','E','L','L','O',' ','W','O','R','L','D','!',0x0000};
```

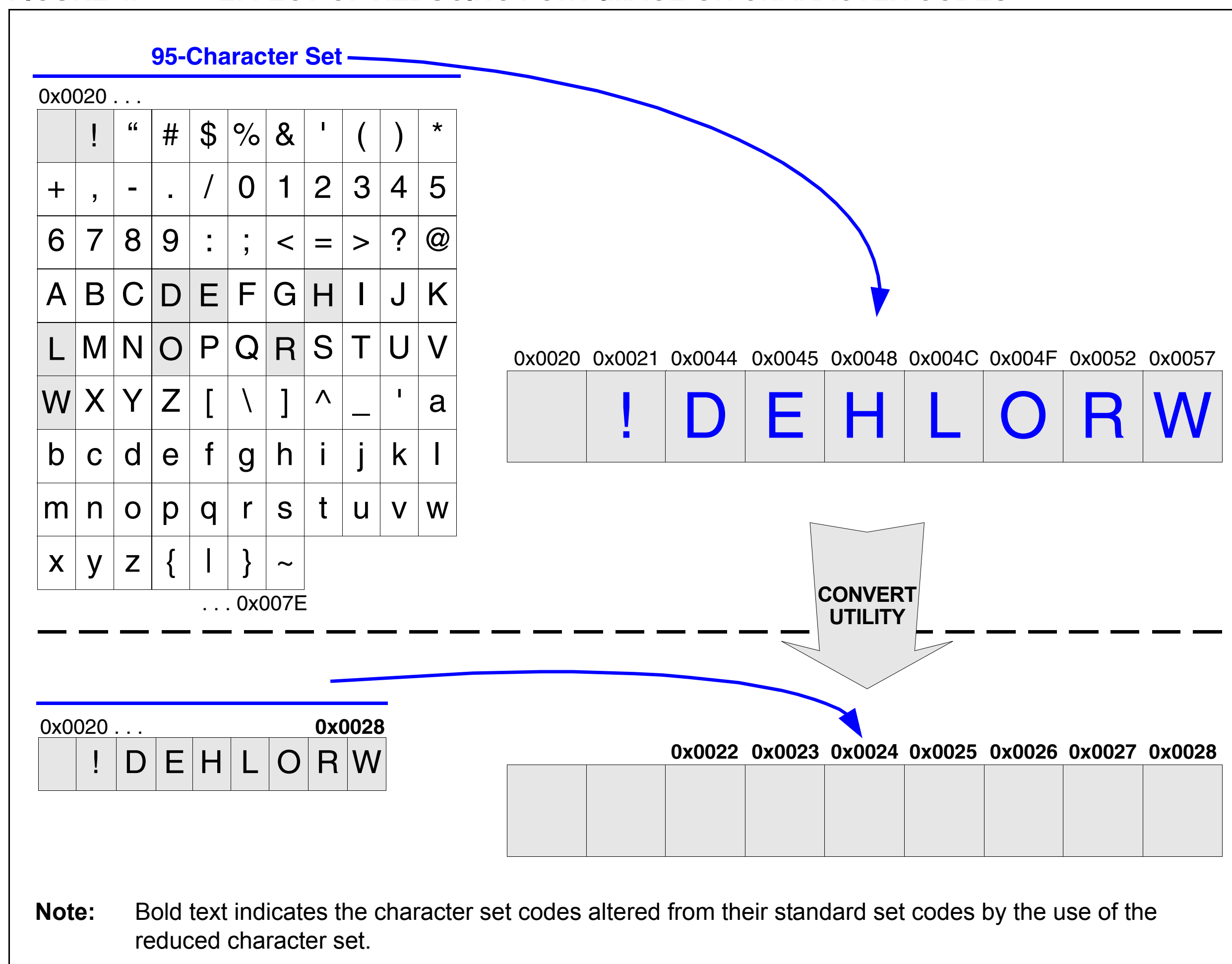
`USE_MULTIBYTECHAR` or `USE_UNSIGNED_XCHAR` can be defined in the file, `GraphicsConfig.h`. This file is required for any application using the Microchip Graphics Library.

EXAMPLE 6: MODIFIED XCHAR ARRAY DECLARATION USING A REDUCED CHARACTER SET

```
XCHAR EngStr[] = {     0x0024,0x0023,0x0025,0x0025,0x0026,0x0020,
                       0x0028,0x0026,0x0027,0x0025,0x0022,0x0021,
                       0x0000};
```

Figure 4 shows the transformation of the original font to the reduced character set form. The character codes in bold are the ones modified because of the use of a reduced font image.

FIGURE 4: EFFECT OF REDUCING FONT IMAGE ON CHARACTER CODES



Note: Bold text indicates the character set codes altered from their standard set codes by the use of the reduced character set.

The reduction of the font image and the arrays of converted character codes should be performed automatically by a software utility. This removes the error prone procedure of manually converting the original character codes to new values.

In the “HELLO WORLD!” example, the reduced font image required 90% less memory space. The character set was reduced from 95 (Codes 32 to 127) to nine.

As more characters are included in the reduced character set, the memory savings become negligible for European languages with small standard character sets. But for languages with thousands of characters, the memory savings from reduced character sets are very significant. Limited Flash memory can easily accommodate more than one font when font images are reduced to a few characters.

The extent of the memory savings for non-European languages can be demonstrated by translating the earlier example (“HELLO WORLD!”) into Japanese. The translation of the phrase is “みなさん こんにちは” with the character conversion utility producing the code shown in Example 7.

EXAMPLE 7: JAPANESE CHARACTER ARRAY DECLARATION USING REDUCED FONT IMAGE

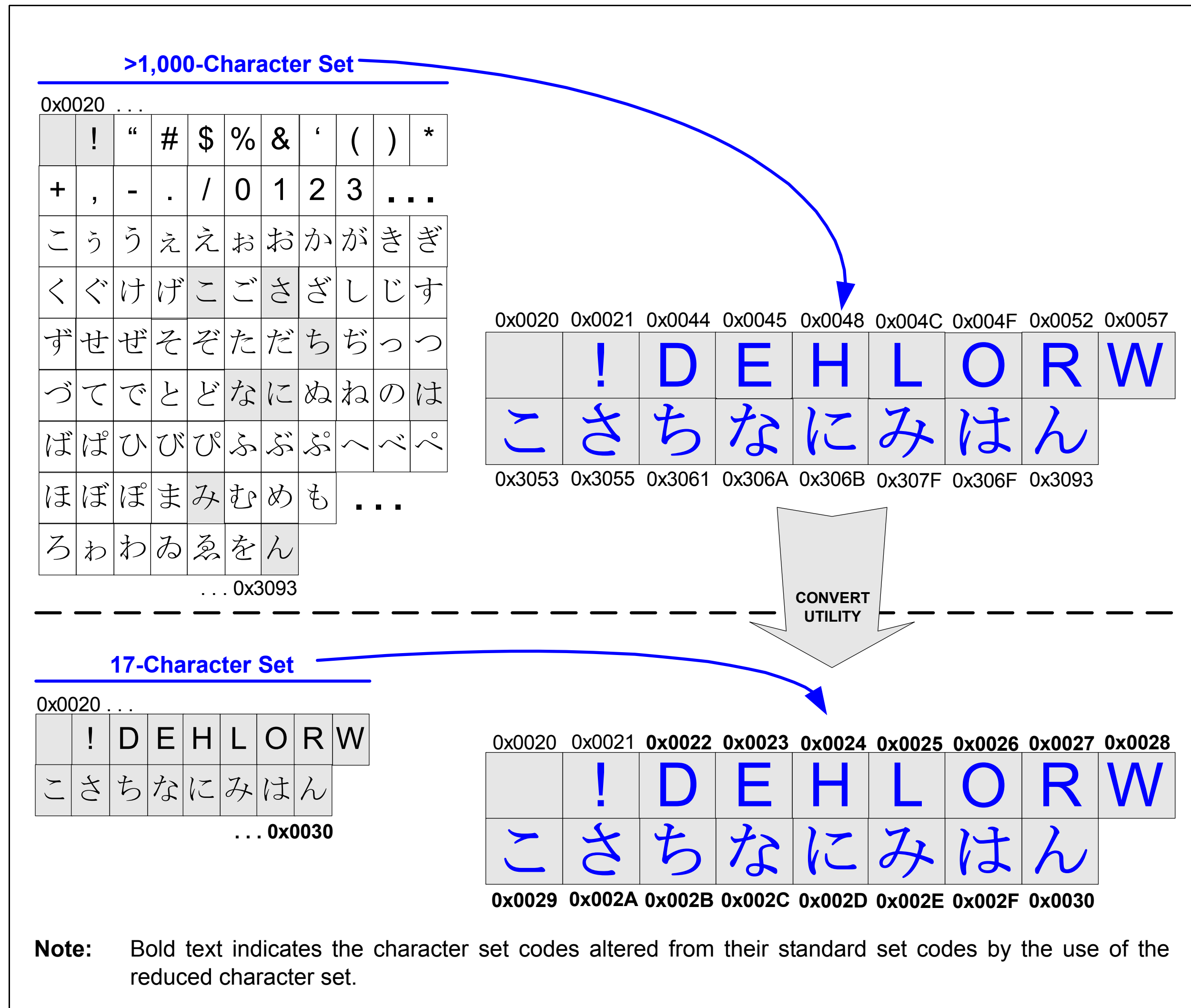
```
XCHAR JpnStr [] = {        0x002E, 0x002C, 0x002A, 0x0030, 0x0020,
                           0x0029, 0x0030, 0x002D, 0x002B, 0x002F,
                           0x0000};
```


© 2012 Microchip Technology Inc.

Figure 5 shows how the reduced character set utility produces the font image for the Japanese phrase.

Until now, the examples have been of only one font. If an application added a Chinese version of the same phrase, an additional font style would be needed and the conversion utility would have to generate another font encoding.

FIGURE 5: REDUCED FONT IMAGE WITH JAPANESE CHARACTERS



Note: Bold text indicates the character set codes altered from their standard set codes by the use of the reduced character set.

GENERATING REDUCED CHARACTER SETS

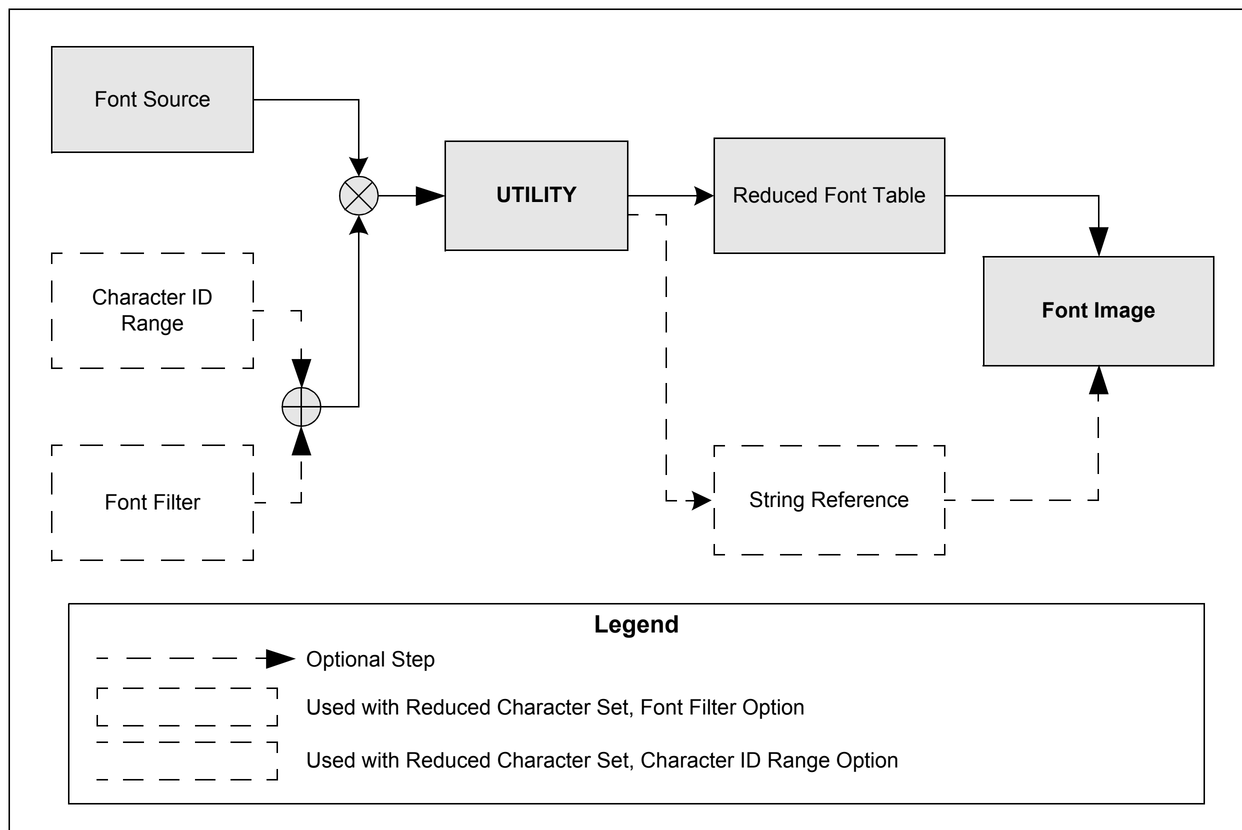
This section describes:

- The requirements for generating the reduced character sets
- The files needed to access character glyph bit maps for the generated font set

Generating a reduced character set for the Microchip Graphics Library requires character code translation. If a character set of more than 255 characters is being used, XCHAR must be defined as 2 bytes.

The inputs and outputs of the character code translation are shown in Figure 6.

FIGURE 6: GENERATING FONT IMAGE INPUTS AND OUTPUTS



Font Source – A font file or a font that is installed as part of the operating system.

Typefaces used by fonts are usually licensed. Before using the font, ensure that it is properly licensed.

Character Code Range – A user-defined setting that specifies the range of character codes to be used. This is set when converting with the utility.

Using this option means that the character's codes will not be changed. The Reduced Font Table will contain all the characters specified in the range.

Font Filter – A formatted text file (`*.txt`) containing all of the character array strings used in the application.

Using this option will change the character codes, so generation of a reference array is required. (See the **"String Reference"** section.)

Utility – The software that automates the generation of the string reference arrays whenever the font filter option is used to generate a reduced character font image.

For more information on the utility, see the documentation of the font and bit map converter that comes with the graphics library installer.

Reduced Font Table – The output file of the utility that contains the required characters, as specified by either the character code range or the font filter file.

String Reference – A set of C arrays used to reference strings defined in the filter. The arrays are created with the new character IDs of the font image. Application may use these arrays to refer to the strings.

Example 6 and Example 7 show how these character arrays will appear.

Font Image – The image of the characters or strings as they appear on the display screen.

© 2012 Microchip Technology Inc.

Using Font Filters

To illustrate the required inputs and generated outputs for reduced character sets, this section expands the “HELLO WORLD!” example by translating the phrase into different languages. Font filters are used to display the different languages’ translations on a screen.

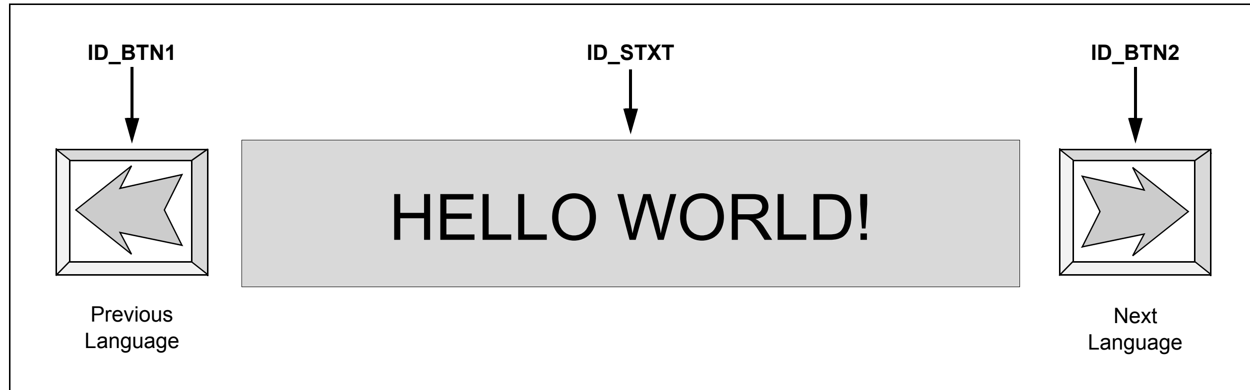
The screen display is driven by a PIC microcontroller with a static text widget specifying the text.

Two buttons are used to change the display’s translation to another language, moving sequentially, forward or backward, through a circular selection of the available languages.

Figure 7 shows the assigned codes for the two buttons and the static text widget.

For more information on code, see the code listings in **Appendix A: “Code Examples”**.

FIGURE 7: “HELLO WORLD” EXAMPLE OUTPUT



The available languages in this particular example include English, Chinese, Japanese, Korean, German, Dutch, Russian, Italian and French.

Table 5 lists the font sources for the different languages.

TABLE 5: SUMMARY OF FONTS USED

Language	Internet Source	Font Source
English	http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&cat_id=FontDownloads	Gentium
French		
Italian		
German		
Dutch		
Chinese	http://www.unifont.org/fontguide	FireFlySung
Japanese		
Hindi	http://salrc.uchicago.edu/resources/fonts/available/hindi/jaipur.shtml	Jaipur
Thai	http://www.thai-language.com/downloads/dbtt.ttf	DB ThaiText

The font filter file will be used to generate the reduced font image, following a format required by the graphics library utility. Since the Gentium font is defined as the graphics library’s default font, it will be used for the examples’ translations.

To generate the remaining translations, five font filter files must be created, one file for each translation.

The format of the font filter file is shown in Example 8.

EXAMPLE 8: FONT FILE FORMAT

```
StringName:    text in selected language              // comment
```

Example 9 through Example 12 show the font filter file contents for each of the translations of the "HELLO WORLD!" phrase.

> **Note:** When viewing this pdf file, it is possible that some characters will not show up properly if the proper fonts are not installed with the viewer.

EXAMPLE 9: CHINESE FONT FILTER FILE CONTENTS

```
ChineseStr:    你好世界！                    // In Chinese
```

EXAMPLE 10: JAPANESE FONT FILTER FILE CONTENTS

```
JapaneseStr:   みなさんこんにちは！            //In Japanese
```

EXAMPLE 11: HINDI FONT FILTER FILE CONTENTS

```
HindiStr:      नमस्ते विश्व!                    // In Hindi
```

EXAMPLE 12: THAI FONT FILTER FILE CONTENTS

```
ThaiStr:       สวัสดีชาวโลก!                   // In Thai
```

The filter file's colon (`:`) and two forward slashes (`//`) act as the delimeters. The utility parses the filter file and saves the `StringName` with the corresponding foreign text character codes. Based on these saved character codes, the reduced font image is created.

The `StringName` must follow the standard C coding guidelines for variable names because the output string reference header file will be using the same `StringName` for the generated character array of the string. The text after the `//` will be used as comments for the character array.

© 2012 Microchip Technology Inc.

Example 13 shows the utility's string reference file output for the previously shown font filter files.

EXAMPLE 13: STRING REFERENCE OUTPUT

```
/***************************************************************************
 * SECTION:  ChineseFont
 ***************************************************************************/

const XCHAR ChineseStr[] = {   0x0022, 0x0023, 0x0021, 0x0024, 0x0020, 0x0000    };    // In
Chinese

/***************************************************************************
 * SECTION:  JapaneseFont
 ***************************************************************************/

const XCHAR JapaneseStr[] = {   0x0027, 0x0024, 0x0022, 0x0028, 0x0021, 0x0028, 0x0025, 0x0023,
0x0026, 0x0020, 0x0000    };    // In Japanese

/***************************************************************************
 * SECTION:  HindiFont
 ***************************************************************************/

const XCHAR HindiStr[] = {   0x0024, 0x0023, 0x0028, 0x0025, 0x0020, 0x0027, 0x0020, 0x0022,
0x0026, 0x0029, 0x0026, 0x0021, 0x0000    };    // Hindi Font: Jaipur

/***************************************************************************
 * SECTION:  ThaiFont
 ***************************************************************************/

const XCHAR ThaiStr[] = {   0x0026, 0x0025, 0x0027, 0x0026, 0x0023, 0x0029, 0x0022, 0x0028,
0x0025, 0x002A, 0x0024, 0x0021, 0x0020, 0x0000    };    // Thai Font: DB Thai Text
```

As previously discussed and shown in the example, all of the array names are derived from the input font filter files. From this, the utility will generate five font images and the string arrays to be displayed.

For multiple strings, the font filter file contains all of the strings that will be using a particular font (see Example 14).

EXAMPLE 14: JAPANESE FONT FILTER FILE CONTENTS

```
JapaneseStr:     みなさんこんにちは！     // In Japanese
StringUsed2:     An example string 1.    // 2nd string example
StringUsed3:     An example string 2.    // 3rd string example
```

It is important to remember which strings were used to generate the reduced font image. Using the wrong font image will result in the display of an empty string or invalid data.

To cycle the display through the different translations using the library's GUI buttons, as shown in Figure 7, a linked list is created with the languages arranged in a ring. To do this, the desired structure is established and the linked list is initialized by a function (see Example 15).

© 2012 Microchip Technology Inc.

EXAMPLE 15: HELLO WORLD" LINKED LIST STRUCTURE

```
// structure used to rotate around the used fonts and "Hello World" strings
// HW acronym in the code represents Hello World data types, variables and
// definitions.
typedef struct {
    void     *pHWFont;            // pointer to the reduced font
    XCHAR    *pHWStr;             // pointer to the reference string
    void     *pHWPrev;            // pointer to the previous list member
    void     *pHWNext;            // pointer to the next list member
} HWDATA;

#define      HWDATAMAX 9          // number of translations

// array of structures that will hold the strings and its pointers to corresponding
// fonts.  this will be configured as a ringed linked list
HWDATA HWLang[HWDATAMAX];
// global pointer to the linked list.
HWDATA *pHWData;

    void     InitHWData(void)
{
    int      i;

  // Get all the translation of "Hello World" and store them into
  // the list.
  for(i = 0; i < HWDATAMAX; i++)
  {
    switch(i)
     {
    case 0:    HWLang[i].pHWFont = (void *) &GOLFontDefault; HWLang[i].pHWStr = (XCHAR *)EnglishStr; break;
    case 1:    HWLang[i].pHWFont = (void *) &ChineseFont; HWLang[i].pHWStr = (XCHAR *)ChineseStr; break;
    case 2:    HWLang[i].pHWFont = (void *) &JapaneseFont; HWLang[i].pHWStr = (XCHAR *)JapaneseStr; break;
    case 3:    HWLang[i].pHWFont = (void *) &GOLFontDefault; HWLang[i].pHWStr = (XCHAR *)ItalianStr; break;
    case 5:    HWLang[i].pHWFont = (void *) &GOLFontDefault; HWLang[i].pHWStr = (XCHAR *)GermanStr; break;
    case 6:    HWLang[i].pHWFont = (void *) &GOLFontDefault; HWLang[i].pHWStr = (XCHAR *)DutchStr; break;
    case 7:    HWLang[i].pHWFont = (void *) &GOLFontDefault; HWLang[i].pHWStr = (XCHAR *)FrenchStr; break;
    case 9:    HWLang[i].pHWFont = (void *) &HindiFont; HWLang[i].pHWStr = (XCHAR *)HindiStr; break;
    case 10:   HWLang[i].pHWFont = (void *) &ThaiFont; HWLang[i].pHWStr = (XCHAR *)ThaiStr; break;
    default:   break;
        }
 // make the list a ring list
        if(i == (HWDATAMAX - 1))
        {
            HWLang[i].pHWNext = (void *) &HWLang[0];
            HWLang[i].pHWPrev = (void *) &HWLang[i - 1];
        }
        else if(i == 0)
        {
            HWLang[i].pHWNext = (void *) &HWLang[i + 1];
            HWLang[i].pHWPrev = (void *) &HWLang[HWDATAMAX - 1];
        }
        else
        {
            HWLang[i].pHWNext = (void *) &HWLang[i + 1];
            HWLang[i].pHWPrev = (void *) &HWLang[i - 1];
        }
    }

    pHWData = &HWLang[0];
}
```

 © 2012 Microchip Technology Inc.

The static text is controlled by the two buttons. This is accomplished by using the message callback function (see Example 16).

EXAMPLE 16: STATIC TEXT CONTROL

```
WORD MsgAN1182Callback(WORD objMsg, OBJ_HEADER* pObj,
                       GOL_MSG* pMsg) {
        WORD objectID;
        STATICTEXT *pSt;

        objectID = GetObjID(pObj);
        switch (objectID) {
            case ID_BTN1:
                // check if button is pressed
                if (objMsg == BTN_MSG_RELEASED) {
                    pHWData = pHWData->pHWPrev;
                    // get pointer to static text
                    pSt = (STATICTEXT*) GOLFindObject(ID_STXT);
                    // set the new string to be displayed
                    StSetText(pSt, pHWData-> pHWStr);
                    // set the font for the string to be displayed
                    pSt->pGolScheme->pFont = pHWData-> pHWFont;
                    // set redraw state
                    SetState(pSt, ST_DRAW);
                }
                break;
            case ID_BTN2:
                if (objMsg == BTN_MSG_RELEASED) {
                    pHWData = pHWData->pHWNext;
                    // get pointer to static text
                    pSt = (STATICTEXT*) GOLFindObject(ID_STXT);
                    // set the new string to be displayed
                    StSetText(pSt, pHWData-> pHWStr);
                    // set the font for the string to be displayed
                    pSt->pGolScheme->pFont = pHWData-> pHWFont;
                    // set redraw state
                    SetState(pSt, ST_DRAW);
                }
                break;
                default:   break;
            }
            return 1;
}
```

CONCLUSION

To avoid memory related costs and font size inefficiencies that can be associated with displays' font images, the Microchip Graphics Library generates font images with reduced character sets. By doing this, all unused characters are eliminated from the font image. This allows simple embedded designs to integrate more functionality into very limited memory systems and easily adapt their application to different languages.

REFERENCES

For additional information on fonts in the Microchip Graphics Library, refer to following application notes:

- *AN1136, "How to Use Widgets in Microchip Graphics Library"* (DS01136), P. Tamayo and A. Alkhimenok, Microchip Technology Inc., 2007.
- *"Microchip Graphics Library Help"* (`Microchip Graphics Library Help.chm`), Microchip Technology Inc., 2008.
- "*Graphics Resource Converter Help*" (`Graphics Resource Converter Help.chm`), Microchip Technology Inc., 2011.

© 2012 Microchip Technology Inc.

APPENDIX A: CODE EXAMPLES

The listed items are example application code files that are part of an application code installed with the Microchip Graphics Library. (Refer to the "<install directory>/Graphics/AppNotes" subdirectory after installing the Microchip Application Libraries which can be downloaded from the web page: http://www.microchip.com/MAL.) Relevant files to run the demo are given on the indicated page. The `AppNotes` code runs on various development boards supported by Microchip Graphics Library. Refer to the "Microchip Graphics Library Help" file or the Application Note documentation link, located in the given directory, for the list of supported development boards.

The Application Note's main file incorporates the "Hello World" demo code that translates the "HELLO WORLD!" phrase into different languages, as shown in Example 9 through Example 14.

For the application code, refer to the `AppNotes` Demo available along with the Microchip Graphics Library.

NOTES:

© 2012 Microchip Technology Inc.

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PIC32 logo, rfPIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MTP, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

Analog-for-the-Digital Age, Application Maestro, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Omniscient Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, REAL ICE, rfLAB, Select Mode, SQI, Serial Quad I/O, Total Endurance, TSHARC, UniWinDriver, WiperLock, ZENA and Z-Scale are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

GestIC and ULPP are registered trademarks of Microchip Technology Germany II GmbH & Co. & KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2012, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.

ISBN: 978-1-62076-569-2

QUALITY MANAGEMENT SYSTEM CERTIFIED BY DNV

═ ISO/TS 16949 ═

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Osaka
Tel: 81-66-152-7160
Fax: 81-66-152-9310

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

11/29/11

© 2012 Microchip Technology Inc.