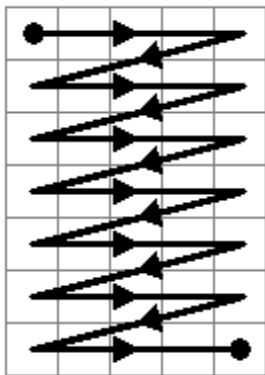


## 1. Introduction

The main purpose of BGL (Brad's Graphic Library) is to paint text, lines and rectangles on a color LCD module in a way that exploits the windowing capability of typical small LCD controllers. The code is optimized for 16-bit color (5:6:5 RGB) format but provides limited support for 18-bit to 24-bit color.

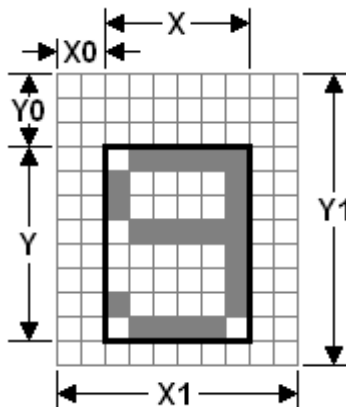
### 1.1 Font representation.

Bitmap fonts are supplied in source form as text files. One or more fonts are compiled into application ROM for use by rendering code. This rendering code sets up the LCD module to fill a rectangular field with pixels and then paints a monochrome image of the character using user-defined foreground and background colors. A character that is 7 x 9 pels, for example, results in 63 pixels being written to the display. The LCD hardware maps the 63-word block onto the designated 7x9 rectangular region.



When storing pixel data, the LCD controller is expected to index left to right, top to bottom. Small graphic modules typically have commands such as a window address function to set the left and right columns as well as the upper and lower rows.

The raw character (n) pixels are packed into  $(n+7)/8$  bytes with the first pixel encoded in the LSB of the first byte. A '1' bit uses the variable FG\_COLOR as data. A '0' bit doesn't plot.



There are six dimensions of interest in the character field. The character is to be drawn at  $(\text{cursX} + X0, \text{cursY} + Y0)$  on the screen. The upper left corner of the text is at  $(\text{cursX}, \text{cursY})$ .

$X0$  and  $Y0$  are the whitespace to the top and left of the drawn region.

$X$  and  $Y$  are the dimensions of the drawn region.

$X1$  and  $Y1$  are the overall field dimensions, used for character spacing.

$Y1$  is the same for all characters in the font. All other dimensions precede the character data in this order:  $X1\ X0\ X\ Y0\ Y$

The fonts use a simple markup language:

<b>NEWFONT</b>	<b>( k d t y1 &lt;name&gt; -- )</b>	Creates a font table in ROM.
<b>ADDCHAR</b>	<b>( rows char -- )</b>	Adds a character to the table.
<b>ENDFONT</b>	<b>( -- )</b>	Resolves the font's tree search structure.

ADDCHAR accepts text data from the input stream and calculates X, X0, Y0 and X1. It expects each row of the character bitmap to be on a new line and composed of '-' and '\*' characters.

One or more fonts are available to an application. They are assumed to be in ROM. The pointers to them are **FONTA** and **FONTB**. **FONTA** is searched first, then **FONTB**. **FONTA** is in primary ROM. **FONTB** may be provided by applications. In the above example, the font would be selected with:

**ShortFont FONTA SETFONT**

Compiled font data is held in a binary tree structure. The links in the tree structure are built last so that the tree is balanced. A 255-character font requires traversal of no more than eight nodes to get to the character data. Links are offsets relative to the beginning of the font table. The endianness of 16-bit data matches the target. Character data is composed of:

16-bit	Unicode ID	Standard identifier
16-bit	left_link	Link to left sub-tree, 0 if none
16-bit	right_link	Link to right sub-tree, 0 if none
8-bit	X1	Width of character
8-bit	X-1	Add to X0 to get right column
8-bit	X0	Offset to left column
8-bit	Y-1	Add to Y0 to get bottom row
8-bit	Y0	Offset to top row
(X*Y)-bit	character data	Monochrome image of character
optional 8-bit empty byte if 16-bit alignment is needed		

The font itself has a short preamble:

- 16-bit link to the root node
- 8-bit Y1 (height of font in pels)
- 8-bit font type
- 8-bit text direction
- 8-bit kerning value (pels to the right of each character)

Before painting a message string, you might have to clear the background (**LCLEAR**) because what's already there won't be completely overwritten by pixels. The drawn regions vary from character to character and various whitespace regions aren't drawn.

Bitmap fonts (which are old-school by today's standards) are not covered by copyright. Although fonts on your PC are copyrighted, the courts have decided several times that the bitmaps they paint on your screen aren't. *Disclaimer: I'm not a lawyer so do your own research.* If you use TrueType fonts (for example) to put text on your computer screen, you can cut and paste the rendered text all you want so as to get any bitmap fonts you need. Make sure the images don't use dithering or other resolution-enhancing technologies because they need to convert to 2-color. The **messages** folder contains tools to automate this task. **WINFONT.F** is a Win32forth program that builds a font containing characters in

the messages. See **MAKEFONTS.F** for examples. When you're running **WINFONT.F**, don't click to other windows (taking focus off the rendering window) until conversion is finished.

Several font types are available:

- |   |                                 |
|---|---------------------------------|
| 0 | Standard 1 bit/pel bitmap font  |
| 1 | High-resolution 3 bits/pel font |

### High-resolution fonts

LCD screens with RGB color order may exploit sub-pixel rendering to get higher resolution. Microsoft calls this new technique (invented over 22 years ago by Steve Wozniak) ClearType®.

R/G/B pixels are individually loaded to get higher horizontal resolution. There are restrictions to its use, however:

- The color order of the LCD (or OLED) must be RGB, not BGR.
- The text color must be Black on White or vice versa.
- You can't use it in landscape mode.
- With this implementation, fixed-spacing fonts aren't supported.

The improvement of sub-pixel rendering versus plain bitmap is subjective. You get three times the horizontal resolution but the antialiasing makes the characters slightly fuzzy. It's also more sensitive to imbalances in display gamma correction (intensity linearity). For example, the intensity levels of a run of 3 adjacent sub-pixels is:

Dot with normal rendering		Dot with sub-pixel rendering	
R	255	R	85
G	255	G	170
B	255	B	255
R	0	R	170
G	0	G	85
B	0	B	0
Total	765	Total	765

Both cases should have the same intensity, but if the gamma correction is off then one white dot will be dimmer than the other.

## 1.2 Low level drawing primitives

High level code interfaces to the graphics hardware via a few key low-level words that you should code in assembly. They vary with the type of controller chip, so they are defined with the other hardware dependencies.

<b>GRCOLS</b>	( min max -- )	\ Set the left and right bounds of the drawing window
<b>GRROWS</b>	( min max -- )	\ Set the top and bottom bounds of the drawing window
<b>GRFILL</b>	( n color -- )	\ Write n pixels into the drawing window
<b>GRTEXT</b>	( ROMaddr n -- )	\ Write n pixels using packed ROM data as bitmap
<b>GRTEXTS</b>	( ROMaddr y x -- )	\ Write x*y subpixels using packed ROM bitmap

These primitives are used to render text, clear the screen, and draw bars and rectangles. Most controllers appear as an array of data. Configuration words and other housekeeping functions use one more low level primitive in a hardware dependent way:

<b>GR!</b>	( data addr -- )	\ Send command or data to the graphic controller
------------	------------------	--

Text colors vary depending on the display and how good the gamma correction is. Also, colors can be 18-bit, 16-bit, etc and in RGB or BGR format. These hardware dependencies are removed (preferably at compile time) by RGB:

<b>RGB</b>	( R G B -- color )	\ Each color plane has a range of 0..255
<b>&gt;RGB</b>	( color -- R G B )	\ Separate into color planes

Depending on how the module is connected, you may be able to read out the RAM for screenshots. A screenshot may have to be processed in sections due to RAM limitations. After setting up a window (full screen or a sliver), you can use **GRREAD**. **GRWRITE** is used with **GRREAD** for BITBLT operations. Only the 16-bit formats (like 5:6:5 RGB) are supported.

<b>GRREAD</b>	( addr n -- )	\ Read n 16-bit pixels to target RAM
<b>GRWRITE</b>	( addr n -- )	\ Write n 16-bit pixels to display

High level code can be used to initialize and turn off the display:

<b>GROPEN</b>	( -- )	\ Turn on and initialize the LCD
<b>GRCLOSE</b>	( -- )	\ Power down the LCD

**GRFILL** is simple: send a color pixel to the display n times.

**GRTEXT** unpacks n pixels from a ROM address, LSB first, using FGCOLOR to represent '1'. A '0' plots nothing.

**GRTEXTS** is somewhat less trivial but not as hard as you might imagine. The font data is the same as that used by **GRTEXT** but the characters are three times as wide. Bits are unpacked from a sequence of bytes in ROM, as with **GRTEXT**, and mapped onto individual R,G,B color planes. Since this results in some color artifacts, each bit affects its left and right neighbors. Each bit weights itself and its left and right neighbors with 1/3 full scale. For example, if an RGB triplet has “010” bitmap data associated with it, the green color will be at 33% intensity. If **FGCOLOR**<>0 the text is assumed to be white on black. Otherwise, it's black on white.

Pay attention to your display's gamma correction. Some controllers have better facilities for this than others. For example: two adjacent pixels, one at 1/3 brightness and the other at 2/3 brightness should put out the same amount of light as a single pixel at full brightness. These corrections should be implemented in **RGB** and **>RGB**.

**GRTEXTS** is optional. You can forgo sub-pixel rendering and avoid some coding and ROM usage.

The simulated LCD has some gamma correction as well as words to show how linear the colors are. They display a 63x126 checkerboard pattern of 100% and 0% (full-on and dark) pixels on the left side of a 126x126 square. A similar pattern of 67% and 33% pixels are displayed on the right side. If the colors are linear, the left and right sides should have equal brightness.

**REDS**        ( -- )    \ test red pixels  
**GREENS**    ( -- )    \ test green pixels  
**BLUES**      ( -- )    \ test blue pixels

### 1.3 Text primitives

<b>variable</b> FGCOLOR	\ Foreground text color
<b>variable</b> BGCOLOR	\ Default background
<b>variable</b> cursX	\ Current X,Y position
<b>variable</b> cursY	
<b>variable</b> charspacing	\ Pels of extra spacing between characters
<b>variable</b> language	\ Current language for multi-language messages
<b>variable</b> direction	\ Direction of text output: 0 = L → R, 1 = R → L
<b>variable</b> FONTA	\ Primary font
<b>variable</b> FONTB	\ Secondary font
<b>constant</b> X0	\ Width of LCD screen
<b>constant</b> Y0	\ Height of LCD screen
<b>AT</b> ( x y -- )	\ Set new X,Y position
<b>AT?</b> ( -- x y )	\ Get current X,Y position
<b>+AT</b> ( dx dy -- )	\ Offset X,Y position
<b>GEMIT</b> ( n -- )	\ Paint a character and move the cursor
<b>GTYPE</b> ( addr len -- )	\ Paint a UTF-8 string and move the cursor
<b>GTYPEC</b> ( ROMaddr len -- )	\ Paint a UTF-8 string from ROM
<b>GTYPEZ</b> ( addr -- )	\ Paint a zero-terminated string and move the cursor
<b>GTYPEZC</b> ( ROMaddr -- )	\ Paint a zero-terminated string from ROM
<b>PRINT</b> ( ROMaddr -- )	\ Paint a counted UTF-8 string considering language
<b>GWIDTH</b> ( addr len -- width )	\ Get the width of a string
<b>GWIDTHC</b> ( ROMaddr len -- width )	\ Get the width of a string from ROM
<b>GWIDTHZ</b> ( addr -- width )	\ Get the width of a zero-terminated string
<b>GWIDTHZC</b> ( ROMaddr -- width )	\ Get the width of a zero-terminated string from ROM
<b>GHEIGHT</b> ( -- height )	\ Get the height of font A
<b>GSPACE</b> ( -- )	\ Output a blank
<b>GCR</b> ( -- )	\ Start a new line
<b>GRECT</b> ( width height color - )	\ Fill rectangular region at the cursor
<b>GCLEAR</b> ( width height -- )	\ Fill region at the cursor with the background color
<b>CLS</b> ( -- )	\ Clear the screen and home the cursor

When text is rendered, it's painted a character at a time onto what should be a blank background. You may paint any background you like (such as gradient effects), then paint text on it. Text may be rendered left to right or right to left. If **direction** is '0', **cursX** is incremented by the character width (X1) plus **charspacing** after the character is painted. If **direction** is '1', **cursX** is decremented before the character is painted so the text origin is at the far right edge of the text field.

## **1.4 International Support**

Multi-language strings are encoded using UTF-8. These strings are stored as files and compiled to ROM by **MESSAGE:**. Each message string needs a separate file, but this approach allows the use of off-the-shelf editing tools (for example, Notepad in Windows 2K/NT) to support multi-language messages. Each language starts on a new line, each line being terminated by a LF and optional CR. When a message is to be displayed the Nth line is used, where N-1 is held in **LANGUAGE**. If there are less than N languages in the list, the first (default) language is used.

**MESSAGE:** ( <name> *filename* -- ) \ Compile multi-language message (compile time)  
*child* ( -- **ROMaddr** ) \ Address of counted substring (run time)

The data is converted to counted strings that may be used to traverse a list of multilingual messages. As before, the endianness of 16-bit data matches the target.

8-bit length **n1** of first string  
**n1** bytes of data to display  
...  
8-bit length **nx** of first string  
**nx** bytes of data to display  
8-bit 0  
8-bit "255" end-of-record marker

### **Sample Usage:**

**MESSAGE: HELLO .\messages\helloworld.txt**

**: Hi ( lang -- ) LANGUAGE ! HELLO PRINT ;**

Blank lines are treated as "no translation needed" messages. The run-time code substitutes the default language (English) when **LANGUAGE** selects an empty string. The run-time action of a message is to return a pointer to the string list.

## 1.5 Embedded Bitmap Images

Images such as icons are expected to be in 24-bit BMP format.

**BMP:** ( <name> *filename* -- ) \ Compile 24-color Windows BMP (compile time)  
**child** ( -- **ROMaddr** ) \ Address of compressed BMP (run time)

The data is compiled to ROM in run-length-encoded format:

```

8-bit X
8-bit Y
16-bit n
n 24-bit fields of:
{ 8-bit runlength-1
  5-bit red
  6-bit green
  5-bit blue
} optional 8-bit empty byte if 16-bit alignment is needed

```

The BMP is painted at the cursor position using **GBMP**. The cursor is not affected.

**GBMP** ( **ROMaddr** -- ) \ Paint BMP at the cursor position

## 1.6 Other graphic primitives

**RECTANGLE** ( **x y color** -- ) \ Draw a rectangle at the cursor  
**PROGRESS** ( **x y color frac** -- ) \ Draw a progress bar

The progress bar can be drawn inside a border rectangle, first choosing the upper left corner with AT. Frac is an unsigned scale factor between 0 and (2<sup>cellbits</sup>)-1. The X and Y dimensions should be two pels less than the border rectangle.

Example:

```

: PERCENT ( n -- ) \ Update progress bar using percentage
  655 * >r \ 65535 is full-scale for 16-bit cells
  5 50 AT 90 9 WHITE RECTANGLE
  6 51 AT 88 7 GREEN r> PROGRESS ;

```



## 2. Using BGL

### 2.1 Running the simulation

SIM.F is a sample usage that uses Win32forth to simulate a console window. You can run it by loading SIM.F and typing “GO”.

<b>Plain</b>	( -- )	\ Use normal text rendering
<b>HiDef</b>	( -- )	\ Use sub-pixel text rendering
<b>DEMO</b>	( -- )	\ Runs a multilingual text demo
<b>..</b>	( n -- )	\ Displays a number in big sub-pixel digits
<b>B/W</b>	( -- )	\ Use black text on a white background
<b>W/B</b>	( -- )	\ Use white text on a black background

### 2.2 Targeting BGL at your Embedded System

BGL uses the proposed Cross Compiler Wordset (XCtext5.pdf) in use at MPE and Forth Inc.

By default, the target is assumed to be little-endian and require 16-bit words to be aligned on even addresses. You can tell **fontcmp.f** to use other options by declaring:

<b>1 constant BigEndian</b>	\ target is big-endian processor
<b>0 constant NeedAlign</b>	\ 16-bit words may be at odd addresses

**fontcmp.f** extends the target compiler (or host Forth if it's plain ANS) to provide words that compile fonts and bitmaps. For a target-compiling environment, **HERE** is assumed to return an address on the target. The host environment contains an image of the target ROM, whose base address must be returned by **CDimage**. Define **CDimage** as the host address of the CData image built by your cross compiler. If you are using **fontcmp.f** in a host environment (for testing, etc.), leave **CDimage** undefined.