

---

## Fonts in the Microchip Graphics Library

---

<i>Author: Paolo Tamayo Microchip Technology Inc.</i>
---

### 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<sup>®</sup> microcontrollers with peripherals capable of connecting to display devices, such as the Parallel Master Port (PMP), and which are supported by the Microchip Graphics Library.

For an overview of the library's architecture and uses, see AN1136, *"How to Use Widgets in Microchip Graphics Library"*. For further details, see the library's Help documentation that comes with the library software. (The library can be downloaded from [www.microchip.com/graphics](http://www.microchip.com/graphics).)

## FONT IMAGE FORMAT

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

**TABLE 1: IMAGE STRUCTURE FOR FONTS**

Block	Char	Name	# of Bits	Description
Font Header		(Reserved)	8	Reserved for future use (must be set to '0')
		Font ID	8	User-assigned ID number
		First Char	16	Character code of the first font character (for example, 32)
		Last Char	16	Character code of the last font character (for example, 3006)
		(Reserved)	8	Reserved for future use (must be set to '0')
		Height	8	Character height in pixels
Character Table	First Character	Offset MSb	8	Offset from the start of font image (8 MSbs)
		Width	8	Width in pixels
		Offset LSb	16	Offset from the start of font image (16 LSbs)
	Second Character	Offset MSb	8	Offset from the start of font image (8 MSbs)
		Width	8	Width in pixels
		Offset LSb	16	Offset from the start of font image (16 LSbs)
	...	...	...	...
	Last Character	Offset MSb	8	Offset from the start of font image (8 MSbs)
		Width	8	Width in pixels
		Offset LSb	16	Offset from the start of font image (16 LSbs)
Font Bitmap	First Character		†	Bitmap data of first character
	Second Character		†	Bitmap data of second character
	...	...	...	...
	Last Character		†	Bitmap data of last character

† Character bitmap bit sizes vary with the character width.

The Font Header block defines the height of all the characters and the range from the First Character to 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 Character Table block defines an array of character entries, each consisting of two two-byte words. The first byte and second word of each entry is the 24-bit offset from the beginning of the image to the character bitmap. The second byte of each entry is the character width.

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

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

$$\text{Offset}[23:0] = \text{Offset MSb}[7:0]:\text{Offset LSb}[15:0]$$

2. The obtained Offset[23:0] is added to 0x0040, as shown.

$$\text{First bitmap} = \text{Offset}[23:0] + 0x0040$$

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

The number of character entries in the font image is calculated as (LastChar – FirstChar) + 1.

The Font Bitmap block contains the character bitmap definitions. Each character is stored as a contiguous set of bytes. Each bit represents one pixel, since fonts glyphs are always treated as 1 bit-per-pixel (bpp) images. Pixels are stored left-to-right, top-to-bottom.

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 details on the font structure, see "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 LANGUAGE (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 bitmaps, 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 can occupy about 3.8 Kbytes of memory. The actual memory requirement will vary depending on the kind of font used 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.

For PIC devices, with internal Flash memory sizes ranging from 16-256 Kbytes for PIC24F and 32-512 Kbytes for PIC32, usually it is not possible to handle such fonts in the usual way.

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 that draws the lines of 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 fonts' 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-printing) 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 characters glyphs would have to be stored in memory.

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

By storing only characters that are going to be used, the second solution is more efficient and has larger reductions in the fonts' memory requirement.

In applications that use static strings, only the pre-defined set of characters 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 would be what is 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};
```

Except for the space and exclamation mark character, all the character codes have changed. That 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 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**

```

#ifdef USE_MULTIBYTECHAR
    define XCHAR short           // 2 bytes
#else
    define XCHAR char           // 1 byte
#endif

```

- If `USE_MULTIBYTECHAR` is not defined:
  - `XCHAR` is defined as `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 6, using only `char` as data type.
- When `USE_MULTIBYTECHAR` is defined:
  - `XCHAR` is defined as `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 MULTI-BYTE XCHAR DEFINITION**

```

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

```

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

By using the reduced character set, the character array code will convert to what is shown in Example 6.

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

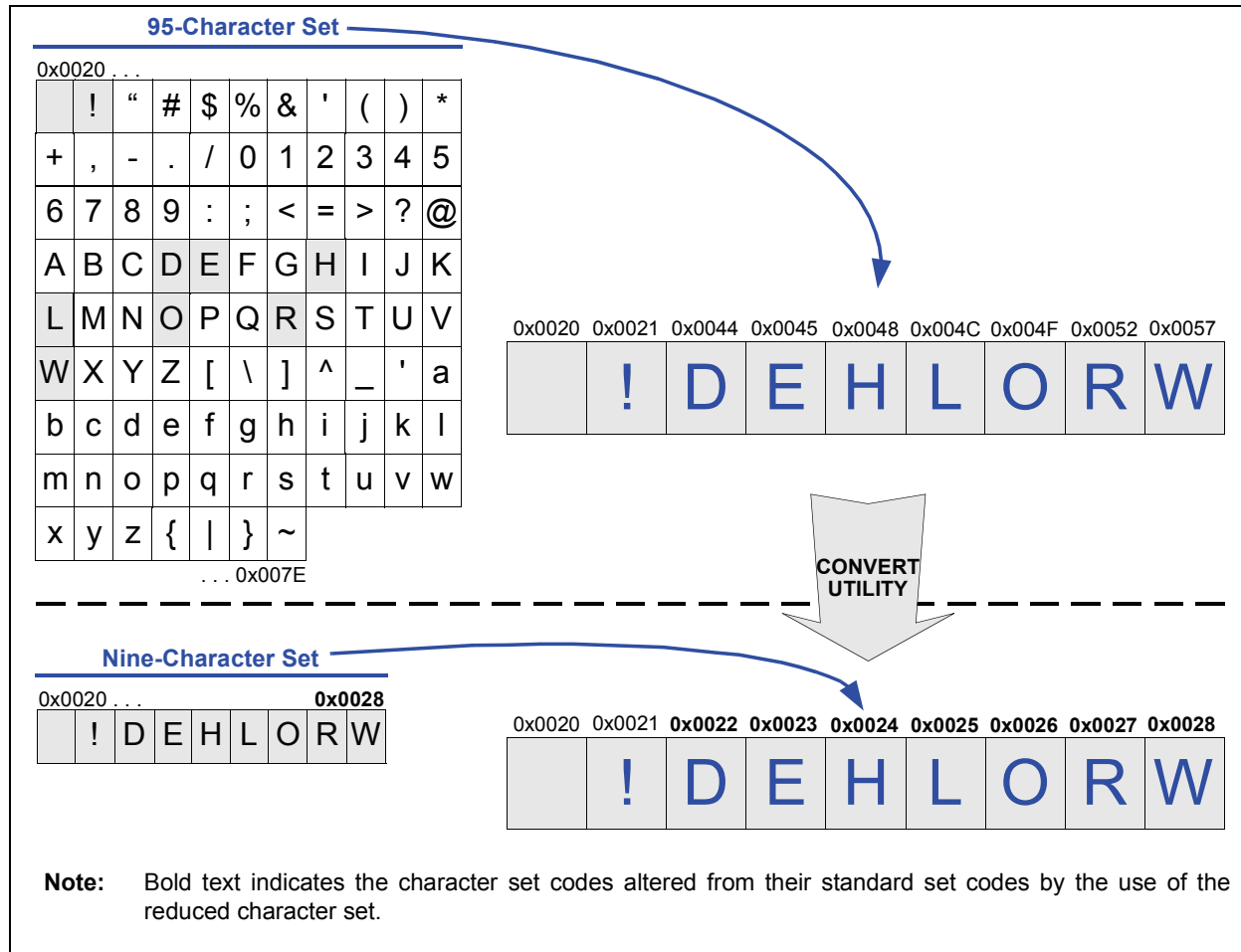
```

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

```

Figure 1 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 1: EFFECT OF REDUCING FONT IMAGE ON CHARACTER CODES**



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 becomes negligible for European languages with small standard character

sets. But for languages with thousands of characters, the memory savings from reduced character sets is very significant. Limited Flash memory easily can 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.

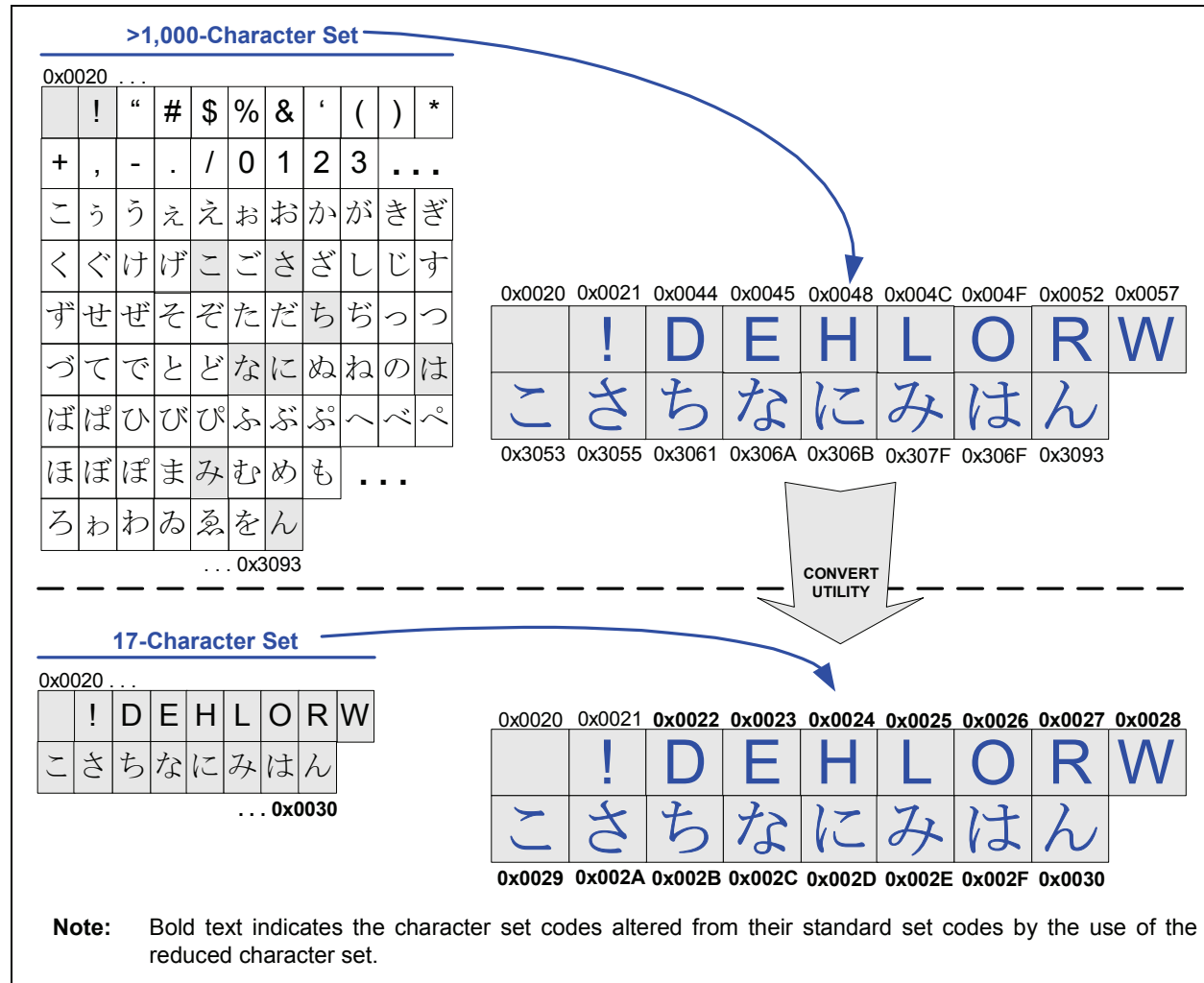
**EXAMPLE 7: JAPANESE CHARACTER ARRAY DECLARATION USING REDUCED FONT IMAGE**

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

Figure 2 shows how the reduced character set utility produces the font image for the Japanese version of the 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 2: REDUCED FONT IMAGE WITH JAPANESE CHARACTERS**



## GENERATING REDUCED CHARACTER SETS

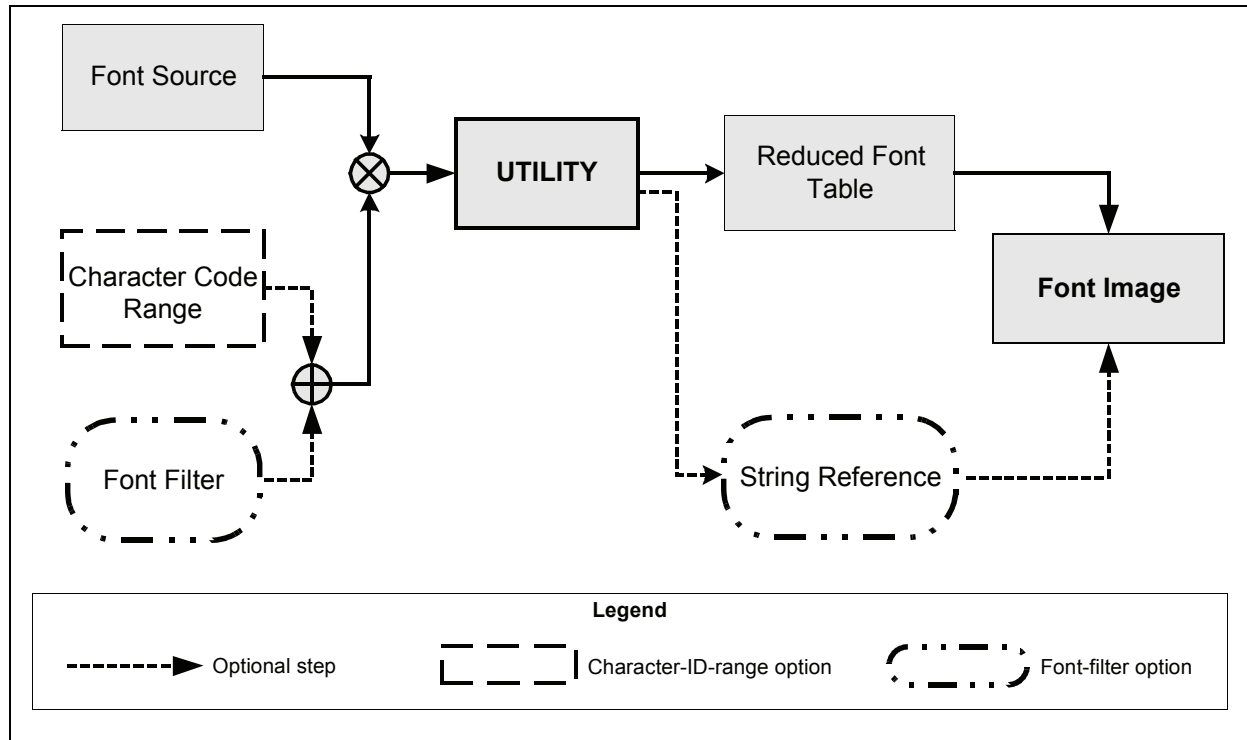
The benefits of reduced character sets having been demonstrated, this section describes:

- The requirements for generating the reduced character sets
- The files needed to access character glyphs bitmaps 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 3.

**FIGURE 3: 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 file is required. (See “**String Reference**” paragraph.)

**Utility** – Software that automates the generation of the string reference file whenever the font filter option is used to generate a reduced character font image.

For details on the utility, see the documentation of the font and bitmap converter that comes with the graphics library installer.

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

**String Reference** – A C header file (\*ref.h) generated if the font filter is used. The file lists all of the string arrays to be used in the application.

Applications use this generated file to refer to the strings defined in the font filter file.

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.



## 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.

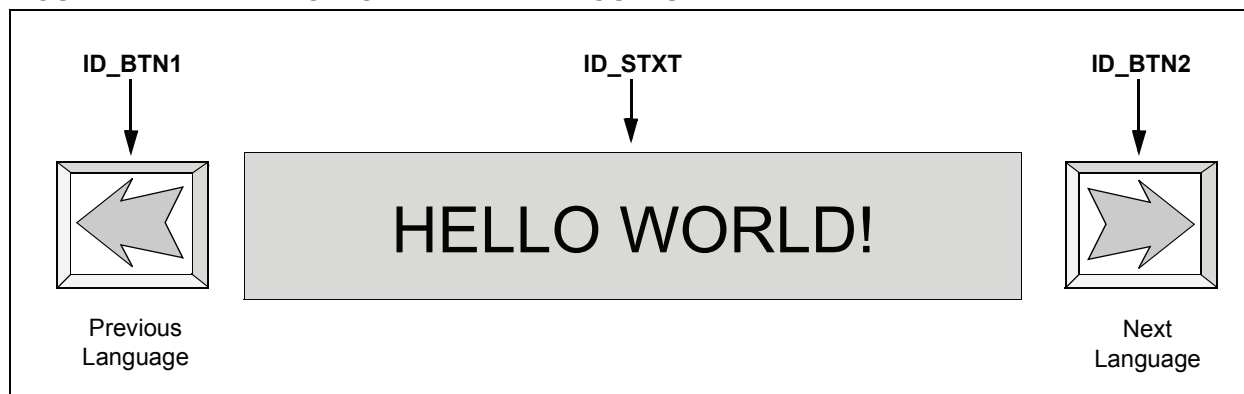
The screen display is driven by a PIC® microcontroller with a static text widget specifying the text.

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

For more details, see the code listing in **Appendix A: “Code Examples”**.

**FIGURE 4: “HELLO WORLD” EXAMPLE OUTPUT**



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

Table 2 lists the font sources for the different languages.

**TABLE 2: SUMMARY OF FONTS USED**

Language	Internet Source	Font Source
English	<a href="http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&amp;cat_id=FontDownloads">http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&amp;cat_id=FontDownloads</a>	Gentium
French		
Italian		
German		
Dutch		
Chinese	<a href="http://www.unifont.org/fontguide">http://www.unifont.org/fontguide</a>	FireFlySung
Japanese		UnBatang
Russian		
Korean		

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
```

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

## 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: KOREAN FONT FILTER FILE CONTENTS

```
KoreanStr:   안녕하세요 . 세계인여러분 !  // In Korean
```

## EXAMPLE 12: RUSSIAN FONT FILTER FILE CONTENTS

```
RussianStr:  Здравствуй Мир!                 // In Russian
```

The filter file's colon (:) and two forward slashes (//) act as the delimiters. 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.

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

**EXAMPLE 13: STRING REFERENCE FILE OUTPUT**

```
#include "Graphics\Graphics.h"
// Automatically generated reference arrays for the Japanese.txt file.
XCHAR JapaneseStr[] = { 0x0028, 0x0025, 0x0023, 0x0029,
                        0x0022, 0x0029, 0x0026, 0x0024,
                        0x0027, 0x0021, 0x0000}; // In Japanese
// Automatically generated reference arrays for the Chinese.txt file.
XCHAR ChineseStr[] = { 0x0023, 0x0024, 0x0022, 0x0025,
                       0x0021, 0x0000}; // In Chinese
// Automatically generated reference arrays for the Russian.txt file.
XCHAR RussianStr[] = { 0x0022, 0x0026, 0x0029, 0x0024,
                       0x0025, 0x002A, 0x002B, 0x0025,
                       0x002C, 0x0028, 0x0020, 0x0023,
                       0x0027, 0x0029, 0x0020, 0x0021,
                       0x0000}; // In Russian
// Automatically generated reference arrays for the Korean.txt file.
XCHAR KoreanStr[] = { 0x0028, 0x0024, 0x002C, 0x0027,
                      0x002A, 0x0022, 0x0027, 0x0023,
                      0x002B, 0x0029, 0x0025, 0x0026,
                      0x0021, 0x0000}; // In Korean
```

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.	// 2 <sup>nd</sup> string example
StringUsed3:	An example string 2.	// 3 <sup>rd</sup> 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 demonstrated in Figure 4, a linked list must be created with the languages arranged in a ring. To do this, the desired structure is established and the linked list initialized by a function (see Example 15).

## 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;
    for (i = 0; i < HWDATAMAX; i++) {
        switch (i) {
            case 0:    HWLang[i].pHWFont = (void *)&GOLFontDefault;
                      HWLang[i].pHWStr = EnglishStr;
                      break;
            case 1:    HWLang[i].pHWFont = (void *)&ChineseFont;
                      HWLang[i].pHWStr = ChineseStr;
                      break;
            case 2:    HWLang[i].pHWFont = (void *)&JapaneseFont;
                      HWLang[i].pHWStr = JapaneseStr;
                      break;
            case 3:    HWLang[i].pHWFont = (void *)&GOLFontDefault;
                      HWLang[i].pHWStr = ItalianStr;
                      break;
            case 4:    HWLang[i].pHWFont = (void *)&RussianFont;
                      HWLang[i].pHWStr = RussianStr;
                      break;
            case 5:    HWLang[i].pHWFont = (void *)&GOLFontDefault;
                      HWLang[i].pHWStr = GermanStr;
                      break;
            case 6:    HWLang[i].pHWFont = (void *)&GOLFontDefault;
                      HWLang[i].pHWStr = DutchStr;
                      break;
            case 7:    HWLang[i].pHWFont = (void *)&GOLFontDefault;
                      HWLang[i].pHWStr = FrenchStr;
                      break;
            case 8:    HWLang[i].pHWFont = (void *)&KoreanFont;
                      HWLang[i].pHWStr = KoreanStr;
                      break;
            default:break;
        }
    }
}
```

**EXAMPLE 15: “HELLO WORLD” LINKED LIST STRUCTURE (CONTINUED)**

```

        // make the list a ring list
        if (i == (HWDATAMAX-1)) {
            HWLang[i].pHWNNext = (void*)&HWLang[0];
            HWLang[i].pHWPPrev = (void*)&HWLang[i-1];
        } else if (i == 0) {
            HWLang[i].pHWNNext = (void*)&HWLang[i+1];
            HWLang[i].pHWPPrev = (void*)&HWLang[HWDATAMAX-1];
        } else {
            HWLang[i].pHWNNext = (void*)&HWLang[i+1];
            HWLang[i].pHWPPrev = (void*)&HWLang[i-1];
        }
    }
    pHWData = &HWLang[0]; // initialize global pointer
}

```

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 GOLMsgCallback(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->pHWPPrev;
                // 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->pHWNNext;
                // 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 the 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

P. Tamayo and A. Alkhimenok, *AN1136, "How to Use Widgets in Microchip Graphics Library"* (DS01136), Microchip Technology Inc., 2007.

*"Microchip Graphics Library Help"* (Microchip Graphics Library Help.chm), Microchip Technology Inc., 2008.

*"Bitmap and Font Converter Utility"* (Help documentation), Microchip Technology Inc., 2008.

<http://www.microchip.com/graphics>, Microchip Technology Inc., 2008.

## APPENDIX A: CODE EXAMPLES

The listed items are example application code files that are installed with the Microchip Graphics Library. (Please refer to “*Graphics AN1182*” subdirectory under the “*Microchip Solutions*” directory after installing the library files.) Each file’s contents is given on the indicated page. The code runs on Explorer 16 using PIC24FJ128GA010 and PIC32MX360F512L connected to either the Graphics PICtail™ Plus Board Version 1 or Version 2. Use the compile switches documented in the “*Microchip Graphics Library Help*” file to switch from Version 1 and 2 of the Graphics PICtail Plus Board.

The main file (Example A-1) shows the translation of the “HELLO WORLD!” phrase into different languages. The example assumes a touch screen module is included and that the fonts have been generated and use the names of the font images declared in the C file.

<b>Main C File .....</b>	<b>15</b>
<b>“HELLO WORLD” Font Reference File .....</b>	<b>21</b>
<b>Chinese Font Filter File .....</b>	<b>22</b>
<b>Japanese Font Filter File .....</b>	<b>22</b>
<b>Korean Font Filter File .....</b>	<b>22</b>
<b>Russian Font Filter File .....</b>	<b>22</b>

The other example application file is HelloWorldFonts.C which shows the utility’s output for the Chinese, Japanese, Korean and Russian font images for the “HELLO WORLD!” phrase.

### EXAMPLE A-1: MAIN C FILE (FILENAME: AN1182Demo.c)

```

/*****
/*                               Main C File
*****/
#include "HelloWorldFontsfontref.h"
// Configuration bits
#ifdef __PIC32MX__
    #pragma config FPLLODIV = DIV_2, FPLLMUL = MUL_18, FPLLIDIV = DIV_1, FWDTEN = OFF,
    FCKSM = CSECME, FPBDIV = DIV_8
    #pragma config OSCIOFNC = ON, POSCMOD = XT, FSOSCEN = ON, FNOSC = PRIPLL
    #pragma config CP = OFF, BWP = OFF, PWP = OFF
#else
    _CONFIG2(FNOSC_PRIPLL & POSCMOD_XT) // Primary XT OSC with PLL
    _CONFIG1(JTAGEN_OFF & FWDTEN_OFF) // JTAG off, watchdog timer off
#endif

////////////////////////////////////
//                               OBJECT'S IDs
////////////////////////////////////
#define ID_BTN1      10          // Button 1 ID
#define ID_BTN2      11          // Button 2 ID
#define ID_STXT      20          // Static text ID
////////////////////////////////////
//                               OBJECT DIMENSIONS DEFINES
////////////////////////////////////
// static text dimension
#define STXWIDTH      280        // static text width
#define STXHEIGHT     40         // static text height
#define STXXPOS       20         // static text left/top pos
#define STXYPOS       80

// string select buttons dimensions
#define SELBTNYPOS     STXYPOS+STXHEIGHT+3 // button left/top pos
#define SELBTNXPOS     STXXPOS
#define ARROWHEIGHT    27+(GOL_EMOSS_SIZE*2) // button height
#define ARROWWIDTH     30+(GOL_EMOSS_SIZE*2) // button width

#define HWDATAMAX      9          // # of "Hello World" strings
#define HELLOWORLDDDELAY 30       // default animation delay

```

```
////////////////////////////////////
//                                LOCAL PROTOTYPES
////////////////////////////////////
void CheckCalibration(void); // check if calibration is needed
void InitHWDData(void);      // initialize string struct arrays
void CreateHelloWorld(void); // create the components

////////////////////////////////////
//                                IMAGES USED
////////////////////////////////////
extern const BITMAP_FLASH redLArrow; // bitmap used for button 1
extern const BITMAP_FLASH redRArrow; // bitmap used for button 2
////////////////////////////////////
//                                FONTS USED
////////////////////////////////////
extern const FONT_FLASH RussianFont; // font for Russian translation
extern const FONT_FLASH ChineseFont; // font for Chinese translation
extern const FONT_FLASH JapaneseFont; // font for Japanese translation
extern const FONT_FLASH KoreanFont; // font for Korean translation
////////////////////////////////////
//                                DEMO STATES
////////////////////////////////////
typedef enum {
    SS_CREATE_HELLOWORLD = 0,
    SS_DISPLAY_HELLOWORLD,
} SCREEN_STATES;

////////////////////////////////////
//                                GLOBAL VARIABLES FOR DEMO
////////////////////////////////////
volatile DWORD tick = 0; // tick counter
GOL_SCHEME*altScheme; // alternative style scheme
STATICTEXT*pSt; // pointer to the static text object
WORD update = 0; // variable to update customized
// graphics

// strings that will use the GOL default font.
XCHAR EnglishStr[] = {'H','e','l','l','o',' ','W','o','r','l','d','!',0};
XCHAR FrenchStr[] = {'B','o','n','j','o','u','r',' ','M','o','n','d','e','!',0};
XCHAR GermanStr[] = {'H','a','l','l','o',' ','W','e','l','t','!',0};
XCHAR ItalianStr[] = {'C','i','a','o',' ','M','o','n','d','o','!',0};
XCHAR DutchStr[] = {'H','e','l','l','o',' ','W','e','r','l','d','!',0};

// structure used to rotate around the used fonts and "Hello World" strings
typedef struct {
    void *pHwFont; // pointer to the font used
    XCHAR *pHwStr; // pointer to the string
    void *pHwPrev; // pointer to the previous list member
    void *pHwNext; // pointer to the next list member
} HWDATA;
```



---

```

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

int main(void){
    #ifdef __PIC32MX__
        INTEnableSystemMultiVectoredInt();
        SYSTEMConfigPerformance(GetSystemClock());
    #endif

    GOL_MSG msg;          // GOL message structure to interact with GOL

    TouchInit();          // initialize touch screen
    GOLInit();             // initialize graphics library &
                          // create default style scheme for GOL

    // initialize the list of Hello World translation
    InitHWDData();
    // set screen to WHITE
    SetColor(WHITE);
    ClearDevice();

    // Create and assign colors to the alternate style scheme
    // create alternative style scheme
    altScheme = GOLCreateScheme();
    altScheme->TextColor0 = BRIGHTBLUE; // set the color of text
    // initialize the font used for the style scheme
    altScheme->pFont = pHWDData->pHWFFont;

    CreateHelloWorld();
    while(1){
        if (GOLDraw()) { // Draw GOL object
            TouchGetMsg(&msg); // Get message from touch screen
            GOLMsg(&msg); // Process message
        }
    }
}

```

```
////////////////////////////////////
// Function: WORD GOLMsgCallback(WORD objMsg, OBJ_HEADER* pObj, GOL_MSG* pMsg)
// Input: objMsg - translated message for the object,
//        pObj - pointer to the object,
//        pMsg - pointer to the non-translated, raw GOL message
// Output: if the function returns non-zero the message will be processed by default
// Overview: it's a user defined function. GOLMsg() function calls it each
//          time the valid message for the object received
////////////////////////////////////
WORD GOLMsgCallback(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) {
                // adjust global pointer to previous string
                pHWData = pHWData->PHWPrev;
                // get pointer to static text
                pSt = (STATICTEXT*) GOLFindObject(ID_STXT);
                // change font used in static text
                pSt->pGolScheme->pFont = pHWData->pHWFFont;
                // set the new string
                StSetText(pSt, pHWData->pHWStr);
                SetState(pSt, ST_DRAW); // set redraw state
            }
            break;
        case ID_BTN2:
            if (objMsg == BTN_MSG_RELEASED) {
                // adjust global pointer to next string
                pHWData = pHWData->PHWNext;
                // get pointer to static text
                pSt = (STATICTEXT*) GOLFindObject(ID_STXT);
                // change font used in static text
                pSt->pGolScheme->pFont = pHWData->pHWFFont;
                // set the new string
                StSetText(pSt, pHWData->pHWStr);
                // set redraw state
                SetState(pSt, ST_DRAW);
            }
            break;
        default:
            break;
    }
    return 1;
}
```

```

////////////////////////////////////
// Function: WORD GOLDrawCallback()
// Output: if the function returns non-zero the draw control will be passed to GOL
// Overview: it's a user defined function. GOLDraw() function calls it each
//           time when GOL objects drawing is completed. User drawing should be done
//           here.
//           GOL will not change color, line type and clipping region settings while
//           this function returns zero.
////////////////////////////////////
WORD GOLDrawCallback(){
    return 1;
}

////////////////////////////////////
// Function: void CreateHelloWorld()
// Output: none
// Overview: Create the objects that will show the Hello World Demo.
//           This is composed of two buttons and a static text.
//           The two buttons will select the string that will be displayed.
//           The strings are arranged in a linked list configured as a ring.
////////////////////////////////////
void CreateHelloWorld(void)
{
    altScheme2->pFont = pHWData->pHWFont;
    StCreate(    ID_STXT,
                STXXPOS, STXYPOS, STXXPOS+STXWIDTH, STXYPOS+STXHEIGHT,
                ST_CENTER_ALIGN|ST_DRAW|ST_FRAME,
                pHWData->pHWStr,
                altScheme2);

    BtnCreate(  ID_BTN1,                                // object's ID
                SELBTNXPOS, SELBTNYPOS,
                SELBTNXPOS+ARROWWIDTH,
                SELBTNYPOS+ARROWHEIGHT,                    // object's dimension
                0,                                          // radius of the rounded edge
                BTN_DRAW,                                  // draw the object after creation
                (void*)&redLArrow,                         // bitmap used
                NULL,                                       // use this text
                NULL);                                     // use default style scheme

    BtnCreate(ID_BTN2,
                SLIDERXPOS+SLIDERWIDTH,
                SLIDERYPOS,
                SLIDERXPOS+SLIDERWIDTH+ARROWWIDTH,
                SLIDERYPOS+ARROWHEIGHT,
                0,
                BTN_DRAW,
                (void*)&redRArrow,
                NULL,
                NULL); // use default style scheme
}

```

```
////////////////////////////////////
// Function: void InitHWData()
// Output: none
// Overview: Initialize the ring linked list.
////////////////////////////////////
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 = EnglishStr;
                break;
            case 1:
                HWLang[i].pHWFont = (void *)&ChineseFont;
                HWLang[i].pHWStr = ChineseStr;
                break;
            case 2:
                HWLang[i].pHWFont = (void *)&JapaneseFont;
                HWLang[i].pHWStr = JapaneseStr;
                break;
            case 3:
                HWLang[i].pHWFont = (void *)&GOLFontDefault;
                HWLang[i].pHWStr = ItalianStr;
                break;
            case 4:
                HWLang[i].pHWFont = (void *)&RussianFont;
                HWLang[i].pHWStr = RussianStr;
                break;
            case 5:
                HWLang[i].pHWFont = (void *)&GOLFontDefault;
                HWLang[i].pHWStr = GermanStr;
                break;
            case 6:
                HWLang[i].pHWFont = (void *)&GOLFontDefault;
                HWLang[i].pHWStr = DutchStr;
                break;
            case 7:
                HWLang[i].pHWFont = (void *)&GOLFontDefault;
                HWLang[i].pHWStr = FrenchStr;
                break;
        }
    }
}
```

```

        case 8:
            HWLang[i].pHWFont = (void *)&KoreanFont;
            HWLang[i].pHWStr = KoreanStr;
            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];
    }
}
pHWDData = &HWLang[0];
}

```

**EXAMPLE A-2: “HELLO WORLD” FONT REFERENCE FILE  
(FILENAME: HelloWorldFontsref.h)**

```

/*****
/*          Font Reference Header File
*****/
#include "Graphics\Graphics.h"
// Automatically generated reference arrays for the HWJapanese.txt file.
XCHAR JapaneseStr[] = { 0x0028, 0x0025, 0x0023, 0x0029, 0x0022,
                        0x0029, 0x0026, 0x0024, 0x0027, 0x0021,
                        0x0000}; // In Japanese

// Automatically generated reference arrays for the HWChinese.txt file.
XCHAR ChineseStr[] = { 0x0023, 0x0024, 0x0022, 0x0025, 0x0021,
                        0x0000}; // In Chinese

// Automatically generated reference arrays for the HWRussian.txt file.
XCHAR RussianStr[] = { 0x0022, 0x0026, 0x0029, 0x0024, 0x0025,
                       0x002A, 0x002B, 0x0025, 0x002C, 0x0028,
                       0x0020, 0x0023, 0x0027, 0x0029, 0x0020,
                       0x0021, 0x0000}; // In Russian

// Automatically generated reference arrays for the HWKorean.txt file.
XCHAR KoreanStr[] = { 0x0028, 0x0024, 0x002C, 0x0027, 0x002A,
                      0x0022, 0x0027, 0x0023, 0x002B, 0x0029,
                      0x0025, 0x0026, 0x0021, 0x0000}; // In Korean

```

# AN1182

---

## **EXAMPLE A-3: CHINESE FONT FILTER FILE (FILENAME: HWChinese.txt)**

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

## **EXAMPLE A-4: JAPANESE FONT FILTER FILE (FILENAME: HW HWJapanese.txt)**

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

## **EXAMPLE A-5: KOREAN FONT FILTER FILE (FILENAME: HWKorean.txt)**

KoreanStr: 안녕하세요 . 세계인여러분 ! // In Korean
--

## **EXAMPLE A-6: RUSSIAN FONT FILTER FILE (FILENAME: HWRussian.txt)**

RussianStr: Здравствуй Мир! // In Russian
---

---

**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, Accuron, dsPIC, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC<sup>32</sup> logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA 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.

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

© 2008, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

**QUALITY MANAGEMENT SYSTEM**  
**CERTIFIED BY DNV**  
**== ISO/TS 16949:2002 ==**

*Microchip received ISO/TS-16949:2002 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://support.microchip.com>  
Web Address:  
[www.microchip.com](http://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

#### Dallas

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

#### Detroit

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

#### Kokomo

Kokomo, IN  
Tel: 765-864-8360  
Fax: 765-864-8387

#### 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: 86-10-8528-2100  
Fax: 86-10-8528-2104

#### China - Chengdu

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

#### 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 - Xiamen

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

#### China - Xian

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

#### China - Zhuhai

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

### ASIA/PACIFIC

#### India - Bangalore

Tel: 91-80-4182-8400  
Fax: 91-80-4182-8422

#### 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 - 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: 886-3-572-9526  
Fax: 886-3-572-6459

#### Taiwan - Kaohsiung

Tel: 886-7-536-4818  
Fax: 886-7-536-4803

#### 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