# The GUI Software Toolkit for the QScreen Controller

**Kernel Verson 4.4**

# Using the GUI Toolkit

**INTRODUCTION** ........ 3
**The GUI Toolkit** ........ 3
**Setting Up Your Tools** ........ 3
Setting Up the Development Environment ........ 3
Installing the GUI Toolkit ........ 4
Installing the GUI Toolkit Firmware ........ 4
**Verifying the Development Environment with the Demo program** ........ 4
Forth Language ........ 4
C Language ........ 5
**Elements of a GUI Based Project** ........ 5
**HOW TO DESIGN YOUR GUI** ........ 6
**Design Tips** ........ 6
**The LCD** ........ 6
Drawing to the LCD Screen ........ 6
Screen Geometry ........ 6
**GUI OBJECTS** ........ 8
Graphics ........ 8
Graphic Objects Implementation Detail ........ 10
**Buttons** ........ 10
Button Objects Implementation Detail ........ 11
Menus ........ 13
Using Offsets ........ 14
Menu Objects Implementation Detail ........ 15
**DESIGNING YOUR GRAPHICS** ........ 17
**The Demo Graphics Library** ........ 17
**Drawing Your Own Graphics** ........ 18
**Image Conversion Program** ........ 18
Image Conversion Program Interface ........ 18
Image Conversion Program Errors ........ 19
**CODING YOUR APPLICATION** ........ 20
**For C Programmers** ........ 20
**For Forth Programmers** ........ 20
**Organization** ........ 23
**Notes on Updated Drivers** ........ 24
**Notes on Multipage Applications In C** ........ 26
**Touchscreen Calibration** ........ 26
**TIPS AND TRICKS** ........ 28
**Double Size Buttons** ........ 28
**Modal Selectors and File Tabs** ........ 28
**Custom Fonts** ........ 29
Custom Fonts in Text Mode ........ 30
**Check Boxes** ........ 31
**Pop-up Dialog Boxes** ........ 31
**SUMMARY** ........ 32
**APPENDIX 1 – MODAL BUTTON SELECTION EXAMPLE** ........ 33
**APPENDIX 2 – TEXT MODE CUSTOM FONT EXAMPLE** ........ 36
**APPENDIX 3 – LCD WORKSHEET** ........ 38

# Introduction

This document describes all of the features of the GUI Toolkit, and presents code examples plus a complete Glossary of Functions to help you get the most out of this comprehensive software package.

## The GUI Toolkit

The goal of a user interface is to provide an intuitive and straightforward way to control an instrument. A well designed graphical user interface (GUI) provides an excellent means of instrument control. Without high level tools, designing such a graphical user interface can be a formidable task, particularly for complicated instruments. The GUI Toolkit provides a set of high level functions that enable you to design the interface in terms of graphics, buttons, and menus instead of pixels and touchscreen scan codes. The QScreen Controller running the GUI Toolkit empowers you to craft an intuitive touch-sensitive front panel for your instrument.

There are three primary object types used in the GUI Toolkit:

<u>Graphics</u> are bitmapped images that may be displayed directly on the screen or used as elements of buttons or menus.

<u>Buttons</u> are objects that contain references to graphics, user handler code, and configuration flags. A button is a structure that describes how an area on the screen looks before, during, and after that area has been pressed, as well as what actions to take when it is pressed and released.

<u>Menus</u> are arrays of buttons or graphics. Buttons and graphics are placed into menus along with their desired screen locations. When a menu is drawn, all of its constituent objects are drawn at their prescribed locations on the screen. The top level function, Wait_Then_Service_Touch, scans the touchscreen and handles button presses for active menus.

Each of these objects is discussed in detail below.

## Setting Up Your Tools

### *Setting Up the Development Environment*

The QScreen Controller includes an interactive operating system that communicates serially with your PC via a terminal program. This communications link enables you to transfer programs to the QScreen, and to interactively debug the programs. We recommend that you use the terminal program which is available at our web site (www.mosaic-industries.com) and is part of our Mosaic IDE. Refer to the documentation provided with that software for installation details.

Once the Mosaic Terminal program is properly installed on your PC and the QScreen is connected and turned on, you should see an ‘ok’ prompt when you hit enter in the terminal program. Note, however, that new QScreen starter kits are shipped with a pre-installed demo program that automatically runs each time the QScreen is turned on. If the demo is running, the controller won’t respond to serial input. To establish communications, you can prevent the demo program from automatically running by performing the factory cleanup procedure as explained earlier. Briefly, to perform a factory cleanup, do the following:

1. Install a jumper on J2;

2. Press the reset button; and,

3. Remove the jumper.

Once you have installed and tested the terminal software, you are ready to transfer your program to the QScreen.

If programming in Forth, you can use your favorite editor to create programs to download to the QScreen.

To program in C, the Fabius C development software package is required. It is not included with the GUI Toolkit and must be purchased from Mosaic Industries. Follow the detailed instructions in the second chapter to install and test the C development package.

### *Installing the GUI Toolkit*

The GUI Toolkit is available as a kernel extension. Briefly, kernel extensions are modular software add-ons that provide either C or Forth with additional functions that implement drivers or toolkits such as the GUI Toolkit. The web based tool you use is called the Kernel Extension Manager (KEM). The KEM allows you to choose from an ever growing library of useful kernel extensions and generates a single set of files that allow you to install and use all of the functions of the various kernel extensions you selected. You may access the Kernel Extension Manager using the URL that is provided in the kernel extension quick guide that is supplied with your order.

### *Installing the GUI Toolkit Firmware*

Starter kits are usually shipped with a running demo program that may be cleared by performing a factory cleanup using Jumper J2 and the reset switch as explained in the section titled *Setting Up the Development Environment*. From your terminal, type

```
COLD
```

You should see the "Coldstart" and "QED-Forth V4.4x" messages. If you do not see any text, try the factory cleanup procedure again. If you are still having trouble, or if the displayed version number doesn't begin with V4.4, call Mosaic Industries at 510-790-1255.

Now use the terminal program to send the file, install.txt, included in the packages.zip file from the Kernel Extension Manager or in the Kernel_Extension directory found in \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT_PTC directory. Now you're ready to compile and run your software. Once you've installed the kernel extensions, you're ready to compile and run code of your own or any of the provided demo programs.

## Verifying the Development Environment with the Demo program

Now let's test the development environment by installing the GUI Demo program. Before installing the demo program, you need to send the images for the demo to the QScreen. The images or graphics for the demo program are in the \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT_PTC\IMAGES directory. There is also a pre-built graphics library named image_data.txt in the same directory that is the concatenation of all of the images. Download the image_data.txt file into the board using the Mosaic Terminal program, just as with the install.txt file form the kernel extension. Now, your graphics are installed in the flash. In order to speed downloads, graphics are not kept with the your software you download into the controller; thus you only have to download your graphics once.

### *Forth Language*

If you are programming in Forth, the file named library.4th from the kernel extension should be downloaded to the board before any other Forth software. The easiest way to do this is to simply #include it at the top of any software uses those functions. This file contains all of the headers, constants, etc. to allow you to use the functions in the kernel extensions. The demo program, gui_demo.4th already #includes the library.4th from the default kernel extension directory. To start the demo program after you've downloaded it, type at your terminal:

```
start
```

to begin program execution. You should see the same demo that we ship with new starter kits. Cycle the power or toggle the reset button to regain control of the unit. Since no autostart was set, the demo won't restart.

### *C Language*

For C users, start the Mosaic IDE Editor (TextPad) and load the demo C source file, gui_demo.c. Click the single page make icon or select 'Make' from the tools menu. Examine the output to verify that no errors were reported by the compiler. There will likely be a warning that an expression is always true. This is caused by a while(1) statement used in the program to implement an endless loop. If the compile was successful, then download the resulting target file, gui_demo.dlf, to the QScreen. To start the demo program, type

```
main
```

The demo should begin running. To regain control of the unit, cycle the power or toggle the reset switch. Since the autostart wasn't set, the program will not restart when the power is turned back on.

## Elements of a GUI Based Project

Before beginning your development effort, we recommend that you consider the organization of your project and choose a structure that makes your work easy to maintain and offers reasonable version control.

The heart of any application is the code itself. An application program may consist of one source file containing all the code, but it is often desirable to modularize the code into individual files that are either #included by a top level source code file or compiled separately and linked together into a target. All these code and header files may be kept in a single folder or arranged into subdirectories. If you are taking advantage of multipage C compilation, all files to be processed by the compiler must reside in the same directory. See the section titled *Notes on Multipage Applications In C* below.

# How to design your GUI

## Design Tips

A well crafted GUI can have a dramatic impact on the attractiveness of an instrument, and ultimately, its success as a product. The designer of a GUI should have a thorough understanding of what the instrument does and the typical procedures an operator will follow. Often, the structure of the internal workings of an instrument is reflected in its user interface, but this may not be optimal for the user. If two functions of an instrument operate similarly, but are used for completely different purposes, then they should not be placed together on the same menu or screen.

For example, if the purpose of an instrument is to draw in certain amount of a liquid, analyze it, then expel it, then the GUI should not necessarily have the controls for pumping in and pumping out next to each other on the screen while the analysis controls are elsewhere. Rather, if the operator will always be performing the steps in the same order (pump in, analyze, pump out), then that should be made evident in the GUI by properly grouping the sequential steps on the front panel interface. The temptation for the programmer is to group controls for pumping in the liquid and expelling it into one area while putting the controls for analysis in a different area. While this seems more logical based on the internal operation of the instrument, it is not as clear to the operator who doesn't care about internal workings.

## The LCD

### *Drawing to the LCD Screen*

The Liquid Crystal Display (LCD) device drivers provide 2 methods for writing graphics objects to the screen. Graphics may either be placed in the graphics array stored in RAM or sent directly to the LCD's RAM. The most efficient user interfaces use both of these techniques.

When graphics data is placed in the graphics array, Update_Graphics must be called to make the modified graphics array appear on the LCD. This technique is faster than drawing many different graphics and buttons individually to the screen directly. Menus and static graphics are displayed this way. This is the standard method of placing graphics on the screen.

Direct drawing to the LCD (bypassing the graphics array) is useful when it is desirable to only change a small portion of the screen. When drawing only a small number of objects covering a small part of the screen, direct drawing is faster than using the graphics array and Update_Graphics since only the affected portion is updated. Button presses commonly use direct drawing since the buttons are single objects and occupy a small part of the screen. This provides a much faster response than having to draw to the graphics array and then update the entire display. When using direct screen drawing, keep in mind that any subsequent calls to Update_Graphics will cause the entire graphics layer to be overwritten with the contents of the graphics array, thereby overwriting any data that was directly drawn to the screen.

Text mode does not have this distinction. The GUI Toolkit always writes text into the text array and uses Update_Text to send it to the display. Text requires an eighth the amount of data as graphics for a given area. For this reason, there is not an appreciable speed advantage to directly writing text to the display. However you can write text directly to the display by using Update_Here_With.

### *Screen Geometry*

The LCD on a QScreen controller has a resolution of 240 horizontal pixels by 128 vertical pixels. The analog touchscreen is divided into 20 areas of 48 pixels by 32 pixels each to remain backwarks compatible with the Panel-Touch Controller. Since the graphics memory is organized such that 6 pixels are stored in 1 byte of display memory in a horizontal raster-like fashion, there is an imposed granularity of graphics placement of 6 pixels horizontally. The vertical granularity is 1 pixel. Unless otherwise specified, when screen coordinates are mentioned, the columns are expressed in terms of these 6-pixel units.

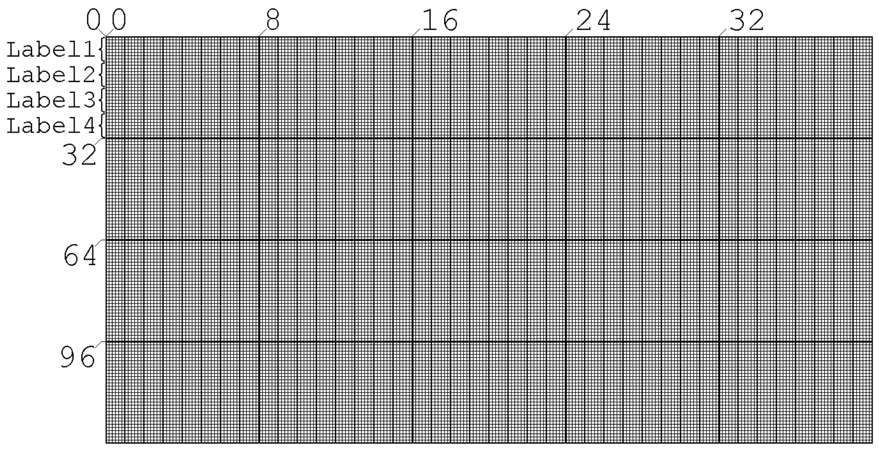


**Figure 1. LCD screen geometry**

In Figure 1, the smallest squares represent pixels, and the emphasized vertical lines show division of columns every 6 pixels. Each 48 by 32 pixel button area has room for 4 lines of text with each line no more than 8 characters long. Label1 through Label4 in the upper left part of the figure refer to the optional button labels specified in calls to FASTBUTTON, NORMBUTTON, and BLANKBUTTON. The large 5 by 4 grid represents the 20 touch sensitive areas on the touch panel.

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 9 | 13 | 17 |
| 2 | 6 | 10 | 14 | 18 |
| 3 | 7 | 11 | 15 | 19 |
| 4 | 8 | 12 | 16 | 20 |

**Figure 2. Touchscreen button numbering**

Figure 2 illustrates the touchscreen button numbering. When placing a “single size” button (that is, a button that is the same size as a single touch-sensitive area) on the screen with ADD_TOUCH_BUTTON, the button number is automatically determined from the screen coordinates. When a “double size” button is used, you must also use ADD_BUTTON for the second half of the button; this requires the specification of the button number. See the “Tips and Tricks” section below for more information about large buttons.

# GUI Objects

There are three principle types of objects in GUI Toolkit: graphics, buttons, and menus. Graphics are generally produced on a PC using any drawing package, and are converted for use with the QScreen using the graphics conversion software provided with the GUI Toolkit called the Image Conversion Program. Button objects are structures that are built using the macros FASTBUTTON, NORMBUTTON, and BLANKBUTTON. Menus are arrays of structures which contain references to graphic or button objects as well as screen locations and configuration flags. Menus are created with the macros NEW_MENU, ADD_GRAPHIC, ADD_BUTTON, ADD_TOUCH_BUTTON, and BUILD_MENU. The relationships among these object types are illustrated in Figure 3.

A GUI object has two main parts: code and data. For any given object type, there is a section of code, called a handler, that is part of the GUI Toolkit firmware. One or more data structures comprise the data part of a GUI object. To use an object, you must make a call to the handler for the object type you are using and pass it a pointer to the object data itself. Methods of creating objects vary as described below. Although some object types use multiple data structures, they are internally connected so that all objects have one method of referencing them to their handlers.

An object may be used multiple times in a program. For example, the same button object may be used in more than one menu and may have different screen locations in each use. An object's screen location is not stored in the object itself; thus an object's location is not locked down at the time of its definition. Menus contain references to objects and their relative screen locations, but the menu's own location is determined by the offsets specified when Init_Menu is called. These offsets are in turn applied to the relative locations of the objects in the menu.

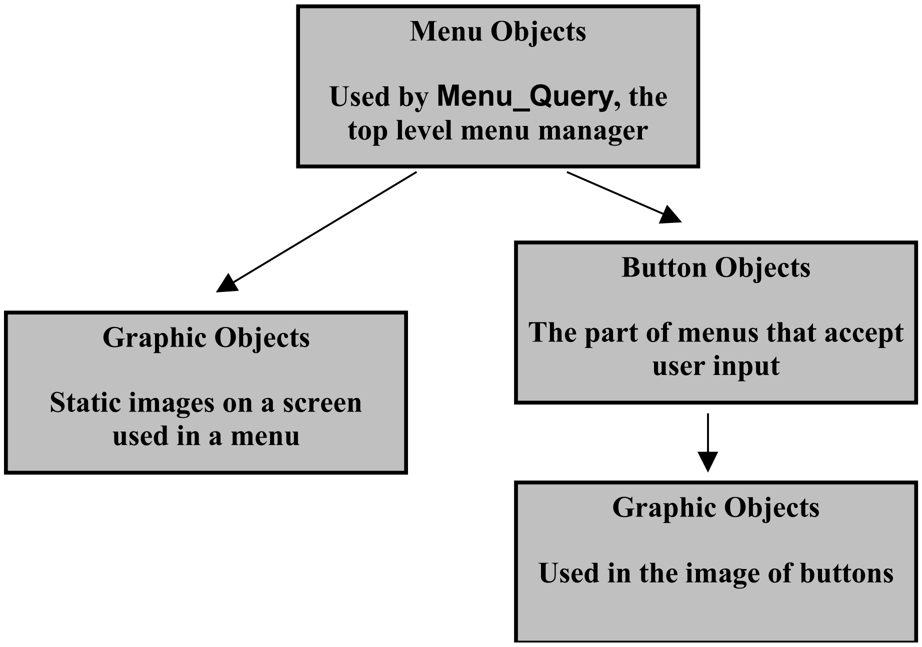

Menu Objects
Used by Menu_Query, the top level menu manager
Graphic Objects
Static images on a screen used in a menu
Button Objects
The part of menus that accept user input
Graphic Objects
Used in the image of buttons

**Figure 3. Relationships among different object types**

## *Graphics*

Graphic objects, as stated above, are created using the Image Conversion Program. This allows the integration of company logos, diagrams, icons, and large text displays. Graphics can be used: (1) as the visual part of a button; (2) by themselves as elements of menus; or (3) they may be called directly using the graphic object handler, Do_Graphic, at runtime. Refer to Listing 1 for examples of these three cases which are described below.

From image_headers.h generated automatically from the Image Conversion Program…

```
.
.
#define MY_ICON1_PCX         0x00072484
#define LOGO_PCX             0x0007252a
#define CAUTION_PCX          0x000725d0
.
.
In main source code file written by user….
.
.
#include “..\images\image_headers.h”

// Scenario 1 – use of a graphic in a button
FASTBUTTON(                  // New button
PRESS_HANDLER_FLAG,          // Execute action upon press
MY_ICON1_PCX,                // Draw graphic
MY_ICON1_PCX,                // Release graphic
SBLACK_PCX,                  // Press graphic
my_handler,                  // Function to execute when pressed
"",                          // No text for the first line
"",                          // No text for the second line
"",                          // No text for the third line
"",                          // No text for the fourth line
my_button );                 // Instantiate new button

.
.

// Scenario 2 – use of a graphic in a menu
NEW_MENU pump_ctrl[14] =
{
  .
  .
  ADD_GRAPHIC(     24,  17,  DRAW_MASK, LOGO_PCX ),
  .
  .
  ADD_TOUCH_BUTTON(  0,   38,  DRAW_MASK, my_button ),
  .
  .
};
BUILD_MENU( alnum_keypad, 14);
.
.
// Scenario 3 – calling a graphic directly
void my_function (int status)
  {
  if (status==2)
    Do_Graphic(8, 32, TVARS, DRAW_ACTION, CAUTION_PCX);
  .
  .
  }
```

**Listing 1. Graphics object example**

In scenario 1 shown in Listing 1, a button, my_button for example, might use a 48 pixel by 32 pixel graphic, ICON1_PCX, as its image. If a menu containing my_button is initialized, all objects contained in it including my_button will be initialized as well. When my_button is initialized, its draw graphic object, ICON1_PCX, will also be placed at the location specified in the menu. Refer to the shaded portion of the scenario 1 section of Listing 1.

In the second scenario, a menu containing a reference to a graphic object draws the graphic (and all other objects in the menu) at the location specified in the menu when the menu is initialized. Scenario 2 in Listing 1 has a highlighted example of the use of graphics in a menu. Many separate graphic objects can be drawn with a single function call when they are grouped into a menu.

Finally, in the third scenario, some situations such as status displays or simple animation might require the ability to display a graphic object with a direct function call at runtime. That can easily be done by simply calling the graphic object's handler, Do_Graphic, and passing it the desired screen position of the graphic, its pointer, and what action to take (draw, erase, etc.). See the highlighted section in scenario 3 in Listing 1. Details on the syntax of **Do_Graphic** are described in the *GUI Toolkit Glossary*.

### *Graphic Objects Implementation Detail*

This section describes low level implementation details, and need not be studied to use the GUI Toolkit.

Graphic objects are data structures that are formatted in the same way as the graphics array. The data is stored in a linear raster fashion beginning at the upper left corner and scanning from left to right and top to bottom. There are two parts to a graphic object, the array parameter field (pf) and the array data. The array parameter field is identical to that of a Forth array. The parameter field structure is described in the *GUI Toolkit Glossary* under **Array_pf struct**. It specifies the size, geometry, and actual memory xaddress of the array data. The array data contains no additional formatting. When drawing a graphic object, the columns of the array are copied one at a time to the graphics array, which shadows the memory of the display. Since Forth arrays are column dominant, the array column represents the display row. When Update_Graphics is called, the graphics array is sent to the LCD. Alternately, a graphic object may be drawn directly to the display. In this case, the columns of the graphic object array are sent directly to the LCD without involving the graphics array.

The Image Conversion Program, creates both parts of the graphic object as S-records in the file image_data.txt. The xaddresses assigned to the macros in the image_headers.h file are those of the Forth array parameter fields. Graphic objects may, however, be any 2 dimensional Forth array. For example, a set of advanced plotting routines could write data into a 2 dimensional variable Forth array which can then be passed to Do_Graphic or used as part of a menu or button just like a static graphic object. This would allow real time plotting of data on the front panel of an instrument.

## Buttons

Button objects provide a mechanism for binding several graphic objects together with user press/release action handlers. The behavior of a button object can be customized to provide the desired look and feel. Like graphic objects, button objects may be invoked directly via the Do_Button handler which can draw the button as pressed, released, or undraw it. Typically, however, Do_Button is not called directly; rather, the button is used as a part of a menu. The menu manager manages the menu's buttons based on the input received from the touchscreen panel.

The macros **FASTBUTTON, NORMBUTTON,** and **BLANKBUTTON** simplify the creation of buttons, and provide for automatic management of the graphical actions that occur when a button is pressed and released. See the *GUI Toolkit Glossary* for detailed descriptions of these macros. FASTBUTTON is the most common and efficient type since the resulting button will use direct drawing techniques for the press and release. This technique provides a high degree of responsiveness to the user. NORMBUTTON produces a button object that uses the graphics memory for all drawing. This is a bit slower than FASTBUTTON, but may be useful in applications where many tasks have to write to the screen at the same time. BLANKBUTTON creates a button with no default flags. It is typically used by advanced users who wish to manually specify the contents of the button structure. The example in Listing 1 demonstrates how to define a button. The glossary entries for NORMBUTTON and BLANKBUTTON also contain useful examples.

## *Button Objects Implementation Detail*

This section describes low level implementation details, and need not be studied to use the GUI Toolkit.

Button objects require only a single data structure and a pointer to that data structure. The button object structure is described in the *GUI Toolkit Glossary* under **Button Object**. The button structure of type BUTTON uses a set of bitmapped flags to control its behavior. The flags each occupy their own bit fields so that several flags can be ORed together. One 16 bit word contains all of the flags. The macros NORMBUTTON, FASTBUTTON, and BLANKBUTTON create this structure using parameters supplied to the macro. The macros set certain flags by default, but also accept additional flags as the first parameter. The default flags for these macros are defined as constants named NORMBUTTON_FLAGS and FASTBUTTON_FLAGS as described below. All button building macros in C have the flag C_STYLE_TEXT_FLAG set; this flag bit is automatically clear if the Forth programming language is being used. BLANKBUTTON has no other flags set by default. Table 1 is a list of the default flags that are set for each of the button building macros. Note that both NORMBUTTON and FASTBUTTON include flags that direct the appropriate graphical actions to occur when the button is drawn, pressed and released.

| Constant | Flags |
|---|---|
| FASTBUTTON_FLAGS | DRAW_GRAPHIC_FLAG<br>RELEASE_GRAPHIC_FLAG<br>DIR_RELEASE_GRAPHIC_FLAG<br>PRESS_GRAPHIC_FLAG<br>DIR_PRESS_GRAPHIC_FLAG<br>C_STYLE_TEXT_FLAG |
| NORMBUTTON_FLAGS | DRAW_GRAPHIC_FLAG<br>RELEASE_GRAPHIC_FLAG<br>PRESS_GRAPHIC_FLAG<br>C_STYLE_TEXT_FLAG |
| BLANKBUTTON<br>(hard codes the flag) | C_STYLE_TEXT_FLAG |

**Table 1. Default button flags for different button types**

If the FASTBUTTON macro used as follows…

```
FASTBUTTON(                  // New button
PRESS_HANDLER_FLAG,          // Execute action upon press
MY_ICON1_PCX,                // Draw graphic
MY_ICON1_PCX,                // Release graphic
SBLACK_PCX,                  // Press graphic
my_handler,                  // Function to execute when pressed
```

```
    "",                         // No text for the first line
    "",                         // No text for the second line
    "",                         // No text for the third line
    "",                         // No text for the fourth line
    my_button );                // Name of new button
```

…then the code generated will look like this:

```
    BUTTON my_button_struct_=
    {(0x0080) | ( 0x0001 |0x0002 |0x0010 |0x0004 |0x0020 |0x4000 ),
    ((xaddr) ((((0x07))<<16)+ (0xFFFF & ((0x077E))))),
    0x00072484,
    0x00072484,
    0x0004642a,
    my_handler,
    my_handler,
    (""),
    (""),
    (""),
    ("") };
    BUTTON * my_button=&my_button_struct_;
```

**Listing 2. Button building macro expansion**

Listing 2 shows how the FASTBUTTON macro is expanded by the compiler. In the second part of Listing 2, line 1 defines a new struct of type BUTTON which is initialized in the subsequent lines.

Line 2 contains the flags that are set. The first number, 0x0080 (where the 0x prefix indicates that it is a hexadecimal number), is the flag for PRESS_HANDLER_FLAG which was supplied in the flags parameter in the call to FASTBUTTON. This flag directs that the user-supplied action specified by the my_handler function is to be executed when the button is pressed. This action depends on the application; it may involve turning a valve on or off, displaying a value, etc. The remaining flags on that line are the defaults that are associated with FASTBUTTON.

Line 3 specifies the xaddress (extended 32bit address) of the internal GUI Toolkit firmware function that handles the graphic objects that follow. This value is automatically initialized by all of the button building macros. The code on line 3 is generated by means of an intermediate statement (transparent to the user) of the form:

```
    TO_XADDR(DO_GRAPHIC_ADDR,DO_GRAPHIC_PAGE)
```

This statement returns the code xaddress of the graphic object handler.

Lines 4, 5, and 6 specify the xaddresses of the draw, release, and press graphic objects respectively. The xaddresses are defined in image_headers.h which is generated by the Image Conversion Program.

Lines 7 and 8 are the respective xaddresses of the programmer-defined press and release handlers. The linker replaces the function name (my_handler in this example) with the function's 32 bit numerical xaddress at link time. Note that in the FASTBUTTON and NORMBUTTON macros, only one handler function is passed. It is duplicated in the press and release handler fields. Because most buttons don't have distinct user-specified actions for press and release, placing the handler xaddress in both fields allows the flags to determine whether the action will be executed upon press or upon release. (It would be possible, but not very useful, to specify both PRESS_HANDLER_FLAG and RELEASE_HANDLER_FLAG; the resulting button would execute the user-specified handler code when it is pressed and again when it is released.) If separate actions for press and release are required, BLANKBUTTON should be used to build the button. Note that a system crash will occur if a flag is set for press or release action when there is not a valid handler xaddress stored in that field of the button structure.

Lines 9-12 contain string xaddresses to be used as labels placed inside the button. These string pointers can point to constant (ROM- or flash-resident) or variable (RAM-resident) strings. For an example of RAM-resident dynamic button labeling, see the alphanumerical keypad section of the gui_demo.c file in the \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT_PTC\C directory of the GUI distribution.

GUI objects are defined under the C compiler's large memory model which treats memory addresses as full 32-bit xaddresses. While the C compiler can process macros in the large memory model, code must be compiled in the small or medium memory model which only supports 16 bit addressing. The full xaddress of each object must be extracted while still under the large model in order to access it in the runtime code after the model has been switched back to the small or medium model. For this reason, line 13 defines and initializes a pointer to the structure. The 32-bit contents of the pointer can be accessed from any memory model. This pointer has the name that is specified when calling the button building macro, while the actual structure name (which is not typically directly referenced by the programmer) includes the _struct_ suffix.

## *Menus*

Menus are the high level building blocks of a graphical user interface. You can think of a menu as a "container" for button and graphic objects. It also relates each constituent object to a relative screen location within the menu. A menu object is built with the macros NEW_MENU and BUILD_MENU. The macros ADD_BUTTON, ADD_TOUCH_BUTTON, and ADD_GRAPHIC insert objects into a menu. Menus can contain graphics and buttons, but not other menu objects. Each of these macros is demonstrated in the example in Listing 3.

Init_Menu and Uninit_Menu are the most common functions used with menus. These functions cause all objects inside the menu to have their handlers called with the directive to draw or erase respectively. Functions called by Init_Menu and Uninit_Menu also post and delete information about touchscreen buttons to an array called the keymap array. This allows the touchscreen menu manager, Wait_Then_Service_Touch, to handle touchscreen input and call the appropriate buttons when they are pressed. Multiple menus may be active at the same time; however, note that overlapping buttons or graphics may cause undesired operation. One exception to this is when background images are desired. For example, a large background image may be placed on the screen with buttons overlaying it by simply specifying the image in the menu prior to the buttons as objects are drawn in the order they are specified. Any graphic objects in the menu (those added with ADD_GRAPHIC) do not affect the keymap array because static graphics do not respond to touchscreen input.

```
#include "..\images\image_headers.h"
.
.
.
NEW_MENU mmain[12] =
{            // add buttons to the menu by initializing array of structs
// first and second columns
  ADD_GRAPHIC(       01,  33,  DRAW_MASK, PUMP_PCX ),      // The pump diagram
  ADD_BUTTON(   4,    0,   0,          0,   sht_dn_button ), // ...is Doublewidth
  ADD_TOUCH_BUTTON(   0,   0,  DRAW_MASK, sht_dn_button ), // Shutdown button..
// Second Column
  ADD_TOUCH_BUTTON(   8,  66,  DRAW_MASK, s_flow_button ), // Set flow button
// third column
  ADD_TOUCH_BUTTON(  16,  12,  DRAW_MASK, config_button ), // Config button
  ADD_TOUCH_BUTTON(  16,  66,  DRAW_MASK, s_title_button), // Set Title button
// forth column
```

```
  ADD_TOUCH_BUTTON(  24,  12,  DRAW_MASK, stats_button  ), // Stats button
  ADD_TOUCH_BUTTON(  24,  66,  DRAW_MASK, s_power_button), // Set Power button
// fifth column
  ADD_TOUCH_BUTTON(  32, 12,   DRAW_MASK, exit_button    ), // Exit button
  ADD_TOUCH_BUTTON(  32, 72,   DRAW_MASK, increase_button), // Increase button
  ADD_GRAPHIC(       32, 90,   DRAW_MASK, PWRLABEL_PCX   ), // Label for inc/dec
  ADD_TOUCH_BUTTON(  32,104,   DRAW_MASK, decrease_button)  // Decrease button
};
BUILD_MENU( mmain, 12);
```

**Listing 3. Menu building example**

Wait_Then_Service_Touch is the top level function that manages the touchscreen and handles button presses. It is typically called within a main runtime loop. Wait_Then_Service_Touch repeatedly scans the touchscreen for any user input. If an area of the touchscreen corresponding to any of the buttons currently on the screen is being touched by the user, Wait_Then_Service_Touch takes the appropriate action as defined by the button. If no button is associated with the part of the touchscreen being pressed, the press is ignored. Wait_Then_Service_Touch exits after each press/release cycle. Since Wait_Then_Service_Touch itself executes the user handlers associated with the buttons, they will run in the same task environment as the call to Wait_Then_Service_Touch. This is particularly relevant for multitasking applications.

## *Using Offsets*

Button offsets allow a menu to be placed in different places on the screen while still maintaining the relative grouping of objects in the menu. The objects in a particular menu may be optionally shifted to a different portion of the screen from the relative column and row positions and the relative button locations specified in the menu definition. The relative column and row positions are passed as parameters when an object is added to a menu. The ADD_BUTTON macro additionally requires the relative button location. The column, row, and button offset parameters passed to Init_Menu and Menu_Install are added to the relative positional information for each object in the menu. If the offsets are zero, the positional information in the menu will be absolute.

A call to Init_Menu for the menu in Listing 4 causes the menu to be drawn in the upper left corner of the screen. Nonzero offsets may be passed to Init_Menu to cause the menu to be drawn elsewhere on the screen. Suppose that Init_Menu is called with column, row, and button offsets of **8**, **32**, and **5** respectively. The entire menu would be shifted to the right by **8** columns (48 pixels) and down by **32** pixels. Refer to Figure 1 and Figure 2 and notice that offsetting the button location by **5** causes the touch sensitive area of the button to shift down and to the right by one button area. Recall that buttons are 8 columns wide and 32 pixels tall. The value of the button offset must be chosen to be congruent with the values used for the column and row offsets. When Init_Menu, called with the above offset parameters for the menu in Listing 4, places this menu on the screen, the TITLEBAR_PCX graphic will be placed at the screen location 8, 32. Likewise, the continue_button will be placed at 8, 41 in button location 5 and the cancel_button will be placed at 16, 41 in button location 9.

```
NEW_MENU confirm[3] =
{            // add buttons to the menu by initializing array of structs
  ADD_GRAPHIC(        0,   0,  DRAW_MASK, TITLEBAR_PCX  ), // The pump diagram
  ADD_TOUCH_BUTTON(   0,   9,  DRAW_MASK, continue_button ),// Continue button
  ADD_TOUCH_BUTTON(   8,   9,  DRAW_MASK, cancel_button ), // Set flow button
};
BUILD_MENU( confirm, 3);
```

**Listing 4. Small menu offset example**

For menus that are only to be used in one place on the screen, there is no reason to use offsets. Offsets are more useful for small dialog boxes such as "Are you sure? yes no" that may be used in different contexts that require it to appear in various places on the screen. Since the menu in Listing 3 is a full screen menu, using offsets would cause part of the menu to exceed the boundaries of the screen and thus produce and error.

## *Menu Objects Implementation Detail*

This section describes low level implementation details, and need not be studied to use the GUI Toolkit.

A menu is implemented as a 1-dimensional Forth array of structures of type MENU_ENTRY. This structure contains a substructure called KEYMAP_ENTRY. The Forth array comprises a data portion in the heap area, and an array parameter field (pf) that describes the geometry of the array and contains a "handle" to the heap-resident data. The xaddress of the Forth array pf (xpfa) is passed to the functions that use menus to allow the data in the array to be manipulated. The KEYMAP_ENTRY and MENU_ENTRY structure types are described in Table 2 and Table 3.

| Offset | Type | Element name | Description |
|---|---|---|---|
| 0x0000 | uint | row | Relative row position for the button graphics |
| 0x0002 | uint | col | Relative column position for the button graphics |
| 0x0004 | xaddr | object | Xaddress of the object structure |
| 0x0008 | xaddr | obj_handler | Xaddress of the object handler |

**Table 2. KEYMAP_ENTRY structure**

| Offset | Type | Element name | Description |
|---|---|---|---|
| 0x0000 | KEYMAP_ENTRY | keymap_entry | A substructure of type KEYMAP_ENTRY that is copied into the keymap array when the menu is installed. |
| 0x000C | uint | action_mask | Bitmask used to control what action flags may be passed through to the object for Do_Menu. Any action flag passed to do menu is ANDed with the action_mask before being passed to the underlying object handler. |
| 0x000E | uint | button | The relative keymap index of a button object. |

**Table 3. MENU_ENTRY structure**

A related data structure is called the "keymap array", or simply "keymap", which is a 1-dimensional RAM-resident Forth array of KEYMAP_ENTRY structures. The keymap substructure is copied into the keymap array when the menu is installed. This keymap is dynamically loaded with keymap entries to be accessed at runtime by the touchscreen menu manager, Wait_Then_Service_Touch. When an area of the touchscreen is pressed, the touchscreen hardware driver returns a button number is which is used to index the keymap array and, in turn, call the appropriate object handler to respond to the press.

When a menu is to be placed on the screen during an application's runtime, two things must happen. First, the menu must be drawn out onto the screen. Second, the touchscreen menu manager must be made aware of the buttons that occupy the touch areas on the screen.

Do_Menu performs the screen draw. When Do_Menu is called, it must be passed a column and row offset as explained below, an action flag, and the menu object's pointer. Do_Menu sequentially parses the array starting at element 0. For each element in the array, the object handler, specified in the menu, is called to process the object specified in that element of the menu. The column and row are offset by the offset values specified in the call to Do_Menu (which may be zero). The action flag passed to the object handler is the result of ANDing the menu's action mask (specified when the menu was defined) with the action flag specified in the call to Do_Menu.

Menu_Install copies the keymap portion of the menu entries into the keymap array. It uses the passed button offset to adjust the placement in the keymap array as described below. Once the elements of the menu array have been copied into the keymap array, button presses detected by the menu manager will result in the appropriate action. When a menu is deactivated, it is deleted from the keymap array by the function Menu_Remove.

Do_Menu, Menu_Install, and Menu_Remove are called by the higher level functions Init_Menu and Uninit_Menu. Init_Menu calls Do_Menu with the DRAW_ACTION parameter, and then calls Menu_Install to place the menu entries in the keymap array. Uninit_Menu calls Do_Menu with the ERASE_ACTION parameter, and then calls Menu_Remove to remove the menu entries from the keymap array.

# Designing Your Graphics

## The Demo Graphics Library

Figure 4 shows the graphic library that is used in the demo as well as several additional images. Each button image should have a “pressed” version for visual feedback when the operator touches the button on the touchscreen. This library includes solid filled versions of each button size to be used as the pressed graphics for the buttons. You may freely use these pre-designed graphics in your application.

A simple generic user interface can also be built using buttons that are rectangular empty boxes, together with solid filled versions of the buttons to indicate the image while the button is pressed. Several sizes of blank buttons and their corresponding filled versions are available for your use in the pre-designed graphics library shown in Figure 4. The advantage of this “text in blank button” technique is that it saves memory space because the same graphics can be used to implement many different buttons.

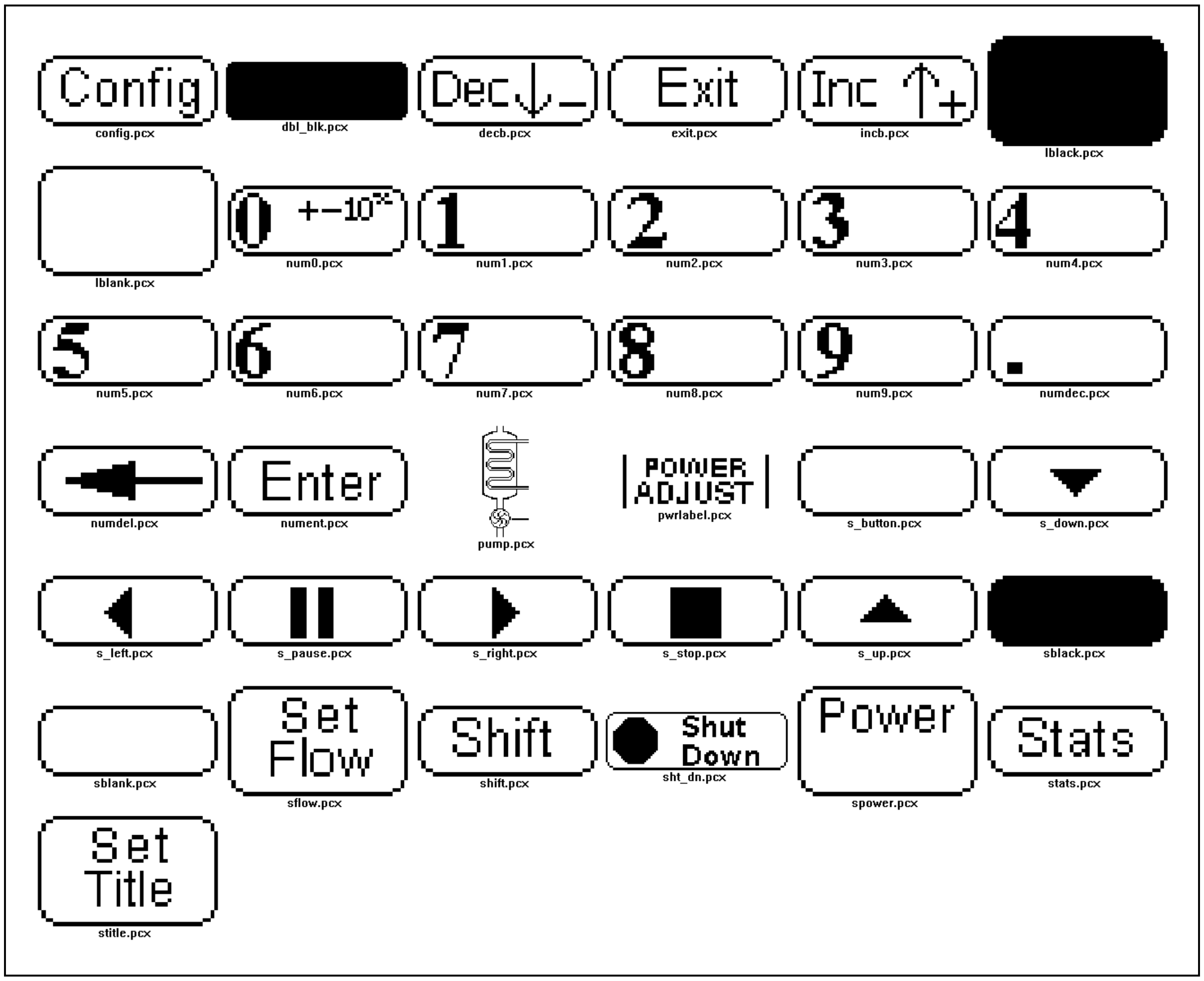


**Figure 4. Default graphics library used with the demo (not shown to the same scale)**

## Drawing Your Own Graphics

Custom graphics can endow a user interface with a unique and professional look. A wide variety of graphical editing software is available for nearly all platforms. Any graphical editing package may be used as long as it can produce a 1 bit color depth (line art) monochromatic image in the pcx or bmp format. Color and grayscale images will be rejected by the Image Conversion Program.

Programs that are suitable for custom graphics design include Adobe Photoshop, Paint, and ZSoft's PC Paintbrush. All of the graphics in the demo graphics library are provided in pcx form inside the \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT_PTC\IMAGES directory of the distribution, and they provide useful examples. If your graphics editing package supports it, a visible drawing grid may be helpful when drawing smaller graphics.

To create a custom button, it is often helpful to start with one of the blank buttons in the graphics library and draw custom text or icons inside the rectangle. Save the new image with a different filename and note which image contains a solid version of the rectangle to be used as the pressed graphic when the button is defined. There is no limitation on the size or shape of button graphics other than the 6 pixel horizontal granularity that affects the size and placement of all graphics.

Some user interfaces may require a button that contains or is part of a diagram with oddly shaped symbols to be touched by a user. In such a case, it might be advantageous to design customized "pressed graphics" (for example, an inverted image of the symbol in the button) for each button instead of a generic solid rectangle. This can result in a more elegant and aesthetically pleasing user interface.

## Image Conversion Program

Mosaic's Image Conversion Program allows you to easily transfer images created on a PC to your QScreen Controller for use with the GUI Toolkit. The image conversion program takes all of the images in a selected directory and concatenates them into a single text file called an Image Data File. The Image Data File is stored in the selected directory and is sent to the Controller using the Mosaic Terminal, the terminal program provided by Mosaic Industries. The Image Data File, by default, puts the converted image data into RAM on page 0x01 and then transfers the data into Flash at page 0x10. For more information about the memory map of the QScreen Controller, see chapter 6.

Another file, called an Image Header File, is created to associate the names of the images (which is just the filename of the image) with their location in memory on the QScreen. The Image Header File contains constants named after the file names of the images and must be included in your program to tell the GUI Toolkit where the images are stored. Supported image formats are mono-chrome bitmap (*.bmp) and PCX (*.pcx) files with 1 pixel bit depth. The following sections describe the user interface and the error messages returned by the Image Conversion Program.

### *Image Conversion Program Interface*

The user interface for Mosaic's Image Converions Program has a main screen, an advanced options screen, and a help screen. On the main screen there are controls that select the directory that contain your image files, specify the type of image file (PCX or BMP) to convert, and select the product that you're using. A button labeled "Convert Files Now" starts the conversion process. In the advanced options screen there are controls that select your programming language, desired target memory location for the image data, and filenames for the Image Data and Image Header files. All of the advanced options are set to default values that will work for most applications. The help screen provides additional information for each of the options and controls.

## *Image Conversion Program Errors*

Mosaic's Image Conversion Program detects and reports the following error conditions:

> "Error changing to the specified directory". The directory does not exist, it was moved, or deleted. A new directory has to be specified.
>
> "Error opening an image file, Image Data File, or Image Header File".
>
> "Not a valid bitmap file (no valid file identifier) or pcx file (no valid manufacturer or encoding)".
>
> "Bit depth of the image file does not match the specified value".
>
> "Image width does not fall on an 6-pixel boundary". All images must have a width that is a multiple of six pixels. This is required to quickly draw the images onto screens. Please crop or stretch the image using any photo or image editing program such as Photoshop or Paint.

The next section explains how to incorporate the output files of the Image Conversion Program when developing an application in either C or Forth.

# Coding Your Application

## For C Programmers

The C development software must be given some special directives before and after defining the GUI objects. Although the syntaxes are the same for all GUI routines in C and Forth, the following statements must be used in a C program source file before GUI objects are defined:

```
#pragma option init=.doubleword
#include <mosaic\gui_tk\to_large.h>
```

The first statement places the initialized variables into the area known to the C linker as .doubleword, which is located in the FLASH program area instead of RAM. The second statement is a header file that contains preprocessor directives to instruct the compiler to operate in a 32 bit addressing mode. This accommodates full 24 bit addressing of the GUI objects, a necessary requirement of the GUI toolkit routines. However, no code may be compiled under the 32 bit addressing mode. Thus, at the end of the GUI object definitions, the following two statements must be present to reset the compiler to its original state:

```
#pragma option init=.init
#include <mosaic\gui_tk\fr_large.h>
```

Only GUI object definitions should be placed between these two sets of preprocessor directives; no code (function definitions, calls, etc.) are allowed. The statement,

```
#include "..\images\image_headers.h"
```

may appear outside of the region bounded by these preprocessor directives.

The graphics file, image_data.txt, which is generated from the Image Conversion Program, only needs to be downloaded to the board after the graphics have been changed. It you have trouble downloading it, type

```
cold
```

and try to download it again, then download the .dlf file of your application generated from the C compiler.

C programmers may skip ahead to the *Organization* section which describes how the preprocessor directives, object definitions and application code are combined to create a source code file.

## For Forth Programmers

**Figure 5** shows a recommended code downloading sequence for Forth programmers. The first file to be loaded is install.txt, the kernel extension install file. This file contains the pre-compiled GUI Toolkit firmware. See → *Setting Up Your Tools* → *Installing the GUI Toolkit Firmware* for more information on installing the firmware. As described in that section, install.txt may already be present on your QScreen Controller as part of the pre-loaded demo program on starter kits.

Next, the image_data.txt file should be sent to the QScreen Controller. It contains graphics data that is loaded temporarily onto page 1, and from there is transferred into a predetermined memory location (typically in flash on page 10) as determined by the Image Conversion Program.

The library.4th file contains the Forth headers associated with the GUI and other kernel extension firmware, and it also sets up the memory map for general applications. By default, the definitions pointer (DP) is set to 0000\4 (address 0000 on page 4), the names pointer (NP) is set to 0000\6, and the variable pointer (VP) is set to 8E00\0 (in common RAM). Although this is acceptable for most applications, you may simply edit the file Library.4th if you prefer a different memory map. If there is a `cold` restart (say, because the QScreen crashes during program development), the memory map pointers will be lost. To recover and continue programming, you may either type `RESTORE` (consult the Forth glossary for its definition), or you may reload library.4th and all subsequent files.

The next file to load is image_headers.h which contains a list of xconstants specifying the locations of the graphic objects. Because these constants occupy space in the definitions area and the names area, they must be loaded after the memory map has been set up by library.4th. Of course, your can #include these files from your own code rather than having to manually download each one of them.

The final file (or set of files) comprises your application code. **Figure 6** shows the elements of a typical source code file. It is good practice to execute `DOWNLOAD.MAP` at the start of the file to ensure that pages 4 through 6 can accept the download into RAM. You need not set up a memory map, as that was done by library.4th. That is, you do not need to execute `4 USE.PAGE` or equivalent statements. To simplify the reloading of software during development, each of your application code files should start with an "ANEW" statement as explained in the software manual. At the bottom of your final source code file, the PAGE.TO.FLASH statements shown in **Figure 6** transfer the contents of pages 4 through 6 to flash, and STANDARD.MAP restores the standard memory map.

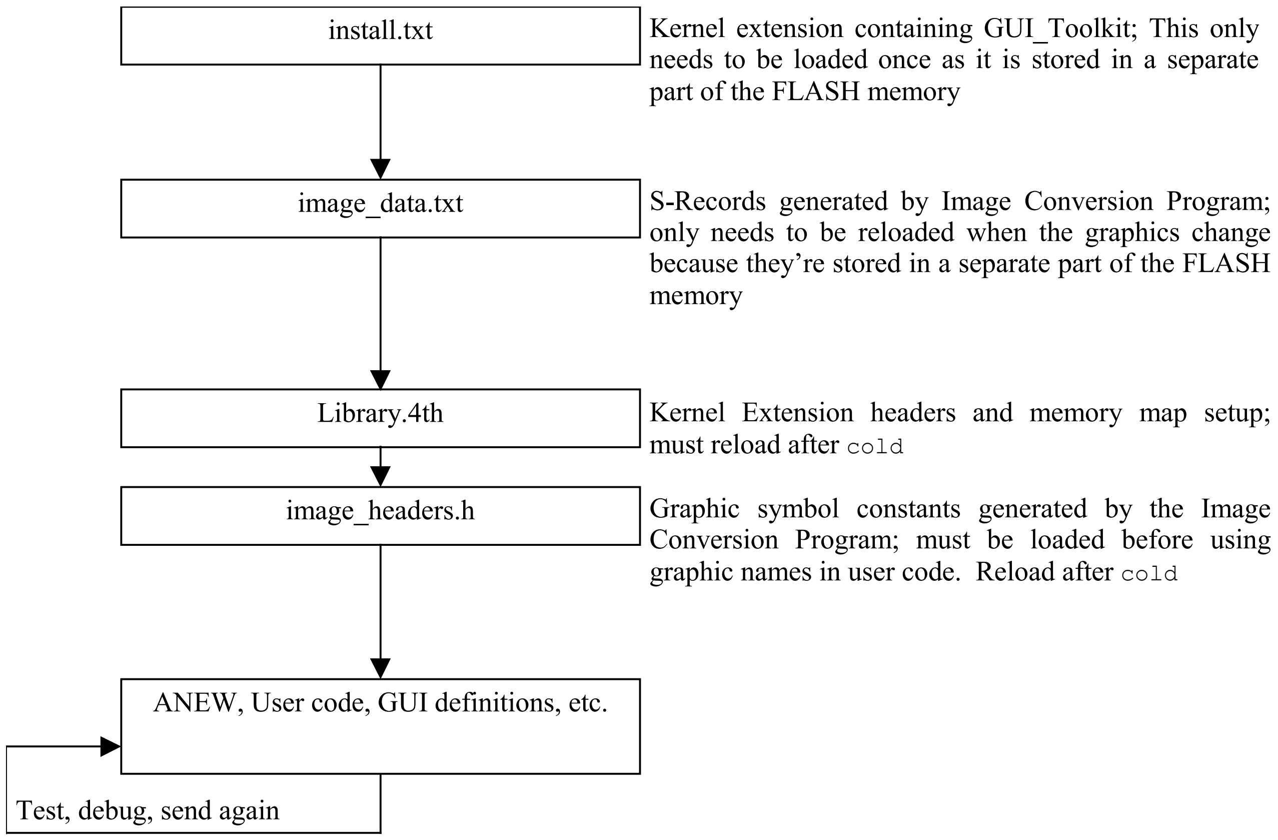
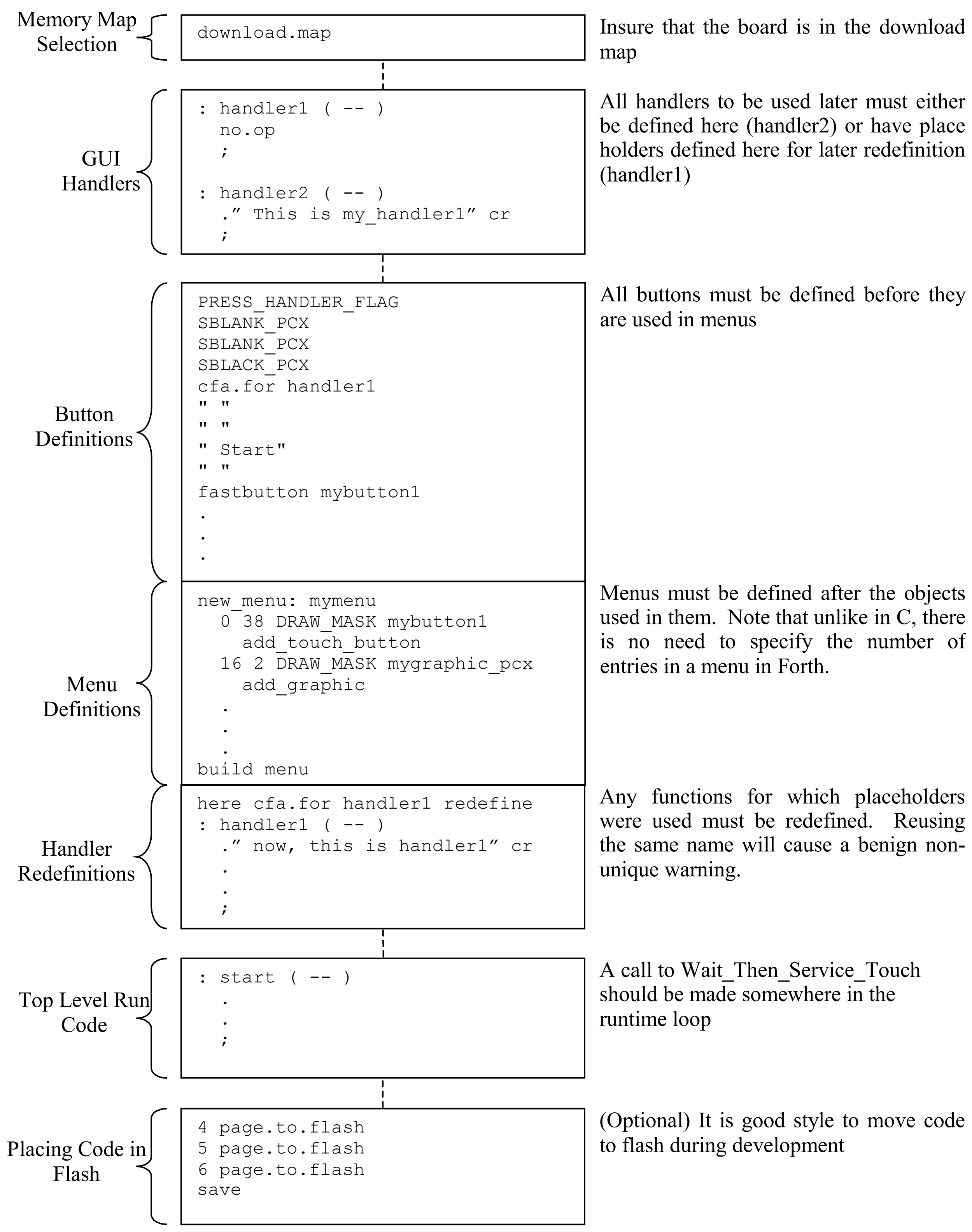

install.txt
Kernel extension containing GUI_Toolkit; This only needs to be loaded once as it is stored in a separate part of the FLASH memory
image_data.txt
S-Records generated by Image Conversion Program; only needs to be reloaded when the graphics change because they're stored in a separate part of the FLASH memory
Library.4th
Kernel Extension headers and memory map setup; must reload after cold
image_headers.h
Graphic symbol constants generated by the Image Conversion Program; must be loaded before using graphic names in user code. Reload after cold
ANEW, User code, GUI definitions, etc.
Test, debug, send again

**Figure 5. Forth Download Sequence**



```
download.map
```

Insure that the board is in the download map

```
: handler1 ( -- )
  no.op
  ;

: handler2 ( -- )
  .” This is my_handler1” cr
  ;
```

All handlers to be used later must either be defined here (handler2) or have place holders defined here for later redefinition (handler1)

```
PRESS_HANDLER_FLAG
SBLANK_PCX
SBLANK_PCX
SBLACK_PCX
cfa.for handler1
" "
" "
" Start"
" "
fastbutton mybutton1
.
.
.
```

All buttons must be defined before they are used in menus

```
new_menu: mymenu
  0 38 DRAW_MASK mybutton1
    add_touch_button
  16 2 DRAW_MASK mygraphic_pcx
    add_graphic
  .
  .
  .
build menu
```

Menus must be defined after the objects used in them. Note that unlike in C, there is no need to specify the number of entries in a menu in Forth.

```
here cfa.for handler1 redefine
: handler1 ( -- )
  .” now, this is handler1” cr
  .
  .
  ;
```

Any functions for which placeholders were used must be redefined. Reusing the same name will cause a benign non-unique warning.

```
: start ( -- )
  .
  .
  ;
```

A call to Wait_Then_Service_Touch should be made somewhere in the runtime loop

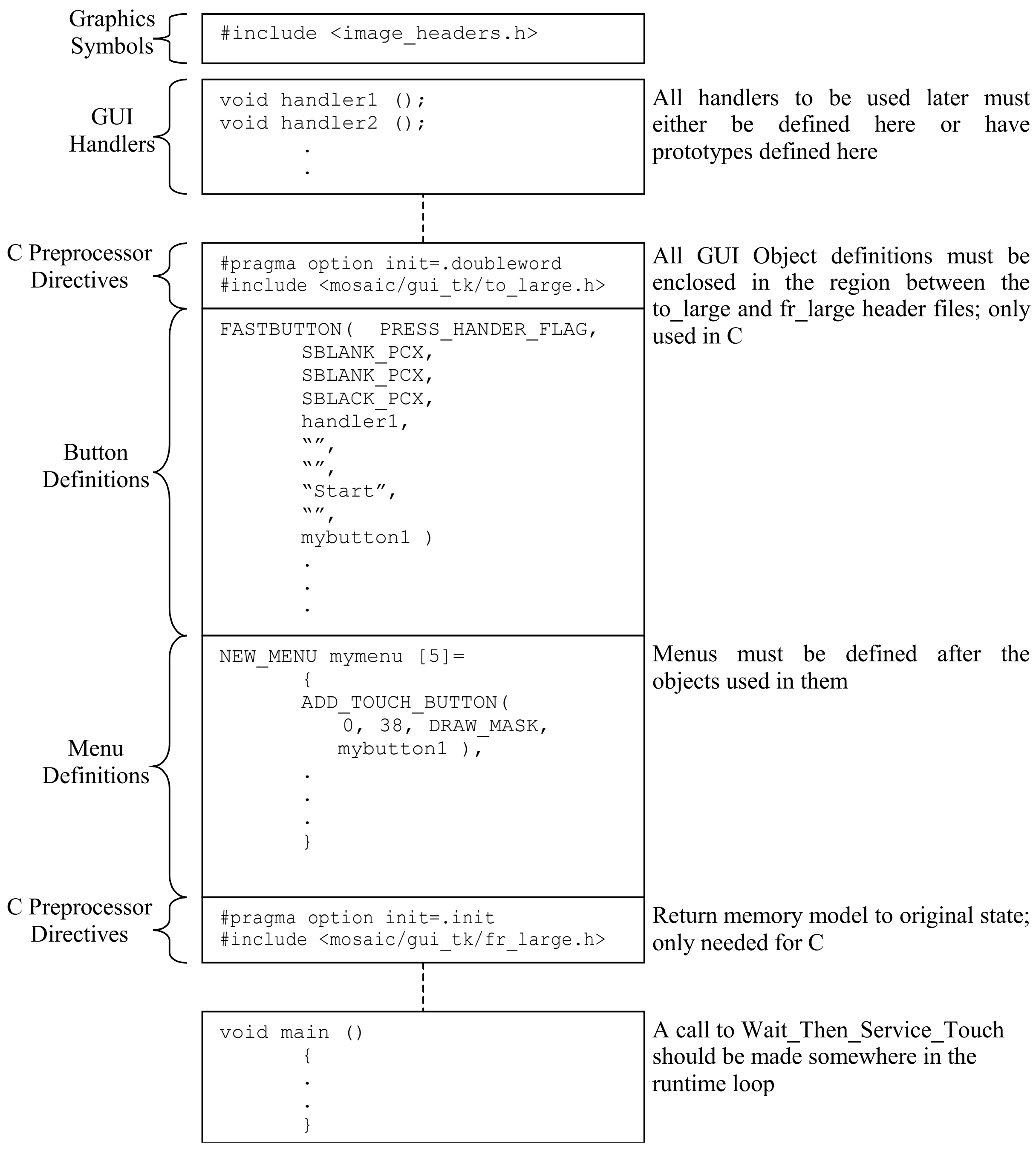
```
4 page.to.flash
5 page.to.flash
6 page.to.flash
save
```

(Optional) It is good style to move code to flash during development

**Figure 6. Forth program construction**

## Organization

In a typical GUI based application, there are a number of menus consisting of buttons and graphics which must be organized in a certain order. The image header file must be included before using any of the graphics objects. Menus must be defined after the buttons that comprise them. Figure 7 shows the hierarchy of a simple application in which all the buttons are defined at once, and then one or more menus are defined. Although shown in C, this structure is applicable to Forth with the exception of the C preprocessor directives which are not necessary in Forth.



```
#include <image_headers.h>
```

```
void handler1 ();
void handler2 ();
       .
       .
```

All handlers to be used later must either be defined here or have prototypes defined here

```
#pragma option init=.doubleword
#include <mosaic/gui_tk/to_large.h>
```

All GUI Object definitions must be enclosed in the region between the to_large and fr_large header files; only used in C

```
FASTBUTTON(  PRESS_HANDER_FLAG,
       SBLANK_PCX,
       SBLANK_PCX,
       SBLACK_PCX,
       handler1,
       “”,
       “”,
       “Start”,
       “”,
       mybutton1 )
       .
       .
       .
```

```
NEW_MENU mymenu [5]=
       {
       ADD_TOUCH_BUTTON(
          0, 38, DRAW_MASK,
          mybutton1 ),
       .
       .
       .
       }
```

Menus must be defined after the objects used in them

```
#pragma option init=.init
#include <mosaic/gui_tk/fr_large.h>
```

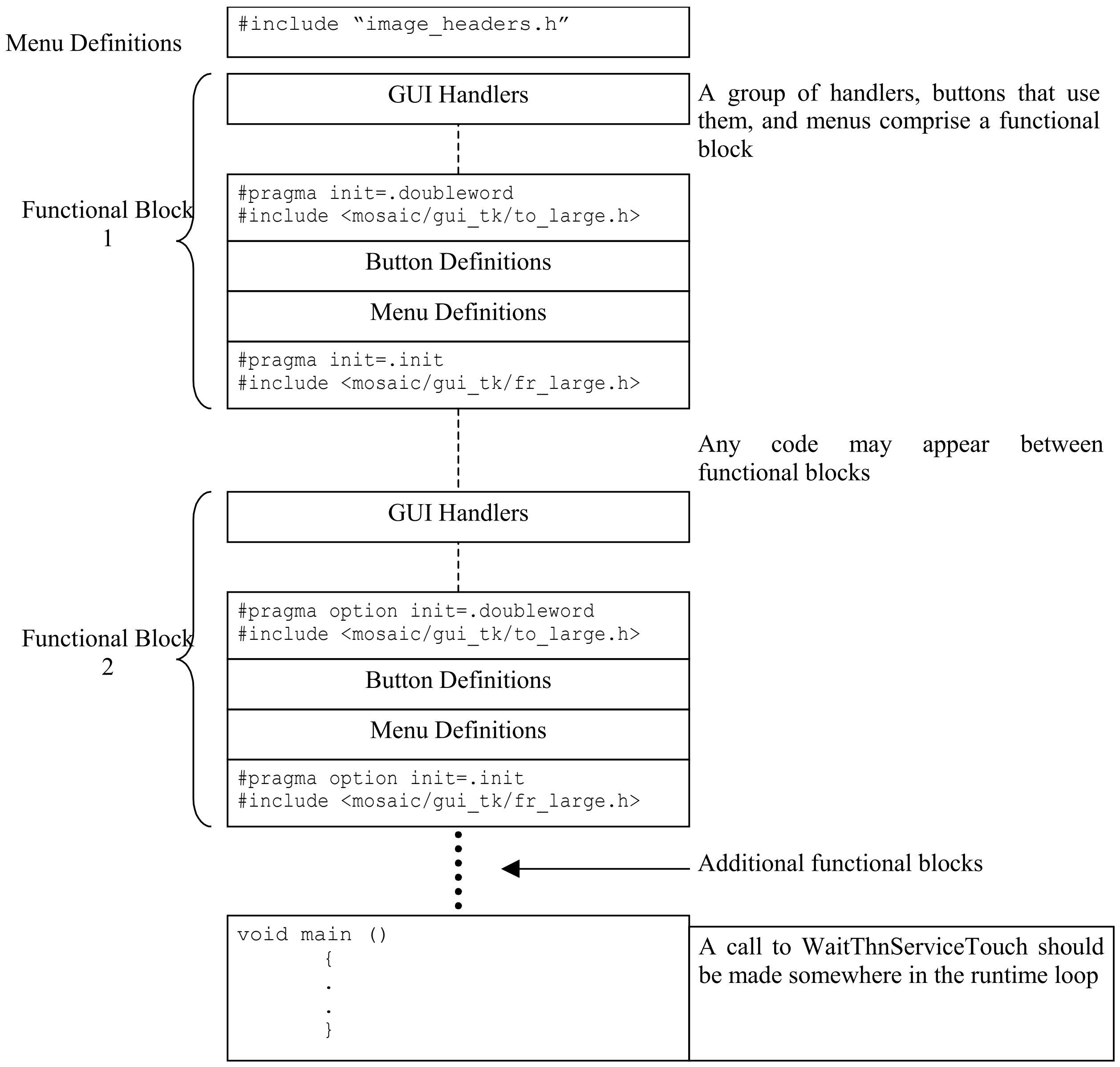
Return memory model to original state; only needed for C

```
void main ()
       {
       .
       .
       }
```

A call to Wait_Then_Service_Touch should be made somewhere in the runtime loop

**Figure 7. C simple program construction**

Most applications require many menus making this structure difficult to work with. To solve this problem, the construction in Figure 7 can be repeated for the different menus and/or functional blocks in a program. Figure 8 shows the grouping of different functional blocks in a program. It is also possible to place the blocks in separate files to be #included in the main program. Be sure to enclose all button definitions and menu definitions between the C preprocessor directives as explained earlier. Also, specifying the #include files, to_large.h or fr_large.h, more than once in a program will not cause an error because they only contain preprocessor directives instead of symbol definitions.



Menu Definitions
#include "image_headers.h"
Functional Block 1
GUI Handlers
A group of handlers, buttons that use them, and menus comprise a functional block
#pragma init=.doubleword
#include <mosaic/gui_tk/to_large.h>
Button Definitions
Menu Definitions
#pragma init=.init
#include <mosaic/gui_tk/fr_large.h>
Any code may appear between functional blocks
Functional Block 2
GUI Handlers
#pragma option init=.doubleword
#include <mosaic/gui_tk/to_large.h>
Button Definitions
Menu Definitions
#pragma option init=.init
#include <mosaic/gui_tk/fr_large.h>
Additional functional blocks
void main ()
{
.
.
}
A call to WaitThnServiceTouch should be made somewhere in the runtime loop

**Figure 8. C complex program construction**

## Notes on Updated Drivers

The QScreen's kernel contains older device drivers for the LCD and touchscreen which are superceded by the GUI Toolkit's drivers. See Table 4 for a summary of the changes to the device drivers. Only the C versions of the old function names are shown. Do not use the old functions that are shaded in any application that uses the GUI Toolkit.

Although the high level functions in the GUI Toolkit should provide all the necessary flexibility to most users, some programmers may wish to use the hardware through low level calls. The GUI Toolkit primitive function, Update_Here_With, offers the lowest level of control that works robustly in all multitasking environments. It allows an 2 dimensional array of any size to be sent to an arbitrary location in the LCD's internal memory space. Calls below that level must manage the display resource variables manually if they are to be multitasking friendly.

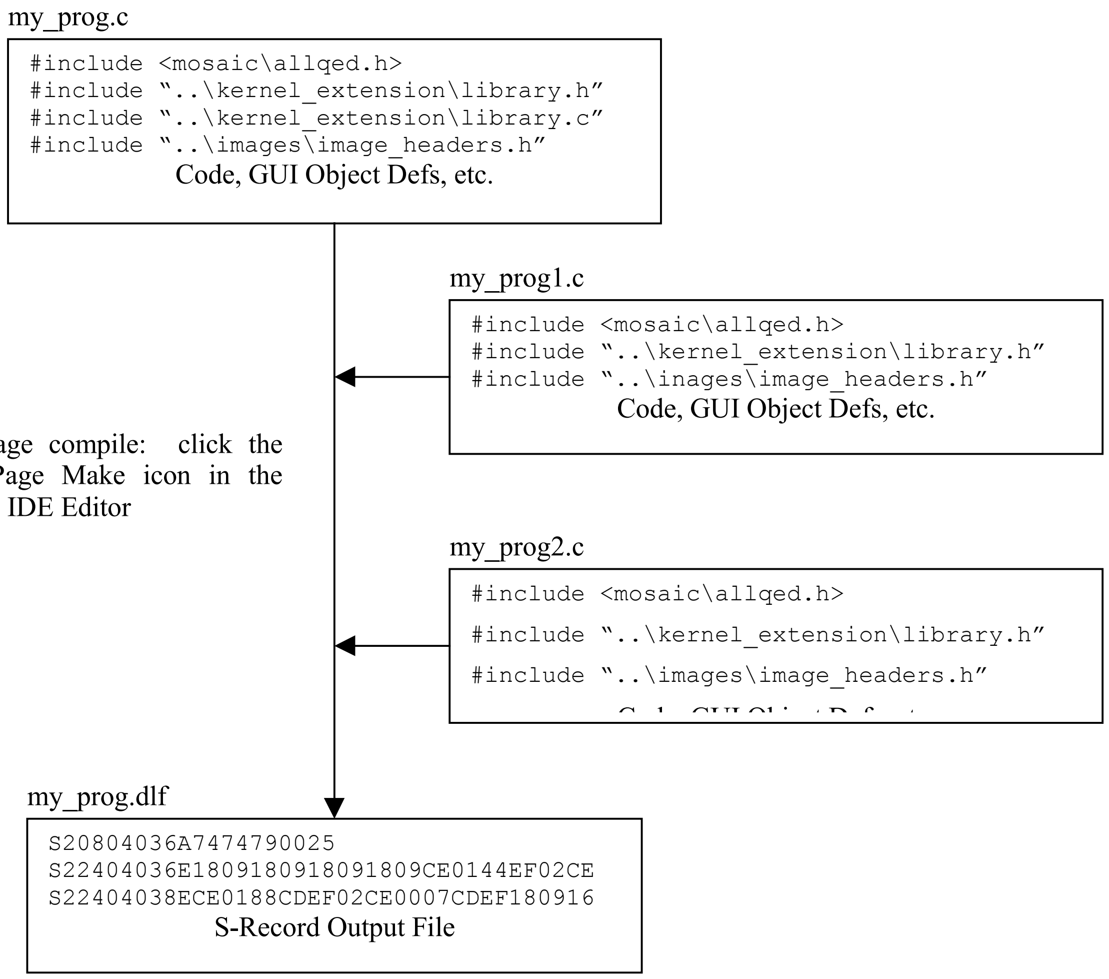
The touchscreen input is managed by the touchscreen menu manager function named Wait_Then_Service_Touch.

| Old function name | Replacement | Comment |
|---|---|---|
| UpdateDisplay | Update_Graphics<br>Update_Text | Either layer may be updated separately |
| UpdateDisplayRam | Update_Here_With | Only used under special circumstances |
| UpdateDisplayLine | Obsolete | Handled with direct draw capability |
| ClearDisplay | Clear_Graphics<br>Clear_Text | |
| Is_Display | Config_Display | |
| InitDisplay | Init_Display | See Config_Display in the *GUI Toolkit Glossary* |
| DisplayOptions | Set_Display_State<br>Set_Cursor_State | |
| IsDisplayAddress | Obsolete | Part of Update_Here_With |
| DISPLAY_HEAP | tvars.display_heap_top<br>tvars.display_heap_bottom | See tvars in *GUI Toolkit Glossary* |
| LinesPerDisplay<br>CharsPerDisplayLine | tvars.graphic_rows<br>tvars.text_rows<br>tvars.graphic_cols<br>tvars.text_cols | See tvars in *GUI Toolkit Glossary* |
| ScanKeyPad | Read_Touchscreen +<br>Wait_For_Release | Old driver used for the digial touchscreen and keypad |
| ScanKeyPress | Read_Touchscreen | Old driver used for the digial touchscreen and keypad |
| Keypad | Wait_For_Press +<br>Read_Touchscreen +<br>Wait_For_Release | Old driver used for the digial touchscreen and keypad |
| CharToDisplay<br>CommandToDisplay | none | These low level drivers should be used by advanced users with extreme care |
| GARRAY_XPFA | none | Returns the display shadow array for the text layer. Use tvars.graphics_garray to access to the shadow array for the graphics layer. |
| DisplayBuffer | irrelevant | Only useful for ascii displays |
| BufferPosition | irrelevant | Only useful for ascii displays |
| StringToDisplay | none | Writes a string into the text layer. See its glossary entry in the *Forth/C Glossary* |
| PutCursor | none | Positions the cursor. See its entry in the *Forth/C Glossary* |

**Table 4. Updated device drivers**

## Notes on Multipage Applications In C

The Control C cross compiler uses multiple compiler passes to allow programs written for the QScreen to span across multiple 32 Kbyte pages of memory. When using the GUI Toolkit with single page applications, the library.c file must be included prior to using any of the GUI Toolkit functions. Since library.c contains code, it must only be included once when an application consists of several files to linked together into a large application. In order for the function names to be accessible to the other files, the Library.h header file must be `#included` in the other source code files. Also, keep in mind that the graphics symbols file, image_headers.h, also must be `#included` in all files that need access to the graphics names. Figure 9 illustrates the organization of multipage applications. The main compilation file is my_prog.c; compiling this file also causes the compilation of the like-named files my_prog1.c and my_prog2.c, resulting in the S-record output file named my_prog.dlf which is downloaded to the QScreen.


my_prog.c
Code, GUI Object Defs, etc.
my_prog1.c
Code, GUI Object Defs, etc.
my_prog2.c
Multipage compile: click the Multi-Page Make icon in the Mosaic IDE Editor
my_prog.dlf
S-Record Output File

```
#include <mosaic\allqed.h>
#include “..\kernel_extension\library.h”
#include “..\kernel_extension\library.c”
#include “..\images\image_headers.h”
```

```
#include <mosaic\allqed.h>
#include “..\kernel_extension\library.h”
#include “..\inages\image_headers.h”
```

```
#include <mosaic\allqed.h>
#include “..\kernel_extension\library.h”
#include “..\images\image_headers.h”
```

```
S20804036A7474790025
S22404036E1809180918091809CE0144EF02CE
S22404038ECE0188CDEF02CE0007CDEF180916
```

**Figure 9. Multipage C programming diagram**

## Touchscreen Calibration

The analog touchscreen is tested and then calibrated after each QScreen Controller is assembled and before it is shipped from Mosaic Industries. If the touchscreen calibration changes (i.e. you push a button and nothing happens or another button registers the press event), you will have to recalibrate the touchscreen. To calibrate the touchscreen it is recommended that you acquire the uncalibrated touchscreen readings from three points. The points are chosen to avoid non-linearities (points that are not too close to the edge), minimize scaling errors (points that are not too close to each other), and yield non-redundant simultaneous equations. The recommended calibration points are (34,19), (290,123), and (162,219). Once you obtain the three points, simply call the Calibrate_Touchscreen method to calculate and store the new calibration coefficients for the touchscreen into Flash memory for later use. The calibration coefficients will then be applied to the uncalibrated touchscreen readings each time someone touches the touchscreen.

An example calibration routine is located in the gui_demo code. It shows how to calibrate the touchscreen. The demo code prompts the user to press three locations on the touchscreen, captures the user’s touches using the Get_Raw_Coords routine, and finally calls the Calibrate_Touchscreen method.

# Tips and Tricks

There are a number of useful techniques for building elegant user interfaces described below that may not be immediately obvious to many users. The following topics are described from a C perspective; however, the information is equally relevant to Forth programmers.

## Double Size Buttons

Many system control panels require large buttons for major control functions. For example, the demo program that is shipped with the QScreen Controller uses a double width button for the "Shut Down" button. The graphic object associated with a button object may be of any size, even if it is larger than the typical size occupied by one touchscreen-sized button area (see Figure 1. LCD screen geometry). In order for all the area occupied by the large button to be sensitive to touchscreen presses, the menu definition must contain an object for each touchscreen location that overlays the button.

For example, suppose that a button is twice the usual width, covering button locations 5 and 9 in Figure 2. In Listing 5 below, the call to ADD_TOUCH_BUTTON adds the button to the button location corresponding to the upper left corner (button location 5), but ADD_BUTTON must be used to manually add the button to the additional button location 9. The graphical coordinates for the upper left corner remain the same since the button is still in the same place on the screen, regardless of which half is pressed. Due to hardware constraints on the QScreen, double width buttons that are repeating are not recommended because pressing 2 horizontally adjacent button locations simultaneously may cause an early release.

```
FASTBUTTON(                    // New button
PRESS_HANDLER_FLAG,            // Execute action upon press
BIG_ICON_PCX,                  // Wide draw graphic
BIG_ICON_PCX,                  // Wide release graphic
DBLBLACK_PCX,                  // Wide press graphic
my_handler,                    // Function to execute when pressed
"",                            // No text for the first line
"",                            // No text for the second line
"",                            // No text for the third line
"",                            // No text for the fourth line
my_big_button );               // Name of new button

.
.

NEW_MENU mymenu [5]=
{
ADD_TOUCH_BUTTON( 8, 33, DRAW_MASK, my_big_button ),
ADD_BUTTON( 9,    8, 33, DRAW_MASK, my_big_button ),
.
.

}
```

**Listing 5. Code for large buttons**

## Modal Selectors and File Tabs

Another common technique used in instrument control panel design is modal selection. For example, a machine may operate in one of three modes named Auto, Manual, and Test. We can design a menu with three mode buttons labeled Auto, Manual and Test, only one of which is “emphasized” at any given time to indicate the current mode. When the mode is changed by pressing another button, the emphasized version of the previous mode should return to the de-emphasized version and the new mode button should become emphasized. In such a case, three different graphics are required for the button: the unselected (de-emphasized) version, the selected (emphasized) version, and the pressed (user touching the button on the touchscreen) version. In the button definition, these correspond to the *draw graphic*, *release graphic*, and *press graphic*. If the draw graphic is different from the release graphic, then the button will appear with the image of the draw graphic after the menu is first drawn, but after being pressed and released, the different released graphic will appear instead.

The C source code for a sample implementation of such a modal instrument is presented in Appendix 1 – Modal Button Selection Example. When the menu manager processes a button press, it stores information about the currently pressed button into global variables contained within the **tvars** structure, explained in the *GUI Toolkit Glossary.* This information is then accessible to the programmer-defined handler. When a button is pressed, its handler should cause the previously emphasized button to be drawn in its de-emphasized state. To do this, the handler needs to know which button was previously selected and its location. Using that information, the handler can call Do_Button with the DRAW_ACTION and/or DIR_DRAW_ACTION to draw the previously selected button in its unselected state. Next, the handler should save its own pointer and screen location into some user variables that identify it as the currently selected mode so that it can be de-emphasized when future modes are selected.

Another similar application of modal button selection the implementation of file tabs along the edge of the screen. File tabs are useful in displaying different screens consisting of different controls or displays. In Figure 10 file tabs are used to select different screens containing different menus. The file tabs are modal selectors in which the emphasized (front-most) and de-emphasized (rear) versions of the file tab buttons are used to provide an illusion of layering. The emphasized graphic is drawn without a horizontal line below its label, while all of the de-emphasized graphics are shown with a horizontal line that makes it look like they are behind the emphasized file tab.

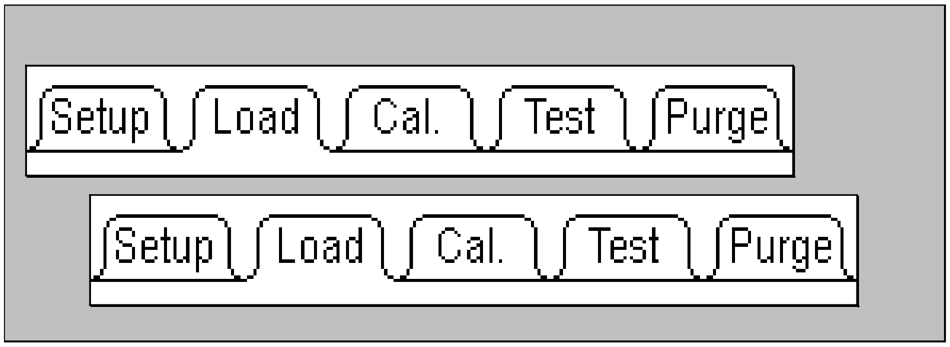


**Figure 10. File tab selectors**

## Custom Fonts

There are two ways to place text on the LCD screen: using text mode, and rendering fonts as bit maps. The most common method is to use the text mode of the LCD in conjunction with the function StringToDisplay. The text string is written to the display’s text array at the location specified in the call to StringToDisplay. The entire text array is sent to the display by calling Update_Text_Display. This technique is extremely fast because text is sent in its compact ASCII form to the LCD (that is, text is not sent as a bitmapped image). The disadvantage to text mode is that all characters must occupy a 6 pixel wide by 8 pixel high space.

The second method is to render the text as graphical objects onto the graphics layer of the display. This method allows text to be of any size and shape since the letters are drawn as graphics from regular graphic objects. The disadvantage is that graphical text placement is much more time consuming for the processor. If you have special needs regarding graphical font rendering, please contact Mosaic Industries.

## *Custom Fonts in Text Mode*

The LCD has a built-in character generator to provide a text display mode. Text mode operates independently of graphics mode as a separate layer overlaid onto the graphical layer. The font used by text mode is stored in the ROM of the LCD's T6963C display controller. Custom text mode fonts can be downloaded to the LCD module's RAM to be used in addition to or in place of the built-in ROM font, depending on the setting of the text mode (see **Set_Text_Mode** in the *GUI Toolkit Glossary*). The characters that comprise a font are stored in the LCD in the same order as their ASCII values in the display. By default, the display uses its built-in ROM font for text mode and has an offset of 0x20. This means that each byte of the text data sent to the display must have 0x20 subtracted from the ASCII value in order to map to the correct character of the font. StringToDisplay does this modification for you when writing strings into the text array. Any data manually placed in the text array will be sent directly to the display without the offset. ASCII characters below 0x20 are control characters and have no printable representations. The ROM font consists of only 128 characters. Bytes with values above 128 will be printed to the screen using the corresponding RAM font characters. This allows the ROM based font to be used in conjunction with additional custom characters. The default ROM font is shown in Figure 11. The numbers at the top and bottom of each pillar are the ASCII values that correspond with the characters assuming that StringToDisplay is used.

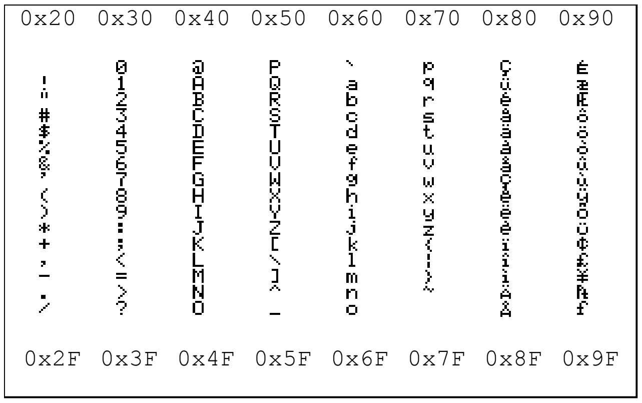

0x20 0x30 0x40 0x50 0x60 0x70 0x80 0x90
0x2F 0x3F 0x4F 0x5F 0x6F 0x7F 0x8F 0x9F

**Figure 11. LCD Built in ROM font for text mode**

Any text mode font must be 6 pixels (1 display column) wide by 8 pixels high. Thus each character of the font requires 8 bytes of memory. *Appendix 2 – Text Mode Custom Font Example* shows how to use a set of graphic objects to download a font to the display for use in text mode. By building the graphic objects in pillars of 16 characters as shown in Figure 11, a font identical to the ROM font can be represented with 8 graphic objects. In the example program in Appendix 2, the 8 graphic objects from left to right in Figure 11 are f0.pcx through f7.pcx. You can use your favorite graphics editor to modify them, and follow the example in Appendix 2 to generate your own custom characters in text mode. 16 pillars can be used to create a full 256 character font, the maximum allowed. Of course, the higher order characters are not type-able in a string and must be manually placed in the text array. The code example in Appendix 2 shows an example that demonstrates how to write bytes directly into the text array.

## Check Boxes

Check boxes are useful for requiring an operator to acknowledge certain information before proceeding. Buttons can be built that "stick" in the checked position whenever they are pressed and have handlers that check to see if all the other check boxes have been checked. When all of the check boxes have been checked, the handler can call another menu or cause some other event to occur. This would be a good way to implement a safety checklist that the operator must go through prior to commencing a risky function on a machine.

The trick behind building checkbox buttons is really a simplification of the technique previously described for modal buttons. The empty unchecked version of a check box button graphic is specified for the draw_graphic field of the button object definition and the checked version of the button graphic is specified for the release_graphic field of the definition. The graphic specified for the press_graphic field of the definition is the image for the button when the button is actually being held down by the operator. When each check box is pressed and released, the draw_graphic that was drawn when the menu was first placed on screen will be replaced with the press_graphic, and subsequently the release_graphic which is an image of the box with a check mark in it. Use the handlers for the check boxes to provide the logic to produce the desired behavior when the boxes are checked.

## Pop-up Dialog Boxes

Dialog boxes provide a means of alerting an operator to an important condition or to get a quick button press response to a question on the screen. The most common use for dialog box menus is the query for "Are you sure? $\boxed{\text{Yes}}$ $\boxed{\text{No}}$" It is generally desirable for dialog boxes to be superimposed over a menu that is already displayed and return the screen to its original state after receiving a response from the operator. The best way to accomplish this is to take advantage of the difference between direct drawing to the screen and drawing using the graphics array. These two techniques for placing graphic data on the screen are described in *How to design your GUI→The LCD→Drawing to the LCD Screen*.

The graphics array is typically used for drawing regular menus to the display, while button presses on that menu take advantage of the rapid response of direct drawing. For dialog boxes, the entire dialog menu can be drawn using direct drawing without disturbing the graphics array at all. You must call Menu_Remove to make the buttons on the background menu insensitive to touch while the dialog box menu is active. When the dialog box has received the response from the operator, the background menu that is partially covered by the dialog box menu can be restored with a simple call to Update_Graphics, assuming that the background menu was drawn using the graphics array. A call to Menu_Install will restore the background menu buttons' sensitivity to touch. If the same dialog box is to be used in conjunction with various background menus, then a variable containing a pointer to the background menu and its screen location must be used by the dialog box code to provide the parameters to Menu_Remove and Menu_Install.

Since Init_Menu draws the objects of a menu to the graphics array, it cannot be used for dialog boxes that do not disturb the graphics array. Instead, use the functions Do_Menu with the DIR_DRAW_ACTION flag and Menu_Install. (Init_Menu does the same thing except that it uses the DRAW_ACTION flag). The dialog menu must use the FASTBUTTON macro for the buttons so that their press/releases will be done without using the graphics array.

# Summary

This document has explained how to set up your development environment, load the GUI Toolkit firmware, and craft a customized graphical user interface for your product. We want to help you succeed. If you have any questions, problems, or requests, please contact Mosaic Industries and we will do our best to help you.

# Appendix 1 – Modal Button Selection Example

The following example can code can be found at \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT\C\MODAL\modal.c. This code demonstrates how to create buttons that behave in a modal fashion in which each button "sticks" in a selected position while "unsticking" the previously selected button.

```
#include <\mosaic\allqed.h> // include all of the qed and C utilities

// The following lines are include the default demo kernel extension.
// See the alternative below.
#include "..\..\kernel_extension\library.h"
#include "..\..\kernel_extension\library.c"
// When you're using other drivers/software toolkits, you must generate
// your own kernel extension files using the web based kernel extension
// manager. The following commented out lines can be used to #include the
// library files for your own kernel extension build assuming you
// unzipped them into the my_kernel_extensions directory.
// #include "..\..\my_kernel_extensions\library.h"
// #include "..\..\my_kernel_extensions\library.c"

#include "PCX\IMAGE_HEADERS.H"
// NOTE: DOWNLOAD .\PCX\IMAGE_DATA.TXT BEFORE EXECUTING

GUI_VARS  tvars;            // Create an instance of the touchscreen
                            // variable structure.  All programs
                            // that use the Gui Toolkit must declare
                            // an instance of this structure.
uint firsttime;             // Flag that indicating whether or not
                            // deselect_last is executing for the first time.
uint lastrow, lastcol;      // Graphical row and column of the previously
                            // selected mode button object
xaddr lastbutton;           // 32 bit pointer to the previously selected
                            // mode button object

#define TOP_OF_HEAP       0x0F7FFF
#define BOTTOM_OF_HEAP    0x0F4800

void deselect_last ()
{
  if (lastbutton!=tvars.current_button)
  {
    // This call causes the last button's deselected version of its image
    // to be directly drawn to the screen.  Comment out if not using
    // direct to screen drawing.
    Direct_Draw_Graphic( lastcol, lastrow, TVARS,
      ((BUTTON *)lastbutton)->draw_graphic );

    // This call causes the last button's deselected version of its image
    // to be drawn to the graphics array thus requiring a call to
    // Update_Graphics for the change to be appearent on the screen.
    // Comment this out and use the previous call if no calls to
    // Update_Graphics are going to occur.
    Do_Button( lastcol, lastrow, TVARS, DRAW_ACTION, lastbutton );

    lastrow=tvars.current_row;        // update the last button used info
    lastcol=tvars.current_col;
    lastbutton=tvars.current_button;  // The 32 bit pointer to the mode
  }                                   // button object
}
```

```
void automode_handler ()
{
  printf("Auto button pressed\n");
  deselect_last();
}

void manmode_handler ()
{
  printf("Man button pressed\n");
  deselect_last();
}

void testmode_handler ()
{
  printf("Test button pressed\n");
  deselect_last();
}

#include <mosaic/gui_tk/begingui.h>

FASTBUTTON( PRESS_HANDLER_FLAG | DRAW_TEXT_FLAG,
   LBLANK_PCX,
   L_EMPH_PCX,
   LBLACK_PCX,
   automode_handler,
   "",
   "  Auto",
   "  Mode",
   "",
   automode_button );


FASTBUTTON( PRESS_HANDLER_FLAG | DRAW_TEXT_FLAG,
   LBLANK_PCX,
   L_EMPH_PCX,
   LBLACK_PCX,
   manmode_handler,
   "",
   " Manual",
   "  Mode",
   "",
   manmode_button );


FASTBUTTON( PRESS_HANDLER_FLAG | DRAW_TEXT_FLAG,
   LBLANK_PCX,
   L_EMPH_PCX,
   LBLACK_PCX,
   testmode_handler,
   "",
   "  Test",
   "  Mode",
   "",
   testmode_button );

// The following constants must match values in the mode_selector menu
// of the initial mode button.
#define MODE_INIT_COL 8                     // Initial mode column
#define MODE_INIT_ROW 33                    // Initial mode row
#define MODE_INIT_BUTTON automode_button    // Initial mode button ptr

NEW_MENU mode_selector[3] =
{            // add buttons to the menu by initializing array of structs
```

```
  ADD_TOUCH_BUTTON(  8,   33,  DRAW_MASK, automode_button ),
  ADD_TOUCH_BUTTON(  16,  33,  DRAW_MASK, manmode_button ),
  ADD_TOUCH_BUTTON(  24,  33,  DRAW_MASK, testmode_button )
};

BUILD_MENU( mode_selector, 3);

#include <mosaic/gui_tk/endgui.h>

void init_mode_menu ()
{
  Init_Menu( 0, 0, 0, TVARS, mode_selector_menu ); // draw & install menu

  // When menu is first drawn, draw the selected graphic of the initial
  // mode button object.  The graphical location must correspond with the
  // location in the menu that that button occupies.

  lastrow=MODE_INIT_ROW;
  lastcol=MODE_INIT_COL;
  lastbutton=(xaddr)MODE_INIT_BUTTON;

  Draw_Graphic( MODE_INIT_COL, MODE_INIT_ROW, TVARS,
      ((BUTTON *)((addr)MODE_INIT_BUTTON) )->release_graphic );
  Update_Text_And_Graphics(TVARS);    // Commit the graphics array to LCD
}

void main ()
{
  IsHeap( BOTTOM_OF_HEAP, TOP_OF_HEAP );  // Specify the heap
  Std_Display( TVARS );         // Config the display hardware
  Init_Display( TVARS );        // Initialize the display hardware
  Init_Touch( TVARS );          // Initialize touchscreen hardware & vars

  init_mode_menu( );            // Finally, initialize newly defined menu
                                // and display it on the screen

  while(1)                      // infinite loop runs application
  {
    Wait_Then_Service_Touch( TVARS ); // Get a single touch from the user
    PauseOnKey();               // Comment out this line in a real app.
  }
}
```

# Appendix 2 – Text Mode Custom Font Example

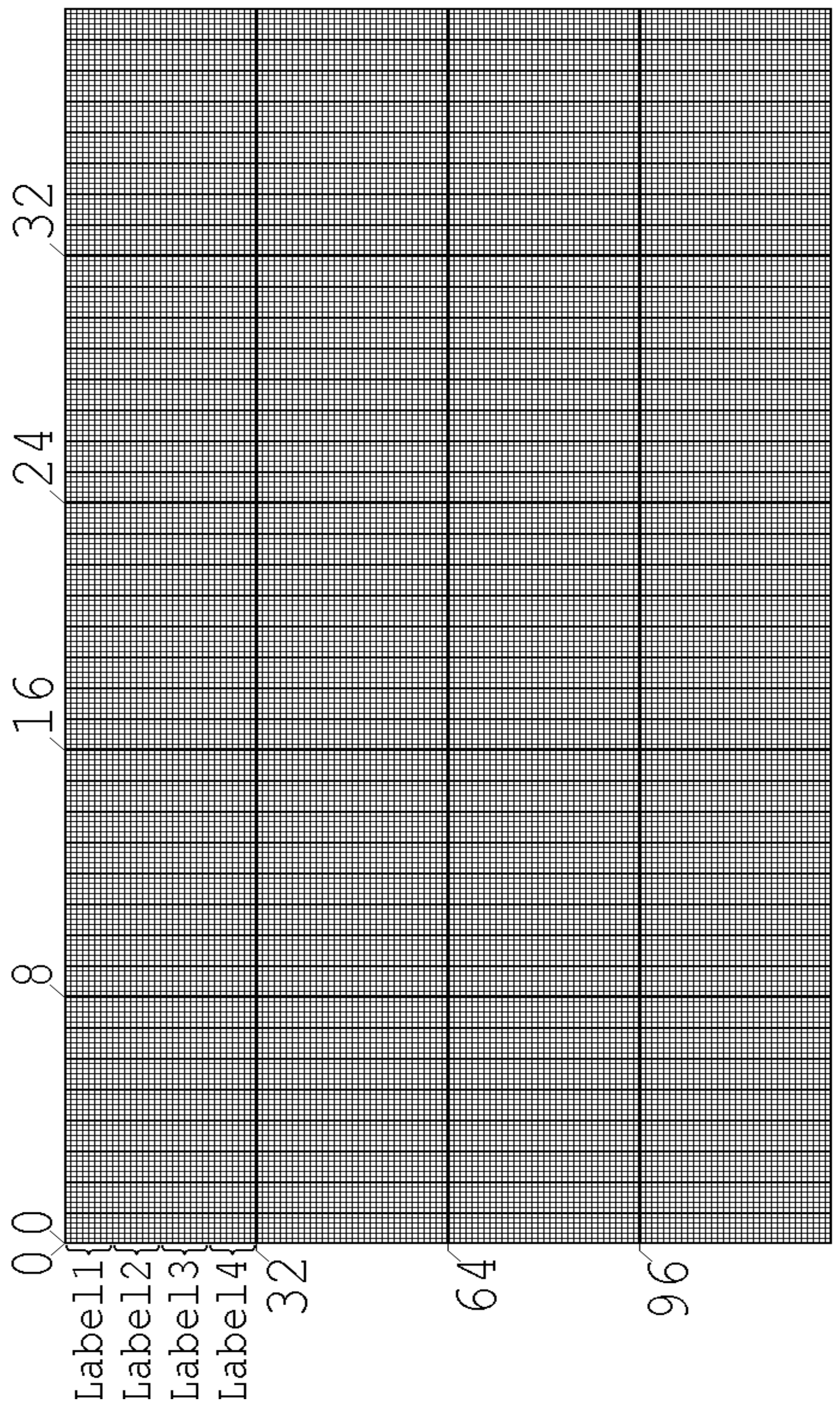
The following example demonstrates downloading graphics objects as custom fonts into the LCD module for use on the text layer. The path for this program is \MOSAIC\DEMOS_AND_DRIVERS\GUI_TOOLKIT_PTC\C\FONTS\custchar.c. The graphics used are in the pcx directory. They are identical to the ROM based fonts built into the display but can be modified for your application and used in conjunction with the code below.

```
#include <\mosaic\allqed.h>

// The following lines are include the default demo kernel extension.
// See the alternative below.
#include "..\..\kernel_extension\library.h"
#include "..\..\kernel_extension\library.c"
// When you're using other drivers/software toolkits, you must generate
// your own kernel extension files using the web based kernel extension
// manager.  The following commented out lines can be used to #include
// the library files for your own kernel extension build assuming you
// unzipped them into the my_kernel_extensions directory.
// #include "..\..\my_kernel_extensions\library.h"
// #include "..\..\my_kernel_extensions\library.c"

#include "pcx\image_headers.h" // The graphics symbol file generated from
                            // the pcx converter
                            // The font graphics in image_data.txt must
                            // first be downloaded to the board.

GUI_VARS  tvars;            // Create an instance of the touchscreen
                            // variable structure.  All programs
                            // that use the Gui Toolkit must declare
                            // an instance of this structure.

#define RAM_CG_OFFSET 0x1800
#define OFFSET_REGISTER_SET_CMD 0x22

_Q void installfonts( )
    {
    // Here, we use the low level function, Update_Here_With, to load the
    // font table into the display's RAM. A full 256 character font table
    // requires 0x800 bytes (2k) of display memory, one quarter of total
    // memory.  We locate this table in the top 2 k of memory in the
    // display, above the graphics area.  In the default ROM font mode,
    // character codes 00-7f map to the ROM font table.  Character codes
    // 80-FF map to the area starting half way through the font table. In
    // this case, 80 would map to the font character stored at 1C00-1C07.
    // 81 would map to 1C08-1C0F and so on. If RAM font mode is selected,
    // then all 00-FF character codes will map to area starting at 1800.
    // Also, TC6963C has a 0x20 ascii offset.  That is, all ascii codes
    // must have 0x20 subtracted from them before being sent to display
    // or stored in the garray.  That is handled automatically by
    // StringToDisplay.
    // The font loading should be done after the display is initialized.
    Update_Here_With( RAM_CG_OFFSET+0x0000, &tvars.display_resource,  THIS_PAGE,
XADDR_TO_ADDR(F0_PCX),XADDR_TO_PAGE(F0_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0080, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F1_PCX),XADDR_TO_PAGE(F1_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0100, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F2_PCX),XADDR_TO_PAGE(F2_PCX) );
```

```
    Update_Here_With( RAM_CG_OFFSET+0x0180, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F3_PCX),XADDR_TO_PAGE(F3_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0200, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F4_PCX),XADDR_TO_PAGE(F4_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0280, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F5_PCX),XADDR_TO_PAGE(F5_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0300, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F6_PCX),XADDR_TO_PAGE(F6_PCX) );

    Update_Here_With( RAM_CG_OFFSET+0x0380, &tvars.display_resource, THIS_PAGE,
XADDR_TO_ADDR(F7_PCX),XADDR_TO_PAGE(F7_PCX) );

    // The following 3 lines tell the display hardware where to point to
    // the RAM fonts.

    CharToDisplay( (char) RAM_CG_OFFSET>>11 );      // low byte
    CharToDisplay( 0 );                             // high byte
    CommandToDisplay( OFFSET_REGISTER_SET_CMD );

    Set_Text_Mode (0x08);     // Set the text mode to external RAM fonts

    }

_Q void init_lcd ()
    {
    Std_Display( TVARS );         // Sets the display type to the default
    Init_Display( TVARS );        // Initialize the display
    }

void main( ){

    IsHeap ( 0x0F5000, 0x0F7fff);     // Set up a heap

    init_lcd();                       // initialize the display
    installfonts();                   // upload the fonts to the display

    StringToDisplay("Testing Text Mode using",(0xff00|THIS_PAGE),8,10);
    StringToDisplay("the custom fonts table",(0xff00|THIS_PAGE),9,10);
    StringToDisplay("displayed to the left.",(0xff00|THIS_PAGE),10,10);

    // Below is an example of accessing additional custom characters from
    // beyond the base 128 char set.
    // Since these char codes are not typable, they are stored directly
    // into the garray.
//    ARRAYSTORE(0x80,16,5,GARRAY_XPFA);  // Notice that coordinates are
//    ARRAYSTORE(0x81,17,5,GARRAY_XPFA);  // in reverse order from
//    ARRAYSTORE(0x82,18,5,GARRAY_XPFA);  // StringToDisplay.
//    ARRAYSTORE(0x83,19,5,GARRAY_XPFA);
    Update_Text( TVARS );
    Do_Graphic( 0,0,TVARS,DIR_DRAW_ACTION,F0_PCX);  // Draw the fonts on
    Do_Graphic( 1,0,TVARS,DIR_DRAW_ACTION,F1_PCX);  // the left side of
    Do_Graphic( 2,0,TVARS,DIR_DRAW_ACTION,F2_PCX);  // screen as regular
    Do_Graphic( 3,0,TVARS,DIR_DRAW_ACTION,F3_PCX);  // graphic objects
    Do_Graphic( 4,0,TVARS,DIR_DRAW_ACTION,F4_PCX);
    Do_Graphic( 5,0,TVARS,DIR_DRAW_ACTION,F5_PCX);
    Do_Graphic( 6,0,TVARS,DIR_DRAW_ACTION,F6_PCX);
    Do_Graphic( 7,0,TVARS,DIR_DRAW_ACTION,F7_PCX);
    }
```

# Appendix 3 – LCD Worksheet

The following Figure is an enlarged version of Figure 1, the diagram of the LCD geometry. This image may be photocopied and used as a guide for sketching GUI prototypes.


0 0
8
16
24
32
Label1
Label2
Label3
Label4
32
64
96