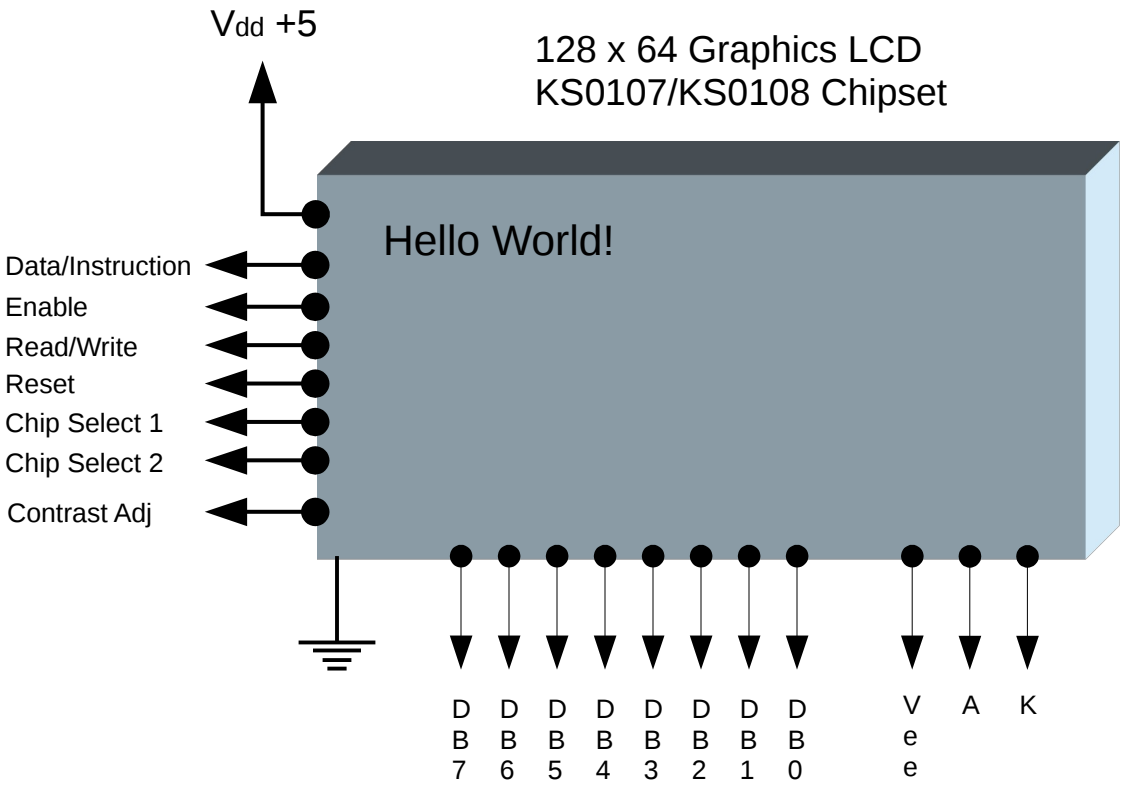
Graphics LCD Display C Library for the PIC18F4550 MCU (or similar advanced microcontrollers)

Version 1.00 – 5/2013 **www.muniac.com**



Introduction:

The purpose of this guide is to make installing and using the GLCD library as easy as possible. Herein are detailed instructions on what needs to be accomplished to build the GLCD library and get a KS0107/KS0108 based 128x64 GLCD working properly. The easiest of which would be to use the default wiring, configuration and settings. Perhaps the best starting point for new projects. As specific applications deviate from the defaults, more steps will need to be undertaken to create a working build of the GLCD library. Every effort has been made to make this as easy and straight forward as possible. Creating a custom build, in most cases, requires only simple edits to the HWSpec.h header file. None of which are difficult. Directions on how to perform these edits have been included in this guide. In these cases, no source code modifications will be required. All header and source files have been thoroughly commented.

All the core functions of the GLCD library have been written in C using straight forward programming techniques. The goal being to make the C code as easy to read, understand and modify as possible. Having an understanding of C and MCU architecture will allow a more in depth understanding of the GLCD library, its installation and usage. Although not required at the expert level, this document assumes the end user possesses such knowledge.

For some of the C functions, optional assembly language programs have been included to provide a slightly smaller program that runs quicker. The extra functions provided in this package that allow reading/writing of EEPROM and flash program memory have also been written in assembly language. They have nothing to do with the core functions of the GLCD library. They have been provided as ancillary tools that might be helpful in certain applications. Using them is optional.

The library concept of providing a collection of functions to an application through a link library allows for efficient inclusion of individual modules. Any modules that aren't called in the main program and/or called by dependencies won't become part of the final build. This keeps program size to the minimum required and avoids wasting valuable memory space with code that never gets called.

Development Platform:

All the programs included in the GLCD library were developed using Microchip's MPLAB IDE V8.70 and the ICD 3 programmer running under Windows XP. The PC and O/S software were Oracle's VirtualBox virtual machine running Windows XP as a guest hosted by Linux Mint 14 running on a Dell Inspiron 1501. The MPLAB tool suite was as follows:

- C18 Optimizing Compiler (mcc18.exe V3.45)
- MPASM Macro Assembler (mpasmwin.exe V5.48)
- MPLINK Object Module Linker (mplink.exe V4.46)
- MPLIB Object Module Librarian (mplib.exe V4.46)

The library has received extensive testing. Included with the GLCD library are a simple “Hello World” demo and a comprehensive demo that exercises all the features of the GLCD library. It's best to

start with the simple demo first and increase complexity from there. A combination of ports, RESET control, serial and direct connections of control lines have been tested. This was done to verify the conditional compilation and resulting code generated from changes in HWSpec.h For each test case, the GLCD library was completely rebuilt to check its portability. A new build of the demo program (GLCD_Demo.c) was then used to exercise all the features and functions of the GLCD. Although these test cases are not exhaustive, they do utilize enough variations to verify all the features of the GLCD library code, header files and ease of changeability. It's assumed that if functionality is correct for ports A and C then port B can be substituted for either A or C with no ill affects. This assertion is correct to the extent a given port provides adequate signal type, direction and I/O pins. For example, the 40 pin PIC18F4550 ports (A->E) aren't all 8 bits wide. A port(s) that doesn't provide digital I/O on all 8 pins can't be used for the data lines. Owing to this MCU hardware restriction, complete interchangeability of ports isn't possible. The test cases are thus restricted to utilizing ports that are appropriate for control and data line connections with the respective connection methods. The following 11 test cases were run successfully on a 40 pin PIC18F4550 from MPLAB ICD:

RESET not tied to Vdd, serial control lines and PORT D (0-7) used for data lines. Strobe, Data and Clock Pulse derived from the following ports and pins:

PORT A (bits 0-2) Test case 1 using 74HC4049
PORT A (bits 3-5) Test case 2 using 74HC4049
PORT C (bits 0-2) Test case 3 using 74HC4049

RESET tied to Vdd, serial control lines and PORT D (0-7) used for data lines. Strobe, Data and Clock Pulse derived from the following ports and pins:

PORT C (bits 0-2) Test case 4 using 74HC4049
PORT E (bits 0-2) Test case 5 using 74HC4049

RESET included with dedicated parallel control lines on PORT A. Data lines connected to different ports as follows:

PORT B (0-7) Test case 6
PORT D (0-7) Test case 7

Data lines connected to PORT D and RESET included with control lines connected on different ports as follows:

PORT A (0-5) Test case 8
PORT B (0-5) Test case 9

Data lines connected to PORT B and RESET tied to Vdd. Control lines connected on different ports as follow:

PORT A (0-4) Test case 10
PORT D (0-4) Test case 11

Compiler Options, Memory and Architecture:

The PIC18F4550 MCU uses the Harvard Architecture computer model. This means separate program and data memory buses exist which allow simultaneous fetching of both data and instructions. This results in faster program execution. It also results in a slightly more complex memory model which must be dealt with when writing a compiler and developing application software. The C ANSI/ISO standard allows for code and data to be in different places but it isn't sufficient to locate data in the code space as well. The C18 compiler introduces the ***rom*** and ***ram*** qualifiers which denote objects located in program or data memory respectively. If the GLCD C source code will be compiled on a different platform (something other than C18), replacing the C18 ***rom*** and ***ram*** qualifiers will be required to avoid compilation errors. Otherwise the C source code complies with the ANSI/ISO standards. No in line assembly code is used.

None of the GLCD functions use recursion. Thus function arguments can be declared as either ***static*** or ***auto*** during compile time. Arguments declared as ***auto*** are pushed onto the software stack. Those declared as ***static*** are allocated globally. For example, the C18 compiler provides a compiler switch (for choosing ***static*** or ***auto***) accessible via the command line or through the IDE. Choosing either ***static*** or ***auto*** changes the amount of program/data memory required for the final build. It also has implications on program execution speed. Arguments accessed via the stack tend to be slightly slower owing to more instructions required to access data. Arguments placed on the stack also use significantly less data memory. So there are trade offs which need to be selected based on specific applications. The table below summarizes ***static*** *and* ***auto*** builds for GLCD_Demo.c:

Comparing Builds for C18 Compiler Options STATIC and AUTO			
Compiler Option	Data Memory Used	Program Memory Used	Execution Speed
AUTO	219 Bytes	25,547 Bytes	Slower
STATIC	342 Bytes	22,989 Bytes	Faster
Values are decimal. For PIC18F4550 w/32k program ROM and 2k data RAM			

C18 default optimizations were enabled along with the traditional instruction set, small memory model, large data model and multi-bank stack model for each build shown above. It's also important that all compiler switches be set identically when compiling the application and creating the GLCD library. Failure to do this may result in an incorrect build. In other words, something that may appear to build correctly but won't run properly.

Linker Options:

Linking is the process by which all object files (.o or .O) are brought together into a single executable file. Object files are those coming from the application project as well as those contained in one or more link libraries (.lib files). As object files get assembled into an executable, addresses are resolved and memory space is allocated as required to name just a few basic linker operations. With MPLINK, a linker script (.lkr file) is provided to allow control over the linker process by setting options within the script. One area of importance with using the GLCD library is instructing the linker where to look for library files. More information about searching libraries is provided later in this document.

Building a Library:

A link library is a single file that contains a collection of relocatable object modules. In general, libraries are built around a common theme of functionality like I/O, math, string, etc. For example, the GLCD library contains all the functions relating to displaying fonts and graphics. Libraries offer a convenient and efficient way of using functions. Including a single library in the linker process provides access to all the functions contained within it. This could run into hundreds of functions for large complex applications. Only those functions called from the library, however, become part of the executable. Thus no dead code gets created. Multiple libraries can be included in a search path allowing easy access to diverse processes.

Building libraries requires the use of a library utility program like MPLIB or equivalent. The librarian assembles object files into a single library file. Formats for these files are usually manufacturer specific. Thus it's best to remain within a single development environment and tool suite to ensure a high level of compatibility. More information about building the GLCD library is provided later in this document.

The GLCD Package:

The GLCD package includes 4 PDF documents, demo video and a source code disk. The PDF documents and demo video are available free on line. The source code disk must be purchased. The major components in the package are listed below:

- Introduction Document (Free PDF file)
- Functional Reference (Free PDF file)
- Installation Guide (Free PDF file)
- Schematics (Free PDF file)
- Source Code CD (**Must Be Purchased**)

The source code disk contains all the C source, assembly source, header files, linker script, batch library creation file, GLCD library files, demo video and all the documentation. Please review the documentation to determine if the GLCD package is appropriate for a specific MCU application before purchasing the source code CD.

Installation:

MCU applications using the MPLAB IDE and a PIC18Fxxxx family of advanced microcontrollers may be able to use one of the two GLCD.lib files already built. The ***Schematics*** PDF shows how the 128x64 GLCD must be wired to be compatible with the two GLCD.lib builds. The ports used for control and data lines must also be A and B respectively. One build uses the ***static*** storage class for arguments while the other build uses ***auto***. Make sure the application code compilation matches the library build type. In addition, project build options must be set as follows:

- Extended Instruction Set Disabled
- Small Program Memory Model
- Large Data Memory Model
- Multi-Bank Stack Model
- Default Optimizations

To make use of the GLCD functions in the GLCD library, the library must be included in the collection of libraries the linker requires to resolve function calls. This can be accomplished either by including the library as part of the MPLAB IDE project in Library Files or modifying the linker script. Below is an example linker script file for the PIC18F4550 which shows the two script lines (in red) that were added to include the GLCD library.

```
// File: 18f4550_g.lkr
// Generic linker script for the PIC18F4550 processor
// 4/20/2013 LIBPATH Modified to search glcd.lib first

#DEFINE _CODEEND _DEBUGCODESTART - 1
#DEFINE _CEND _CODEEND + _DEBUGCODELEN
#DEFINE _DATAEND _DEBUGDATASTART - 1
#DEFINE _DEND _DATAEND + _DEBUGDATALEN

LIBPATH "c:\X Drive\GLCD";.

#IFDEF _CRUNTIME
 #IFDEF _EXTENDEDMODE
   FILES c018i_e.o
   FILES clib_e.lib
   FILES p18f4550_e.lib

 #ELSE
   FILES c018i.o
   FILES glcd.lib
   FILES clib.lib
   FILES p18f4550.lib
 #FI
```

The LIBPATH in the above example assumes the GLCD library is in the directory "c:\X Drive\GLCD". This is a project specific name and should be changed to reflect the current GLCD library location. Note the above will cause the GLCD library to be searched first. Projects that don't conform to the default wiring, use different port assignments, use a different MCU family or different IDE will require a custom build. This is easy to create.

Building A Custom GLCD Library:

Before a custom build can be created, decisions need to be made about which ports will be used for control and data lines. Whether a direct connection or serial method will be used for the control lines and if RESET will be software controlled or tied to Vdd. And what mapping is being used to wire the control and data lines to the MCU ports. The 5 questions below summarize all that needs to be considered before attempting a custom build:

- What port will be used for the control lines?
- What port will be used for the data lines?
- Will the control lines be a direct connection or serial method?
- Is RESET tied to Vdd?
- What mapping is being used to connect GLCD control and data lines to the I/O port pins?

The port used for the control lines needs to be output only. Its width can be anywhere from 3 to 6 bits wide. The port used for the data lines needs to provide 8 bits of both input and output. If port pins are tight, it may be advantageous to use a serial method for the control lines. Extra circuitry is required for this and a tested circuit can be found in ***Schematics***. This will allow just 3 port pins to manage the control lines. Serially managing the control lines introduces only a very slight degradation in speed.

The hardware RESET signal does some GLCD housekeeping by resetting registers, addresses and the top line. It also turns off the GLCD. If RESET is tied to Vdd, the housekeeping is automatically simulated in software. Again, there is a very small performance penalty for this when a RESET is called for. Most applications would not find this even noticeable. On the other hand, freeing up an additional port pin might offer outweighing advantages.

For simplicity sake, consider mapped port pins in ascending order by pin number for control and data lines. There is nothing mandating this beyond making the wiring as easy to understand as possible. The default GLCD wiring in ***Schematics*** shows an example of ascending pin function mapping. That said, use whatever a specific application benefits from the most. It's also a very good idea to make a schematic diagram of the specific MCU and GLCD wiring that is being used. Like all software/hardware systems, only one mistake in wiring and/or a parameter being set will render an entire system dysfunctional. A schematic creates an organized road map of a system and may help diagnose problems later on. Above all, make sure that any and all details are carefully attended to. Check grounds, power feeds and digital wiring to ensure it's correct. Always use good circuit design practices. Once all the above have been resolved, the first step in a custom build can be attempted. That being making modifications to HWSpec.h This single header file contains all the hardware specific information. Editing this file to reflect a specific application's requirements will “tune” all of the GLCD C source code.

Applications using another IDE, MCU, compiler, linker and librarian will need to make the necessary adjustments for that system. Although some details of this may differ between IDE platforms, the compile, link and library concepts are basically the same. The C code provided herein is very basic in the features of the language used. As such it should compile with any quality C compiler.

Making Changes To HWSpec.h:

It's assumed the reader understands the C **#include** and how header files are located and processed. HWSpec.h may be found on the GLCD CD in a subdirectory named GLCD. This header file should remain with the source code in a dedicated directory. Editing this file should be done with a programmer grade editing tool that processes end-of-line, tabs and other whitespace characters properly. If these whitespace characters aren't handled properly, seemingly meaningless errors can creep into the compile process.

With a copy of the GLCD library on the target PC, open HWSpec.h and make sure its contents are as expected. The header file includes liberal comments, explanations and other helpful data. It's a good idea to take a few minutes and scan through the header file to gain a preliminary understanding of its contents and organization.

STEP 1:

Lines 28-30 contain three **#include** statements near the beginning of HWSpec.h which are processor specific. Examine these and adjust accordingly to the specific processor being used. This is required so port and register defines are correct for the specific MCU being used.

STEP 2:

At lines 41 and 42 are two **#defines** that identify the names of the direction control register (TRISx) and I/O port (LATx) being used for the control lines. These are defaulted to TRISA and LATA for the PIC18Fxxxx family of advanced MCUs. Note that the tokens TRISA and LATA are defined in the processor specific header file (for example pic18f4550.h). Make the necessary changes to these two lines. For example, substitute TRISB and LATB if port B will be used for the control lines.

STEP 3:

If the control lines are direct connected (one control line per port pin) 5 (RESET tied to Vdd) or 6 (RESET connected to port) bits will be required. Lines 71-78 define the mapping between the GLCD control lines and port pins. If wiring has been done in ascending order by pin number no changes are required. Each control line has a mask which isolates a specific bit for that control line. Adjust these masks as required to match the GLCD control line wiring being used.

If RESET is tied to Vdd, its mask must be set to 0 as follows: **#define S_RST 0** Otherwise set a mask for this according to which bit is controlling this signal.

STEP 4:

If a serial method (extra circuitry required) of managing the control lines is being used then 3 bits of the control port communicate with a shift register (74HC4049 in this example). The masks defined above then relate to the Q outputs of the shift register instead of the MCU I/O port pins. The concept is the same just with different connections (Q outputs instead of port pins).

Lines 112-114 define the masks required to isolate the control bits for STROBE, DATA and CP (clock pulse) on the 74HC4049. The default is these being wired in ascending order by pin number. The circuit presented in **Schematics** shows a working example. If something other than port bits 0-2 are going to be used, adjust the masks appropriately. A table of 8 masks has been included in the comments just about the **#define** statement. Three masks will be required to map STROBE, DATA and CP to their corresponding port pins.

STEP 5:

Line 49 defines NShift which is the length of the bit stream shifted into the 74HC4049. If the GLCD control lines are direct connected, set NShift to 0. For serially managed control lines, NShift will be set to 5 (RESET tied to Vdd) or 6 (RESET controlled via a Q output). Spare outputs have been provided and if used will require a larger NShift value. User code must be written to make use of the spare bits. The header file includes explanatory comments about all the hardware and configuration specific data items.

STEP 6:

Lines 150-153 define which MCU port will be used for the GLCD data lines. This 8 bit wide port must be capable of both input and output. The default is port B. These are structured to provide the C statements necessary to set port's data direction. For the PIC18Fxxxx family of advanced MCUs, TRISx is the generic data direction register. 00 sets each bit as output and ff sets each bit as input. Applications using other MCUs will need to convert this to the appropriate system. Define statements are also included to provide the C statements to read a port and write a port. Edit these to reflect the desired port. For example, simply replace “B” with “D” in 4 places (shown in red) if port D is the chosen data line port. The header statements for port D would look as follows:

```
#define     SetP_OUT    TRISD=0x00;     /* Code to set port(s) as output */
#define     SetP_IN     TRISD=0xff;     /* Code to set port(s) as input */
#define     RDPORT      byte=PORTD;     /* Read port(s) code */
#define     WRLAT       LATD=byte;      /* Write port(s) code */
```

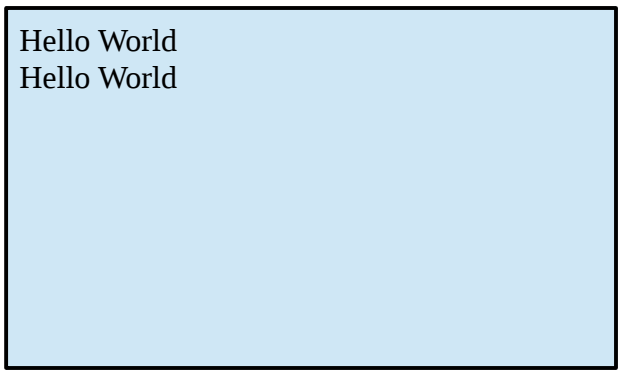
With steps 1-6 completed, HWSpec.h should now reflect the control/data ports being used, control line method (direct connection or serial), how RESET is being controlled and the processor specific header files. Double check these edits making sure all are correct. Also check the masks against the wiring to ensure bit positions match control/data line wiring. With the HWSpec.h file up to date, a custom build of the GLCD library can be completed.

Building The GLCD Library:

A batch file (.bat) has been provided to compile all the source code and create a link library file (.lib). This file uses the MPLAB tool suite. Paths to the header files and executables will need to be setup correctly. These would include those of the C standard library, IDE specific files and those required by the macro assembler if the application will be using the extra non GLCD functions. For example, EEPROM read/write.

Select an argument type of ***static*** or ***auto*** by changing the switch in the compiler command line. If the extra assembly functions will be used this must be set to ***auto*** to make sure all arguments are pushed onto the stack. The batch file includes comments on how to set this switch. From the command prompt, switch to the GLCD directory and run the batch file. Fix any errors that may occur due to incorrect or missing paths. The compiler and assembler will process all the files that end in .c and .asm Assembly files (6 in total) have been given the extension .assembly to force these to be excluded. Replacing the .assembly extension with .asm (via ren *.assembly *.asm) will make them visible to the assembler. Note the assembly files are optional and aren't required for the GLCD library to function. C object files use a lowercase “o” while assembler object files use an uppercase “O”. If everything has been done properly the batch file should compile all the source code and create glcd.lib which will contain a collection of .o files (and .O if assembly files have been included). Note the module count may vary based on the disposition of the object files processed by the librarian. Once built, the GLCD library can now be used by the linker to resolve any GLCD function calls that have been included in the application's main program. Check to make sure all the C source files compiled without any errors. An unresolved address (can't find error) from the linker usually indicates a missing module in the library. The most likely cause is a compilation error. Another common cause is a function name has been misspelled. Remember that C is case sensitive.

For MPLAB IDEs, the glcd.lib file can simply be added to the project under library files. Or by modifying the linker script as explained on page 7. Either method is fine and the goal being to make the GLCD functions available to the application's main program. With a correctly built glcd.lib file in place and accessible to the linker a main program test build can be attempted. A good place to start is with HelloWorld.c This simple program has been included with the GLCD software and does nothing more than display **Hello World** on the GLCD in two places. One at row 1 column 1 by issuing a call to **Print_Str** and another at row 2 column 1 by issuing a call to **printf**. If all is working properly, running HelloWorld.c should cause the display to look like that below:



HelloWorld.c contains two **#include** statements that require proper paths to the header files. Set these paths to reflect where on the target system the include files are located. Failure to do this will result in the compiler issuing missing token and syntax errors.

If HelloWorld.c displays its two messages as expected then the most basic of functions are working properly. This means control and data lines are connected correctly and there are no timing or port issues to contend with. A full blown demo program called GLCD_Demo.c has been included with this package to exercise all of the available features. The next logical step is to compile GLCD_Demo.c and get it running properly.

Trouble Shooting:

If the display isn't working as expected trouble shooting will be required. This may take several minutes, several hours or more depending on the problem(s). A systematic step by step approach will be required to verify that each part of the interface is working properly. Begin with the most basic things first. Check power supply connections, grounds and back lighting. Check the contrast wiring and make sure the voltage on the contrast control varies as expected. Double check all the parameters in HWSpec.h to make sure they are set properly. Verify the masks to make sure they isolate the correct bits for each of the control lines. Check the RESET signal to make sure it is at the correct level. Inspect and check control and data line wiring for pin to pin correctness between the GLCD and MCU. If a timing problem is suspected, increase the delay by changing the parameters in HWSpec.h If all of the above are correct then signals at the port pins will need to be checked.

LEDs with an appropriate series current limiting resistance are helpful with monitoring port pin levels. A bank of 8 LEDs will cover any and all of the important signals. A good place to start is with the control lines. Connect an LED to each of the control lines. Wire the LEDs such that they turn on with a 1 and off with a 0. A small test program can easily be written to set bits in CByte. Passing this byte to function **GLCD_Ctrl_Out** should set the corresponding control line bits as specified in CByte. Masks can be built from those defined in HWSpec.h to set/clear bits as a more thorough check. For example, **S_EN** should set/clear the bit that controls the enable signal. Make sure each control line turns on/off with a 1/0 in each bit position. This will be 5 bits if RESET is tied to Vdd. 6 bits if RESET has been wired to a port pin. Make sure these signals are appearing on the GLCD control lines too. Carefully check pin numbers and their functions.

Using the same LED technique described above, check the 8 signals on the data line port. Direct calls to function **PORT_Out** can be used for this. Pass in test bytes and make sure each of the port pins toggles properly. Again, a simple test program can be written to accomplish this. Using a delay function and shift within a loop, a 1 bit can be walked through each of the 8 bits. A 1 or 2 second delay will slow down the signal changes so they can be seen with the naked eye. Make sure this is happening on the GLCD pins too. If all of the signals are working properly, check the GLCD command structure with that presented in this documentation. Make sure the proper command bytes are being sent to the GLCD. For example, a simple error with turning the GLCD on would cause output to fail. The same would be true for the write data command. Chip select problems would also cause the display to not function. As a final step, make sure the display being used is 100% compatible with the Samsung KS0107/KS0108 protocol.

Keep in mind that a garbled display usually indicates timing problems and/or mixed up data line signals. A completely dark display usually indicates control line signals missing, mixed up or not of the proper level. Make sure active high and active low signals are of the proper logic sense. For example chip select is active high on KS0107/KS0108 devices. RESET is active low.

Running GLCD_Demo.c:

Once problems are located and rectified, the full blown demo program can be compiled, linked and run. At this point the "Hello World" test should be working properly. The GLCD_Demo.c program includes code to exercise all the GLCD features. By default the extra assembly language functions that read/write flash program memory, EEPROM and handle serial control lines (via a 74HC4049) are excluded. With the assembly functions excluded from the GLCD library, calls to these functions are also excluded in the demo program. A **#define** parameter has been provided in the beginning of the demo program to take care of this. The default value (0) is to exclude the assembly functions. If this isn't the desired action, edit the program, locate this parameter and follow the instructions in the comments to include the assembly calls. This is very simple to do. It will also require these to be assembled and included in the GLCD library.

For MPLAB IDE platforms, remove HelloWorld.c from the project and replace it with GLCD_Demo.c Initiate a new build in the usual way. This should result in a successful compile and link. Program the PIC as before and run the demo. If everything is correct, the demo program will take the GLCD through all of its features. A successful demo means the hardware/software interface is correct and the GLCD functions can now be used in an application program. Use the ***Function Reference*** included with this package to understand how calls should be made.

Assembly Language Files:

Included with this package are 6 assembly language files. These have been written for the PIC18Fxxxx family of advanced microcontrollers. They assume the traditional instruction set, small memory model and that C arguments will be passed on the stack. None of these "extra" assembly files are required for the GLCD functions. They have been provided to help with certain applications and as useful examples of how C callable assembly language functions are written. These assembly functions read/write flash program memory and provide access to EEPROM. There is an assembly version of GLCD_Ctrl_Out.c which may provide smaller and faster code for applications using a 74HC4049 to serially manage the control lines. Tuning this assembly program will require editing several **#define** statements. Refer to the comments in the source file for more information.

Reading/writing/erasing flash program memory as well as the EEPROM require assembly instructions. All of this is well documented in the PIC18Fxxxx data sheet to include examples. The assembly programs provided herein to accomplish this are based on the examples provided in the data sheet. All are callable from C. Certain limitations and protocol must be followed precisely in order for these to work properly. For example, a special buffer area needs to be defined which lands on a 64 byte boundary. An example of how to set this up has been included in GLCD_Demo.c Refer to the comments included in the source code for more information.

Other Header Files:

A total of four header files are required for the GLCD library. **HWSpec.h** has already been explained and is where all the hardware specific information is contained. This file requires editing to "tune" the GLCD functions to a specific application. All the header files include generous comments. Refer to these for more information and specifics. In most cases a user application won't need to include **HWSpec.h** in the main program.

Fonts.h – Contains the font bitmaps and housekeeping information required to display characters on the GLCD. Remember a GLCD does not contain a character generator so each character must have a bitmap image. In addition to the bitmaps, a collection of global variables are defined in this header file. They include row and column information, pointers to fonts, fill tables and the flash program read/write buffers. CByte is defined here too and its bits set all the control line signals.

GraphicsLCD.h – Contains a collection of **#define** statements to make the code more readable and transportable. In addition, the font control structure is defined here. This allows switching between the two fonts. This structure contains all font specific data items to include a pointer to the bitmap table. All the GLCD function prototypes are included as well.

ExternVars.h – Contains an external listing of all the global variables required to make variables known throughout the GLCD source files. These aren't definitions but simply indications of external declarations. They tell the C compiler these are legitimate variables with their definitions in an external file. In most cases a user application won't need to include this header file in the main program.

The typical list of includes for an application's main program are as follows:

```
#include <stdio.h>
#include "c:\GLCD\GraphicsLCD.h"
#include "c:\GLCD\Fonts.h"
```

Make sure paths are correct for the target system's directory so these header files can be found.

References and Acknowledgments:

The internet contains a wealth of information about GLCDs, MCU interfacing techniques and software. The Samsung KS0107/KS0108 protocol is quite common. It's always a good idea to search there and see what pops out. Dedicated blogs are another place to troll around for help, information and suggestions. Below is a short list of other resources that may be helpful:

http://www.microchip.com - Microcontrollers, technical data and software
http://www.digikey.com - Electronic parts
http://www.arduino.cc/ - Electronics prototyping
http://www.adafruit.com - Electronic parts and GLCDs
http://www.vishay.com - GLCDs
http://store.elsevier.com/categoryController.jsp?categoryId=EST_IMP-73 – Technical books
http://embedded-lab.com/blog/?p=2398 – GLCD interfacing project
http://www.futurlec.com/Boards.shtml – MCU prototyping boards

MUNIAC, LLC
scott@muniac.com