ng

Alternative Sparkfun Serial Graphic LCD Backpack Firmware

User Guide
Version 1.27
Jon Green

sparkfun
ELECTRONICS
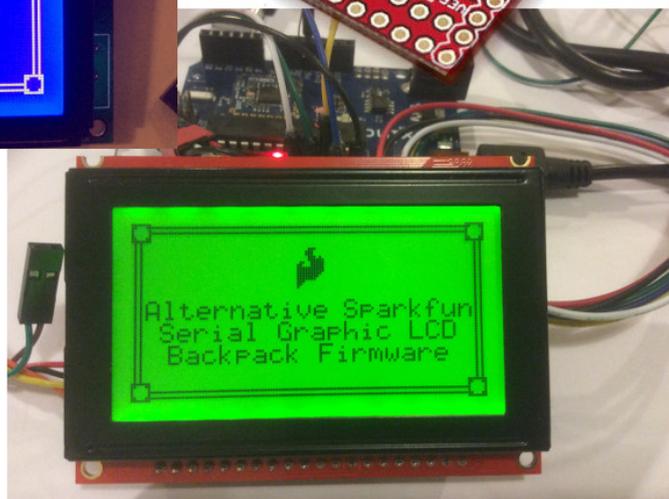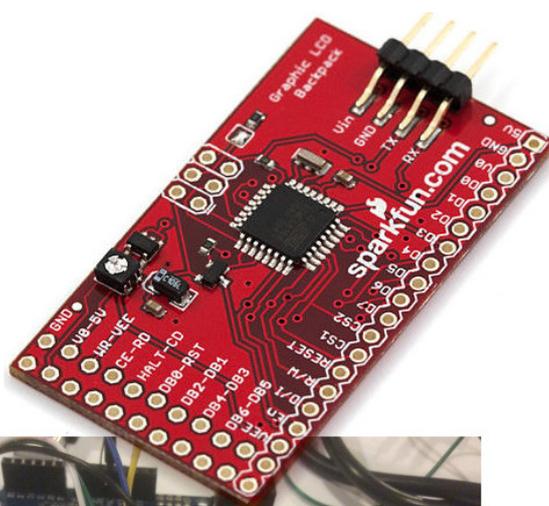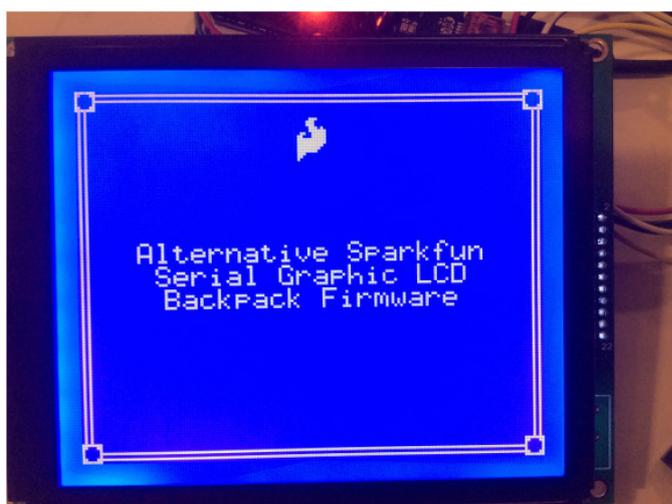
Jon Green
Eton, United Kingdom

No Warranty: The software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. in no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

Title:	User Guide
Reference:	GLCD
Version	v1.27
Date:	2015/07/06 22:46:12
Location:	www.jasspa.com/serialGLCD.html
Contact:	jon@ja**pa.com

Typeset on a MacBook Pro with the TeX Live 2013 LaTeX Documentation System.

Acknowledgements

Mike Hord, SparkFun Electronics

These works are derived in part from the previous works of Mike Hord, SparkFun Electronics dated 02 May 2013 who released a software bundle under Creative Commons Attribution Share-Alike 3.0 license. The main code used from Sparkfun Electronics is the back light control, T6963 160x128 driver and command syntax.

Some pictures and graphics appearing in this document have been taken from Sparkfun published material appearing in the public domain on their web site and are not subjected to any of the licensing terms stated in this document.

Copyright (c) 2010 Jennifer Holt

The baulk of the code has been inspired by Jennifer Holt who provided a 128x64 firmware library implementation. Significantly the serial control, bitblt and font handling. This was distributed with a MIT License.

This implementation is mainly based on the software by the Jennifer Holt and similarly inherits the MIT Licensing with acknowledgement for the Mike Hord, SparkFun Electronics implementation.

Disclaimer

The author is in no way connected with Sparkfun Electronics and no claim is made that Sparkfun Electronics acknowledge or endorse any of the materials provided herein.

MIT License

Copyright © 2105 Jon Green

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

1 Introduction

This document attempts to provide all of the information to upgrade and use the Sparkfun Serial Graphic LCD backpack and has been collected and collated from different sources on the Internet. The document describes this implementation of the Sparkfun Serial Graphic LCD backpack and includes additional information on modifying an re-programming the backpack.

This is replacement firmware for the Sparkfun Serial Graphic LCD Backpack (Figure 1) and is suitable for both the Sparkfun 128x64 and 160x128 LCD screens. The firmware replaces the existing Sparkfun firmware and retains the principal commands of the original.
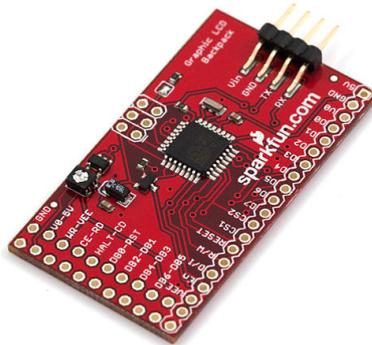
Figure 1: Sparkfun Graphic LCD backpack

The software has been re-written after using both the original Sparkfun software and the software created by Jennifer Holt. Jennifer Holt's version added software flow control which significantly improved the reliability of the screen communication in addition to a bitblt operation which improved bitmap rendering. Unfortunately the Jennifer Holt version did not include support for the 160x128 screen. This new version is a re-write of the firmware for both screens 160x128 and 128x64.

Development started when trying to save program memory for draw commands when used on an Arduino project. That development has since spiralled out of control and consumed much too much of my time. Having proceeded so far then it seemed sensible to try and make it accessible to as many people as possible and attempt to try to write some reasonable documentation to accompany that work. There are many questions asking how to re-program the backpack and how to manage serial software flow control, hopefully those are answered here. It transpires that writing the documentation is a task that seems to take considerably longer than writing the software but the hope is that it is worthwhile and may be useful to somebody else.

1.1 New Features

The new features of this firmware are summarised as follows:

1.1.1 KS0108B Driver

The KS0108B driver for the 128x64 display has been re-written against the Samsung chip specification. The driver now uses the waveform timings of the specification and polls the command status to determine when the command has completed. This method of writing commands considerably speeds up the driver as timed waits are removed.

The KS0108B chip is sensitive to changes in the command lines which lead to anomalies in the rendered display. When addressing the device the command lines should be atomic (i.e. the command lines should

not include transient states when changing the control lines). The data lines should be kept kept as inputs to avoid bus conflicts when not used.

1.1.2 Graphics Mode

A Graphics Mode concept has been introduced which allows multiple drawing commands to be batched and sent to the display without transmitting the command character `0x7c`. Graphics mode may be exited explicitly via a special command or by sending a command prefixed with the command character.

Graphics mode saves one byte per draw command, reducing memory requirements, in addition reduces the serial communications overhead.

1.1.3 Draw mode

A Draw mode concept has been introduced allowing the drawing mode to be set on the LCD device and is used as the default drawing mode without explicitly sending the draw mode on each command. Typically the draw mode is the same for a batch of drawing commands and this method saves one byte per draw command reducing the communications overhead and memory requirements of the caller.

The draw mode is also enhanced for all commands; the original Sparkfun implementation used '`1`' to set the pixel on and '`0`' to set the pixel off. In this implementation then the original Sparkfun parameter is retained but is now extended with bitwise operations for:

or bitwise-or the pixel with the screen pixel, typically used for setting pixels.

xor Exclusive-or the pixel with the screen pixel, may be used to draw by toggling the setting of the pixel.

nand Not-and the pixel with the screen pixel, typically used for clearing the bit.

fill Fills the shape rather than drawing the outline in the specified colour.

Refer to Introduction to Drawing Modes for further information.

1.1.4 Bitblt

A Draw bitblt operation is implemented which is inherited from Jennifer Holt's version of the firmware and forms the basis of the character and sprite drawing. Bitblt is implemented at the driver level for the best performance.

The drawing capabilities of the KS0108B (128x64) and T6963 (160x128) are very different and the bitblt operation hides the differences in the underlying hardware screen operation. The KS0108B is organised as vertical columns of pixels while the T6963 is organised as rows of pixels. The T6963 provides the best performance as the command cycle is shorter and facilitates a pixel set command which allows a pixel to be written without reading in a block of pixels.

1.1.5 Non-EEPROM Command Variants

The Backlight level and Reverse screen commands previously saved their status to EEPROM. This method was not ideal for temporary settings i.e. dimming the screen to sleep etc. A new set of commands have been introduced which change the settings without writing to EEPROM, their settings are lost on a reset.

1.1.6 Splash Screen Logo

The splash screen logo is moved to EEPROM which allows the splash screen logo to be replaced if required. A Factory Reset command exists which restores the device to the shipping condition, restoring the Sparkfun logo.

The splash screen includes three states now, *no logo*, *logo with information*, *logo only*. The default is *logo with information* which shows the screen configuration information in addition to the logo.

The splash screen is shown at start up and remains on-screen until any serial character is received. This provides feedback on the configuration of the screen, see Figure 2.

Figure 2: Information splash screen

1.1.7 Serial Flow Control

The software flow control implemented by Jennifer Holt is included in this version. Software flow control sends XON (0x11) when the screen is ready to receive more data and XOFF (0x13) when the buffer is full and the screen is requesting that the Host stop sending.

The flow control buffer positions are configurable, see Set LCD; a 256 character serial receive buffer is used and the XON and XOFF positions are defined as 20 and 166. This XOFF position has been set for the worst case scenario where the screen is driven from a Mac using a FTDI cable which can send around 70 characters before reacting to an XOFF. For an Arduino then the XOFF position can be increased to around 230 assuming that XOFF is serviced quickly. See Serial Overview for a more in-depth discussion of serial communication.

1.1.8 Sprites

The concept of sprites previously introduced by Jennifer Holt has been retained and extended to allow the sprites to be stored in EEPROM, see Sprite upload and Sprite draw. There are 6 RAM and 14 EEPROM sprite locations which stores sprites of up to 34 bytes. RAM sprites have identities starting from 0x00. EEPROM sprites have identities commencing from `0x80`.

The sprite size is not checked so it is the callers responsibility to ensure that the sprites fit into the available memory. This allows some caller flexibility and permits larger sprites to occupy multiple sprite entries where required without enforcing any size capping.

1.1.9 Polygons

Draw commands have been introduced for polygon line drawing (Draw polygon) and multiple connected lines (Draw lines) where where the last point is not the first point. Polygons may be filled but are restricted

to convex polygons which must be defined in a clockwise vertex order. Polygon fill handles some convex polygons BUT NOT ALL. Polygon filling uses a horizontal scan line algorithm which is computed as the polygon is drawn. When defining filled polygons then it is important to choose the starting point such that a scan line does not over-write part of the form of the shape.

1.1.10 Rounded Box

Draw commands have been introduced for a rounded box i.e. a rectangle with rounded corners which may be filled. See Draw rounded box.

1.1.11 Information Commands

New commands have been introduced to allow the Host to synchronise with the display where required. The Echo character command allows the Host to send a character which is echoed over the serial when executed. The echo allows commands to be batched and once the drawing has completed then the display returns the echo character over serial which the Host may use to trigger some other operation.

A *query* command is introduced which allows the Host to query the current settings including the screen size. See Query LCD.

1.1.12 Character Set

Two characters sets are installed in the display, the default 6x8 (WxH) and Tom Thumb 4x6 bitmap character sets. The 6x8 source code for the characters was supplied with the original Sparkfun source bundle and indicates that it originated from **Sinister 7**, the origin is unknown. The original characters have been manually adjusted to clean them up a little i.e. the existing '4' was, to my mind, ugly and has been replaced with a more aesthetic version of the character.

For better presentation of characters then the bitmap font may be presented as a proportional font, see Figure 3. This is performed by removing any white space from the character resulting in variable length characters.

Proportional fonts may be enabled on either of the supplied character sets by changing the Font mode and applying proportional mode with `GLCD_MODE_FONT_PROPORTIONAL`. The differences in the font mode are shown in Figure 3.

When a proportional font is used then the rendered length of the string is not known by the caller; the Set string position serial command may be used to perform string rendering with centre and right justification positioning for labels etc.

Figure 3: Fonts compared: fixed (left) and proportional (right)

1.2   Resources

The following are resources around the Internet that may be useful for anybody attempting to upgrade their Sparkfun Serial GLCD backpack. The Sparkfun web site provides a good source of information:

Sparkfun Serial Graphic LCD 128x64

Sparkfun Serial Graphic LCD 160x128

Github: Sparkfun Graphic Serial Backpack software

The original Jennifer Holt version of the software may be found here:

SourceForge: SerialGLCD

Some of the diagrams in this document have been generated with the tools from **Fritzing** which provide some good free tools for documenting electronic circuits:

Fritzing.org Website

The Arduino IDE has been used as an ISP programmer and for building an Arduino library to use the backpack:

Arduino Website

Development has been performed on a Mac using the CrossPack AVR©️ tools:

CrossPack for AVR©️ Development

Development has been made easier by developing some of the algorithms off the backpack on a computer using a USB-TTL serial cable to interface directly with the device.

FTDI USB-TTL serial cables

2 Operational Overview

Communication with the Graphcis LCD works by sending high level draw commands and ASCII text characters over the serial line which are processed and rendered by the display. Refer to Serial Overview for a introduction to the serial communications.

The software supports both the Sparkfun small (128x64) and large (160x128) format screens. The (0,0) coordinate is defined as the top left corner of the screen, all positions are defined relative to this point, see Figure 4.
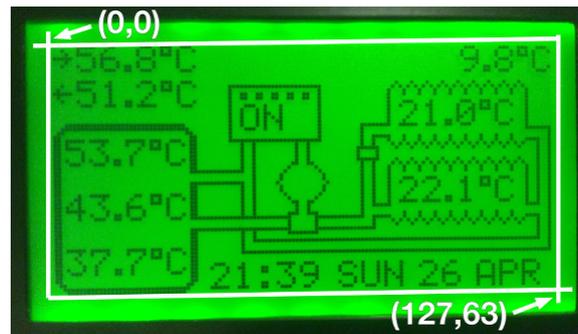
Figure 4: Screen coordinate space

ASCII characters are directly rendered on the screen at the current position (see Set position command), the character is rendered and the position is advanced to the next character position. The characters carriage return CR 0x0d or '\r' and line feed NL 0x0a or '\n' are recognised and advance the line. The CRLF mode controls the operation of CR/LF, see the set LCD command. When CRLF is on then a NL performs a automatic carriage return (UNIX mode). When disabled then NL advances the line but does not change the character position in the line. A CR moves the cursor to the start of the line in both modes.

The ASCII characters `0x20` (space) to `0x7e` (tilde) are supported. The character `0x7c` ('|') is reserved as the graphics command character; to render character `0x7c` then the character must be escaped with `0x7c` i.e. sending the character sequence `0x7c 0x7c`.

Graphics commands are formatted as follows:

`0x7c <command> <arg1> ... <argn>`

Commands are variable length, the number of arguments depends on the command being sent. The principal argument types are discussed in the following sections:

2.1   Drawing Commands

Operation of some of the basic drawing commands are outlined in the following sections.

2.1.1   Box

A rectangular region or box is described by a pair of coordinates (x1, y1) and (x2, y2). This describes a rectangular region as a diagonal as shown in Figure 5.
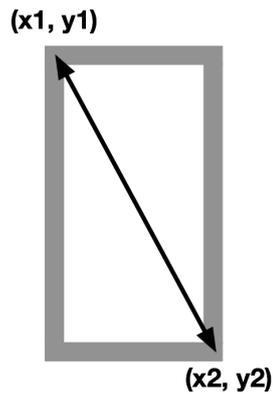
Figure 5: Box drawing

2.1.2   Multiple Joined Lines

Multiple joined lines are described as a list of coordinates pairs (x0, y0), (x1, y1) ... (xn, yn). The end of the list of is marked by the yn coordinate which is OR'ed with 0x80, indicating the last coordinate pair, as shown in Figure 6.
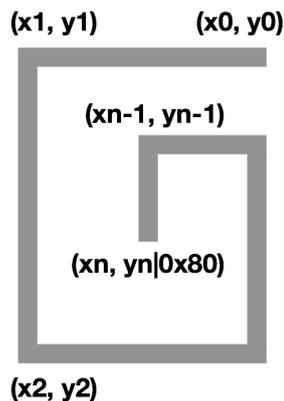
Figure 6: Multiple line drawing

2.1.3   Polygons

Polygons are described as a list of coordinates pairs (x0, y0), (x1, y1) ... (xn, yn). The end of the list of is marked by the yn coordinate which is OR'ed with 0x80, indicating the last coordinate pair. The last coordinate point is automatically joined to the first coordinate to form a polygon as shown in Figure 7.
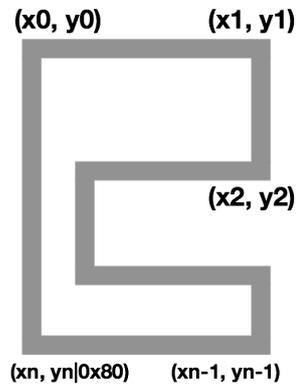
Figure 7: Polygon drawing

When defining a polygon to be filled then the coordinate pairs must be defined in a clockwise direction and shall not cross. Polygon filling only works reliably with convex polygons although a limited number of concave polygons render correctly. The filling algorithm works by drawing the perimeter and fills using horizontal scan lines between the opposite sides of the polygon. As the perimeter is drawn then the horizontal **x** position is saved for each vertical **y** position, when a subsequent point **x'** is draw at the horizontal position **y** then the two points **x** and **x'** are filled by drawing a scan line between them. Provided that the polygon does not fold in on itself then the polygon should render correctly. The concave polygon shown in Figure 7 fills correctly, conversely the polygon defined in Figure 8 fills correctly when the first point is defined as **P**, but fills incorrectly if the first point is defined as **Q**.
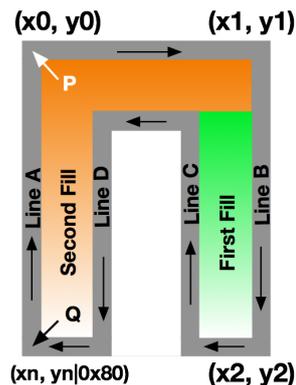
Figure 8: Polygon filling

The polygon starting at **Q** fails because on processing line **Line B** the fill creates horizontal scan lines from **Line A** which incorrectly fills the concave hollow as lines **Line C** and **Line D** have not yet been encountered. Conversely when the starting point is defined as **P** then **Line B** is drawn followed by **Line C** which causes a fill operation (there are now two values of x on a horizontal line). Then **Line D** is drawn followed by **Line A**

which fills the horizontal scan line to **Line D** and then to **Line B** at the upper part of the shape.

2.1.4   Drawing Modes

The drawing mode, Draw mode, is specified with each draw command and defines how the pixels of the image are drawn. Drawing performed in *Reverse* mode automatically corrects for screen reversal. i.e. when the screen is in reverse then drawing a '1' clears the pixel. The draw mode is a bit mask defined in Table 1.

b7	b6	b5	b4	b3	b2	b1	b0
Reserved for future use				Fill mode	Merge mode		Colour
0000 = Zero				**0** = No fill	**00** = Overwrite		**0** = Clear
				1 = Fill	**01** = OR		**1** = Set
					10 = XOR		
					11 = NAND		

Table 1: Draw mode field

Where the fields are defined as follows:

colour: The colour used to draw or fill the shape. A value of '0' clears or reverses the pixel, a value of '1' sets the pixel.

When used with Font mode, sprites then '0' draws the character in reverse and '1' draws in normal mode.

Merge mode: A 2-bit field that defines how the drawing is merged with the background. The default drawing mode overwrite `00` performs a copy over where the background pixel is replaced with the drawn pixel. A non-zero value performs a bitwise merge operation as follows:

`01` bitwise OR i.e. `screen | pixel`.

`10` bitwise exclusive-OR (XOR) i.e. `screen ↑ pixel`

`11` bitwise not-AND (NAND) i.e. `screen & ∼pixel`

Where the *pixel* value is defined by the *colour* field.

Fill mode: A 1-bit field that specifies whether a shape should be filled with the *Colour* or outlined. A value of '0' draws the shape outline, a value of '1' fills the shape.

The resultant drawing for each mode is shown in Table 2. The background is specified as *Normal* or *Reverse* and specifies whether reverse mode has been enabled.
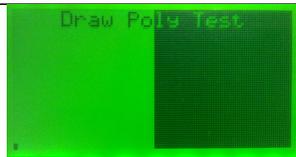
Image	Background	Left	Middle	Right
	Normal	Normal Background	Normal Background	Normal Background
	Reverse	Reverse Background	Reverse Background	Reverse Background
	Normal	0x01 = Normal	0x05 = XOR	0x00 = Reverse
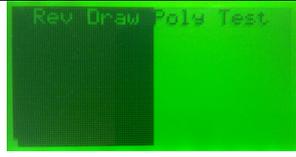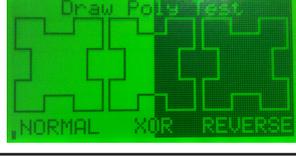
Table 2: Draw modes (continued ...)
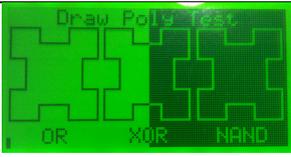
Image	Background	Left	Middle	Right
	Normal	0x03 = Bitwise OR	0x05 = XOR	0x07 = Bitwise NAND
	Reverse	0x01 = Normal	0x05 = XOR	0x00 = Reverse
	Reverse	0x03 = Bitwise OR	0x05 = XOR	0x07 = Bitwise NAND
	Normal	0x09 = Fill+Normal	0x0d = Fill+XOR	0x08 = Fill+Reverse
	Normal	0x0b = Fill+Bitwise OR	0x0d = Fill+XOR	0x0f = Fill+Bitwise NAND
	Reverse	0x09 = Fill+Normal	0x0d = Fill+XOR	0x08 = Fill+Reverse
	Reverse	0x0b = Fill+Bitwise OR	0x0d = Fill+XOR	0x0f = Fill+Bitwise NAND
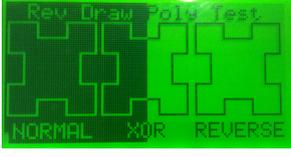
Table 2: Draw modes

2.1.5 Sprite Data

The sprite data is organised as shown in Figure 9:
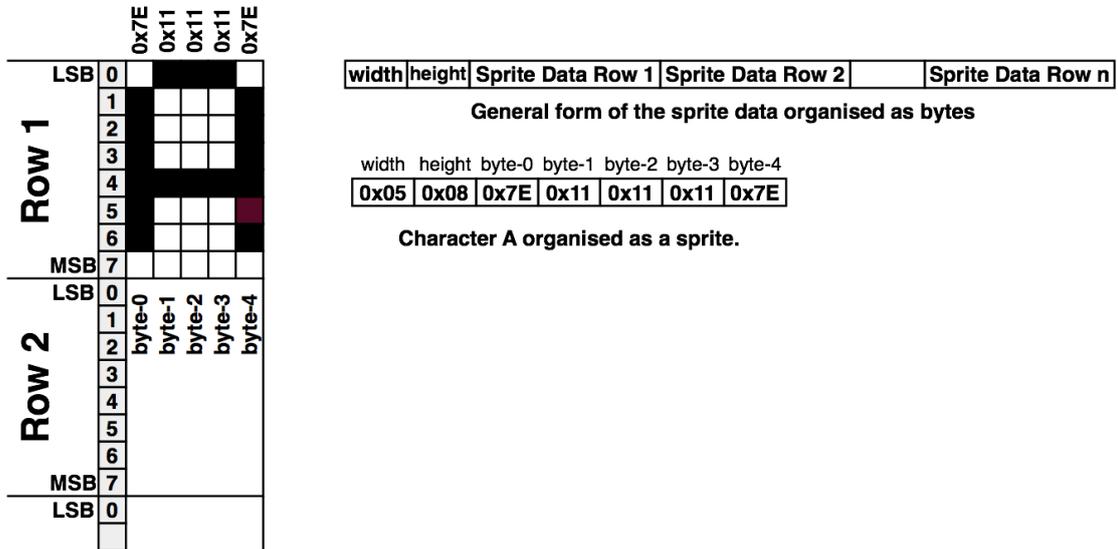
Figure 9: Sprite Data Organisation

The sprite commences with two bytes which are the *width* and *height* of the image in pixels. The pixel data is organised as rows of 8 vertical pixels per byte where the least significant bit (LSB) is the top-left pixel and the most significant bit (MSB) tends towards the bottom-left pixel. A complete row of 8 vertical pixels across the image width comprises the first row, this is then followed by the next row of 8 vertical pixels and so on.

Where the image height is not an exact multiple of 8 bits then any unused bits are typically set to zero (although this does not matter). As an example, a sprite 10 pixels wide and 9 pixels high requires 10 x ((9 + (8 - 1)) / 8) = 20 bytes of pixel data, the sprite would occupy 22 bytes with the *width* and *height* pre-pended to the sprite data.

2.2   Serial Overview

The Graphical Serial LCD provides a serial interface between the Host and the display (Figure 10).
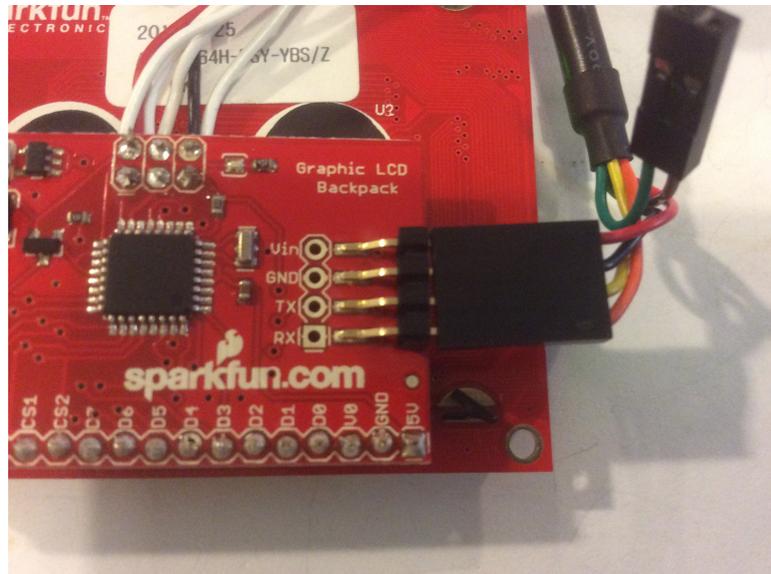
Figure 10: Serial Connection

A 4-wire serial connection is required, a pair for power **Vin** +5v and **GND** ground; a pair for data communication **TX** and **RX**. **TX** is the Host transmit line Host→LCD. **RX** is the Host receive line LCD→Host. The connections are summarised in the Table 3

Arduino	Graphic LCD
5v	Vin
GRD	GRD
TX	TX
RX	RX

Table 3: Serial Connections

Figure 11 shows the Arduino connection to the hardware serial pins. Figure 12 shows a possible Arduino connection using software serial; the pins used may be different to those shown.
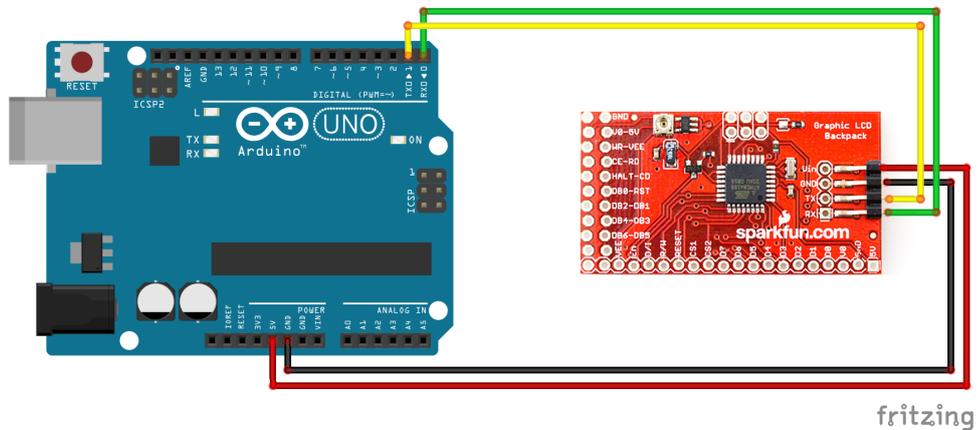
Figure 11: Arduino connection using hardware serial
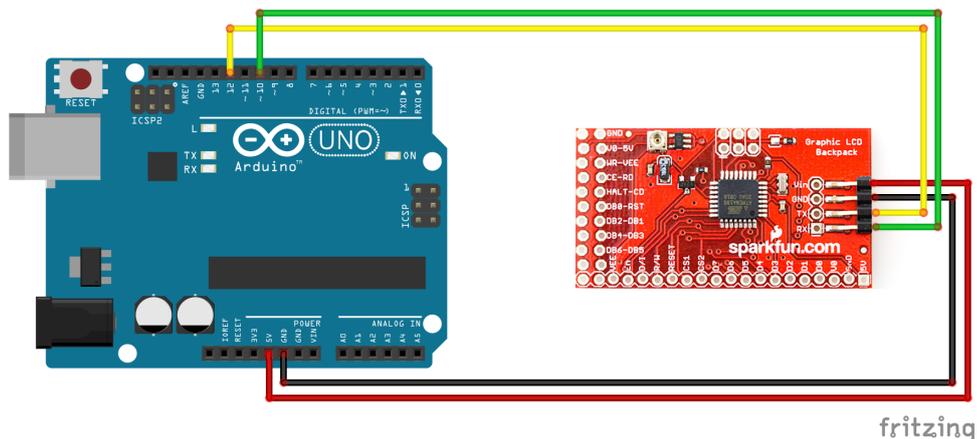
Figure 12: Arduino connection using software serial

The default communication baud rate is 115200bps, 8-bit, no parity. The baud rate may be changed with the Baud rate command which may be configured to work with six different baud rates 4800, 9600, 19200, 57600 and 115200. The fastest speed is recommended and is well supported by the Arduino native serial port in addition to the Arduino Software Serial library. When using Software Serial with the Arduino then version 1.6.1 or above is recommended as this version of the Arduino IDE includes a new improved version of the Software Serial library.

The Serial LCD may also be connected directly to a computer. Development of this software was performed on a Mac using a Future Technology Devices International Ltd (FTDI) USB to TTL 5v RS232 serial cable, part number TTL-232RG-VSW5V-WE. Noted this part is supplied with bare wire ends the alternative part TTL-232R-5V includes a 0.1" header.

To use the RX (0), TX (1) pins of the Arduino then some modification of the supplied Arduino library is required to remove *SoftwareSerial* and is straight forward.

2.2.1   Software Flow Control

The serial communication uses software flow control to signal to the Host when to stop transmission and allow the display to catch up. This simple mechanism was not implemented in the original Sparkfun version of the software and first appeared in the version by Jennifer Holt. The same implementation created by Jennifer Holt is used in this implementation and is extremely effective in managing the serial buffer, preventing over-running which causes corruption of the display. When using software flow control then communication is speeded up as the Host device does not need to add artificial delays which slow down communication.

The software flow control works by monitoring the space available in the serial buffer of the display. When the buffer fills faster than the screen is able to process commands then the internal serial buffer length increases i.e. the number of bytes in the queue waiting to be processed increases. When the serial buffer reaches or exceeds a high water mark then the display sends an XOFF character (0x13). On receipt of XOFF then the Host should stop sending any more characters. The display sends a single XOFF character for every character received over and above the XOFF high water threshold. After sending an XOFF the display continues to process commands from the serial buffer; assuming that the Host has stopped sending, then the number of bytes in the serial buffer decreases. When the serial buffer nears empty, or reaches the low water mark, then the display sends a single XON (0x11) character. This indicates to the Host that it may commence sending characters again. The XON character is only sent by the display when a XOFF has been sent previously, a single XON character is sent only.

Software flow control may be implemented natively by the Host. On the Arduino the serial library does not support software flow control and is implemented in the GLCD library on behalf of the caller. The flow control is simple to implement; on each command send the library polls the serial port for any characters from the display, if no XOFF characters have been received then the command is sent. For long commands i.e. bitblt, then chunks of data (around 32 bytes) are sent before checking the serial for XOFF again. During the check, if an XOFF character is received then the Host blocks on the serial input waiting for an XON character. When an XON is received the Host is then able to continue sending commands again, continuing to periodically poll the serial RX line for an XOFF character.

The serial display uses a 256 byte serial buffer and the default XON and XOFF thresholds are set to 20 and 166 characters, respectively. The XOFF threshold is very conservative and has been defined for use with the FTDI cable on a MAC which can overrun the serial buffer by 60-80 characters. This is a result of internal buffering on the MAC (noted **tcdrain**() is used but this does not significantly decrease the overrun). The XON and XOFF positions may be configured to better match the Host send and receive characteristics using the Set LCD command.

The display sends a `0xff` character in the event that the serial buffer overruns i.e. when the serial buffer has more than 256 characters. The character should be tracked by the sending Host and indicates that the XON/XOFF position should be adjusted.

The serial buffer size must be minimally larger than 128 bytes. For the small display (128x64) then the Draw bitblt operation may look ahead (peek) the serial buffer by up to 128 bytes. The bitblt operation uses the serial buffer as a storage area before consuming the data. When peeking into the serial buffer and XOFF is in currently in effect then an XON is sent by the display if there are insufficient bytes remaining in the buffer and more data is required.

3 Serial Commands

The serial commands are defined in the following section. A summary of the commands is included in Table 4.

Description	Format	ASCII Cmd
Backlight level (non-persistent)	`0x7c 0x42 <percentage>`	
Backlight level (persistent)	`0x7c 0x02 <percentage>`	`Ctrl-B`
Baud rate	`0x7c 0x07 <baud_rate>`	`Ctrl-G`
Clear Screen	`0x7c 0x00`	`Ctrl-@`
Demo (Show splash screen)	`0x7c 0x04`	`Ctrl-D`
Draw bitblt	`0x7c 0x16 <x> <y> <mode> <width> <height> <sdata>*`	`Ctrl-V`
Draw box	`0x7c 0x0f <x1> <y1> <x2> <x2> <mode>`	`Ctrl-O`
Draw box (draw mode)	`0x7c 0x4f <x1> <y1> <x2> <x2>`	
Draw circle	`0x7c 0x03 <x> <y> <radius> <mode>`	`Ctrl-C`
Draw circle (draw mode)	`0x7c 0x43 <x> <y> <radius>`	
Draw line	`0x7c 0x0c <x1> <y1> <x2> <y2> <mode>`	`Ctrl-L`
Draw line (draw mode)	`0x7c 0x4c <x1> <y1> <x2> <y2>`	
Draw lines	`0x7c 0x11 <mode> [<x> <y>]* <xn> <yn>|0x80`	`Ctrl-Q`
Draw lines (draw mode)	`0x7c 0x51 [<x> <y>]* <xn> <yn>|0x80`	
Draw mode	`0x7c 0x0d <mode>`	`Ctrl-M`
Draw pixel	`0x7c 0x10 <x> <y> <mode>`	`Ctrl-P`
Draw pixel (draw mode)	`0x7c 0x50 <x> <y>`	
Draw polygon	`0x7c 0x1a <mode> [<x> <y>]* <xn> <yn>|0x80`	`Ctrl-Z`
Draw polygon (draw mode)	`0x7c 0x5a [<x> <y>]* <xn> <yn>|0x80`	
Draw rounded box	`0x7c 0x09 <x1> <y1> <x2> <x2> <radius> <mode>`	`Ctrl-I`
Draw rounded box (draw mode)	`0x7c 0x49 <x1> <y1> <x2> <x2> <radius>`	
Echo character	`0x7c 0x17 <char>`	`Ctrl-W`
Erase block	`0x7c 0x05 <x1> <y1> <x2> <x2>`	`Ctrl-E`
Factory reset	`0x7c 0x1f`	
Fill box	`0x7c 0x06 <x1> <y1> <x2> <x2> <pattern>`	`Ctrl-F`
Fill box (draw mode)	`0x7c 0x46 <x1> <y1> <x2> <x2>`	
Font mode	`0x7c 0x0a <mode>`	`Ctrl-J`
Font set (non-persistent)	`0x7c 0x48 <font>`	
Font set (persistent)	`0x7c 0x08 <font>`	`Ctrl-H`
Graphics mode off	`0x7c 0x41`	
Graphics mode on	`0x7c 0x40`	
Query LCD	`0x7c 0x1e`	
Reset LCD	`0x7c 0x20`	
Reverse mode (non-persistent)	`0x7c 0x52`	
Reverse mode (persistent)	`0x7c 0x12`	`Ctrl-R`
Set LCD	`0x7c 0x1b 0xc5 <id> <value>`	
Set x and y position	`0x7c 0x58 <x> <y>`	
Set x position	`0x7c 0x18 <x>`	`Ctrl-X`
Set y position	`0x7c 0x19 <y>`	`Ctrl-Y`
Set string position	`0x7c 0x59 <x> <y> <string> 0xff`	
Splash screen toggle	`0x7c 0x13`	`Ctrl-S`
Sprite draw	`0x7c 0x14 <x> <y> <id> <mode>`	`Ctrl-T`
Sprite upload	`0x7c 0x15 <id> <width> <height> <sdata>*`	`Ctrl-U`

Table 4: Serial Commands

3.1 Backlight level

NAME

Set backing duty cycle

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_SET_BACKLIGHT   0x02
#define GLCD_CMDX_SET_BACKLIGHT 0x42

SYNOPSIS

0x7c 0x02 *<percentage>* - Persistent
0x7c 0x42 *<percentage>* - Non-persistent

DESCRIPTION

Sets the the back light brightness and saves the value to EEPROM. Use command 0x42 where the backlight brightness is temporarily changed without saving the value to EEPROM.

The argument <percentage> is the brightness specified as a percentage 0-100.

EXAMPLE

0x7C 0x02 0x32 sets the back light to 50%.

SEE ALSO

GLCD::setBacklight(), GLCD::updateBacklight().

3.2   Change Baud Rate

NAME

Change the baud rate.

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_CHANGE_BAUD_RATE 0x07

SYNOPSIS

0x7c 0x07 *<baud_rate>*

DESCRIPTION

Changes the default baud rate, the setting is persistent over power cycles. Where <baud_rate> is a value 1-6 or ASCII '1'-'6' defined as follows:

1 - 4800
2 - 9600
3 - 19200
4 - 38400
5 - 57600
6 - 115200 (Default)

EXAMPLE

0x7c 0x07 0x05 sets the baud rate to 57600.

SEE ALSO

GLCD::setBaud(), GLCD::restoreDefaultBaud().

3.3   Clear Screen

NAME

Clears the screen.

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD           0x7c
#define GLCD_CMD_CLEAR_SCREEN   0x00
```

SYNOPSIS

`0x7c 0x00`

DESCRIPTION

Clears or sets all pixels on the screen (depending on whether reverse is set). The command sets the `x` and `y` character position to (0, 0).

SEE ALSO

GLCD::clearScreen().

3.4   Demo

NAME

Shows the splash screen information page.

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_DEMO   0x04

SYNOPSIS

0x7c 0x04

DESCRIPTION

Command shows the information splash screen page which shows information on the current settings of the LCD. The display remains visible until a character is received on the serial.

SEE ALSO

GLCD::demo().

3.5   Draw bitblt

NAME

Draw a bitmap graphic to the screen.

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_BITBLT        0x16
```

SYNOPSIS

0x7c 0x16 *<x> <y> <mode> <width> <height> <sdata>**

DESCRIPTION

The Bitblt command allows graphics to be immediately drawn to the screen like sprites, but the data does not have to be uploaded first. There are no size restrictions other than the data must fit the drawable area of the display. The arguments are defined as follows:

  <x> The x coordinate 0-127 (small) / 0-159 (large)

  <y> The y coordinate 0-63 (small) / 0-127 (large)

  <mode> The drawing mode (XOR, Copy etc.). See draw mode. Normal drawing (0x01) mode is generally used and this performs a copy operation, over-writing anything on screen. The logical operator modes may be used with Bitblt.

  <width> The width of the sprite in pixels.

  <height> The height of the sprite in pixels.

  <sdata> The image data bytes. There are <width> * <height> bytes of image data.

The width and height are ordered so that sprite data may be sent directly to BitBlt from a file with a terminal program. Send the x, y, mode bytes first and then send the file.

SEE ALSO

Sprite data format, Sprite draw, Sprite upload, GLCD::bitblt(), GLCD::drawSprite(), GLCD::loadSprite().

3.6   Draw box

NAME

Draw a rectangular box

LIBRARY

#include <AltSerialGraphicLCD.h>

```
#define GLCD_CHAR_CMD           0x7c
#define GLCD_CMD_DRAW_BOX       0x0f
#define GLCD_CMDX_DRAW_BOX      0x4f
```

SYNOPSIS

0x7c 0x0f *<x1> <y1> <x2> <x2> <mode>*
0x7c 0x4f *<x1> <y1> <x2> <x2>*

DESCRIPTION

This command draws an outline of a box with opposing corners (x1, y1) and (x2, y2). <mode> defines the line type which respects reverse; when omitted then *draw mode* is used.

EXAMPLE

0x7c 0x0f 0x04 0x05 0x0f 0x10 0x01 draws a box from (4,5) to (15,16).

SEE ALSO

Draw rounded box, Erase block, Fill box,
GLCD::drawBox(), GLCD::eraseBox(), GLCD::fillBox(), GLCD::drawRoundedBox().

3.7   Draw circle

NAME
Draw a circle

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD           0x7c
#define GLCD_CMD_DRAW_CIRCLE     0x03
#define GLCD_CMDX_DRAW_CIRCLE    0x43
```

SYNOPSIS

0x7c 0x03 *<x> <y> <radius> <mode>*
0x7c 0x43 *<x> <y> <radius>*

DESCRIPTION
Draws a circle of radius `<radius>` with the centre point defined by (`<x>`, `y`). The line colour is defined by the `<mode>` field; when omitted then *draw mode* is used.

SEE ALSO
Draw mode, GLCD::drawCircle(), GLCD::drawMode().

3.8   Draw line

NAME

Draw a line

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_DRAW_LINE   0x0c
#define GLCD_CMDX_DRAW_LINE   0x4c

SYNOPSIS

0x7c 0x0c *<x1> <y1> <x2> <y2> <mode>*
0x7c 0x4c *<x1> <y1> <x2> <y2>*

DESCRIPTION

Draws a line on the screen from screen coordinate (<x1>, y1) to coordinate (x2, y2) inclusive. The line colour is defined by the <mode> field; when omitted then *draw mode* is used.

SEE ALSO

Draw lines, Draw mode, GLCD::drawLine(), GLCD::drawLines(), GLCD::drawMode().

3.9   Draw lines

NAME
Draw multiple connected lines

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_DRAW_LINES   0x11
#define GLCD_CMDX_DRAW_LINES   0x51

SYNOPSIS
0x7c 0x11 *<mode>* [*<x> <y>*]* *<xn> <yn>*|0x80
0x7c 0x51 [*<x> <y>*]* *<xn> <yn>*|0x80

DESCRIPTION
Draws multiple connected lines from a list, the last coordinate pair in the list (<xn>, <yn>) is identified as the end of the list by ORing 0x80 with the <yn> value. The colour of the line is defined by the <mode>; when omitted then *draw mode* is used.

SEE ALSO
Draw line, Draw mode, GLCD::drawLine(), GLCD::drawLines(), GLCD::drawMode().

3.10   Draw mode

NAME

Set the current draw mode

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_DRAW_MODE     0x0d

#define GLCD_MODE_NORMAL       0x01
#define GLCD_MODE_REVERSE      0x00
#define GLCD_MODE_OR           0x02
#define GLCD_MODE_XOR          0x04
#define GLCD_MODE_NAND         0x06
#define GLCD_MODE_FILL         0x08
#define GLCD_MODE_CENTER       0x08
```

SYNOPSIS

0x7c 0x0d *<mode>*

DESCRIPTION

Sets the default drawing mode which is used in draw commands that omit the `<mode>` parameter. The `<mode>` is a bit field which is interpreted as follows:

`0x01` Bit 0 determines whether the drawing is rendered in normal (1) or reverse (0) mode. Where normal means that a pixels is set and reverse the pixel is cleared. When the LCD is in reverse mode then normal mode clears the bit.

`0x02` OR mode; the pixel is bitwise OR'ed with the screen i.e. `(screen | pixel)`. This sets the screen at the pixel.

`0x04` XOR mode; the pixel is bitwise XOR'ed with the screen i.e. `(screen ^ pixel)`. This inverts the pixel at the screen when the pixels are different.

`0x06` NAND mode; the pixel is bitwise NAND'ed with the screen i.e. `(screen & ~pixel)`. This clears the screen at the pixel.

`0x08` Fill mode; any shape is filled, this operates with polygons, boxes and circles.

`0x08` Center mode; this flag is only valid with Sprite Draw and draws the sprite centred at the (x,y) position rather than using the top left corner as the location point..

EXAMPLE

`0x7c 0x0d 0x01` - The normal rendering mode.
`0x7c 0x0d 0x05` - Draw using XOR.
`0x7c 0x0d 0x0d` - Fill using XOR.

SEE ALSO

Introduction to Drawing Modes
Sprite draw, Font Mode, GLCD::drawMode(), GLCD::fontMode(), GLCD::drawSprite().

30

3.11 Draw pixel

NAME

Draw a pixel

LIBRARY

#include <AltSerialGraphicLCD.h>

```
#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_DRAW_PIXEL    0x10
#define GLCD_CMDX_DRAW_PIXEL   0x50
```

SYNOPSIS

0x7c 0x10 *<x> <y> <mode>*
0x7c 0x50 *<x> <y>*

DESCRIPTION

Sets or clears a pixel at coordinate (<x>, <y>) with colour defined by the <mode> field; when omitted then *draw mode* is used.

Draw pixel with x,y set to 0xff, 0xff may be effectively used as a NULL command as the pixel is off-screen and is not rendered. This is the suggested NULL command used for reseting the screen. See Reset LCD.

SEE ALSO

Draw mode, GLCD::drawMode(), GLCD::drawPixel().

3.12 Draw polygon

NAME

Draw a polygon

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_DRAW_POLYGON   0x1a
#define GLCD_CMDX_DRAW_POLYGON  0x5a
```

SYNOPSIS

0x7c 0x1a *<mode>* [*<x> <y>*]* *<xn> <yn>*|0x80
0x7c 0x5a [*<x> <y>*]* *<xn> <yn>*|0x80

DESCRIPTION

Draws a polygon which commences from (<x>, y) and draws a line to the next point defined by the next (<x>, <y>) pair. The last point of the polygon is defined by setting the top bit of the <yn> coordinate to 0x80. The last coordinate (<xn>, <yn>) is connected to the first coordinate. This command is essentially the same as Draw lines which does not close the polygon.

The line colour is defined by the <mode> field; when omitted then *draw mode* is used.

BUGS

Fill polygon is not robust with concave shapes, when filling then the first vertex should not be a horizontal line. There are also some issues with XOR mode when drawing straight lines.

SEE ALSO

Draw mode, GLCD::drawMode(), GLCD::drawPolygon().

3.13   Draw rounded box

NAME

Draw a rectangular box with rounded corners

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_DRAW_ROUNDED_BOX 0x09
#define GLCD_CMDX_DRAW_ROUNDED_BOX 0x49

SYNOPSIS

0x7c 0x09 *<x1> <y1> <x2> <x2> <radius> <mode>*
0x7c 0x49 *<x1> <y1> <x2> <x2> <radius>*

DESCRIPTION

This command draws an outline of a box with opposing corners (x1, y1) and (x2, y2). The box is drawn with rounded corners a radius of <radius>. <mode> defines the line type which respects reverse; when omitted then *draw mode* is used.

SEE ALSO

Draw box, Draw mode, Fill box, GLCD::drawBox(), GLCD::drawMode(), GLCD::drawRoundedBox().

3.14   Echo character

NAME

Echo a character on the serial port

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_ECHO   0x17

SYNOPSIS

0x7c 0x17 *<char>*

DESCRIPTION

Echos the character <char> on the RX port when the command is executed.

The command may be used as a trigger by the Host to inform when the drawing has completed i.e. for timing command execution or synchronising with the graphics when some operation has completed.

EXAMPLE

0x7c 0x17 0x53 - Echo the character 'S' on the serial when executed.

SEE ALSO

GLCD::echo(), GLCD::echoWait(), GLCD::waitc().

3.15   Erase block

NAME

Erases a rectangular region of the screen

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_ERASE_BLOCK   0x05
```

SYNOPSIS

0x7c 0x05 *<x1> <y1> <x2> <x2>*

DESCRIPTION

Clears a block on the screen honouring the current reverse mode. The arguments (<x1>, <y1>) and (<x2>, <y2>) are the coordinates of two opposite corners describing the block.

EXAMPLE

`0x7c 0x05 0x00 0x00 0x0f 0x0f` clears the rectangular block from (0,0) to (15,15) inclusive which is a block of 16x16 pixels.

SEE ALSO

Draw box, Fill box, GLCD::drawBox(), GLCD::eraseBox(), GLCD::fillBox().

3.16   Factory reset

NAME
Perform a factory reset

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD           0x7c
#define GLCD_CMD_FACTORY_RESET  0x1f
```

SYNOPSIS
0x7c 0x1f

DESCRIPTION
Command performs a factory reset by resetting all of the EEPROM persistent values and parameters back to their default values. The default values are defined as follows:

Baud rate = 115200

Splash = on / information splash screen

CRLF = on (0)

Reverse = off

Back light = 100%

Scroll = on (0)

XON = 20

XOFF = 166

It is suggested that a Reset LCD is issued following a factory reset, the screen will then start up with the new settings.

SEE ALSO
Query LCD, Reset LCD, GLCD::factoryReset(), GLCD::query(), GLCD::reset().

3.17   Fill box

NAME

Fills the block with a vertical bit pattern

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD           0x7c
#define GLCD_CMD_FILL_BOX       0x06
#define GLCD_CMDX_FILL_BOX      0x46
```

SYNOPSIS

0x7c 0x06 *<x1> <y1> <x2> <x2> <pattern>*
0x7c 0x46 *<x1> <y1> <x2> <x2>*

DESCRIPTION

This draws a filled box much like the draw box command which is described by the opposing corners (<x1>, <y1>) and (<x2>, <y2>), but fills the box with the <pattern>.

The <pattern> fill byte describes an 8-pixel high vertical stripe that is repeated every column and every 8 pixel rows. The most useful values are 0x00 to clear the box and 0xff to fill it.

When the <pattern> field is omitted then the line colour and fill is defined by the *draw mode*.

SEE ALSO

Draw box, Erase block, GLCD::drawBox(), GLCD::eraseBox(), GLCD::fillBox().

3.18 Font mode

NAME

Set the current font rendering mode

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_FONT_MODE      0x0a

#define GLCD_MODE_PROP_FONT     0x20
```

SYNOPSIS

0x7c 0x0a *<mode>*

DESCRIPTION

Sets the default drawing mode which is used in font rendering. The <mode> is a bit field which is interpreted in the same way as draw mode.

<mode> may be defined OR'ed with the value 0x20 GLCD_MODE_PROP_FONT which renders the fixed font as a proportional font, removing any excess space in characters. The proportional font results in cleaner looking text.

SEE ALSO

Draw mode, Font set, GLCD::drawMode(), GLCD::fontFace(), GLCD::fontMode(), GLCD::setFontFace().

3.19   Font set

NAME

Set the current font

LIBRARY

#include <AltSerialGraphicLCD.h>

```
#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_FONT_SET       0x08
#define GLCD_CMDX_FONT_SET      0x48
```

SYNOPSIS

0x7c 0x08 *<font>*
0x7c 0x48 *<font>*

DESCRIPTION

Sets the current font, the <font> argument defines the current font to use where 0x00 is the default 6x8 font and 0x01 is the Tom Thumb 4x6 font. Both fonts use a 1 pixel space separation between character; the actual rendered character is 5x8 and 3x6 respectively.

The Tom Thumb font is courtesy of https://www.pjrc.com/teensy/td_libs_GLCD.html.

Setting the font with the 0x08 (CMD_FONT_SET) persistently stores the font in EEPROM as the default font. Setting the font with 0x48 (CMDX_FONT_SET) temporarily changes the font without saving the preference to EEPROM.

SEE ALSO

Draw mode, Font mode, GLCD::drawMode(), GLCD::fontFace(), GLCD::fontMode(), GLCD::setFontFace().

3.20   Graphics mode

NAME

Set the graphics mode on/off

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD            0x7c
#define GLCD_CMDX_GRAPHICS_ON    0x40
#define GLCD_CMDX_GRAPHICS_OFF   0x41
```

SYNOPSIS

0x7c 0x40 - Graphics mode on
0x7c 0x41 - Graphics mode off

DESCRIPTION

Graphics mode moves the LCD screen into a graphics only function so that each command does not need to be prefaced by the 0x7c command character. When graphics mode is entered then the 0x7c command leader is dropped and commands only should be sent to the display with no characters, allowing drawing commands to be batched together.

Graphics mode may be disabled when drawing has completed, moving back to a normal command and character interaction. Graphics mode is exited with command 0x41 or by sending a command prefixed with 0x7c.

EXAMPLE

The following example shows how the graphics mode is used to batch together drawing commands.

```
0x7c 0x40                          - Enter graphics mode
0x0c <x1> <y1> <x2> <y2> <mode>    - draw a line
0x0c <x1> <y1> <x2> <y2> <mode>    - draw a line
0x10 <x> <y> <mode>                - set pixel
0x41                               - exit graphics mode.
0x7c <x1> <y1> <x2> <y2> <mode>    - draw a line (non graphics)
```

SEE ALSO

GLCD::setGraphics().

3.21 Query/Set LCD

NAME

Query or set the LCD state

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD              0x7c
#define GLCD_CMD_QUERY             0x1e
#define GLCD_CMD_SET               0x1b

// The Set LCD check byte used as the 1st parameter with GLCD_CMD_SET
#define GLCD_CMD_SET_CHECKBYTE     0xc5

// The query/set identities
#define GLCD_ID_MAGIC              0x00 /* Magic number to handle new install */
#define GLCD_ID_BAUDRATE           0x01 /* Baud rate */
#define GLCD_ID_BACKLIGHT          0x02 /* Backlight level */
#define GLCD_ID_SPLASH             0x03 /* Splash screen enabled */
#define GLCD_ID_REVERSE            0x04 /* Reverse the screen */
#define GLCD_ID_XON_POS            0x07 /* XON position */
#define GLCD_ID_XOFF_POS           0x08 /* XOFF position */
#define GLCD_ID_SCROLL             0x09 /* Scroll on/off */
#define GLCD_ID_LARGE_SCREEN       0x0a /* Large screen. */
#define GLCD_ID_FONT               0x0b /* Selected characterset */

#define GLCD_ID_VERSION_MAJOR      0x20 /* Firmware version major (Read only) */
#define GLCD_ID_VERSION_MINOR      0x21 /* Firmware version major (Read only) */
#define GLCD_ID_EEPROM_SPRITE_SIZE 0x22 /* EEPROM sprite byte size (Read only) */
#define GLCD_ID_EEPROM_SPRITE_NUM  0x23 /* Number of EEPROM sprites (Read only) */
#define GLCD_ID_RAM_SPRITE_SIZE    0x24 /* RAM sprite byte size (Read only) */
#define GLCD_ID_RAM_SPRITE_NUM     0x25 /* Number of RAM sprites (Read only) */

#define GLCD_ID_X_DIMENSION        0x40 /* Screen X dimension (Read only) */
#define GLCD_ID_Y_DIMENSION        0x41 /* Screen Y dimension (Read only) */

#define GLCD_ID_ESPRITE_WIDTH_0    0x80 /* EEPROM sprite[0] width (Read only) */
#define GLCD_ID_ESPRITE_HEIGHT_0   0x81 /* EEPROM sprite[0] height (Read only) */

// For EEPROM sprite[1..n] then add 2 for each sprite.
// i.e. sprite[4].width = (GLCD_ID_ESPRITE_WIDTH_0 + (4*2))
```

SYNOPSIS

0x7c 0x1e *id* - Query
0x7c 0x1b 0xc5 *id value* - Set

DESCRIPTION

Query (`0x7c 0x1e`) the internal state of the system. The call has a single parameter *id* which identifies the information that is being requested. The current settings of the screen are returned to the caller via the serial. The format of the returned data commences with an ASCII 'Q' (`0x51`) followed by a single byte which is the information corresponding to the *id* requested. If the *id* is not recognised then `0xff` is returned.

The version request is the only command that returns more than 1 byte, this returns multiple ASCII characters which are nil (`0x00`) terminated.

The set command (`0x7c 0x1b`) updates the EEPROM setting of the screen. The first parameter is `0xc5` used as confirmation that this really is an intentional write operation, if any other value is received then the command is ignored. *id* which identifies the information to be updated with the new *value*.

The values of *id* are defined below in Table 5, the **R/W** indicates if the index may be written where `RW` is read and write, `RO` is read only and cannot be updated with the **set** command.

Value	GLCD_ID_ Name	R/W	Description
0x00	MAGIC	RW	The EEPROM magic number identifying the layout of EEPROM. Modifying this parameter causes a factory reset to be performed on the next reset or power-on.
0x01	BAUDRATE	RW	The current baud rate in EEPROM. Values are `1` = 4800, `2` = 9600, `3` = 19200, `4` = 38400, `5` = 57600 and `6` = 115200. Modifying this parameter has no effect until the next reset or power-on.
0x02	BACKLIGHT	RW	The backlight level expressed as a percentage `0` = off, `100` = full brightness. Modifying this parameter has no effect until the next reset or power-on.
0x03	SPLASH	RW	Splash screen setting. Values are defined as `0` = off, `1` = information splash screen and `2` = logo splash screen. Modifying this parameter has no effect until the next reset or power-on; use Toggle splash.
0x04	REVERSE	RW	Reverse screen setting. Values are defined as `0` = Reverse screen and `1` = Normal screen. Modifying this parameter causes all subsequent draws to use reverse mode however the display is not reversed; use Reverse mode instead.
0x05		RW	Reserved for future use.
0x06	CRLF	RW	CRLF Line ending handling. Values are defined as `0` = a LF (\n) advances to the beginning of the next line. `1` = a LF (\n) advances to the next line in the same position. Modifying this parameter causes all subsequent draws to use the new mode. See GLCD::setCRLF().
0x07	XON_POS	RW	The XON position. See GLCD::setXon().
0x08	XOFF_POS	RW	The XOFF position. GLCD::setXoff().
0x09	SCROLL	RW	Causes the screen to scroll at the bottom of the page. Values are defined as `0` = the screen is scrolled up by 1 lines when the character position advances off the bottom of the page. `1` = the *y* position moves back to the top of the page when the character position advances off the bottom of the page. See GLCD::setScroll().

Table 5: Query identities (continued ...)

Value	GLCD_ID_ Name	R/W	Description
0x0a	LARGE_SCREEN	RW	The screen format. The values are defined as `0x00` = 128x64 small screen. `0x08` = 160x128 large screen. Modifying this parameter has no effect until the next reset or power-on but may cause the screen to stop functioning if the incorrect parameter is sent; the value should be defined automatically by Factory reset.
0x0b	FONT	RW	The currently selected font. The values are defined as `0x00` = Normal 6x8 font. `0x01` = Small 4x6 font. Modifying this parameter causes all subsequent character rendering to use the new font. See GLCD::setFontFace().
0x20	VERSION_MAJOR	RO	The screen software firmware version major number.
0x21	VERSION_MINOR	RO	The screen software firmware version major number.
0x22	EEPROM_SPRITE_SIZE	RO	The size of the EEPROM sprites in bytes. Current value is 34 bytes.
0x23	EEPROM_SPRITE_NUM	RO	The number of EEPROM sprite identities available. Current value is 14 locations.
0x24	RAM_SPRITE_SIZE	RO	The size of the RAM sprites in bytes. Current value is 34 bytes.
0x25	RAM_SPRITE_NUM	RO	The number of RAM sprite identities available. Current value is 6 locations.
0x40	X_DIMENSION	RO	The width of the screen in pixels.
0x41	Y_DIMENSION	RO	The height of the screen in pixels.
0x80	ESPRITE_WIDTH_0	RO	EEPROM sprite[0] width in pixels.
0x81	ESPRITE_HEIGHT_0	RO	EEPROM sprite[0] height in pixels.
0x82	ESPRITE_WIDTH_0 + 2	RO	EEPROM sprite[1] width in pixels.
0x83	ESPRITE_HEIGHT_0 + 2	RO	EEPROM sprite[1] height in pixels.
0x8n+0	ESPRITE_WIDTH_0 + 2n	RO	EEPROM sprite[n] width in pixels.
0x8n+1	ESPRITE_HEIGHT_0 + 2n	RO	EEPROM sprite[n] height in pixels.

Table 5: Query identities

NOTES

The **set** command allows the XON and XOFF buffer fullness to be modified to match the Host serial buffering.

The display implements XON/XOFF flow control using a 256 byte buffer; when the buffer is nearing full containing <xoff> bytes then a XOFF (0x13) character is sent to the Host to request transmission be stopped. An XOFF is then sent on every additional character received by the LCD until the Host stops sending characters. When an XOFF has been sent then the LCD continues to process commands from the serial buffer emptying it, when the are only <xon> bytes remaining in the buffer then an XON (0x11) character is sent to the Host notifying that transmission may be continued.

The default values are <xon> = 20 and xoff = 166. The <xoff> value is very low and has been set up for Mac (OS-X) running with a FTDI cable which can over-run XOFF by 60-80 characters. The XOFF value

may be increased if required but is unlikely to make any significant difference to the command throughput.

The serial ISR driver will send an `0xff` just as it over-runs the buffer, this error is not recoverable and generally results in screen corruption and commands going out of synchronisation. Should a `0xff` be received by the Host then the XOFF size should be reduced to prevent the buffer from over-running.

The XOFF size depends on how quickly the Host is able to react to the software flow control and stop sending characters, this may depend on the amount of buffering that is performed by the Host in the serial I/O sub-system. The XOFF value also provides a good indication as to the number of bytes that can be safely sent by the Host where the return serial is polled.

The current XON/XOFF values may be determined with the **Query** command.

SEE ALSO

Backlight level, Factory Reset, Reverse mode, Serial Overview, Splash screen toggle, GLCD::factoryReset(), GLCD::query(), GLCD::set(), GLCD::setBacklight(), GLCD::setCRLF(), GLCD::setFontFace(), GLCD::setScroll(), GLCD::setXon(), GLCD::setXoff(), GLCD::toggleSplash(), GLCD::waitc().

3.22 Reset LCD

NAME

Reset the LCD hardware

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_RESET   0x20

SYNOPSIS

0x7c 0x20

DESCRIPTION

Reset the LCD screen and hardware to its initial state. The firmware on the backpack is re-started from the beginning. The screen is cleared, the cursor position is set to (0,0) and any transient states such as reverse screen, screen dimming etc. are removed. RAM based sprites that have been loaded are removed.

The reset has completed once an XON character (0x11) has been received.

It is generally wise to initiate a reset on starting the Host program when a previous run may have terminated mid command leaving the screen in an undefined state.

On the backpack it is not possible to detect and intercept the screen reset directly on the serial input as any byte sequence may be legitimate in a sprite; there are no other control lines that may be used to signal a display reset. The reset is ideally initiated once the screen is in a known state and responding correctly, the Echo character command may be used to determine when the screen is in a good state.

Variable length commands such as Draw bitblt and Draw polygon are potential commands that may cause the display to appear to lock up if insufficient bytes are sent or polygon has not been terminated with 0x80 correctly. Repeatedly sending a Draw pixel command with (x,y) position of (0xff, 0xff) may be used to unblock the screen by finishing an unclosed polygon or feeding a Bit blit with more bytes. *Draw pixel (0xff, 0xff)* is effectively a null command as it is clipped off screen and not drawn.

A suggested display reset sequence is defined as follows:

```
// Loop to flush any blocked or unfinished display commands
REPEAT
    Send Echo Character Command ('R')
    Poll serial for character
    IF (no characters}
    THEN
        // Attempt to flush any blocked command.
        Send Draw Pixel Command (0xff, 0xff)
        Delay a few milliseconds;
    ENDIF
}
UNTIL ('R' received)

// Reset the screen
Send Reset Command ()

// Loop and discard any characters until an XON received.
REPEAT
    Poll serial for character
UNTIL (XON received);

// Screen is reset and now ready
```

SEE ALSO

Factory Reset, GLCD::factoryReset(), GLCD::reset().

3.23 Reverse mode

NAME

Reverse the screen

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_REVERSE_MODE   0x12
#define GLCD_CMDX_REVERSE_MODE   0x52

SYNOPSIS

0x7c 0x12 - persistent
0x7c 0x52 - non-persistent

DESCRIPTION

Toggles reverse (white on black) mode. The new reverse mode inverts the screen in place, it does not clear the screen or change the text drawing position.

The persistent version of the command (0x12) stores the setting in EEPROM which is restored on the next power-on. Use the non-persistent command (0x52) if the reversal is temporary.

SEE ALSO

GLCD::reverseMode(), GLCD::toggleReverseMode().

3.24   Set position

NAME

Set the text cursor position

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_SET_X_OFFSET   0x18
#define GLCD_CMD_SET_Y_OFFSET   0x19
#define GLCD_CMDX_SET_XY_OFFSET 0x58
#define GLCD_CMDX_SET_XY_STRING 0x59
```

SYNOPSIS

0x7c 0x18 *<x>* - Set x position only

0x7c 0x19 *<y>* - Set y position only

0x7c 0x58 *<x> <y>* - Set x and y position simultaneously **0x7c 0x59** *<x> <y> <justification> <string> <0xff>* - Position a 0xff terminated string

DESCRIPTION

Sets the cursor <x> and <y> position. The next character received is rendered at the specified position. The position (0, 0) is the top left of the screen.

The GLCD_CMDX_SET_XY_STRING is used for positioning text labels when using proportional fonts. The <justification parameter specifies the justification of the string a value of 0x00 is centre justification and 0x01 is right justification. <string> are the characters to render, this must not include any new line characters which is terminated with a 0xff to indicate the end of the string.

When the position is set with any of the set methods then the *x* position is stored and defines the left margin on a *carriage return* or *newline* operation. This behaviour allows columns of text to be written using the newline without explicitly changing position.

NOTES

The GLCD_CMDX_SET_XY_STRING is terminated with 0xff as the serial input could hang up if the user has not sent the terminal character. The Reset LCD sends 0xff characters to reset the screen and will cause the command to complete if it has not been terminated correctly.

The GLCD_CMDX_SET_XY_STRING operates by peeking ahead in the serial input queue and calculates the length of the rendered string in pixels. On reaching the 0xff termination character then the XY position is set and adjusted according to the computed pixel length of the string and passed (x, y) coordinates. The position command then completes and knocks off graphics mode, if enabled. The input buffer is then read as normal rendering the characters at the designated position, the last 0xff character is ignored by the text rendering.

SEE ALSO

GLCD::setX(), GLCD::setY(), GLCD::setXY(), GLCD::setHome(), GLCD::setString(), GLCD::xdim, GLCD::ydim.

3.25   Splash screen toggle

NAME

Toggle the splash screen setting

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_TOGGLE_SPLASH  0x13
```

SYNOPSIS

0x7c 0x13

DESCRIPTION

Toggles whether or not the splash screen is displayed on startup. The setting is persistent over power cycles. There are three different settings and the splash screen toggles through each mode. The current mode may be determined with a Query LCD.

The default sprite shown at start up is EEPROM sprite identity zero (`0x80`). The default sprite is the Sparkfun logo, unless explicitly re-written by the user. The Sparkfun logo may be restored to EEPROM via a Factory reset command.

When the splash screen is enabled then the splash screen is shown at start up and is removed on receipt of the first serial character.

  `0` - Splash screen is disabled.

  `1` - Splash screen is enabled. The splash screen is shown at start up with additional information on the current settings of the display. This is the default setting.

  `2` - Splash screen is enabled, the splash screen is show at start up with no information. The sprite is placed in the centre of the screen.

SEE ALSO

Demo (Show splash screen), Factory reset Query LCD, Sprite upload, GLCD::demo(), GLCD::factoryReset(), GLCD::loadSprite(), GLCD::query(), GLCD::toggleSplash().

3.26   Sprite draw

NAME

Draw a sprite on the screen

LIBRARY

#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD   0x7c
#define GLCD_CMD_DRAW_SPRITE   0x14

SYNOPSIS

0x7c 0x14 *<x> <y> <id> <mode>*

DESCRIPTION

Draws a sprite to the screen where <x> and <y> defined the coordinates of the top left corner of the sprite draw position. Sprites are loaded by the Host with the Sprite upload command.

<id> is the identity of the sprite to draw. All sprites are 34 bytes in length, sufficient to hold a 16x16 bitmap with 2 bytes to define the width and the height of the sprite. There are 6 RAM based sprites with identities 0-5 and 14 EEPROM based sprites with identities 0x80-0x8d. EEPROM sprint identity 0x80 is the Sparkfun logo by default.

<mode> is the mode to draw the sprite; see Draw mode for a description of the modes. Where the <mode> includes the fill bit 0x08 set, then the sprite is centred about (<x>, <y>) i.e. the coordinate position defines the location of the centre of the sprite.

EXAMPLE

0x7c 0x14 0x20 0x20 0x01 0xd - XOR RAM sprite 1 with the background at location (32, 32).
0x7c 0x14 128/2 64/2 0x81 0xd - XOR EEPROM sprite 1 with the background at location (32, 32).

SEE ALSO

Sprite data format, Demo (Show splash screen), Sprite draw, Sprite upload, GLCD::demo(), GLCD::drawSprite(), GLCD::loadSprite(), GLCD::toggleSplash().

3.27   Sprite upload

NAME

Upload a sprite to the LCD

LIBRARY

```
#include <AltSerialGraphicLCD.h>

#define GLCD_CHAR_CMD          0x7c
#define GLCD_CMD_UPLOAD_SPRITE  0x15
```

SYNOPSIS

0x7c 0x15 *<id> <width> <height> <sdata>**

DESCRIPTION

Uploads a sprite to memory in the LCD. Sprites may be persistent and stored in RAM or non-persistent and stored in EEPROM. The EEPROM based sprites to not need to be re-loaded through a power cycle of the display. All sprites are 34 bytes in length, sufficient to hold a 16x16 bitmap with 2 bytes to define the width and the height of the sprite.

EEPROM sprite identity `0x80` holds the splash screen logo which is displayed on start up, see Splash screen toggle. The splash screen sprite is is the Sparkfun logo which may be over-written.

`<id>` is the identity of the sprite to draw. There are 6 RAM based sprites with identities 0-5 and 14 EEPROM based sprites with identities `0x80-0x8d`.

`<width>` defines the width of the sprite in pixels.

`<height>` defines the height of the sprite in pixels.

`<sdata>` is sprite data, the format is described in more detail in section Sprite data. The sprite data should be `<width>` * (`height` + 7)/8 bytes long.

Note: There is no requirement to send padding sprite data as required with the Jennifier Holt version.

EXAMPLE

An example sprite is the Sparkfun logo; the upload command is defined as:

```
0x7c 0x0d                                  // Sprite upload command
0x0n                                       // Sprite number
0x0a 0x10                                  // Width = 10, Height = 16
0x80 0xc0 0x40 0x0c 0x3e 0xfe 0xf2 0xe0 0xf0 0xe0 // Row 1
0xff 0x7f 0x3f 0x1f 0x1f 0x1f 0x1f 0x0f 0x07 0x03 // Row 2
0x7c ...                                    // Next command
```

SEE ALSO

Sprite data format, Demo (Show splash screen), Sprite draw, GLCD::demo(),
GLCD::drawSprite(), GLCD::loadSprite(), GLCD::toggleSplash().

4   Updating the Backpack

Updating the firmware of the backpack requires a hardware modification to provide access to the ISP header. The programming may be performed with an Ardunio, it is not necessary to purchase a AVR programmer. This section provides an overview of the programming process.

4.1   Equipment & parts

To perform the update you will minimally need the following equipment and parts.

Soldering Iron used to solder connectors to the ISP connections of the backpack.

Wires with 0.1" connectors 6x pre-crimped wires with female 0.1" connectors are required to make the ISP connector if you do not have a 0.1" crimping tool. These are soldered to the ISP connection to program the device. Ideally use a 3x2 plastic crimp connector or use insulation tape to form a 3x2 block from individual connectors.

ISP programmer or Arduino UNO an ISP programmer or a 5v Arduino may be used to program the backpack. If you are using an Arduino then a 10uF/16v capacitor (or similar) is required to stop the Arduino from resetting when programming.

Note: in the UK you can get a 10uF / 16v capacitor in Maplins with code AT98G.

Male to Male Jumper Wires 6x to connect the ISP female connectors to the Arduino.

Arduino IDE the IDE contains the ISP programmer. Version 1.6.1 is current at the time of writing.

Amtel AVR development environment this includes the compiler and AVRDUDE. If you do not want to compile the code then only AVRDUDE is required to download the hex file of the firmware image.

For OSX then one can use **CrossPack** available from:

https://www.obdev.at/products/crosspack/index.html

CrossPack is simple to download and install, providing the compilers and the AVDUDE utility. Version 20131216 is current at time of writing.

CrossPack 20131216 was used to build this firmware and included components: avarice: 2.13, avr-binutils: 2.23.2, avr-gcc: 4.8.1, avr-libc: 1.8.0, avrdude: 6.0.1, gdb: 7.6.1, libusb: 0.1.12, make: 4.0, sim-ulavr: 0.1.2.7

For Microsoft Windows and Linux refer to the Amtel website www.amtel.com for the availability of AVR tools.

4.2 Modifying the backpack

The ISP programming connector location on the GLCD backpack is shown in Figure 13 with annotations for the pin assignments.
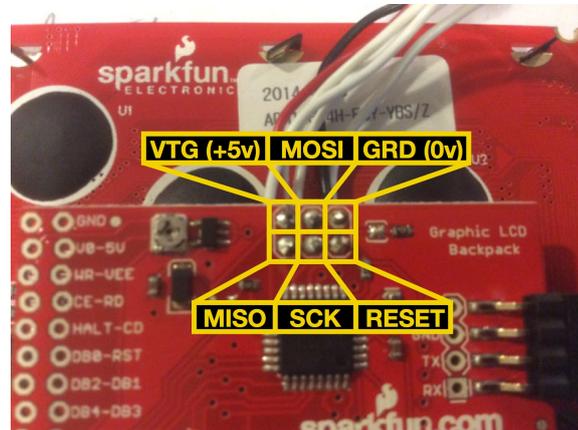
Figure 13: Programming Connectors

Connectors must be soldered to the backpack in order to program the device. The backpack is pre-soldered to the display and it is undesirable to remove the backpack in order to perform the soldering; it was considered easier to solder some fly wires wires with 0.1" female connectors attached inserting them from the underside of the backpack board and soldering from the top of the board. Threading the wires from the underside of the backpack is a little fiddely and it is best to strip the wires and pre-tin them with solder and bend them at the insulation to a 90 degree angle so that the wire can be slid under the backpack and passed through the hole. Ensure that the bend is located at the insulation so that there are no shorts between the different pins as they are located close together.

Figure 14 shows the attached wiring where 0.1" female connectors have been used on a short wire for programming . The orientation of the board connector at the end of the wiring is retained in the same orientation as the backpack using a 3x2 connection block configuration to avoid any confusion when connecting the device for programming.
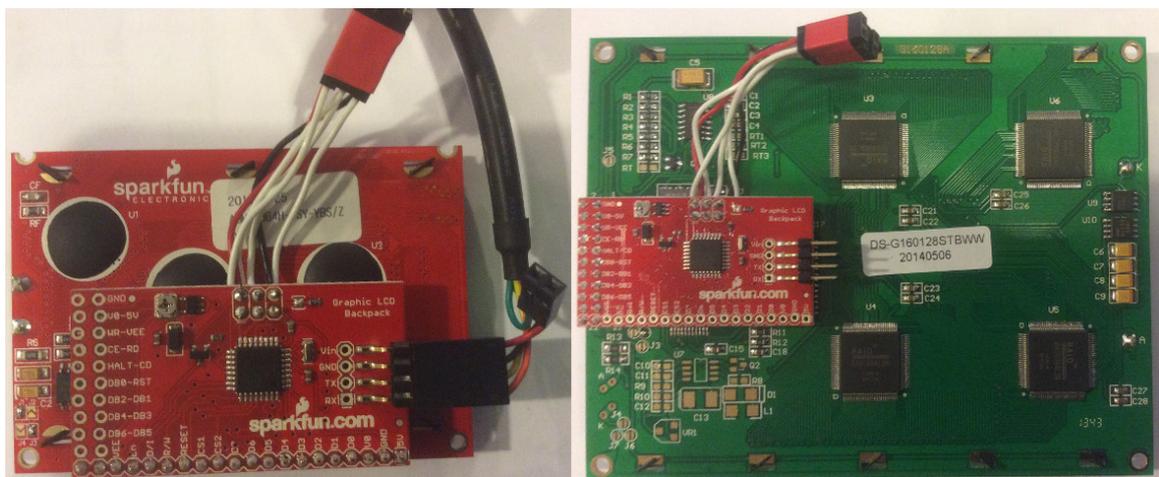
Figure 14: Screen Connectors

4.3 Preparing to program with an Arduino

Where the Arduino UNO is used for programming then the Arduino should be loaded with the **ISP programmer** sketch; this enables the Arduino to be used as a AVR programmer. Run the Arduino IDE and open the sketch **ArduinoISP**, compile and upload to the Arduino. The Arduino is now programmed as a ISP programmer.

Once the Arduino is programmed it may be connected to the screen. The wiring of the Arduino for an ISP programmer is described on the Sparkfun website, refer to Installing an Arduino Bootloader tutorial. The connections between the Arduino and backpack is shown in Figure 15.
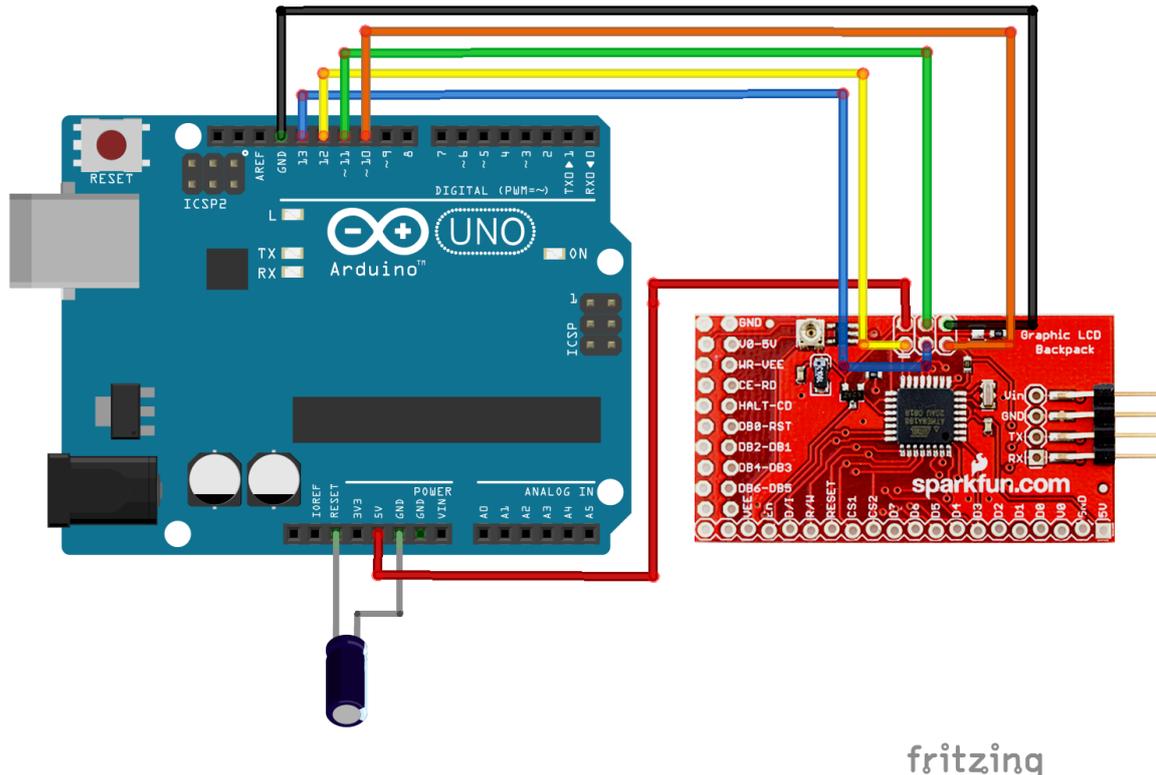
Figure 15: Arduino connections for ISP programming

Table 6 shows the board interconnections, the colours relate to the Fritzing diagram.

Colour	Arduino	Graphic LCD
red	5v/Vcc	Pin 2 / VTG (VCC)
black	GRD	Pin 6 / GRD
green	D11	Pin 4 / MOSI
yellow	D12	Pin 1 / MISO
blue	D13	Pin 3 / SCK
orange	D10	Pin 5 / Reset

Table 6: Arduino ISP connections

Use a 10uF / 16v (or equivalent) capacitor placed between the **Reset** and **GND** pins of the Arduino, this stops the Arduino from reseting when programming. Note: the -ve pin of the capacitor is connected to **GND**.

For programming the author used a Ciseco Xino RF board, this is equivalent to an Arduino Uno board. This is wired up to the serial backpack ready for programming as shown in Figure 16.
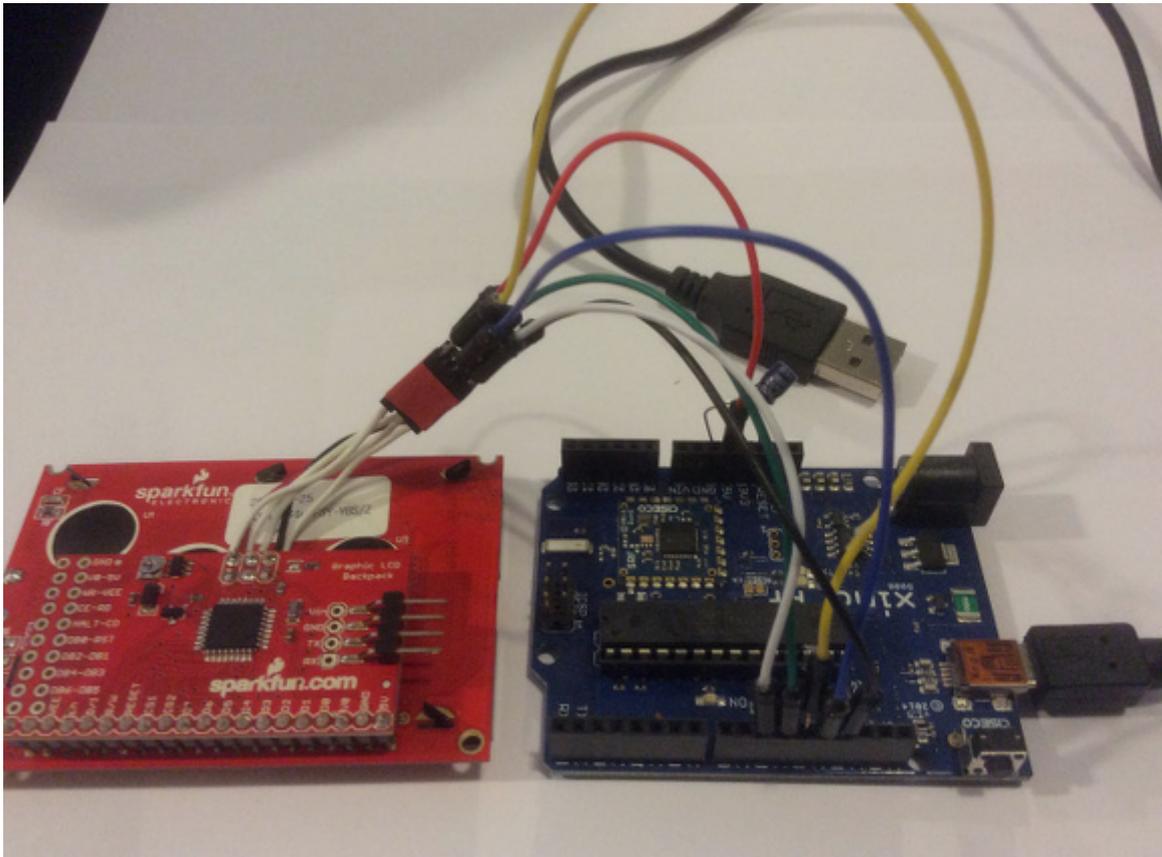
Figure 16: Backpack Programming from an Arduino

4.4 Programming the backpack

Once the Ardunio has been prepared (or ISP connected) the backpack may be programmed with the firmware image.

The **Makefile** in the firmware directory is configured to use an Arduino as the ISP programmer and may need to be changed to match the programmer by changing the Makefile `AVRDUDE_PROGRAMMER` setting. The port used for programming `AVRDUDE_PORT` and `AVRDUDE_BAUD_RATE` also need to be changed to match the device connection.

The simplest option is to issue a `make program` command which builds the firmware and downloads to the backpack in a single operation.

Where the `.hex` file is to loaded without building then **avrdude** may be used directly from the command line as follows:

```
avrdude -p atmega168 -P /dev/cu.usbmodem000001 -c arduino -b 19200 -U flash:w:main.hex
```

the argument `/dev/cu.usbmodem000001` is the serial device which should be changed to match the serial device of your own system.

When the command is issued then the backpack programming proceeds as follows:

```
zsh% make program

This will program the Sparkfun Graphics LCD.

cd /Users/jon/dev/serialGLCD/firmware/trunk/ make program

avrdude -p atmega168 -P /dev/cu.usbmodem000001 -c arduino -b 19200 -U flash:w:main.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.02s

avrdude: Device signature = 0x1e9406
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "main.hex"
avrdude: input file main.hex auto detected as Intel Hex
avrdude: writing flash (11278 bytes):

Writing | ################################################## | 100% 14.32s

avrdude: 11278 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex auto detected as Intel Hex
avrdude: input file main.hex contains 11278 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 8.41s

avrdude: verifying ...
avrdude: 11278 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.
```

The programming command may sometimes fail when the serial device is not ready, the command can usually be repeated. Where the programming command fails repeatedly then it is likely that the ISP connections are incorrect and should be re-checked.

As soon as programming is complete then the display is ready to use.

5 Arduino Alternative Serial Graphic LCD Library

This section describes the `AltSerialGraphicLCD` library for the Arduino which defines a single class `GLCD` to manage the Serial LCD.

5.1 Installation of the Library

The Arduino Alternative Serial Graphic LCD Library is installed into the Arduino IDE by copying the supplied `Arduino/libraries/AltSerialGraphicLCD` directory to the IDE `Arduino/libraries` directory. The whole subtree is copied which includes the GLCD library and some example programs.

When the files are copied then the Arduino IDE should be closed and re-started.

5.2 Example Applications

Once the library is installed in the Arduino IDE then the example code is found when opening a sketch. *Open → libraries → AltSerialGraphicLCD*. All of the examples have been run on a Arduino UNO (328). There are four example programs as follows, all examples run with both the small (128x64) and large (160x128) screens with no changes:

AltSerialGraphicLCDBenchmark this is a simple benchmark application for speed testing. This has been adapted from:

> https://www.pjrc.com/teensy/td_libs_GLCD.html

> and is provided as more of a curiosity to see the drawing performance. The author has not found any alternative definitive benchmark code.

AltSerialGraphicLCDTest this sketch contains the test application which as been used to check the implementation. The sketch comprises of a number of drawing tests that continually loop performing a set of drawing operations and then wait 5 seconds to allow the screen to be inspected before moving onto the next test.

> The test file is large as it contains sprites and other drawing material, some of the tests may need to disabled to run on and Arduino with less resources than the UNO.

SerialGraphicLCDDemo this is the original Sparkfun demo with a few bug fixes and ported to this new library. This has been used as a reference to ensure that the serial commands have been kept as close to the original Sparkfun distribution as possible.

> Running the demo is useful when the baud rate is lost as this application resynchronises the baud rate with the screen.

SimpleApplication this is a simple application to get the new user started and demonstrates how the library is used. The operation of this application is explained in greater detail in the next section which walks through how it is construted.

5.3 Simple Application

The Alternative Serial Graphic LCD Firmware library is used with the **SoftwareSerial** library. A simple sketch using the library is defined as follows which outlines how the library is used with a simple example. This example is provided and may be executed.

```c++
/* -!- c++ -!- ***********************************************************
 * Alternative Serial Graphic LCD Library Demo by Jon Green May 24, 2015
 *
 * This is a simple application to draw some text on the screen. The Sparkfun
 * demo application provided a good starting page for "Hello World".
 */

#include <AltSerialGraphicLCD.h>
#include <SoftwareSerial.h>

// Define the TX and RX pins used to connect the screen. Change these two pin
// values to whichever pins you wish to use (RX, TX).
//#define SERIAL_TX_DPIN    3
//#define SERIAL_RX_DPIN    2
#define SERIAL_TX_DPIN   12
#define SERIAL_RX_DPIN   10

// Initialize an instance of the SoftwareSerial library
SoftwareSerial serial (SERIAL_RX_DPIN,SERIAL_TX_DPIN);

// Create an instance of the GLCD class named glcd. This instance is used to
// call all the subsequent GLCD functions. The instance is called with a
// reference to the software serial object.
GLCD glcd(serial);

static uint32_t counter = 0;          // Counter for number of iterations.
static uint32_t start_time;           // The time we started running.

////////////////////////////////////////////////////////////////////////////
// Perform significant initialisation.
void setup()
{
    // Start the Software serial library we run at 115200 by default.
    serial.begin(115200);

    // The first call is reset to the sceeen. This puts it into a sane state
    // irrespective of the state that we last left it in. Following a reset
    // then the screen is clear and the cursor is at location 0,0. Reset can
    // be called at any time, not just at the start.
    glcd.reset();

    // Initialise our simple clock so we can keep a time count.
    start_time = millis();
}

////////////////////////////////////////////////////////////////////////////
// Execution loop
void loop()
{
    uint32_t diff_time;               // Variable for the time difference
    char buffer [20];                 // Character buffer for strings
    uint8_t x_pos_1_4;                // 1/4 of horizontal screen
    uint8_t x_pos_3_4;                // 3/4 of horizontal screen
    uint8_t radius;                   // Radius of circle.

    // Work out the size of the screen and calculate the 1/4 and 3/4
    // horizontal pixel positions.
    x_pos_1_4 = glcd.xdim / 4;
    x_pos_3_4 = x_pos_1_4 * 3;

    // Prints "Hello World" to the screen and draws a tiny world (circle) in
```

```
    // the right 1/4 of the screen.

    // "Hello" is 6 * 5 = 30 pixels long, place at 3/4 of screen at the top.
    glcd.setXY(x_pos_3_4 - 15, 0);
    glcd.printStr(F("Hello"));          // Print "Hello"

    // "World" is 6 * 5 = 30 pixels long and 8 pixels high place at 3/4 of
    // screen at the bottom.
    glcd.setXY(x_pos_3_4 - 15, glcd.ydim - 8);
    glcd.printStr(F("World"));          // Print "World"

    // Compute the radius of the circle, draw as large as possible
    // considering the size of the screen.
    radius = (glcd.ydim - 24) / 2;
    if (x_pos_1_4 - 1 < radius)
        radius = x_pos_1_4 - 1;

    // Draw a circle in the middle of the screen leaving 12 pixels at the top
    // and bottom of the screen.
    glcd.drawCircle (x_pos_3_4,              // Horizontal 3/4 left
                    (glcd.ydim / 2),         // Vertical middle
                    radius,                  // Radius is 1/2 remaining height.
                    GLCD_MODE_NORMAL);       // Write normally

    // Draw the Sparkfun logo sprite (sprite_id=0x80) as we know it is
    // loaded. Position 1/4 fscreen width from the left and in the middle.
    glcd.drawSprite (x_pos_1_4, glcd.ydim / 2, 0x80,
                    GLCD_MODE_CENTER|GLCD_MODE_NORMAL);

    // Add some animation to spice it up a bit.

    // Print counter of number of iterations at top left of screen, use a
    // long number so it can run for a long time without wrapping.
    glcd.setXY(0, 0);                        // Top of screen
    sprintf (buffer, "%ld", counter++);
    glcd.printStr(buffer);

    // Print our running time at the bottom left of screen. Display hours,
    // minutes, seconds and milliseconds.
    glcd.setXY(0, glcd.ydim - 8);       // Bottom of screen – char height
    diff_time = millis() - start_time;
    sprintf (buffer, "%02d:%02d:%02d.%03d",
            (int)(diff_time / (1000L * 60L * 60L)),
            (int)((diff_time / (1000L * 60L)) % 60L),
            (int)((diff_time / 1000L) % 60L),
            (int)(diff_time % 1000L));
    glcd.printStr(buffer);
}
```

The script operates as follows:

Declaration: include the `AltSerialGraphicLCD.h` and `SoftwareSerial.h` include files, these define the interfaces to use.

```
#include <AltSerialGraphicLCD.h>
#include <SoftwareSerial.h>
```

Configure the Serial Port: change the pin values to reflect the pins that are used for the serial port as connected to your screen. If you are using the normal serial ports then change the definitions to pins 2 and 3. In this case we are using pins 12 and 10.

```
// Define the TX and RX pins used to connect the screen. Change these two pin
```

```
// values to whichever pins you wish to use (RX, TX).
//#define SERIAL_TX_DPIN   3
//#define SERIAL_RX_DPIN   2
#define SERIAL_TX_DPIN   12
#define SERIAL_RX_DPIN   10


// Initialize an instance of the SoftwareSerial library
SoftwareSerial serial (SERIAL_RX_DPIN,SERIAL_TX_DPIN);
```

Instance the Alternative Serial GLCD class: the software serial instance is passed to the **GLCD**() constructor, this informs the GLCD library where the serial port is.

```
// Create an instance of the GLCD class named glcd. This instance is used to
// call all the subsequent GLCD functions. The instance is called with a
// reference to the software serial object.
GLCD glcd(serial);
```

Declare local variables: the example program uses two variables to count the number of times the screen has been redrawn and to calculate the running time of the program. These variables are used to animate the screen so we can see something happening.

```
static uint32_t counter = 0;          // Counter for number of iterations.
static uint32_t start_time;           // The time we started running.
```

Setup Initialisation: perform significant initialisation in the **setup**() function. This is called once when the Arduino starts up. Initialise the serial to run at 115200 which is the default serial rate of the screen. Once the serial is set up then call **reset**() this tells the LCD screen to reset itself. The **reset**() function recovers the screen to an operational state and then re-boots the screen such that it starts from afresh.

Finally we get the current time from the millisecond counter **millis**(), we save the time for the application animation and use it to display the program running time.

The initialisation is then complete.

```
//////////////////////////////////////////////////////////////////////////////
// Perform significant initialisation.
void setup()
{
    // Start the Software serial library we run at 115200 by default.
    serial.begin(115200);

    // The first call is reset to the sceeen. This puts it into a sane state
    // irrespective of the state that we last left it in. Following a reset
    // then the screen is clear and the cursor is at location 0,0. Reset can
    // be called at any time, not just at the start.
    glcd.reset();

    // Initialise our simple clock so we can keep a time count.
    start_time = millis();
}
```

Execution Loop: Now we define the execution loop, this is called continually on each iteration we draw the picture again, note that we do not clear the screen because we do not want the screen to flash. The **reset**() function in **setup**() leaves the screen clear ready to be painted.

```
//////////////////////////////////////////////////////////////////////////////
// Execution loop
void loop()
{
    uint32_t diff_time;                   // Variable for the time difference
    char buffer [20];                     // Character buffer for strings
    uint8_t x_pos_1_4;                    // 1/4 of horizontal screen
```

```
uint8_t x_pos_3_4;                 // 3/4 of horizontal screen
uint8_t radius;                    // Radius of circle.
```

Screen Dimensions: the screen dimensions are read from the screen and are valid after a **reset**() operation has been performed. The class variables **xdim** and **ydim** are the dimensions of the screen. This allows screen independent code to be written and it is not necessary to hard code a specific screen size into the application. Calculate the 1/4 and 3/4 pixel positions based on the screen size.

```
// Work out the size of the screen and calculate the 1/4 and 3/4
// horizontal pixel positions.
x_pos_1_4 = glcd.xdim / 4;
x_pos_3_4 = x_pos_1_4 * 3;
```

Draw Text: Draw the `Hello World` where `Hello` is at the top of the screen and `World` is at the bottom of the screen. The text characters are `6x8` i.e. 6 pixels wide (including the inter-character space) and 8 pixels tall. Use knowledge of the character dimensions to adjust the position of the strings to ensure that they are centred. The character is positioned relative to the top left corner of the character. Use **setXY**() to move the drawing position of the first character, the character position is advanced automatically whenever a character is written.

```
// Prints "Hello World" to the screen and draws a tiny world (circle) in
// the right 1/4 of the screen.

// "Hello" is 6 * 5 = 30 pixels long, place at 3/4 of screen at the top.
glcd.setXY(x_pos_3_4 - 15, 0);
glcd.printStr(F("Hello"));              // Print "Hello"

// "World" is 6 * 5 = 30 pixels long and 8 pixels high place at 3/4 of
// screen at the bottom.
glcd.setXY(x_pos_3_4 - 15, glcd.ydim - 8);
glcd.printStr(F("World"));              // Print "World"
```

Draw circle: Draw a circle in the middle of the screen that represents the *World*. Call the **drawCircle**() command to draw a circle about a centre point with a given radius. The line colour drawn is the foreground colour, called *normal* and sets the pixel (or clears it if the screen was in reverse mode). To clear a pixel then set the colour to *reverse*.

```
// Compute the radius of the circle, draw as large as possible
// considering the size of the screen.
radius = (glcd.ydim - 24) / 2;
if (x_pos_1_4 - 1 < radius)
    radius = x_pos_1_4 - 1;

// Draw a circle in the middle of the screen leaving 12 pixels at the top
// and bottom of the screen.
glcd.drawCircle (x_pos_3_4,            // Horizontal 3/4 left
                 (glcd.ydim / 2),      // Vertical middle
                 radius,               // Radius is 1/2 remaining height.
                 GLCD_MODE_NORMAL);    // Write normally
```

Draw sprite: The screen is a bit vacant so to fill it up then draw the Sparkfun logo in the space on the left-hand mid point of the screen using **drawSprite**(). The Sparkfun logo is used on the splash screen at start up and request it to be drawn using sprite identity `0x80`. The sprite drawn centred i.e. the `x` and `y` coordinates are the centre of the sprite using *centre* mode, if *centre* was not specified then the coordinates specify the top left position of the sprite.

```
// Draw the Sparkfun logo sprite (sprite_id=0x80) as we know it is
// loaded. Position 1/4 fscreen width from the left and in the middle.
glcd.drawSprite (x_pos_1_4, glcd.ydim / 2, 0x80,
                 GLCD_MODE_CENTER|GLCD_MODE_NORMAL);
```

Animation: add some animation to make it a bit more interesting. Display the number of times the screen has been drawn. Use **sprintf**(3) to turn the long integer counter into a ASCII string that can be sent to the screen. The counter is incremented by 1 once converted to ASCII characters. Display the counter at the top left corner of the screen (0,0).

```
// Add some animation to spice it up a bit.

// Print counter of number of iterations at top left of screen, use a
// long number so it can run for a long time without wrapping.
glcd.setXY(0, 0);                    // Top of screen
sprintf (buffer, "%ld", counter++);
glcd.printStr(buffer);
```

Running time display: display the running time in the bottom right of the screen. Get the **millis()** time and subtract the **start_time**; the difference is the elapsed time in milliseconds. Convert the elapsed time into a ASCII string to send to the screen to display using **printStr**() the running time is shown in hours, minutes, seconds and milliseconds.

```
// Print our running time at the bottom left of screen. Display hours,
// minutes, seconds and milliseconds.
glcd.setXY(0, glcd.ydim - 8);        // Bottom of screen - char height
diff_time = millis() - start_time;
sprintf (buffer, "%02d:%02d:%02d.%03d",
         (int)(diff_time / (1000L * 60L * 60L)),
         (int)((diff_time / (1000L * 60L)) % 60L),
         (int)((diff_time / 1000L) % 60L),
         (int)(diff_time % 1000L));
glcd.printStr(buffer);
}
```

The **loop()** has now finished, the Arduino will now run it again, effectively forever. On each loop invocation then the screen will be re-drawn and the display time and count of interactions will increase and visibly change on the screen.

There are no artificial delays in the program and the caller does not need to wait for the display to complete any draw operation. The display is driven as fast as possible. The display will tell the GLCD library when to stop and start sending characters. This management is transparent to the application which is able to send as fast as possible without having to perform any special logic or perform any specific timing management of the screen.

5.4 GLCD Class Methods

The methods of the GLCD class are described in the following sections. The Table 7 provides a summary of the methods.

Method	Description
bitblt() bitblt_P()	Draw a sprite or bitmap directly to the screen.
clearScreen()	Clear the screen.
demo()	Display the information splash screen (was demonstration).
drawBox() fillBox()	Draw a box or rectangle.
drawCircle()	Draw a circle.
drawLine()	Draw a line.

Table 7: GLCD Class Methods (continued ...)

Method	Description
drawLines() drawLines_P()	Draw a series of connected lines.
drawMode() fontMode()	Change the line and font drawing modes.
drawPixel() setPixel()	Draw a pixel.
drawPolygon() drawPolygon_P()	Draw a polygon.
drawRoundedBox()	Draw a box with rounded corners.
drawSprite()	Draw a sprite.
echo() echoWait()	Synchronisation; send a character to the screen to echo back on the serial line.
eraseBox() eraseBlock()	Erase a rectangular screen block.
factoryReset()	Reset the screen to a factory shipping state, reset EEPROM etc.
GLCD()	Class constructor; passed an instance of the Software Serial class.
fontFace() setFontFace()	Set the character font.
loadSprite() loadSprite_P()	Load a sprite or bitmap into the screen memory.
put()	Write a single character to the screen checking for XON/XOFF.
putcmd()	Write a command to the screen checking for XON/XOFF.
putstr() putstr_P() printNum() printStr() nextLine()	Print a nil terminated string to the screen.
query() set() setCRLF() setScroll() setXoff() setXon()	Query or set the screen information state.
ready()	Process XON/XOFF and wait for the screen to be ready.
reset()	Reset the screen, typically called at start up in **setup**().
reverseMode() toggleReverseMode()	Reverse the screen.
setBacklight() updateBacklight()	Change the backlight brightness.
setBaud() restoreDefaultBaud()	Change the baud rate of the screen and serial line or restore the default rate.
setGraphics()	Sets the screen into graphics only mode allowing the command string to be shortened.

Table 7: GLCD Class Methods (continued ...)

63

Method	Description
setX() setY() setXY() setHome() setString()	Change the cursor position for drawing text.
toggleSplash()	Change the setting of the start up splash screen.
waitc()	Wait for the specified character sent from the screen.
write() write_P()	Write a block of commands or data to the screen.
xdim ydim	Variables that define the x and y dimensions of the screen.

Table 7: GLCD Class Methods

5.5   bitblt

NAME
bitblt - Draw a sprite or bitmap directly to the screen.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

void
bitblt (uint8_t *x*, uint8_t *y*, uint8_t *mode*, uint8_t **sprite*);

void
bitblt (uint8_t *x*, uint8_t *y*, uint8_t *mode*, uint8_t *length*, uint8_t **sprite*);

void
bitblt (uint8_t *x*, uint8_t *y*, uint8_t *mode*, uint8_t *width*, uint8_t *height*, uint8_t **sprite_pixels*);

void
bitblt_P (uint8_t *x*, uint8_t *y*, uint8_t *mode*, const uint8_t PROGMEM **sprite*);

void
bitblt_P (uint8_t *x*, uint8_t *y*, uint8_t *mode*, uint8_t *length*, const uint8_t PROGMEM **sprite*);

void
bitblt_P (uint8_t *x*, uint8_t *y*, uint8_t *mode*, uint8_t *width*, uint8_t *height*, const uint8_t PROGMEM **sprite_pixels*);

DESCRIPTION
bitblt writes a sprite, or image, to the screen. The sprite must be completely on screen otherwise it is not drawn i.e. no clipping is performed.

The parameters are defined as follows:

x The x-coordinate to draw the top left of the image.

y The y-coordinate to draw the top left of the image.

mode The drawing mode, `GLCD_MODE_NORMAL` copies the sprite over any other pixels. `GLCD_MODE_NORMAL` inverts the image and copies over any other pixels. `GLCD_MODE_XOR` in conjunction with *normal* or *reverse* performs an XOR operation with whatever is on screen.

sprite is a pointer to the sprite in memory, the sprite should be in the *(w, h, pixels, ...)* format.

The second alternative form of the call takes a *length* in bytes and *sprite* pointer. This call allows a sprite defined in memory to be sent without computing the length of data from the *width* and *height*. The data pointed to by *sprite* os *length* bytes is sent as the sprite data, the sprite should be in the *(w, h, pixels, ...)* format.

The third alternative for of the call takes a *width* and *height* in pixels followed by the *sprite_pixels*, this is a pointer to a sprite excluding the width and height parameters.

The **bitblt_P** invocations are used when the sprite data is stored in `PROGMEM` the call will read the sprite data from flash memory and transfer to the screen.

EXAMPLE

The following example performs a full screen bitblt:

```
GLCD lcd(serial);
...
static const uint8_t bitmap_160x128 [] PROGMEM = {
     0xa0, 0x80   /* Width, Height */
   , 0x00, 0xc0, 0xfc, 0xfc, 0xfc, 0xfc, 0xfc, 0xfc
   , 0xfc, 0xfc, 0xfc, 0xfc, 0xfc, 0xfc, 0xfe, 0xfe
   , 0xfe, 0xfe, 0xfe, 0xfe, 0xf0, 0xf8, 0xf8, 0xf8
   ....
};


...
// Draw a full screen sprite
lcd.bitblt_P (0, 0, GLCD_MODE_NORMAL, bitmap_160x128);
```

When the size of the sprite is known then it may be easier to simply send the data as follows:

```
lcd.bitblt_P (0, 0, GLCD_MODE_NORMAL, sizeof (bitmap_160x128), bitmap_160x128);
```

There may be cases when the *width* and *height* are not available, using the same data set as above, the third form of the call is:

```
lcd.bitblt_P (0, 0, GLCD_MODE_NORMAL,
            bitmap_160x128[0],     // width
            bitmap_160x128[1],     // height
            &bitmap_160x128[2]);   // sprite pixels
```

SEE ALSO

GLCD::bitblt(), GLCD::drawSprite(), GLCD::loadSprite(), Draw bitblt serial command.

5.6   clearScreen

NAME

clearScreen - Clear the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
clearScreen ();

DESCRIPTION

This command takes no arguments and clears the screen honouring the current reverse mode such that all pixels on the screen are cleared (or set when in reverse mode). The command sets the character position to (0, 0).

A clear screen operation is not required after a reset() operation as the screen is cleared. At start up then a splash screen may be displayed, a clear screen is not required since as soon as a character is sent to the display then the splash screen is automatically removed with a clear screen operation before the first character is processed for display.

EXAMPLE

Clear screen is invoked as shown in the following example:

```
GLCD lcd(serial);
...
// Label the test
lcd.clearScreen ();
centreString_P (0, (const char PROGMEM *) F("Complex␣Write␣Test"));
centreString_P (8, (const char PROGMEM *) F("Noted␣a␣128x64␣draw"));
```

SEE ALSO

GLCD::reset(), GLCD::reverseMode(), GLCD::setXY(), Clear screen serial command.

5.7 demo

NAME

demo - Display the information splash screen (was demonstration).

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
demo ();

Description

The **demo** command is a legacy command from the original Sparkfun firmware. In this implementation then the command displays the information splash page as shown in Figure 17.

Figure 17: Information splash screen

On invocation then the screen is cleared and the information splash screen is displayed, irrespective of the current splash screen mode. The splash screen is automatically removed from the screen as soon as a new command is received on the screen serial, when the screen is cleared and the character current position is set to (0, 0).

The splash screen includes a logo to the right of the screen, this logo is defined by the contents of the EEPROM sprite with identity 0x80. The logo may be changed by reprogramming the sprite with loadSprite().

EXAMPLE

The following example displays the splash screen for 5 seconds:

```
GLCD lcd(serial);
...
lcd.demo ();
delay (5000);
...
```

SEE ALSO

GLCD::loadSprite(), GLCD::toggleSplash(), Demo serial command.

5.8 drawBox

NAME
drawBox - Draw a box or rectangle.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

void
drawBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*, uint8_t *mode*);

void
drawBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*);

void
fillBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*, uint8_t *pattern*);

void
fillBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*);

DESCRIPTION
drawBox draws the outline of a box, or rectangle, defined using a diagonal line describing opposing corners (*x1*, *y1*) and (*x2*, *y2*). *mode* defines the colour of the line and whether the rectangle is filled. Where *mode* is omitted then the drawMode() value is used. See drawMode for further information on the drawing modes.

fillBox is similar to **drawBox** and takes the argument *pattern* which is a fill byte describes an 8-pixel high vertical stripe that is repeated every column and every 8 pixel rows. The most useful values are `0x00` to clear the box and `0xff` to fill it. Where *pattern* is omitted then the drawMode() value is used.

EXAMPLE
The following example displays draws a rectangle on the screen:

```
GLCD lcd(serial);
...
// Rectangle in left side of screen
lcd.drawBox (0, 0, 64, 61, GLCD_MODE_NORMAL);
// Rounded rectangle around text area
lcd.drawRoundedBox (68, 0, 68+58-1, 61, 5, GLCD_MODE_NORMAL);
...
```

SEE ALSO
GLCD::drawMode(), GLCD::drawRoundedBox(), GLCD::eraseBlock(), Draw box serial command, Fill box serial command.

5.9   drawCircle

NAME

drawCircle - Draw a circle.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
drawCircle (uint8_t *x*, uint8_t *y*, uint8_t *radius*, uint8_t *mode*);
void
drawCircle (uint8_t *x*, uint8_t *y*, uint8_t *radius*);

DESCRIPTION

drawCircle draws a circle with given *radius* at the centre point defined by *x* and *y*. *mode* defines the colour of the line and whether the circle is filled. Where *mode* is omitted then the drawMode() value is used. See drawMode for further information on the drawing modes.

The circle centre point must exist on screen, the circle is clipped to the screen when draw when required.

EXAMPLE

The following example displays draws a circle on the screen:

```
GLCD glcd(serial);
...
// Draw a circle in the middle of the screen leaving 12 pixels at the top
// and bottom of the screen.
glcd.drawCircle (x_pos_3_4,             // Horizontal 3/4 left
                 (glcd.ydim / 2),       // Vertical middle
                 (glcd.ydim - 24) / 2,  // Radius is 1/2 remaining height.
                 GLCD_MODE_NORMAL);     // Write normally
...
```

SEE ALSO

GLCD::drawMode(), Draw circle serial command.

5.10   drawLine

NAME
drawLine - Draw a line.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

void
drawLine (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*, uint8_t *mode*);

void
drawLine (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*);

DESCRIPTION
Draw a line between two points (*x1*, *y1*) and (*x2*, *y2*). *mode* defines the colour of the line. Where *mode* is omitted then the drawMode() value is used. See drawMode for further information on the drawing modes.

EXAMPLE
```
GLCD lcd(serial);
...
for (int i = 0; i < 62; i += 4)
{
    // draw lines from upper left down right side of rectangle
    lcd.drawLine (1, 1, 63, i, GLCD_MODE_NORMAL);
}
```

SEE ALSO
GLCD::drawLines(), GLCD::drawMode(), Draw line serial command.

5.11 drawLines

NAME

drawLines - Draw a series of connected lines.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
drawLines (uint8_t *mode*, uint8_t **xylist*);

void
drawLines (uint8_t **xylist*);

void
drawLines_P (uint8_t *mode*, const uint8_t PROGMEM **xylist*);

void
drawLines_P (const uint8_t PROGMEM **xylist*);

DESCRIPTION

Draw a series of line connected lines defined by a pointer to *xylist* which is a list of (x, y) coordinates, where a line is drawn between (x_n, y_n) and (x_n+1, y_n+1). The last y-coordinate in the list is OR'ed with `0x80` which indicates the end of the list.

mode defines the colour of the line. Where *mode* is omitted then the drawMode() value is used. See draw-Mode for further information on the drawing modes.

The **drawLines_P** variant of the command accepts a *xylist* that is resident in Flash memory i.e. PROGMEN.

Where the lines are closed and the last coordinate is the same as the first coordinate then use drawPolygon().

EXAMPLE

The following example shows an example of line drawing:

```
GLCD lcd(serial);
...
static void
glcdTestDrawLinesHelper (uint8_t x_offset)
{
    uint8_t y_offset = 9;
    uint8_t width = lcd.xdim / 3;
    uint8_t height = lcd.ydim - 1 - (2 * y_offset);
    uint8_t ii;
    uint8_t lines [40];
    uint8_t width_unit = (width - 1) / 3;
    uint8_t height_unit = (height - 1) / 3;
    uint8_t indent = width / 6;

    // Draw the lines
    ii = 0;
    x_offset++;
    lines [ii++] = x_offset + indent;
    lines [ii++] = y_offset + height_unit * 2;
    lines [ii++] = x_offset + indent;
```

```
    lines [ii++] = y_offset + height_unit * 1;
    lines [ii++] = x_offset;
    lines [ii++] = y_offset + height_unit * 1;
    lines [ii++] = x_offset;
    lines [ii++] = y_offset;
    ....
    lines [ii++] = y_offset + height_unit * 3;
    lines [ii++] = x_offset;
    lines [ii++] = y_offset + height_unit * 2 | 0x80;

    // Draw the connected lines
    lcd.drawLines (GLCD_MODE_NORMAL, lines);
    ...
}
```

SEE ALSO

GLCD::drawLine(), GLCD::drawMode(), GLCD::drawPolygon(), Draw lines serial command.

5.12 drawMode

NAME

drawMode, **fontMode** - Change the line and font drawing modes.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

#define GLCD_MODE_NORMAL 0x01
#define GLCD_MODE_REVERSE 0x00
#define GLCD_MODE_OR 0x02
#define GLCD_MODE_XOR 0x04
#define GLCD_MODE_NAND 0x06
#define GLCD_MODE_FILL 0x08
#define GLCD_MODE_FONT_PROPORTIONAL 0x20

void
drawMode (uint8_t *mode*);

void
fontMode (uint8_t *mode*);

DESCRIPTION

drawMode sets the default drawing mode used for subsequent drawing commands that do not include an explicit *mode* as a argument. See drawMode for further information on the drawing modes. The default following a reset() is `GLCD_MODE_NORMAL`.

fontMode sets the default drawing mode used for rendering characters. The default following a reset() is `GLCD_MODE_NORMAL`.

The *mode* is a bit mask (see drawMode) with constants defined as follows:

GLCD_MODE_NORMAL is normal drawing which sets a pixel in a copy over mode. When the screen is in reverse mode then the pixel is cleared.

GLCD_MODE_REVERSE is reverse drawing which clears a pixel in a copy over mode. When the screen is in reverse mode then the pixel is set.

GLCD_MODE_OR used in conjunction with *normal* or *reverse*; the setting of the screen pixel is the result of the bitwise OR operation *screenPixel | drawnPixel*.

GLCD_MODE_XOR used in conjunction with *normal* or *reverse*; the setting of the screen pixel is the result of the bitwise exclusive-or (XOR) operation *screenPixel ↑ drawnPixel*.

GLCD_MODE_NAND used in conjunction with *normal* or *reverse*; the setting of the screen pixel is the result of the bitwise not-and (NAND) operation *screenPixel & ∼ drawnPixel*.

GLCD_MODE_FILL used in conjunction with *normal* or *reverse* or the bitwise modes; the shape being drawn is filled with pixels.

GLCD v1.27 2015/07/06 22:46:12 www.jasspa.com/serialGLCD.html

GLCD_MODE_FONT_PROPORTIONAL used with **fontMode** only and renders the fixed font as a proportional font.

EXAMPLE

The following example shows how the **drawMode** and **fontMode** are used:

```
GLCD lcd(serial);
...
// Set up an XOR draw.
lcd.drawMode (GLCD_MODE_XOR|GLCD_MODE_NORMAL);
lcd.fontMode (GLCD_MODE_XOR|GLCD_MODE_NORMAL);

// Draw lines and text.
lcd.drawBox (xdim / 2, 0, xdim - 1, ydim - 1);
lcd.setXY ((xdim / 2) + 2, ydim/2);
lcd.putstr ("Title");

// Restore normal drawing.
lcd.drawMode (GLCD_MODE_NORMAL);
lcd.fontMode (GLCD_MODE_NORMAL);
```

SEE ALSO

GLCD::drawMode(), GLCD::fontFace(), GLCD::fontMode(), GLCD::reset(), GLCD::setFontFace(), Draw mode serial command, Font mode serial command, Font set serial command.

5.13   drawPixel

NAME

drawPixel, **setPixel** - Draw a pixel.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
drawPixel (uint8_t *x*, uint8_t *y*, uint8_t *mode*);

void
drawPixel (uint8_t *x*, uint8_t *y*);

void
setPixel (uint8_t *x*, uint8_t *y*, uint8_t *mode*);

void
setPixel (uint8_t *x*, uint8_t *y*);

DESCRIPTION

drawPixel draws a single pixel on the screen at position (*x*, *y*). *mode* defines the colour of the pixel. Where *mode* is omitted then the drawMode() value is used. See drawMode for further information on the drawing modes.

The **setPixel** command is identical **drawPixel** and is retained for backwards compatibility with the original Sparkfun library.

```
GLCD lcd(serial);
...

int xmax = lcd.xdim;
int ymax = lcd.ydim;
int xx;
int yy;

// Clear the screen
lcd.clearScreen ();

// Fill the screen with pixels.
for (xx = 0; xx < xmax; xx++)
    for (yy = 0; yy < ymax; yy++)
        lcd.setPixel (xx, yy, GLCD_MODE_NORMAL);
```

SEE ALSO

GLCD::drawMode(), Draw pixel serial command.

5.14 drawPolygon

NAME
drawPolygon - Draw a polygon.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

void
drawPolygon (uint8_t *mode*, uint8_t **xylist*);

void
drawPolygon (uint8_t **xylist*);

void
drawPolygon_P (uint8_t *mode*, const uint8_t PROGMEM **xylist*);

void
drawPolygon_P (const uint8_t PROGMEM **xylist*);

DESCRIPTION

Draws a polygon defined by a pointer to *xylist* which is a list of (*x*, *y*) coordinates, where a line is drawn between (*x_n*, *y_n*) and (*x_n*+1, *y_n*+1). The last y-coordinate in the list is OR'ed with `0x80` which indicates the end of the list. A line is drawn between the last coordinate and the first coordinate of the list to form a polygon.

mode defines the colour of the line. Where *mode* is omitted then the drawMode() value is used. See draw-Mode for further information on the drawing modes.

When filling the polygon with *mode* `GLCD_MODE_FILL` then then the coordinates should be defined in a clockwise order. Filling works with convex polygons but is not guaranteed to work correctly with convex polygons.

The **drawPolygon_P** variant of the command accepts a *xylist* that is resident in Flash memory i.e. `PROGMEN`.

Where the lines are open and the last coordinate is not the same as the first coordinate then use drawLines().

EXAMPLE
The following example shows an example of polygon drawing:

```
GLCD lcd(serial);
...
static void
glcdTestDrawPolygonHelper (uint8_t x_offset)
{
    uint8_t y_offset = 9;
    uint8_t width = lcd.xdim / 3;
    uint8_t height = lcd.ydim - 1 - (2 * y_offset);
    uint8_t ii;
    uint8_t polygon [40];
    uint8_t width_unit = (width - 1) / 3;
    uint8_t height_unit = (height - 1) / 3;
    uint8_t indent = width / 6;

    // Draw the polygon
```

GLCD v1.27 2015/07/06 22:46:12 *www.jasspa.com/serialGLCD.html*

```
    ii = 0;
    x_offset++;
    polygon [ii++] = x_offset + indent;
    polygon [ii++] = y_offset + height_unit * 2;
    polygon [ii++] = x_offset + indent;
    polygon [ii++] = y_offset + height_unit * 1;
    polygon [ii++] = x_offset;
    polygon [ii++] = y_offset + height_unit * 1;
    polygon [ii++] = x_offset;
    polygon [ii++] = y_offset;
    ....
    polygon [ii++] = y_offset + height_unit * 3;
    polygon [ii++] = x_offset;
    polygon [ii++] = y_offset + height_unit * 2 | 0x80;

    // Draw the polygon
    lcd.drawPolygon (GLCD_MODE_NORMAL, polygon);
    ...
}
```

SEE ALSO

GLCD::drawLine(), GLCD::drawLines(), GLCD::drawMode(), Draw polygon serial command.

5.15   drawRoundedBox

NAME

drawRoundedBox - Draw a box with rounded corners.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
drawRoundedBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*, uint8_t *radius*, uint8_t *mode*);

void
drawRoundedBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*, uint8_t *radius*);

DESCRIPTION

drawBox draws the outline of a box, or rectangle, with rounded corners of *radius*. The rectangle is defined using a diagonal line describing opposing corners (*x1*, *y1*) and (*x2*, *y2*). *mode* defines the colour of the line and whether the rectangle is filled. Where *mode* is omitted then the drawMode() value is used. See drawMode for further information on the drawing modes.

EXAMPLE

The following example displays draws a rounded rectangle on the screen:

```
GLCD lcd(serial);
...
// Rectangle in left side of screen
lcd.drawBox (0, 0, 64, 61, GLCD_MODE_NORMAL);
// Rounded rectangle around text area
lcd.drawRoundedBox (68, 0, 68+58-1, 61, 5, GLCD_MODE_NORMAL);
...
```

SEE ALSO

GLCD::drawMode(), GLCD::drawBox(), GLCD::eraseBlock(), Draw rounded box serial command.

5.16   drawSprite

NAME

drawSprite - Draw a box with rounded corners.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
drawSprite (uint8_t *x*, uint8_t *y*, uint8_t *sprite_id*, uint8_t *mode*);

DESCRIPTION

drawSprite draws a sprite (bitmap) with identity *id* on the screen at position (*x*, *y*).

The sprite identity *id* identifies the sprite to be draw, the sprite must have previously been uploaded to the screen with loadSprite(). Sprites may exist in RAM or EEPROM; sprites with identities greater than `0x80` are EEPROM sprites. Sprite identity `0x80` is the splash screen sprite which is by default the Sparkfun logo.

Storage is provided for 6 RAM and 14 EEPROM sprites of size 34 bytes, the first 2 bytes of a sprite are *width* and *height* leaving 32 bytes for the sprite pixel data. This allows a 16x16 sprite, noted that sprites are not required to be square. The actual sprite number and size may be queried from the screen using query().

mode identifies how the sprite is rendered to the screen. When *mode* is `GLCD_MODE_NORMAL` then the sprite copies over the screen data. `GLCD_MODE_REVERSE` inverts the sprite and copies over the the screen data. The bitwise operations may be used with sprites as defined by drawMode.

The *mode* `GLCD_MODE_CENTRE` is a sprite specific mode and centres the sprite about (*x*, *y*), when omitted then by default the sprite is drawn with the top left corner of the sprite at (*x*, *y*).

EXAMPLE

The following is an example of sprite drawing:

```
GLCD glcd(serial);
...
// Draw the Sparkfun logo sprite (sprite_id=0x80) as we know it is
// loaded. Position 1/4 fscreen width from the left and in the middle.
glcd.drawSprite (x_pos_3_4, glcd.ydim / 2, 0x80,
                 GLCD_MODE_CENTER|GLCD_MODE_NORMAL);
```

SEE ALSO

Sprite data format, GLCD::bitblt(), GLCD::loadSprite(), GLCD::query(), Draw sprite serial command.

5.17 echo

NAME

echo - Synchronisation; send a character to echo back over serial.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
echo (uint8_t *echo_char*);

void
echoWait (uint8_t *echo_char*, int *msdelay*);

DESCRIPTION

echo may be used to synchronise the drawing with the display and sends a character *echo_char* to the screen. When the command is executed then the *echo_char* is returned on the serial line to the host.

echoWait is similar to **echo** and sends a *echo_char* character and then blocks waiting for the character to be received on the serial port before returning to the caller.

RETURN

echoWait returns with the *echo_char* if it is received or -1 if the character was not received and the call timed out after 2 seconds.

EXAMPLE

The following sequence is taken from the reset() command that uses a **echoWait**() command to synchronise the screen before performing a reset.

```
GLCD lcd(serial);
...
// First make sure that we can communicate with the screen. If a bitblt
// or polygon operation was interrrupted accross out reset then the
// screen will be waiting for more characters so we need to feed it
// before we perform the reset.
do
{
    // See if the screen is responsive. We use a non-drawable and
    // non-command character
    if ((cc = this->echoWait (0xf7, 1)) == -1)
    {
        // Un-responsive, push a dummy pixel. This will feed any bitblt
        // in addtion to terminating any polygon line, otherwise the
        // pixel draw will be clipped off-screen and do nothing.
        this->drawPixel (0xff, 0xff);
    }
}
while (cc != 0xf7);
```

SEE ALSO

GLCD::reset(), Echo character serial command.

5.18   eraseBox

NAME

eraseBox - Erase a rectangular screen block.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
eraseBox (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*);
void
eraseBlock (uint8_t *x1*, uint8_t *y1*, uint8_t *x2*, uint8_t *y2*);

DESCRIPTION

eraseBlock clears a rectangular block of the screen, erasing the content by filing the content with the background colour. The rectangular region is defined using a diagonal line describing opposing corners (*x1*, *y1*) and (*x2*, *y2*).

eraseBox is identical in operation to **eraseBlock** and is provided for naming consistency.

eraseBlock is identical in operation to drawBox() with a *mode* of GLCD_MODE_FILL | GLCD_MODE_REVERSE.

EXAMPLE

The following is an example of **eraseBlock**() which is used in the benchmark demonstration application to draw a rotating spinner:

```
GLCD lcd(serial);
...
void
drawSpinner (uint8_t pos, uint8_t x, uint8_t y)
{
    // this draws an object that appears to spin
    switch (pos % 8)
    {
    case 0:
        lcd.drawLine (x, y-8, x, y+8, GLCD_MODE_NORMAL);
        break;
    case 1:
        lcd.drawLine (x+3, y-7, x-3, y+7, GLCD_MODE_NORMAL);
        break;
    case 2:
        lcd.drawLine( x+6, y-6, x-6, y+6, GLCD_MODE_NORMAL);
        break;
    case 3:
        lcd.drawLine( x+7, y-3, x-7, y+3, GLCD_MODE_NORMAL);
        break;
    case 4:
        lcd.drawLine( x+8, y, x-8, y, GLCD_MODE_NORMAL);
        break;
    case 5:
        lcd.drawLine( x+7, y+3, x-7, y-3, GLCD_MODE_NORMAL);
        break;
    case 6:
```

```
        lcd.drawLine( x+6, y+6, x-6, y-6, GLCD_MODE_NORMAL);
        break;
    case 7:
        lcd.drawLine( x+3, y+7, x-3, y-7, GLCD_MODE_NORMAL);
        break;
    }
}

////////////////////////////////////////////////////////////////////////////////
// Loop method – run over and over again
void
loop()
{
    iter = 0;
    startMillis = millis();

    // loop for one second
    while (millis() - startMillis < 1000)
    {
        // Rectangle in left side of screen
        lcd.drawBox (0, 0, 64, 61, GLCD_MODE_NORMAL);
        // Rounded rectangle around text area
        lcd.drawRoundedBox (68, 0, 68+58-1, 61, 5, GLCD_MODE_NORMAL);
        for (int i = 0; i < 62; i += 4)
        {
            // draw lines from upper left down right side of rectangle
            lcd.drawLine (1, 1, 63, i, GLCD_MODE_NORMAL);
        }
        // draw circle centered in the left side of screen
        lcd.drawCircle (32, 31, 30, GLCD_MODE_NORMAL);
        // clear previous spinner position
        lcd.eraseBox (94-8, 40, 94-8+16, 40+16);
        // draw new spinner position
        drawSpinner(loops++, 94, 48);
        // locate curser for printing text
        lcd.setXY (5*6-3, 5*8+3);
        // print current iteration at the current cursor position
        lcd.printNum (++iter);
    }
    // display number of iterations in one second
    // clear the screen
    lcd.clearScreen();
    // positon cursor
    lcd.setXY(13*6,2*8);
    // print a text string
    lcd.putstr("FPS=");
    // print a number
    lcd.printNum(iter);
}
```

SEE ALSO

GLCD::drawBox(), Erase block serial command.

5.19   factoryReset

NAME

factoryReset - Reset the screen to a factory shipping state, reset EEPROM etc.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
factoryReset ();

DESCRIPTION

factoryReset resets the screen to its initial factory shipped default state. All of the EEPOM settings are erased and restored to their default. The Sparkfun sprite is restored as the splash screen logo by over-writing the sprite identity `0x80`, all other EEPROM sprites are deleted.

EXAMPLE

The following is an example of a **factoryReset**() invocation:

```
GLCD lcd(serial);
...
lcd.factoryReset ();
...
```

SEE ALSO

GLCD::loadSprite(), GLCD::reset(), Factory Reset serial command.

5.20   fontFace

NAME
fontFace, setFontFace - Change the default font.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

#define GLCD_FONT_NORMAL 0 #define GLCD_FONT_TOM_THUMB 1

void
fontFace (uint8_t *font*);

void
setFontFace (uint8_t *font*);

DESCRIPTION
fontFace and **setFontFace** define the font set to be used. **fontFace** sets the current font non-persistently, while the **setFontFace** is persistent through power-on events and is retained in EEPROM. The `<font>` argument defines the current font to use where `0x00` is the default 6x8 font and `0x01` is the Tom Thumb 4x6 font. Both fonts use a 1 pixel space separation between characters; the actual rendered character is 5x8 and 3x6 respectively.

The Tom Thumb font is courtesy of https://www.pjrc.com/teensy/td_libs_GLCD.html.

EXAMPLE
The following is an example of a **fontFace**() invocation:
```
GLCD lcd(serial);
...
lcd.fontFace (1);  // 4x6 Tom Thumb font.
...
lcd.fontFace (0);  // 6x8 default font.
```

SEE ALSO
GLCD::fontMode(), Font set serial command, Font mode serial command.

5.21 GLCD

NAME

GLCD - Class constructor.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

GLCD (SoftwareSerial& *software_serial*);

DESCRIPTION

The GLCD library is initialised with the **GLCD** constructor at start up. The constructor takes a single argument *software_serial* which is an instance of the software serial library.

The first command to be executed after the constructor should be reset().

EXAMPLE

The following is an example of a **GLCD**() invocation from the *Simple Application* demonstration code:

```
#include <AltSerialGraphicLCD.h>
#include <SoftwareSerial.h>

// Define the TX and RX pins used to connect the screen. Change these two pin
// values to whichever pins you wish to use (RX, TX).
//#define SERIAL_TX_DPIN    3
//#define SERIAL_RX_DPIN    2
#define SERIAL_TX_DPIN   12
#define SERIAL_RX_DPIN   10

// Initialize an instance of the SoftwareSerial library
SoftwareSerial serial (SERIAL_RX_DPIN,SERIAL_TX_DPIN);

// Create an instance of the GLCD class named glcd. This instance is used to
// call all the subsequent GLCD functions. The instance is called with a
// reference to the software serial object.
GLCD glcd(serial);

static uint32_t counter = 0;          // Counter for number of iterations.
static uint32_t start_time;           // The time we started running.

////////////////////////////////////////////////////////////////////////////
// Perform significant initialisation.
void setup()
{
    // Start the Software serial library we run at 115200 by default.
    serial.begin(115200);

    // The first call is reset to the sceeen. This puts it into a sane state
    // irrespective of the state that we last left it in. Following a reset
    // then the screen is clear and the cursor is at location 0,0. Reset can
    // be called at any time, not just at the start.
    glcd.reset();

    // Initialise our simple clock so we can keep a time count.
```

```
    start_time = millis();
}

////////////////////////////////////////////////////////////////////////////
// Execution loop
void loop()
{
    ...
}
```

SEE ALSO

SoftwareSerial(3), GLCD::reset().

5.22 loadSprite

NAME

loadSprite - Load a sprite or bitmap into the screen memory.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
loadSprite (uint8_t *sprite_id*, uint8_t **sprite*);

void
loadSprite (uint8_t *sprite_id*, uint8_t *length*, uint8_t **sprite*);

void
loadSprite (uint8_t *sprite_id*, uint8_t *width*, uint8_t *height*, uint8_t **sprite_pixels*);

void
loadSprite_P (uint8_t *sprite_id*, const uint8_t PROGMEM **sprite*);

void
loadSprite_P (uint8_t *sprite_id*, uint8_t *length*, const uint8_t PROGMEM **sprite*);

void
loadSprite_P (uint8_t *sprite_id*, uint8_t *width*, uint8_t *height*, const uint8_t PROGMEM **sprite_pixels*);

DESCRIPTION

loadSprite loads the screen resident sprite memory identified by *id* with bitmap data that may be subsequently drawn with a drawSprite() command.

The sprite identity *id* identifies the sprite to be draw, the sprite must have previously been uploaded to the screen with loadSprite(). Sprites may exist in RAM or EEPROM; sprites with identities greater than `0x80` are EEPROM sprites. Sprite identity `0x80` is the splash screen sprite which is by default the Sparkfun logo.

Storage is provided for 6 RAM and 14 EEPROM sprites of size 34 bytes, the first 2 bytes of a sprite are *width* and *height* leaving 32 bytes for the sprite pixel data. This allows a 16x16 sprite, noted that sprites are not required to be square. The actual sprite number and size may be queried from the screen using query().

sprite is a pointer to the sprite in memory, the sprite should be in the *(w, h, pixels, ...)* format.

The second alternative form of the call takes a *length* in bytes and *sprite* pointer. This call allows a sprite defined in memory to be sent without computing the length of data from the *width* and *height*. The data pointed to by *sprite* os *length* bytes is sent as the sprite data, the sprite should be in the *(w, h, pixels, ...)* format.

The third alternative for of the call takes a *width* and *height* in pixels followed by the *sprite_pixels*, this is a pointer to a sprite excluding the width and height parameters.

The **bitblt_P** invocations are used when the sprite data is stored in `PROGMEM` the call will read the sprite data from flash memory and transfer to the screen.

EXAMPLE

The following example performs a sprite load operation:

```
GLCD lcd(serial);
...
```

```
static const uint8_t char_A [] PROGMEM =
{
    0x0a, 0x10,
    0x00, 0x00, 0xe0, 0xfe, 0x9f, 0x9f, 0xfe, 0xe0, 0x00, 0x00, /*A*/
    0x00, 0x0c, 0x0f, 0x07, 0x01, 0x01, 0x07, 0x0f, 0x0c, 0x00
};
...

// Load the sprite into RAM location 2.
lcd.loadSprite_P (2, sprite);

// Draw the sprite in the centre of the screen
lcd.drawSprite (xdim/2, ydim/2, 2,
                GLCD_MODE_CENTER|GLCD_MODE_NORMAL);
...
```

NOTES

The Jennifer Holt version required that the sprite data was padded out to match the size of the sprite memory location. This implementation has no such requirement to pad un-used bytes. It is sufficient to upload the exact size of the sprite defined by the *width* and height dimensions.

SEE ALSO

GLCD::bitblt(), GLCD::drawSprite(), GLCD::query(), Sprite upload serial command.

5.23 put

NAME

put - Write a single character to the screen checking for XON/XOFF.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
put (uint8_t *c*);

DESCRIPTION

put writes a single unsigned 8-bit byte to the screen. The command performs a XON/XOFF check before the byte is sent to the screen.

The **put** command incurs a processing overhead in performing the XON/XOFF check and requires the serial input to be read. With some knowledge of the XON and XOFF positions then the serial overhead may be minimised by sending the first byte using the **put**() command to check for over-flow, subsequent bytes that are known to fit into the serial buffer of the display without over-running may be written using the **serial.write**() command. The normal command send operates using this method as it is not strictly necessary to check for XON/XOFF on every byte sent.

EXAMPLE

The following example performs a **put**() operation:

```
GLCD lcd(serial);
...
// Kick off a dummy bitblt operation and see if we can recover.
lcd.put (GLCD_CHAR_CMD);
lcd.put (GLCD_CMD_BITBLT);
lcd.put (0);
lcd.put (0);
lcd.put (1);
lcd.put (128);
lcd.put (64);

// Perform a reset operation with an open bitblt() operation.
lcd.reset ();
...
```

SEE ALSO

GLCD::putcmd(), GLCD::putstr(), GLCD::write(), **serial**(3).

5.24   putcmd

NAME

putcmd - Write a command to the screen checking for XON/XOFF.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
putcmd (uint8_t *cmd*, uint8_t *argc*, . . .);

DESCRIPTION

putcmd is a low level command that sends GLCD serial commands to the display. This is effectively a *universal command* that is used to implement most of the commands of the class defined in the header file *AltSerialGraphicLCD.h*. The command has variable arguments which are defined as follows:

cmd the serial command to execute.

argc is partially a bit-mask and counter of the number of arguments that follow this argument. The lower nibble (4-bits) is interpreted as a count of the arguments, the upper nibble is a bit-mask that defines arguments that follow the *argc* arguments. The bit-mask values are defined as follows:

GLCD_ARG_SIZEOF The variable argument list is followed by an integer *length* parameter and unsigned 8-bit *data* pointer to bytes to be uploaded to the screen. The command line is effectively:

    void
    putcmd (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_SIZEOF, . . ., int *length*, uint8_t *data);

GLCD_ARG_XY_LIST The variable argument list is followed by a unsigned 8-bit integer pointer to a pair of *x* and *y* coordinate list *xylist*, where the last *x,y* pair y-coordinate is OR'ed with 0x80. This is the same format *xylist* as used in drawPolygon(). The command line is effectively:

    void
    putcmd (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_XY_LIST, . . ., uint8_t *xylist);

GLCD_ARG_SPRITE_WH The variable argument list is followed by sprite data, the next two arguments are the unsigned 8-bit integer sprite dimensions *width* and *height* followed by the unsigned 8-bit integer sprite pixel data. *pixel_data*. This is the same format as used in bitblt(). The command line is effectively:

    void
    putcmd (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_SPRITE_WH, . . .,
    uint8_t *width*, uint8_t *height*, uint8_t *pixel_data);

GLCD_ARG_SPRITE The variable argument list is followed by sprite data, the next argument is the unsigned 8-bit integer sprite data *sprite*. This is the same format as used in bitblt(). The command line is effectively:

    void
    putcmd (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_SPRITE_WH, . . ., uint8_t *sprite);

GLCD v1.27 2015/07/06 22:46:12 www.jasspa.com/serialGLCD.html

GLCD_ARG_FFSTRING The variable argument list is followed by a nil terminated string which is passed through to the serial and the nil (0x00 is replaced with 0xff. This is used in the setString() command. The command line is effectively:

> void
> **putcmd** (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_FFSTRING, …, uint8_t *string);

GLCD_ARG_PROGMEM This is a modifier for the aforementioned byte data commands and modifies the unsigned 8-bit integer data pointer to a Flash memory address of type *const unsigned char **. For example, this modifies the GLCD_ARG_SPRITE argument list type to:

> void
> **putcmd** (uint8_t *cmd*, uint8_t *argc*|GLCD_ARG_SPRITE_WH, …, const uint8_t PROGMEM *sprite);

NOTES

putcmd() performs an XON/XOFF check by sending the first character with a put(), the remaining arguments are sent with a **serial.write**() command. Any bitmask pointer data is then sent using the write() method or periodic XON/XOFF checking is performed using put().

EXAMPLE

As an example then commands from *AltSerialGraphicLCD.h* are highlighted:

```
GLCD lcd(serial);
...
void
drawLine(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, uint8_t mode)
{
    this->putcmd (GLCD_CMD_DRAW_LINE, 5, x1, y1, x2, y2, mode);
};

void
bitblt (uint8_t x, uint8_t y, uint8_t mode, uint8_t *sprite)
{
    this->putcmd (GLCD_CMD_BITBLT, GLCD_ARG_SPRITE|3,
                  x, y, mode, sprite);
}

void
bitblt_P (uint8_t x, uint8_t y, uint8_t mode, int length,
          const uint8_t PROGMEM *sprite)
{
    this->putcmd (GLCD_CMD_BITBLT,
                  GLCD_ARG_PROGMEM|GLCD_ARG_SIZEOF|3,
                  x, y, mode, length, sprite);
}

void
drawPolygon (uint8_t mode, uint8_t *xylist)
{
    this->putcmd (GLCD_CMD_DRAW_POLYGON,
                  GLCD_ARG_XY_LIST|1, mode, xylist);
}

void
 setString (uint8_t posX, uint8_t posY, uint8_t justification, char *s)
{
    this->putcmd (GLCD_CMDX_SET_XY_STRING, GLCD_ARG_FFSTRING|3,
                  posX, posY, justification, s);
```

```
};
```

SEE ALSO

GLCD::bitblt(), GLCD::putcmd(), GLCD::putstr(), setString(), GLCD::write(), **serial**(3).

5.25   putstr

NAME

putstr, **printStr**, **printNum**, **nextLine** - Print a nil terminated string to the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
putstr (char *s*);
void
putstr (const __FlashStringHelper *s*);
void
putstr_P (const char PROGMEM *s*);
void
printStr (char *s*);
void
printNum (int *num*);
void
nextLine ();

DESCRIPTION

putstr sends a nil ('\0') terminated character string *s* to the display. The string may be a character pointer or Flash string (F()).

putstr_P is a variant which handles a string defined in PROGMEM.

printStr is identical to **putstr** and retains backwards compatibility with the original Sparkfun implementation.

printNum converts an integer to a string and sends it to the display.

nextLine sends a new line sequence to the screen, advancing the cursor position to the start of the next line. The character sequence '\r\n' is sent which operates consistently irrespective of the screen CRLF mode.

EXAMPLE

An example of string printing is shown below:

```
GLCD glcd(serial);
...
// "Hello" is 6 * 5 = 30 pixels long, place at 3/4 of screen at the top.
glcd.setXY(x_pos_3_4 - 15, 0);
glcd.printStr("Hello");                 // Print "Hello"

// "World" is 6 * 5 = 30 pixels long and 8 pixels high place at 3/4 of
// screen at the bottom.
glcd.setXY(x_pos_3_4 - 15, glcd.ydim - 8);
glcd.printStr("World");                 // Print "World"
```

SEE ALSO
put(), putcmd(), putstr(), write(), **serial**(3).

5.26 query/set

NAME

query, **set** - Query or set the screen information.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

```
#define GLCD_ID_MAGIC            0x00 /* Magic number to handle new install */
#define GLCD_ID_BAUDRATE         0x01 /* Baud rate */
#define GLCD_ID_BACKLIGHT        0x02 /* Backlight level */
#define GLCD_ID_SPLASH           0x03 /* Splash screen enabled */
#define GLCD_ID_REVERSE          0x04 /* Reverse the screen */
#define GLCD_ID_DEBUG            0x05 /* Reserved for future use */
#define GLCD_ID_CRLF             0x06 /* Line ending CR+LF */
#define GLCD_ID_XON_POS          0x07 /* XON position */
#define GLCD_ID_XOFF_POS         0x08 /* XOFF position */
#define GLCD_ID_SCROLL           0x09 /* Scroll on/off */
#define GLCD_ID_LARGE_SCREEN     0x0a /* Large screen. */

#define GLCD_ID_VERSION_MAJOR    0x20 /* Version number major */
#define GLCD_ID_VERSION_MINOR    0x21 /* Version number minor */
#define GLCD_ID_EEPROM_SPRITE_SIZE 0x22 /* EEPROM sprite size in bytes (Read only) */
#define GLCD_ID_EEPROM_SPRITE_NUM  0x23 /* Number of EEPROM sprites (Read only) */
#define GLCD_ID_RAM_SPRITE_SIZE   0x24 /* RAM sprite size in bytes (Read only) */
#define GLCD_ID_RAM_SPRITE_NUM    0x25 /* Number of RAM sprites (Read only) */

#define GLCD_ID_X_DIMENSION      0x40 /* Screen X dimension (Read only) */
#define GLCD_ID_Y_DIMENSION      0x41 /* Screen Y dimension (Read only) */

#define GLCD_ID_ESPRITE_WIDTH_0  0x80 /* EEPROM sprite[0] width (Read only) */
#define GLCD_ID_ESPRITE_HEIGHT_0 0x81 /* EEPROM sprite[0] height (Read only) */
```

int
query (uint8_t *id*);

void
set (uint8_t *id*, uint8_t *value*);

void
setCRLF (uint8_t *state*);

void
setScroll (uint8_t *state*);

void
setXoff (uint8_t *position*);

void
setXon (uint8_t *position*);

DESCRIPTION

The **query** command fetches information from the screen, the single parameter *id* identifies the information that is requested. The values of *id* are defined in Table 5.

The query command suspends waiting for a response from the screen and returns the integer as a return value. A value of −1 indicates an error which received from the screen or as a result of the command timing out after waiting a maximum of 2 seconds.

Any result from the screen is sent as an unsigned 8-bit integer and the result may be safely masked with 0xff.

set modifies the EEPROM setting for the index *id* assigning it the value *value*. The **set** command does not perform any validation of the new *value* and should be used with caution. The reverse mode should be modified through the reverseMode in order to enact the screen reversal.

setCRLF() changes the default line ending behaviour and changes whether a LF/0x0a/\n advances to the beginning of the next line or just advanced to the next line.

The default setting of *state* is CRLF=0 which is compatible with string endings \n or \r\n. The current setting may be queried with query().

setScroll() changes the scrolling behaviour of the screen. When *state* is 0 then scrolling is disabled; when the cursor position reaches the end of the screen then the cursor position is moved back to the top of the screen. When *state* is 1 then when the cursor position reaches the end of the screen then the display is scrolled up by 1 line and a new clear line is inserted.

setXoff() and **setXon**() modify the XON and XOFF reporting positions. See Serial Overview for a more in-depth discussion of serial communication.

EXAMPLE

The following example checks the current CRLF mode. This is taken from the test code.

```
GLCD lcd(serial);
...
// Change the CRLF mode to 0
lcd.set (GLCD_ID_CRLF, 0);
lcd.printStr (F("Query␣CRLF␣"));
qq = lcd.query (GLCD_ID_CRLF);
if (qq != 0)
{
    lcd.printStr (F("FAILED("));
    lcd.printNum (qq);
    lcd.printStr (F(")"));
    lcd.nextLine ();
    countdown();
}
lcd.nextLine ();
```

SEE ALSO

Serial Overview,
GLCD::printStr(), GLCD::query(), Factory reset, Query LCD.

5.27 ready

NAME

ready - Handle XON/XOFF and wait for the screen to be ready to receive data.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
ready ();

DESCRIPTION

ready() handles the software serial follow control. When the method is invoked then the software flow control state of the display is determined and the call blocks until the display is ready to receive some more data.

The **ready** method should be invoked before sending any data to the screen and then checked periodically to ensure that the display serial buffer does not over-flow. Refer to Serial Overview for a more in-depth discussion of serial communication.

The **ready**() method is used by all of the other methods that send data to the display to manage XON/XOFF handling. The method has deliberately not been made private allowing other sending mechanisms to be implemented.

NOTES

The **ready** implementation drains the serial input to process any XON/XOFF characters, any other characters received are discarded. If an XOFF character is received from the display requesting the Host to stop sending then the serial input is polled for a XON character. If no characters are received within 20 milliseconds then the command exits as if a XON had been received.

The timeout is used to ensure that the Host is not blocked indefinitely in the event that the XON character is lost. 20msec is sufficient time for the display to process the pending commands, if the display still has a backlog of commands pending then it sends an XOFF on any next character received which requests the Host to stop sending again.

EXAMPLE

The following example is from the implementation of the GLCD library to send a character.

```
GLCD glcd(serial);
...
//----------------------------------------------------------------------
// Put a character to the screen. Check that we are not blocked.
void
GLCD::put (uint8_t cc)
{
    // Wait for the screen to be ready.
    this->ready();
    // Send the character, we are not blocked.
    serial.write (cc);
}
```

SEE ALSO

GLCD::demo(), GLCD::factoryReset(), GLCD::loadSprite(), GLCD::set(), Toggle splash serial command, Set LCD serial command, Serial Overview.

5.28   reset

NAME

reset - Reset the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
reset ();

DESCRIPTION

The **reset**() command resets the screen and must be invoked as the first command that is executed, typically in **setup** (). The **reset** performs a hard reset of the screen and any sprites that exist in RAM are deleted and any transient states such as reverse screen are restored to their start-up EEPROM settings.

NOTES

It is not possible to send a serial command to the screen to be enacted immediately, commands such as bitblt and drawPolygon which take a variable number of arguments are read directly from the serial input of the display. If reset is invoked while these commands are actively waiting for input then the data requirements of the pending commands have to be satisfied before the screen enacts a reset command.

The implementation of **reset** handles these open command cases and ensures that the screen is synchronised and ready to enact the next command before sending the Reset LCD serial command.

The screen uses a watchdog timer to enact a reset command, the Arduino watchdog timer is set and allowed to timeout. This causes the screen to re-initialise as if a power-on event had occurred.

EXAMPLE

The **reset**() method is typically invoked from **setup**() as follows:

```
GLCD glcd(serial);
...
////////////////////////////////////////////////////////////////////////////
// Perform significant initialisation.
void setup()
{
    // Start the Software serial library we run at 115200 by default.
    serial.begin(115200);

    // The first call is reset to the sceeen. This puts it into a sane state
    // irrespective of the state that we last left it in. Following a reset
    // then the screen is clear and the cursor is at location 0,0. Reset can
    // be called at any time, not just at the start.
    glcd.reset();
}
...
```

SEE ALSO

GLCD::factoryReset, **setup**(3), Reset LCD serial command.

5.29   reverseMode

NAME

reverseMode, **toggleReverseMode** - Reverse the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
reverseMode (uint8_t *mode*);

void
toggleReverseMode ();

DESCRIPTION

reverseMode sets the screen mode colour defined by *mode*. When *mode* is `GLCD_MODE_NORMAL` then the screen draws with a pixel set (lit up). When *mode* is `GLCD_MODE_REVERSE` then the background is lit and pixels are cleared when drawn. **reverseMode** is temporarily applied and the setting is not stored to EEPROM.

toggleReverseMode toggles reverse (white on black) mode and stores the setting to EEPROM. The *mode* is restored on power-up.

Reverse mode inverts the screen in place and does not clear the screen or change the text drawing position.

The current setting of reverse mode enacted by the screen may be read using the query() command. The **query**() command reads the current state of the reverse mode, following a reset this value reflects the state stored in EEPROM.

EXAMPLE

The following example performs a reverse operation:

```
GLCD lcd(serial);
...
// Change the screen to reverse mode.
lcd.reverseMode (GLCD_MODE_REVERSE);
...
// Change the screen to normal mode.
lcd.reverseMode (GLCD_MODE_NORMAL);
```

SEE ALSO

GLCD::query(), Reverse mode serial command.

5.30   setBacklight

NAME

setBacklight, **updateBacklight** - Change the backlight brightness.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
setBacklight (uint8_t *percentage_level*);

void
updateBacklight (uint8_t *percentage_level*);

DESCRIPTION

setBacklight sets the brightness of the LCD backlight to a percentage value between 0 and 100 defined by *percentage_level*. A value of 0 turns the back light off, a value of 100 sets full brightness. The brightness level is saved to EEPROM and restored on the next power-on.

updateBacklight sets the LCD brightness with the exception that the brightness level is not saved in EEP-ROM. This allows the screen brightness to be temporarily altered e.g. screen dimming whilst idle.

The backlight level is set to a default value of 100 following a Factory reset. The current value of the backlight may be obtained from the screen using the query() command.

EXAMPLE

The following example if from the Sparkfun demo that changes the screen brightness:

```
GLCD LCD(serial);
...
void
backlightDemo()
{
    // This function shows the different brightnesses to which the backlight
    // can be set using the setBacklight() function. You can choose any
    // number from the 0-100 range. If you are having trouble seeing text at
    // different brightnesses, try adjusting the trimpot on the backpack.
    //
    // In this code we use updateBacklight() rather than setBacklight()
    // because this setting is not persistent and does not update the screen.
    LCD.clearScreen();
    LCD.printStr("Change the backlight brightness");
    delay(2000);
    LCD.clearScreen();

    // 0-100 are the levels from off to full brightness
    for(int i = 0; i <= 100; i+=10)
    {
        LCD.updateBacklight(i);
        delay(100);
        LCD.printStr("Backlight = ");
        LCD.printNum(i);
        delay(500);
        LCD.clearScreen();
```

```
    }
}
```

SEE ALSO

GLCD::Factory reset, GLCD::query(), Backlight level serial command.

5.31   setBaud

NAME

setBaud, **restoreDefaultBaud** - Change the baud rate of the screen and serial line or restore the default rate.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
setBaud (uint8_t *baud*);
void
restoreDefaultBaud ();

DESCRIPTION

setBaud changes the baud rate of screen serial communication and local host communication to the new rate *baud*. *baud* is defined as follows:

Character '1' or `0x31` or `49` or `0x01` = 4800bps
Character '2' or `0x32` or `50` or `0x02` = 9600bps
Character '3' or `0x33` or `51` or `0x03` = 19,200bps
Character '4' or `0x34` or `52` or `0x04` = 38,400bps
Character '5' or `0x35` or `53` or `0x05` = 57,600bps
Character '6' or `0x36` or `54` or `0x06` = 115,200bps

restoreDefaultBaud may be invoked when the host and screen baud rate are out of synchronisation and resynchronises with the screen and sets the default baud rate to 115200.

The *baud* is stored in EEPROM and is the rate used on the next power-on. The default baud rate is 115200 and is restored when a factoryReset() is performed.

EXAMPLE

The following code fragment is from the demo application which modifies the baud rate.

```
GLCD LCD(serial);
...
void
baudDemo()
{
    // This function uses the setBaud() function to change the baud rate of
    // the backpack. The deafult rate is 115200bps. If you loose track of
    // what baud rate your LCD is set to, you can use the
    // restoreDefaultBaud() function to restore it back to 115200.
    LCD.clearScreen();
    LCD.printStr("This changes the Baud rate");
    delay(2000);

    LCD.clearScreen();
    LCD.printStr("115200 is the Default rate");
    delay(1500);
    LCD.clearScreen();
    LCD.printStr("Hello @ 115200");
```

```
delay(1000);
LCD.setBaud(53);//set to 57600

LCD.clearScreen();
LCD.printStr("Hello␣@␣57600");
delay(1000);
LCD.setBaud(52);//set to 38400

LCD.clearScreen();
LCD.printStr("Hello␣@␣38400");
delay(1000);
LCD.setBaud(51);//set to 19200
....
```

SEE ALSO

GLCD::factoryReset(), Baud rate serial command.

5.32 setGraphics

NAME
setGraphics - Sets the screen into graphics only mode allowing the command string to be shortened.

CLASS
GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS
#include <AltSerialGraphicLCD.h>

void
setGraphics (uint8_t *mode*);

DESCRIPTION
setGraphics enters or leaves graphics mode defined by the argument *mode*. A *mode* value of 0x01 enables graphics mode, a value of 0x00 disables graphics mode.

When graphics mode is enabled then the `0x7c` command character is dropped and only drawing commands should be sent to the display with no characters. The GLCD class automatically disables graphics mode when a character is sent with printStr().

When the screen receives a graphics mode command then the screen is placed into a graphics mode so that each command does not need to be prefaced by the `0x7c` character. The screen automatically exits graphics mode if a `0x7c` command is received.

Graphics Mode is especially useful for batches of drawing commands, enabling graphics mode reduces the serial communication by 1 byte per command, more importantly it reduces the memory overhead to store the commands. Blocks of pre-formatted draw commands may be sent to the display with a write() command.

Graphics Mode should be used with care and all commands sent to the display must be well formatted as the amount of checking that is performed on the command is limited. A malformed command may not be caught correctly and may result in some miss-rendering.

EXAMPLE
The following example shows the use of graphics mode with pre-formatted draw commands:

```
GLCD lcd(serial);
...
static const uint8_t complex [] PROGMEM =
{
    0x7c, 0x40, /* Graphics mode on */
    0x16, 0x53, 0x09, 0x01, 0x28, 0x04,  /* Bitblt: [83,9] 40x4 */
    0x00, 0x00, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x00, /* Bitblt: Row 0 */
    0x16, 0x53, 0x17, 0x01, 0x28, 0x04,  /* Bitblt: [83,23] 40x4 */
    0x01, 0x01, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x02, 0x02, /* Bitblt: Row 0 */
    0x16, 0x53, 0x1d, 0x01, 0x28, 0x04,  /* Bitblt: [83,29] 40x4 */
    0x00, 0x00, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
```

```
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x00, /* Bitblt: Row 0 */
    0x16, 0x53, 0x2b, 0x01, 0x28, 0x04,   /* Bitblt: [83,43] 40x4 */
    0x02, 0x02, 0x02, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x02, 0x02, /* Bitblt: Row 0 */
    0x06, 0x00, 0x18, 0x01, 0x3b, 0xff, /* Filled Box (0,24) -> (1, 59) = ff [2x36 @ 72] */
    0x06, 0x04, 0x14, 0x1f, 0x15, 0xff, /* Filled Box (4,20) -> (31, 21) = ff [28x2 @ 56] */
    0x06, 0x04, 0x3e, 0x1f, 0x3f, 0xff, /* Filled Box (4,62) -> (31, 63) = ff [28x2 @ 56] */
    0x06, 0x22, 0x2f, 0x23, 0x3b, 0xff, /* Filled Box (34,47) -> (35, 59) = ff [2x13 @ 26] */
    0x06, 0x22, 0x23, 0x23, 0x2c, 0xff, /* Filled Box (34,35) -> (35, 44) = ff [2x10 @ 20] */
    0x06, 0x22, 0x18, 0x23, 0x20, 0xff, /* Filled Box (34,24) -> (35, 32) = ff [2x9 @ 18] */
    0x06, 0x0f, 0x1d, 0x10, 0x1e, 0xff, /* Filled Box (15,29) -> (16, 30) = ff [2x2 @ 4] */
    0x06, 0x0f, 0x2b, 0x10, 0x2c, 0xff, /* Filled Box (15,43) -> (16, 44) = ff [2x2 @ 4] */
    0x06, 0x0f, 0x39, 0x10, 0x3a, 0xff, /* Filled Box (15,57) -> (16, 58) = ff [2x2 @ 4] */
    0x06, 0x12, 0x05, 0x13, 0x06, 0xff, /* Filled Box (18,5) -> (19, 6) = ff [2x2 @ 4] */
    0x06, 0x12, 0x0e, 0x13, 0x0f, 0xff, /* Filled Box (18,14) -> (19, 15) = ff [2x2 @ 4] */
    0x06, 0x2f, 0x0c, 0x30, 0x0d, 0xff, /* Filled Box (47,12) -> (48, 13) = ff [2x2 @ 4] */
    0x06, 0x33, 0x0c, 0x34, 0x0d, 0xff, /* Filled Box (51,12) -> (52, 13) = ff [2x2 @ 4] */
    0x06, 0x34, 0x39, 0x35, 0x3a, 0xff, /* Filled Box (52,57) -> (53, 58) = ff [2x2 @ 4] */
    0x06, 0x34, 0x3d, 0x35, 0x3e, 0xff, /* Filled Box (52,61) -> (53, 62) = ff [2x2 @ 4] */
    0x06, 0x37, 0x0c, 0x38, 0x0d, 0xff, /* Filled Box (55,12) -> (56, 13) = ff [2x2 @ 4] */
    0x06, 0x3b, 0x0c, 0x3c, 0x0d, 0xff, /* Filled Box (59,12) -> (60, 13) = ff [2x2 @ 4] */
    0x06, 0x3f, 0x0c, 0x40, 0x0d, 0xff, /* Filled Box (63,12) -> (64, 13) = ff [2x2 @ 4] */
    0x06, 0x64, 0x13, 0x65, 0x14, 0xff, /* Filled Box (100,19) -> (101, 20) = ff [2x2 @ 4] */
    0x06, 0x64, 0x27, 0x65, 0x28, 0xff, /* Filled Box (100,39) -> (101, 40) = ff [2x2 @ 4] */
    0x06, 0x6e, 0x05, 0x6f, 0x06, 0xff, /* Filled Box (110,5) -> (111, 6) = ff [2x2 @ 4] */
    0x4f, 0x1b, 0x00, 0x1d, 0x02, /* Auto Rectangle: (27,0)->(29,2) 3x3 @ 1 */
    0x4f, 0x77, 0x00, 0x79, 0x02, /* Auto Rectangle: (119,0)->(121,2) 3x3 @ 1 */
    0x4f, 0x1b, 0x09, 0x1d, 0x0b, /* Auto Rectangle: (27,9)->(29,11) 3x3 @ 1 */
    ...
    0x50, 0x41, 0x1e, /* Pixel [65,30] =  1 */
    0x50, 0x41, 0x28, /* Pixel [65,40] =  1 */
    0x50, 0x52, 0x17, /* Pixel [82,23] =  1 */
    0x50, 0x52, 0x2c, /* Pixel [82,44] =  1 */
    0x41  /* Graphics mode off */
};
...
// Clear the screen
lcd.clearScreen ();
// Set the drawMode for the commands
lcd.drawMode (GLCD_MODE_NORMAL);
// Draw a pre-cooked list of commands.
lcd.write_P (complex, sizeof (complex));
```

SEE ALSO

GLCD::printStr(), GLCD::write(), Graphics mode serial command.

5.33   setXY

NAME

setHome, **setString**, **setX**, **setY**, **setXY** - Change the cursor position for drawing text.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

#define GLCD_FONT_CENTER 0
#define GLCD_FONT_RIGHT 1

void
setHome ();

void
setString (uint8_t *posX*, uint8_t *posY*, uint8_t *justification*, char * *s*);

void
setString (uint8_t *posX*, uint8_t *posY*, uint8_t *justification*, const __FlashStringHelper **s*);

void
setString_P (uint8_t *posX*, uint8_t *posY*, uint8_t *justification*, const char PROGMEM **s*);

void
setX (uint8_t *posX*);

void
setY (uint8_t *posY*);

void
setXY (uint8_t *posX*, uint8_t *posY*);

DESCRIPTION

These commands set the screen position for character rendering where the *posX*, *posY* define the x-coordinte and y-coordinate top left position.

setX() changes the x-coordinate independently of the y-coordinate.

setY() changes the y-coordinate independently of the x-coordinate.

setXY() changes both the x-coordinate and y-coordinate at the same time.

setHome moves the cursor to (0,0) and is a shorthand for **setXY**(0,0).

setString is used for positioning and rendering text labels when using proportional fonts. The *justification* parameter specifies the justification of the string a value of `0x00` is centre justification and `0x01` is right justification. The *x* position is adjusted relative to the length of the text *string* in order to render centre or right justification. This is required with proportional fonts as the rendered length of the string is unknown by the caller.

EXAMPLE

The following example changes the cursor position.

```
GLCD glcd(serial);
...
// "Hello" is 6 * 5 = 30 pixels long, place at 3/4 of screen at the top.
```

```
glcd.setXY(x_pos_3_4 - 15, 0);
glcd.printStr("Hello");                    // Print "Hello"

// "World" is 6 * 5 = 30 pixels long and 8 pixels high place at 3/4 of
// screen at the bottom.
glcd.setXY(x_pos_3_4 - 15, glcd.ydim - 8);
glcd.printStr("World");                    // Print "World"
```

The following example uses **setString** to position strings with a proportional font in the centre of the screen. Refer to Figure 3 which shows the resultant rendered text.

```
GLCD lcd(serial);
...
uint8_t xdim = lcd.xdim;   // Screen x dimension
uint8_t ydim = lcd.ydim;   // Screen y dimension
...
// Change to normal proportional font
lcd.fontFace (GLCD_FONT_NORMAL);
lcd.fontMode (GLCD_MODE_NORMAL|GLCD_MODE_FONT_PROPORTIONAL);

// Draw the text in the cente of the screen
lcd.setString (xdim/2, y+0, GLCD_FONT_CENTER, F("Alternative␣Sparkfun"));
lcd.setString (xdim/2, y+8, GLCD_FONT_CENTER, F("Serial␣Graphic␣LCD"));
lcd.setString (xdim/2, y+16, GLCD_FONT_CENTER, F("Backpack␣Firmware"));
```

SEE ALSO

putstr(), Set position serial commands.

5.34 toggleSplash

NAME

toggleSplash - Change the setting of the start up splash screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
toggleSplash ();

DESCRIPTION

toggleSplash() changes the splash start up screen behaviour and cycles through three different modes of operation.

`0` - splash screen is off.

`1` - splash screen shows the splash screen logo and screen information.

`2` - splash screen shows the logo only.

The default is `1`, to show the logo and screen information, this may be invoked with the demo() command.

The logo that is used for the splash screen is EEPROM sprite identity `0x80`. The splash screen logo is the Sparkfun logo which may be modified using loadSprite(); if modified, the Sparkfun logo may be restored using factoryReset().

SEE ALSO

GLCD::demo(), GLCD::factoryReset(), GLCD::loadSprite(), GLCD::query(), GLCD::set(), GLCD::setCRLF(), GLCD::setScroll(), GLCD::setXon(), GLCD::setXoff(), Toggle splash serial command.

5.35 waitc

NAME

waitc - Wait for the specified character sent from the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

int
waitc (uint8_t *expected*, int *msdelay*);

DESCRIPTION

waitc waits for a character *expected* for the specified number of milliseconds *msdelay*. The call returns when the character expected is received on the serial from the screen.

expected is the character that is expected or `0` for any character i.e. the next character received.

msdely is the number of milliseonds to wait for the character.

RETURN

waitc returns the character read from the serial or `-1` when a timeout occurs and there are no characters.

EXAMPLE

The following example shows how the query() command operates using **waitc**(). The **query**() issues a query command to the display and then waits for a 'Q' character followed by the argument.

```
GLCD glcd(serial);
...
int
query (uint8_t id)
{
    int cc;
    int dd;

    // Put the query command.
    glcd.putcmd (GLCD_CMD_QUERY, 1, id);

    // Wait for a 'Q' response to the query and return the next character
    // read.
    if ((cc = glcd.waitc ('Q', 2000)) == 'Q')
        cc = glcd.waitc (0, 500);
    return cc;
}
```

SEE ALSO

GLCD::query().

5.36 write

NAME

write - Write a block of commands or data to the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

void
write (uint8_t *data*, int *length*);

void
write_P (const uint8_t PROGMEM *data*, int *length*);

DESCRIPTION

The **write** command writes a block of data to the screen, given a unsigned 8-bit integer *data* pointer and the *length* specified in bytes. The command manages XON/XOFF.

write_P is a variant which handles a string defined in PROGMEM.

EXAMPLE

The following example shows the use of graphics mode with pre-formatted draw commands which are written to the screen:

```
GLCD lcd(serial);
...
static const uint8_t complex [] PROGMEM =
{
    0x7c, 0x40, /* Graphics mode on */
    0x16, 0x53, 0x09, 0x01, 0x28, 0x04,  /* Bitblt: [83,9] 40x4 */
    0x00, 0x00, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x00, /* Bitblt: Row 0 */
    0x16, 0x53, 0x17, 0x01, 0x28, 0x04,  /* Bitblt: [83,23] 40x4 */
    0x01, 0x01, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x02, 0x02, /* Bitblt: Row 0 */
    0x16, 0x53, 0x1d, 0x01, 0x28, 0x04,  /* Bitblt: [83,29] 40x4 */
    0x00, 0x00, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x02,
    0x01, 0x02, 0x0c, 0x02, 0x01, 0x02, 0x0c, 0x00, /* Bitblt: Row 0 */
    0x16, 0x53, 0x2b, 0x01, 0x28, 0x04,  /* Bitblt: [83,43] 40x4 */
    0x02, 0x02, 0x02, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x03, 0x04,
    0x08, 0x04, 0x03, 0x04, 0x08, 0x04, 0x02, 0x02, /* Bitblt: Row 0 */
    0x06, 0x00, 0x18, 0x01, 0x3b, 0xff, /* Filled Box (0,24) -> (1, 59) = ff [2x36 @ 72] */
```

```
    0x06, 0x04, 0x14, 0x1f, 0x15, 0xff, /* Filled Box (4,20) -> (31, 21) = ff [28x2 @ 56] */
    0x06, 0x04, 0x3e, 0x1f, 0x3f, 0xff, /* Filled Box (4,62) -> (31, 63) = ff [28x2 @ 56] */
    0x06, 0x22, 0x2f, 0x23, 0x3b, 0xff, /* Filled Box (34,47) -> (35, 59) = ff [2x13 @ 26] */
    0x06, 0x22, 0x23, 0x23, 0x2c, 0xff, /* Filled Box (34,35) -> (35, 44) = ff [2x10 @ 20] */
    0x06, 0x22, 0x18, 0x23, 0x20, 0xff, /* Filled Box (34,24) -> (35, 32) = ff [2x9 @ 18] */
    0x06, 0x0f, 0x1d, 0x10, 0x1e, 0xff, /* Filled Box (15,29) -> (16, 30) = ff [2x2 @ 4] */
    0x06, 0x0f, 0x2b, 0x10, 0x2c, 0xff, /* Filled Box (15,43) -> (16, 44) = ff [2x2 @ 4] */
    0x06, 0x0f, 0x39, 0x10, 0x3a, 0xff, /* Filled Box (15,57) -> (16, 58) = ff [2x2 @ 4] */
    0x06, 0x12, 0x05, 0x13, 0x06, 0xff, /* Filled Box (18,5) -> (19, 6) = ff [2x2 @ 4] */
    0x06, 0x12, 0x0e, 0x13, 0x0f, 0xff, /* Filled Box (18,14) -> (19, 15) = ff [2x2 @ 4] */
    0x06, 0x2f, 0x0c, 0x30, 0x0d, 0xff, /* Filled Box (47,12) -> (48, 13) = ff [2x2 @ 4] */
    0x06, 0x33, 0x0c, 0x34, 0x0d, 0xff, /* Filled Box (51,12) -> (52, 13) = ff [2x2 @ 4] */
    0x06, 0x34, 0x39, 0x35, 0x3a, 0xff, /* Filled Box (52,57) -> (53, 58) = ff [2x2 @ 4] */
    0x06, 0x34, 0x3d, 0x35, 0x3e, 0xff, /* Filled Box (52,61) -> (53, 62) = ff [2x2 @ 4] */
    0x06, 0x37, 0x0c, 0x38, 0x0d, 0xff, /* Filled Box (55,12) -> (56, 13) = ff [2x2 @ 4] */
    0x06, 0x3b, 0x0c, 0x3c, 0x0d, 0xff, /* Filled Box (59,12) -> (60, 13) = ff [2x2 @ 4] */
    0x06, 0x3f, 0x0c, 0x40, 0x0d, 0xff, /* Filled Box (63,12) -> (64, 13) = ff [2x2 @ 4] */
    0x06, 0x64, 0x13, 0x65, 0x14, 0xff, /* Filled Box (100,19) -> (101, 20) = ff [2x2 @ 4] */
    0x06, 0x64, 0x27, 0x65, 0x28, 0xff, /* Filled Box (100,39) -> (101, 40) = ff [2x2 @ 4] */
    0x06, 0x6e, 0x05, 0x6f, 0x06, 0xff, /* Filled Box (110,5) -> (111, 6) = ff [2x2 @ 4] */
    0x4f, 0x1b, 0x00, 0x1d, 0x02, /* Auto Rectangle: (27,0)->(29,2) 3x3 @ 1 */
    0x4f, 0x77, 0x00, 0x79, 0x02, /* Auto Rectangle: (119,0)->(121,2) 3x3 @ 1 */
    0x4f, 0x1b, 0x09, 0x1d, 0x0b, /* Auto Rectangle: (27,9)->(29,11) 3x3 @ 1 */
    ...
    0x50, 0x41, 0x1e, /* Pixel [65,30] =  1 */
    0x50, 0x41, 0x28, /* Pixel [65,40] =  1 */
    0x50, 0x52, 0x17, /* Pixel [82,23] =  1 */
    0x50, 0x52, 0x2c, /* Pixel [82,44] =  1 */
    0x41  /* Graphics mode off */
};
...
// Clear the screen
lcd.clearScreen ();
// Set the drawMode for the commands
lcd.drawMode (GLCD_MODE_NORMAL);
// Draw a pre-cooked list of commands.
lcd.write_P (complex, sizeof (complex));
```

SEE ALSO

GLCD::put(), GLCD::putStr(), Graphics mode serial command.

5.37 x/ydim

NAME

xdim, **ydim** - variables that define the *x* and *y* dimensions of the screen.

CLASS

GLCD - Alternative Serial Graphic LCD Library

SYNOPSIS

#include <AltSerialGraphicLCD.h>

uint8_t **xdim**;

uint8_t **ydim**;

DESCRIPTION

The *xdim* and *ydim* define the size in pixels of the of the screen horizontal (*xdim*) and vertical (*ydim*) dimensions. The values are undefined until read from the screen when the reset() method is invoked. The dimensions may be used to write screen size independent drawing logic.

The screen dimensions are queried from the screen using the query() command.

EXAMPLE

The following example shows the use of the screen dimensions in the Simple Application:

```
GLCD glcd(serial);
...
////////////////////////////////////////////////////////////////////////////
// Execution loop
void loop()
{
    uint32_t diff_time;               // Variable for the time difference
    char buffer [20];                 // Character buffer for strings
    uint8_t x_pos_1_4;                // 1/4 of horizontal screen
    uint8_t x_pos_3_4;                // 3/4 of horizontal screen

    // Work out the size of the screen and calculate the 1/4 and 3/4
    // horizontal pixel positions.
    x_pos_1_4 = glcd.xdim / 4;
    x_pos_3_4 = x_pos_1_4 * 3;

    // Prints "Hello World" to the screen and draws a tiny world (circle) in
    // the right 1/4 of the screen.

    // "Hello" is 6 * 5 = 30 pixels long, place at 3/4 of screen at the top.
    glcd.setXY(x_pos_3_4 - 15, 0);
    glcd.printStr("Hello");           // Print "Hello"

    // "World" is 6 * 5 = 30 pixels long and 8 pixels high place at 3/4 of
    // screen at the bottom.
    glcd.setXY(x_pos_3_4 - 15, glcd.ydim - 8);
    glcd.printStr("World");           // Print "World"
    ...
}
```

SEE ALSO

GLCD::query(), GLCD::reset().

6 Firmware

This section provides an overview of the structure of the firmware, a schematic of the software structure is shown in Figure 18.
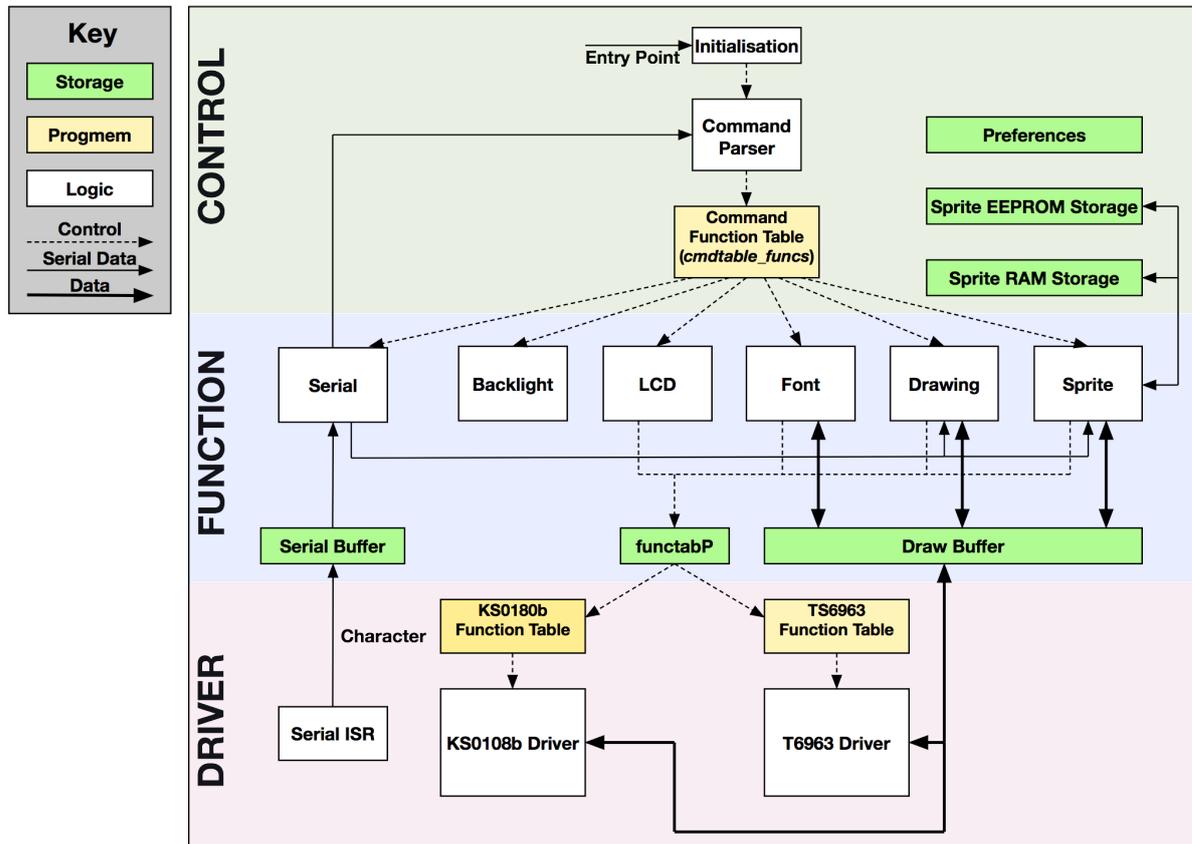

Figure 18: Firmware architectural overview

The software is effectively partitioned into three functional layers:

Control provides control of the whole system, at power-on then the system is initialised, the screen is identified and the appropriate function lookup table to the screen driver is installed. Control then drops into the *Command Parser* which receives serial input and parses commands which are looked-up in a *cmdtable*. The function to invoke is defined within the *cmdtable* which is invoked to process the command.

Function provides the logic to perform the serial command function, the arguments for the command are passed from the *Command Parser*. The commands at this level are grouped into basic sets of functionality. There is a certain amount of cross calling into other functional modules at this level but mainly the driver level is invoked to perform drawing operations by way of the *functabP* pointer which was determined at initialisation.

Commands such as *draw polygon* and `draw sprite` have potentially large data sets that cannot be completely stored on-board the backpack and the serial input is read incrementally to acquire the data.

Commands such as *fill polygon* and *font rendering* require a certain amount of data buffering and use

a 160 byte *Draw Buffer* as a working area. The *Draw Buffer* is also passed to the driver layer to enact any drawing required by the command.

Drivers provide the graphical and serial driver function. The graphical drivers provide basic pixel drawing and screen control primitives which are used at the higher levels to perform more complex drawing operations.

The principle memory areas used in the system are:

Preferences a byte array that stores the user preference i.e. reverse screen, splash screen etc. The settings are read from EEPROM at initialisation. The settings global and are accessed by macros.

Sprite Storage two storage areas exist for the storage of sprites in RAM and EEPROM.

Serial Buffer a 256-byte ring buffer that buffers the serial input data from the ISR ready for processing by the *Command Processor* and some of the drawing commands.

Draw Buffer a 160-byte (the width of the large screen) used by the drawing functions to compose data before rendering to the screen by passing to the driver level. Within the graphical drivers then the drawing buffer may be used to read and merge screen pixel data with the new drawing data before writing out the resultant pixels to the screen.

6.1   Firmware Files

The supplied firmware files are defined in Table 8 and described in the following sections.

Filename	Description
`Makefile`	The build file.
`backlight.c`	Control of the back-light.
`draw.c`	Line drawing and filling methods.
`font.c`	Font rendering methods.
`font_alt_5x8.h`	The 6x8 character set.
`font_tom_thumb_3x6.h`	The 4x6 character set.
`func.def`	Definitions for the function lookup tables.
`glcd.h`	Header file with all global definitions.
`ks0108b.c`	128x64 driver for Samsung KS0108B chipset.
`lcd.c`	Generic commands for LCD control.
`main.c`	Entry point, initialisation and serial command parsing.
`serial.c`	USART Serial device driver.
`sprite.c`	Sprite handling methods.
`t6963.c`	160x128 driver for Toshiba T6963 chipset.

Table 8: Firmware Files

6.1.1   Makefile

The `Makefile` contains the build rules and builds the firmware. Localisation may need to be performed for the following variables when moving to a new system in order to perform the IPS programming.

AVRDUDE_PROGRAMMER The type of ISP programmer to be used. Setting is `arduino` to use the Arduino

as the programming device.

AVRDUDE_PORT The serial port of the host computer used for programming. This should be changed to match the port on your system.

AVRDUDE_BAUD_RATE The baud rate of the IPS programmer.

The `Makefile` may be invoked from the command line, the following are commonly used commands:

```
make          -- Builds the firmware, reports the old and new code size.
make program  -- Builds the firmware and then invokes avrdude to program backpack.
make clean    -- Removes object and executable files.
make spotless -- Cleans the file system, including editor backup files.
```

6.1.2 backlight.c

Methods to control the backlight. The backlight brightness is controlled by a external pin on Port B. The brightness level is controlled by a PWM signal generated by Timer1. The timer configuration configuration reuses the code from the original Sparkfun software and is neat use of the timer.

The backlight level setting is conditionally written to EEPROM if the command used to set the backlight is the persistent command.

6.1.3 draw.c

The main line drawing functions are included in `draw.c` including the polygon, circle, line and box drawing. Polygon filling is implemented separately and performs line scans as described earlier in this document.

The drawing commands use horizontal and vertical line drawing functions `draw_hline`() and `draw_vline`(), respectively. These line drawing primitives are implemented within the screen drivers; allowing the line drawing to be better optimised for each screen.

6.1.4 font.c

All font rendering is performed in `font.c`. The main character rendering function used is bitblt which is implemented within the screen drivers; allowing the line drawing to be better optimised for each screen.

6.1.5 font_alt_5x8.h

This file contains the 6x8 font bitmaps, apparently taken from **Sinister 7**. The character glyphs are 5x8, the 6th character is the single line space that separates characters.

Some of the character glyphs have be modified when compared with the original file distributed by Jennifer Holt. I believe that this is the same file that was distributed with the Sparkfun version.

The font file may be replaced with another file to change the font. There is insufficient program memory remaining for any other font file.

6.1.6 font_tom_thumb_3x6.h

This file contains the Tom Thumb 4x6 font bitmaps. The Tom Thumb font is courtesy of https://www.pjrc.com/teensy/td_1

6.1.7 func.def

The `func.def` file contains the data that is used to build the function tables used for parsing the commands. There are 3 principle table definitions contained within the file.

The data within the file contains macro statements which are expanded by multiple macros defined in `main.c` and enumerations of the variables are made in `glcd.h`. The macro definitions are changed depending on what information is to be extracted from the macro statements of this file.

DEFFUNC defines the screen driver functions. These are the low level functions of the screen driver i.e. `set_pixel()`, `screen_clear()` etc. The definitions for both screen drivers are defined in the same table and are used to generate a function look-up table for each screen. The order of the table is not important as it is indexed with a generated enumeration label.

DEFCMDFUNC defines all of the functions that are called by the command parser which are screen type independent i.e. `draw_circle()`, `lcd_clear_screen()` etc. The order of the table is not important as it is indexed with a generated enumeration label (currently maintained in alphabetical order).

DEFCMD defines all of the serial commands and binds the serial command to a function and defines the parameter organisation. The table is strictly defined in increasing numerical order, where the numeric is the identity of the serial command. New serial commands are added to this table.

> The table is expanded in `main.c` into multiple smaller tables which is used as a binary chop lookup table.

6.1.8 glcd.h

The file `glcd.h` contains all of the external definitions and macro variables used in the system.

6.1.9 ks0108b.c

The Samsung KS0108B is the chipset used by the small display (128x64). This is a re-write of the KS0108b driver which differs significantly from the Sparkfun and Jennifer Holt version; both previous implementations used a timer wait and did not follow the Samsung data sheet for the KS0108b. This version implements the timings from the Samsung data sheet and performs a status check to determine when the chip is ready to accept the next command. This method considerably speeds up the chip access and has significantly changed the structure of the driver code. The abstractions used in the Sparkfun implementation i.e. `io_setup()` have been discarded in favour of explicitly setting the I/O state in the 3 major read/write methods: `ks0108b_read()`, `ks0108b_write()` and `status_check()`.

The `read` and `write` commands are carefully constructed to maintain the integrity of the chip control lines, specifically `RS`, `RW`, `CS1` and `CS2`. The chip control lines, excluding `EN`, are not modified until the `status_check()` operation is performed within the `read()` or `write()` operation. This requires that the `read()` and `write()` operations are passed the control line settings as part of their arguments on invocation. The `status_check()` is performed for the previous command BEFORE the control lines are modified this operation confirms that the previous command execution has completed without changing the previous control line (chip select) settings. Once the *status check* for the previous command has completed then if the chip select of the next command are changed (i.e. `CS1` and/or `CS2` change) then a second *status check* is performed before the new command instruction is initiated. Note: attempts to optimise performance and perform only the second *status check* for the new command is not sufficient, this strategy results in screen corruption. The double status check strategy has been confirmed with a 33 hour run totalling 7 million complete overdraw screen updates with no artefacts.

The basic logic for *status check* is defined as follows:

```
...
// Perform a ks0180b_read/write() operation
Perform status_check() with the last command control line settings.
IF chip select is changed THEN
    Perform status_check() with new command  control line settings.
ENDIF
Set the control lines for the next read/write operation.
Execute the next read/write operation.
...
...
Continue algorithmic processing....
...
...
// Perform a ks0180b_read/write() operation
Perform status_check() with the last command control line settings.
IF chip select is changed THEN
    Perform status_check() with new command  control line settings.
ENDIF
Set the control lines for the next read/write operation.
Execute the next read/write operation.
...
...
Continue algorithmic processing....
...
```

The set column position command has proved to be very problematic and difficult to solve; side 2 was very susceptible to setting an incorrect position intermittently which causes screen corruption (ironically in the *SimpleApplication*). This issue has eventually been isolated and resolved with the revised status_check() operation outlined above where both the previous and new command status is checked when the chip select is changed.

Figure 19 shows the results of soak testing the small screen with a variant of the *Simple Application* which has performed some 16 million screen updates with no intervening clearscreen() operation and shows no screen corruption.
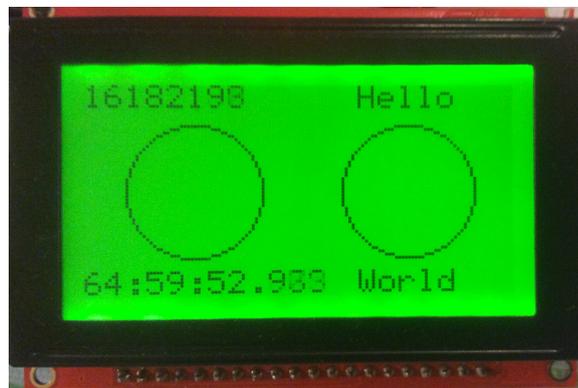

Figure 19: Firmware soak testing

This implementation discards the concept of pages which was used more in the Jennifer Holt version. Rather than maintaining the page state then it is easier to deal with each side of the LCD explicitly in the block/column read and write methods.

The Samsung data sheet uses an X and Y nomenclature for the axis of the screen; these are used incorrectly in the conventional sense and the data sheet uses X for a row address and Y for a column address.

The EN twiddling for chip enable from both of the previous implementations outwardly appeared to be a little strange when reading the code. The Samsung data sheet made reference to this in a note on LCD data reading. The implementation used here performs a full cycle dummy read (i.e. a ks0108b_read() operation)

and the data is discarded; a second and any subsequent reads are performed to clock the data out. The chip requires the first read to transfer the data from the LCD screen to an internal register; it is this first read that is discarded. The second and subsequent read(s) then extract the data from the internal register and place it on the data bus to be picked up externally. When performing multiple reads or writes the column position is auto incremented and subsequent reads will continue to clock the data out so the dummy read is only required on the first read when reading multiple bytes. A dummy write cycle is not required.

One further note. I cannot fully explain why the Jennifer Holt version needed to address the rows with 63 as the top left corner of the screen. This version uses 0 as the top left coordinate. The Samsung datasheet makes mention of the fact that the ADC line voltage changes the coordinate system. There is no information in the LCD datasheet that the ADC can be changed (unless this is related to the E signal).

Extreme care should be taken modifying any of the chip access timings. The chip seems to be very sensitive to timings and control/data line settings which are very difficult to track down and isolate; this might be easier if I had a logic analyzer and could see what was going on. The current implementation is very careful in setting the PORTC lines and ensures that the there are no intermediate transitions when setting states. The data lines (PORTD and PORTB) are disabled and restored to inputs as soon as any write sequence has finished. These two methods together seem to have removed intermittent screen corruption. Changes to the chip read / write sequence are are difficult to identify immediately and may appear to work but may manifest themselves under heavy load and can be spotted by single pixel corruption on screen i.e. a pixel set incorrectly; either set or clear when it should not be. If the timings are modified in any way then it is advised that a lot of testing is required using both sides of the screen to ensure that no regresions are introduced. Noted: that since the double status_check() has been introduced then timing does not seem to be such an issue, it is likely that previous screen corruption was a result of the status check rather than the instruction timing, the chip timings would exaggerate the problem.

The driver level does not deal with reverse screen, this must be handled at the higher levels and propagated through the *draw mode*. When the command *draw mode* is a *reverse* operation then any data that is read from the screen is inverted before processing new pixels which are then merged into the screen data; the data is inverted again when it is written back to the screen.

The principle drawing commands that are implemented in the driver are *bitblt*(), horizontal line draw *hline_draw*() and vertical line draw *vline_draw*().

6.1.10   lcd.c

The file lcd.c implements a motley set of screen functions that did not really fit anywhere else including the *factory reset* and *query* functions.

The LCD reset command uses the AVR watchdog timer to perform the screen reset, the watchdog is configured for a relatively short timeout and then the commands enters an infinite loop allowing the watchdog to trigger and restart the processor.

The reading of the screen size from PINB has also proved to be problematic, especially when performing a watchdog reset. Approximately 1 in 20 watchdog restarts of PINB return an incorrect sense. Increasing the current settling delay of 5us does not seem to make any difference, possibly the ground level is floating high during a watchdog reset which is why it is reading the wrong sense? The reading does not seem to be incorrect from a normal power-on operation. To improve reliability then the screen size sensing is now performed in *factory reset* and the pin sense is read and saved in EEPROM accessed via the *prefs* structure. The screen size value is automatically initialised when the firmware is first run and EEPROM is not configured. On a power-on or restart the stored EEPROM value is used to configure the device rather than relying on sensing the pin. This methof of using EEPROM reduces the number of reads of the PIN so it is less liable to be incorrect. Obviously the pin read may still be wrong when a factory reset is performed, the hope is that it gets it right. The screen size EEPROM setting may be explicitly hard-coded in the lcd_factory_reset()

function if this proves to be a problem for the display.

6.1.11   main.c

The entry point of the program, `main.c` controls the initialisation sequence and serial command parsing. On start-up then the watchdog timer is configured with a 2 second timeout, this re-starts the display if the firmware fails and locks up. The watchdog timer is reset in the serial `getc()` and `peek()` functions, these are the only two locations in the firmware where there may be a significant delay waiting for the next character.

Once the watchdog has been set up the main preferences `prefs` are set up, these are the user preferences for serial speed, reverse screen etc. which are stored in EEPROM. A few sanity checks are performed to ensure that the EEPROM has been initialised, otherwise the EEPROM is reset to default values.

Using the screen type EEPROM size then the variable `functabP` is set up to point to a table of function pointers that point to the driver specific commands for the screen type (defined in `func.def`). The screen and backlight are then partially initialised.

Finally the splash screen is displayed, if required, via the `lcd_demo()` command. The system halts within `lcd_demo()` and maintains the display until a character is received on the serial port when the system drops into a command processing loop.

The command processing loop takes a character at a time from the serial port and draws the character or executes a command. The parser is based on a function lookup table defined in `func.def` which performs a binary chop lookup of the serial character. The function table is retained in Flash memory and uses `pgm_read_byte/word()` to extract it from Flash memory, the `pgm_read` operation is effectively a macro and does not result in a actual function call and appears to be about the same speed when compared to a table in RAM. The function to call is taken from the look-up table and is invoked with the appropriate number of arguments as directed by the look-up table.

The parsing code is quite difficult to read but is effectively replacing a large case statement. The logic of the parsing code should not need to be altered when new commands are added to `func.def` provided that the argument format is consistent with existing commands.

6.1.12   serial.c

`serial.c` contains the code for the serial driver which provides support for sending and receiving characters. The implementation is largely based on the Jennifer Holt implementation.

Serial input is managed through an ISR which adds characters to a 256 byte ring buffer. The ISR measures the fullness of the serial buffer and will send a XOFF character if the buffer fills up too much.

Characters are removed from the serial buffer using `serial_getc()` which de-queues a character from the serial buffer and sends a XON if serial input is stopped and the buffer nears empty. The `serial_peek()` function allows a character to be read from the buffer without de-queueing it and is used by the KS0108B bitblt operation to look ahead in the buffer.

6.1.13   sprite.c

`sprite.c` contains the functions for managing sprites including sprite upload, display in addition to managing the splash screen setting.

6.1.14   t6963.c

The Toshiba T6963c is the chipset used by the large display (160x128). This is a ground up implementation of the T6963c driver which was taken from the Toshiba chipset. This implementation differs from the

Sparkfun Electronics version in that it uses the data auto write and auto read features which enable a greater throughput of data to the screen. Additionally read and write commands have been re-organised with minimal delays using the timings from the Toshiba chipset specification sheet.

This chipset is a lot faster than the KS0180B and seems to be a little less fussy about the setting of control lines. The data organisation is different from the KS0180B and writes horizontal rows of 8-pixels rather then vertical columns. The difference in orientation of pixels means that for bitbit operations, including font rendering, then the sprite data needs to be reorganised from a vertical to horizontal orientation which has to be performed pixel by pixel. The internal character generator of the T6963c is not used.

The principle drawing commands that are implemented in the driver are *bitblt*(), horizontal line draw *hline_draw*() and vertical line draw *vline_draw*().

6.2   To do

Further development items on my list of things to do include the following once I have some more time:

Optimise Circle the circle draw needs some further optimisation, should be possible to reduce the code footprint by some 50%.

Fill Polygon make this method more robust. There are a lot of exception cases and some refinement of the algorithm should be possible resulting in it being more robust.

160x128 Native Font Use the native font in the screen rather than bitblt font. There is a not insignificant overhead in rotating in the bitblt for character rendering.

Vertical text turn the text through 90 degrees. Instead of just moving bits, given the text is defined as vertical strokes, then implementing a horizontal bitbit should be the best solution for the T6963c 160x128 screen; the KS0180B requires the bit processing to rotate.

Latin Characters add extended Latin characters.

Copy Region Export a region of the screen as a bitmap and return to the caller over serial. Allow a screen copy to sprite RAM location allowing replication of parts of the screen to save on Host drawing. Consider vertical and horizontal mirroring on the sprite draw using the draw mode to control the render.

ASCII Drawing Clean up and release ASCII art converter. This allows drawing of the screen in an editor (MicroEmacs) and convert to drawing commands. The current working version needs to be cleaned up.

Host library Release the Host version of the GLCD library used on MAC/UNIX system with FTDI serial cable to prototype the backpack drawing algorithms.

Completed items

Firmware 1.38 - Additional small font http://robey.lag.net/2010/01/23/tiny-monospace-font.html looks to be a good choice giving a 4x6 (3x5 usable pixels) font.

Firmware 1.38 - Bitblt Fixed bitblt issues with non-aligned multi-line images on the small screen,

Revision History

Date	Who	Description	Revision
2015/07/06	JG	Fixed a few typos and added missing firmware file for Tom Thumb font. Corrected the serial connection diagram.	1.27
2015/07/05	JG	Add the Tom Thumb font and new **fontFace** commands. Added GLCD::fontFace(). Added GLCD::setFontFace(). Added GLCD_MODE_FONT_PROPORTIONAL to GLCD::fontMode(). Added setString() for font rendering. Firmware version 1.38	1.25
2015/06/05	JG	Added TODO list. Fixed the CRLF description following testing. Firmware version 1.37	1.23
2015/06/04	JG	Added the **setScroll**() command and simplified the CRLF methods to a basic set method only.	1.21
2015/05/30	JG	Added *See Also* sections and fixed cross references. Added **set**() and **ready**() command.	1.14
2015/05/29	JG	Finished draft of the GLCD class methods.	1.12
2015/05/25	JG	Added a description of the GLCD class.	1.9
2015/05/19	JG	Added architectural overview schematic.	1.6
2015/05/17	JG	Added information on the Reset LCD and watchdog timer. Provided further information on the KS0108b control line logic.	1.5
2015/05/16	JG	Added the ISP programming and outline of the code base.	1.4
2015/05/09	JG	Initial version of the document	1.0

Firmware Change Log

Date	Who	Description	Revision
2015/07/05	JG	Add the Tom Thumb font . Added serial command Font set. Added GLCD::fontFace(). Added GLCD::setFontFace(). Added GLCD_MODE_FONT_PROPORTIONAL to GLCD::fontMode(). Added setString() for font rendering. Fixed bitblt on the small screen, incorrectly handling multi-line non-aligned bitmaps. Fixed the **setXoff**(), **setXon**() and **setScroll**() functions in the header file, incorrect **putcmd**() format.	1.38
2015/06/05	JG	Initial version.	1.37