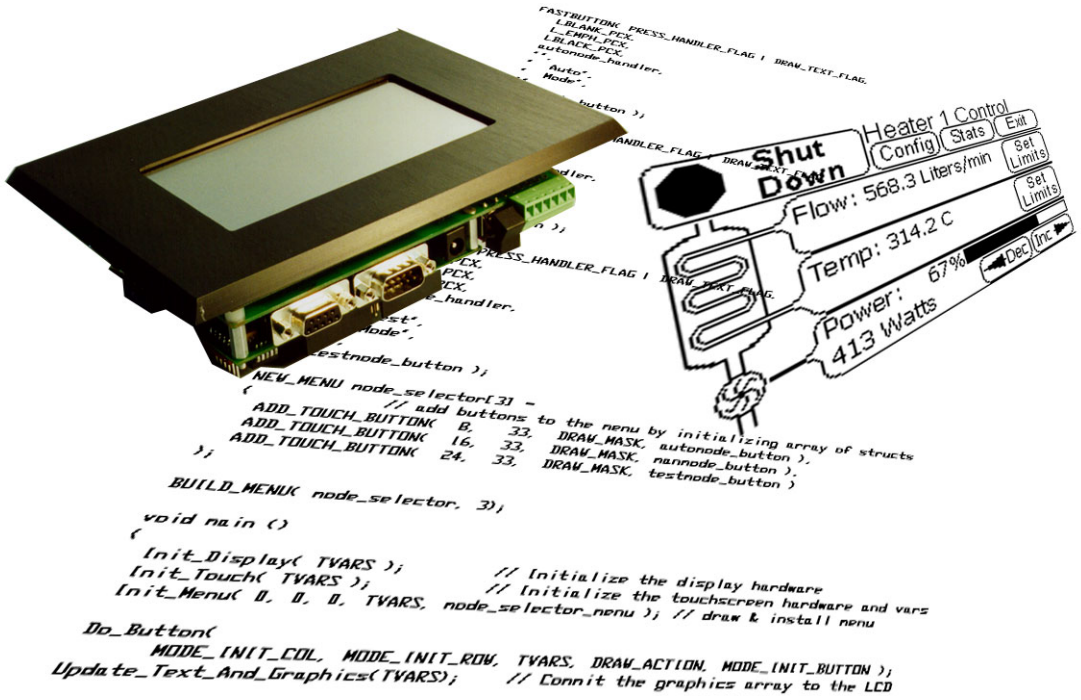
The GUI Software Toolkit Manual
Version 1.6 (Kernel Extension)



Revision 9/18/03

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 Panel-Touch 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:

- Graphics are bitmapped images that may be displayed directly on the screen or used as elements of buttons or menus.
- Buttons 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.
- Menus 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, Menu_Query, 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 QED Board that animates the Panel-Touch 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 QED Board, 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 QED Board is connected and turned on, you should see an ‘ok’ prompt when you hit enter in the terminal program. Note, however, that new Panel-Touch starter kits are shipped with a pre-installed demo program that automatically runs each time the QED Board 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 using dip switches 6 and 7 as explained in detail in the "Getting Started" document. Briefly, to perform a factory cleanup, do the following:

- Turn DIP switch 6 on, and toggle DIP switch 7 on then off;
- Return DIP switch 6 to its off state, and again toggle DIP switch 7.

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

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

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 manual titled "Getting Started with the QED Board Using the Control C Programming Language" to install and test the C development package.

Installing the GUI Toolkit

The GUI Toolkit is now 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 custom 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. If you purchased your equipment prior to October of 2003, then email kxmanager@mosaic-industries.com or call us at (510) 790-1255 for a URL (there is no cost associated with kernel extensions).

If you are using older versions of the QED Control C development software that use WinEdit as the editor, some files may be out of date. If you are not using our Mosaic IDE version 1.164 or later, please contact us and we will upgrade you free of charge. This upgrade addresses all dependency issues with the C compiler and now comprises the additional support files for the GUI Toolkit such as the documentation, demo, and image converter utility needed for converting your own graphics to GUI Toolkit graphic objects.

Installing the GUI Toolkit Firmware

See the readme.txt file that is included with the kernel extension set that you have generated or in the pre-built kernel extension set provided in the examples directory for the GUI_Toolkit_PTC in the Mosaic IDE. Starter kits are usually shipped with a running demo program that may be cleared by performing a factory cleanup using dip switches 6

and 7 as explained in the section titled *Setting Up the Development Environment*. From your 9600 baud terminal communicating with the QED Board as explained in the "Getting Started" manual, type

```
COLD
```

You should see the "Coldstart" and "QED-Forth V4.0x" messages. If you do not see any text, try the factory cleanup procedure using DIP switches 6 and 7 as explained in the section titled *Setting Up the Development Environment*. If you are still having trouble, or if the displayed version number doesn't begin with V4.0, call Mosaic Industries at 510-790-1255.

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. Use your PC terminal program (for example, the Mosaic Terminal program which is available on our web site or with the IDE) 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 the examples directory of the IDE workspace. Now you're ready to compile and run your software.

Verifying the Development Environment with the Demo program

You should now have a fully functional Panel-Touch controller (or QED Board plus display) and development environment. The demo program is available in the in the demo_and_drivers/GUI_Toolkit_PTC section of your IDE workspace. There is also a pre-built graphics library in GUI_Toolkit_PTC/Images that is ready to be installed. Download the grafix.txt file, GUI_Toolkit_PTC/Images/grafix.txt, 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 repeatedly during development. Thus, like the kernel extension install file, you only need to download the graphics file once after it has changed. The following Forth and C procedures depend of having installed the kernel extensions and the graphics.

Forth

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 DIP switch (switch 7) to regain control of the unit. Since no autostart was set, the demo won't restart.

C

For C users, start the Mosaic IDE Editor (TextPad) and load the demo C source file, gui_demo.c. Click the single page build icon (the hammer 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 or gui_demo.txt, to the QED Board. 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 main 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 or sent directly to the LCD. 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. It is possible, however, with a call to the primitive function Update_Here_With.

Screen Geometry

The LCD on a Panel-Touch controller has a resolution of 240 horizontal pixels by 128 vertical pixels. The 5 by 4 touchscreen panel divides this into 20 areas of 48 pixels by 32 pixels each. 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.

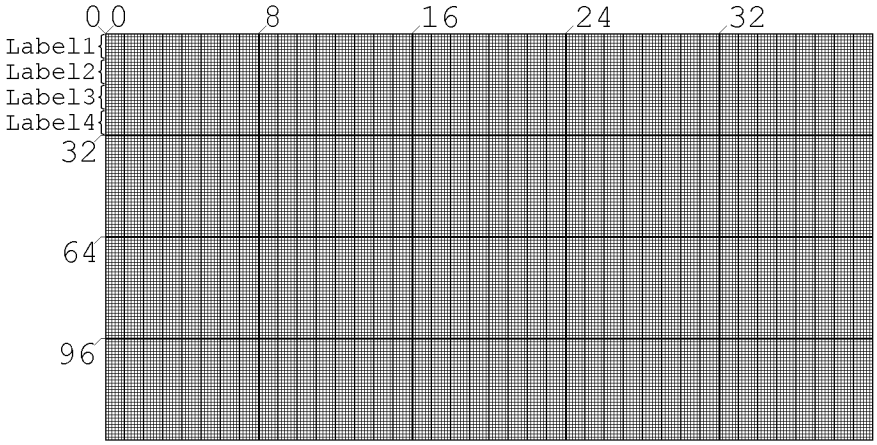


Figure 1. LCD screen geometry

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.

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.

0	4	8	12	16
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19

Figure 2. Touchscreen button numbering

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 QED Board using the graphics conversion software provided with the GUI Toolkit, pcx2qed. 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.

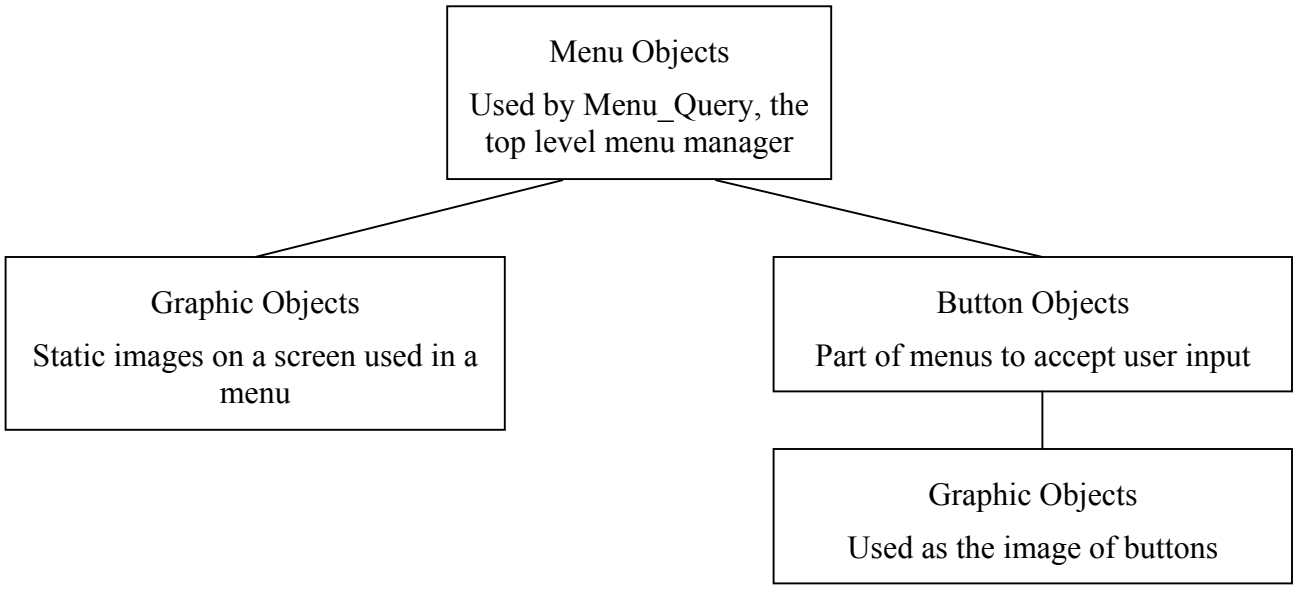


Figure 3. Relationships among different object types

Graphics

Graphic objects, as stated above, are created using the pcx2qed conversion software. 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 grafsyms.h generated automatically from pcx2qed…..

```
.
.
#define MY_ICON1_PCX          0x00072484
#define LOGO_PCX              0x0007252a
#define CAUTION_PCX           0x000725d0
.
.
```

In main source code file written by user….

```
.
.
#include "pcx\grafsyms.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 *Glossary of Functions*.

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 *Glossary of Terms* 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 converter, pcx2qed, creates both parts of the graphic object as S-records in the file grafix.txt. The xaddresses assigned to the macros in the grafsyms.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.

For a more detailed explanation of Forth arrays and their use of the heap memory manager, consult the *QED Software Manual* or *Getting Started with the QED Board Using the Control C Programming Language*.

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 *Glossary of Functions* 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 *Glossary of Terms* 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 RELEASE_GRAPHIC_FLAG DIR_RELEASE_GRAPHIC_FLAG PRESS_GRAPHIC_FLAG DIR_PRESS_GRAPHIC_FLAG C_STYLE_TEXT_FLAG
NORMBUTTON_FLAGS	DRAW_GRAPHIC_FLAG RELEASE_GRAPHIC_FLAG PRESS_GRAPHIC_FLAG C_STYLE_TEXT_FLAG
BLANKBUTTON 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 (line numbers have been added):

```
1.      BUTTON my_button_struct_=
2.      {(0x0080) | ( 0x0001 |0x0002 |0x0010 |0x0004 |0x0020 |0x4000 ),
3.      ((xaddr) ((((0x07))<<16)+ (0xFFFF & ((0x077E))))),
4.      0x00072484,
5.      0x00072484,
6.      0x0004642a,
7.      my_handler,
8.      my_handler,
9.      (""),
10.     (""),
11.     (""),
12.     ("") };
13.     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 grafsyms.h which is generated by the pcx2qed conversion software.

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 \examples 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, Menu_Query, 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 "pcx/grafsyms.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

Menu_Query is the top level function that manages the touchscreen and handles button presses. It is typically called within a main runtime loop. Menu_Query 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, Menu_Query 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. Menu_Query exits after each press/release cycle. Since Menu_Query itself executes the user handlers associated with the buttons, they will run in the same task environment as the call to Menu_Query. 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, Menu_Query. 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 also requires a "pressed" version for display 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.

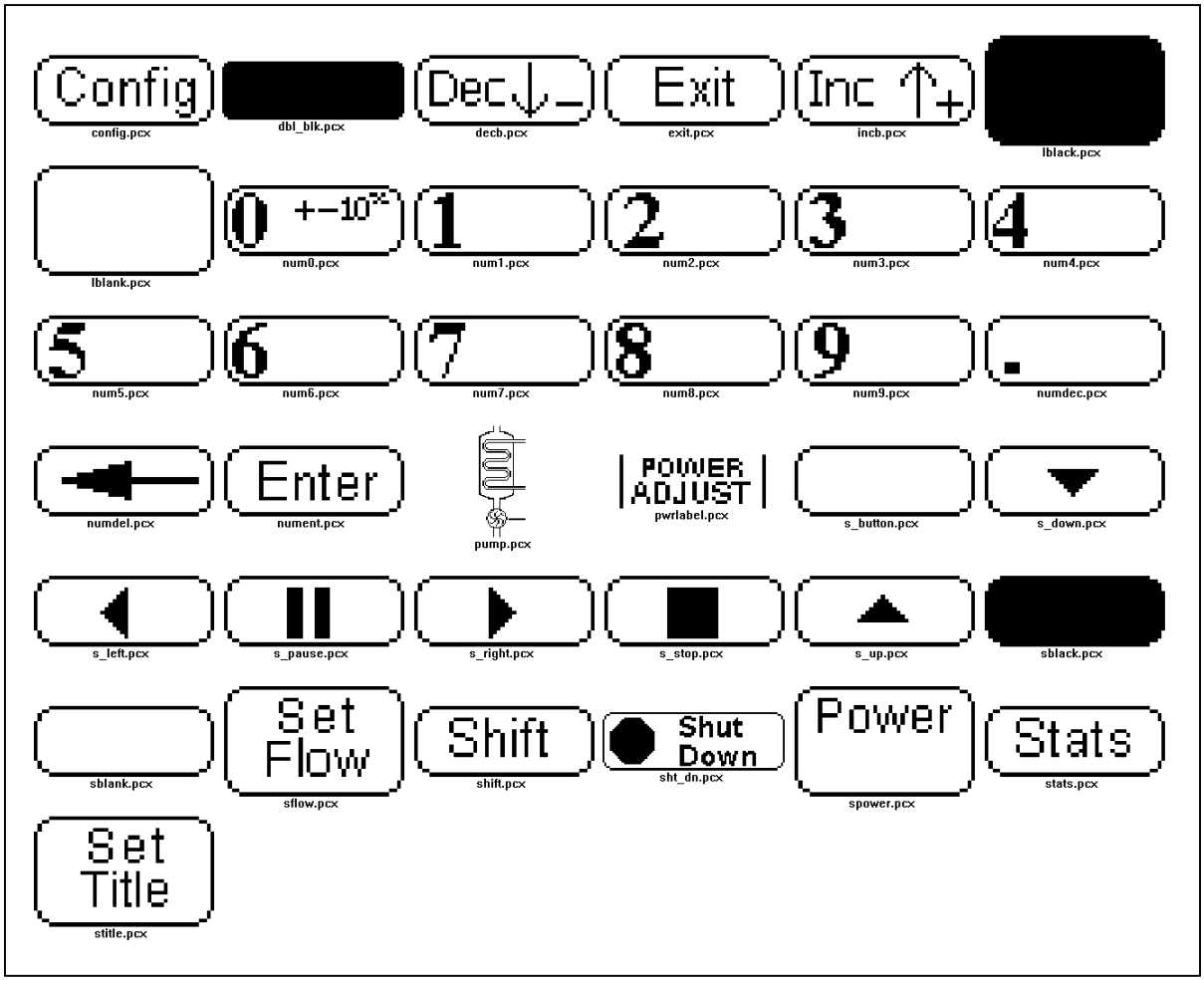


Figure 4. Default graphics library used with the demo (not shown to the same scale)

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 format. Color and grayscale images will be rejected by the pcx2qed conversion software.

Programs that are suitable for custom graphics design include Adobe Photoshop and ZSoft's PC Paintbrush. The Windows paint program is a useful editor, but it does not generate pcx format images. To create graphics for the GUI Toolkit using Windows paint, you would need to download one of the many shareware image format converters available on the web to convert into pcx format images.

All of the graphics in the demo graphics library are provided in pcx form inside the 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.

PCX2QED Conversion Software

The pcx2qed program is a simple DOS command line driven program that converts pcx graphics images into data and symbol files that can be loaded into the memory of the Panel-Touch Controller. In its default configuration, the pcs2qed program scans a directory for pcx image files and generates two files, grafix.txt and grafsyms.h. Grafix.txt is a file containing Motorola S-records of the binary image of a set of graphic objects to be uploaded to the QED Board. For each image, a Forth array parameter field followed by the Forth array data containing the image itself is generated and added to grafix.txt. In addition, pcx2qed places the file gr_pre.4th in front of the S-record data and appends gr_post.4th to the end. Those files contain instructions that facilitate the loading of the memory image into flash. It should not be necessary to alter these preamble or postamble files. It is important that gr_pre.4th and gr_post.4th be located in the same directory as the pcx files.

Grafsyms.h contains a list of constants pointing to the xaddresses of the images. Each constant is named based on the name of the original pcx image. The '.' is replaced with a '_' and all characters are forced to uppercase. The names given to the pcx files must be compatible with C style naming for symbols or an error will occur when the header file is parsed. For this reason, pcx filenames should not begin with numbers. If you are programming in Forth, this restriction does not apply.

For a list of the command line options, type:

```
C:\>pcx2qed -?

PCX Graphic to QED Memory Image Converter              4/20/99 JLW 1999

Help summary
-h <header filename>     Set header output filename
-o <out filename>        Set S record output filename
-p <directory path>      Set the directory in which to search for images
-b <preamble filename>   Set the preamble filename
-a <postamble filename>  Set the postamble filename
   Note: "none" specified for the preamble or postamble filename disables
       Preamble or postamble respectively
-s <c|forth>             Set header output style
-t                       Test without writing any files
-d                       Show lots of ugly debugging information
-i                       Invert image in output file
-r                       Disable paged S record output.  Used for PROM
generation.
-m <start address> <end address>   Set memory RAM range
-e <start address> <end address>   Set memory ROM range
```

The next section explains how to incorporate the output files of the pcx2qed 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\grafsyms.h"
```

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

The graphics file, grafix.txt, which is generated from the pcx2qed conversion software, 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 .txt 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 Introduction → *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 Panel-Touch Controller as part of the pre-loaded demo program on starter kits.

Next, the grafix.txt file should be sent to the Panel-Touch Controller. It contains graphics data that is loaded temporarily onto page 6, and from there is transferred into a predetermined memory location (typically in flash on page 7) as determined by the command line parameters passed to pcx2qed.

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 QED Board 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 grafsyms.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.

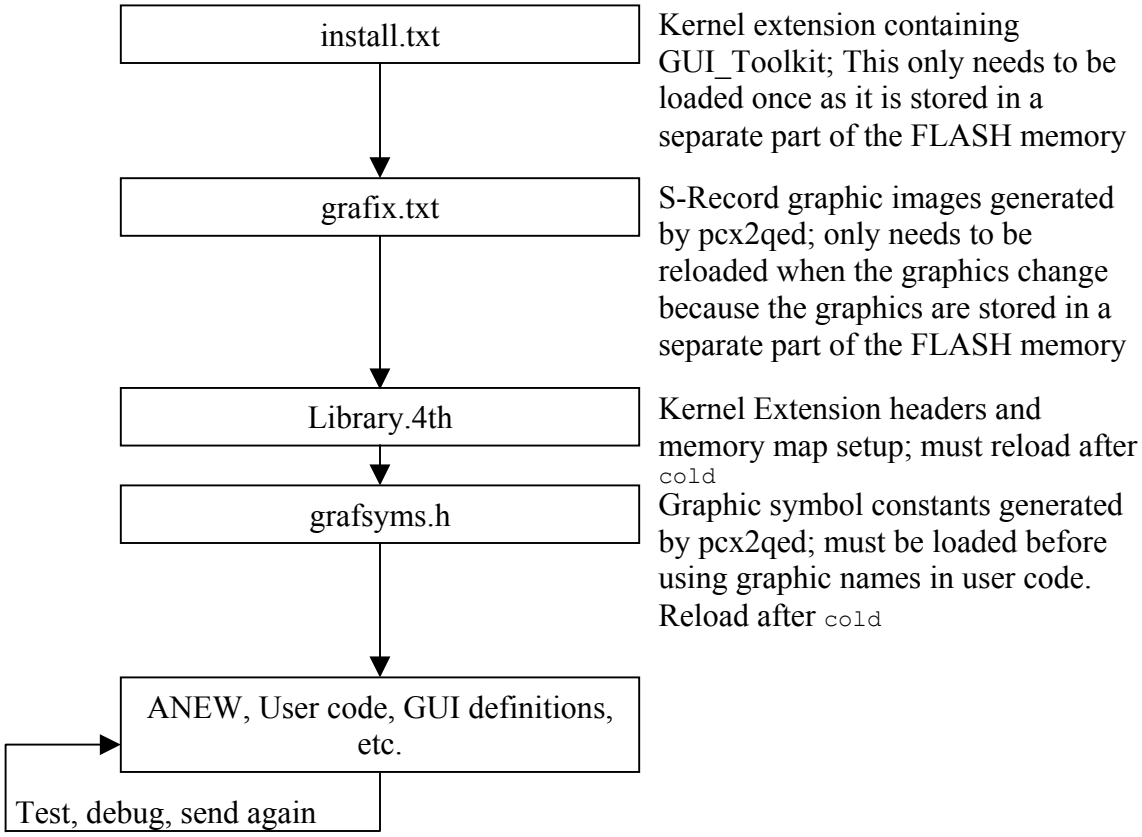


Figure 5. Forth Download Sequence

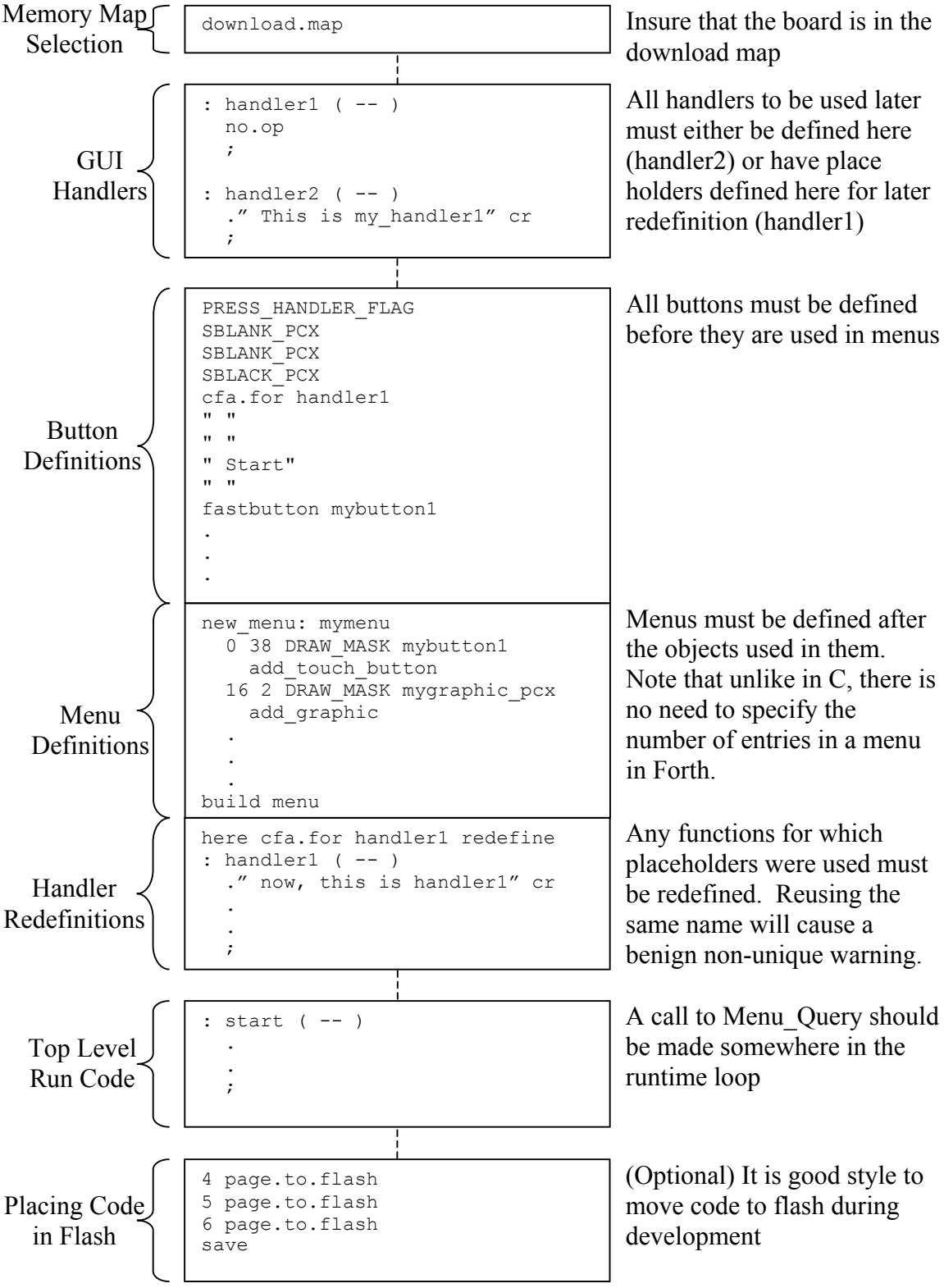


Figure 6. Forth program construction

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 graphics symbols 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.

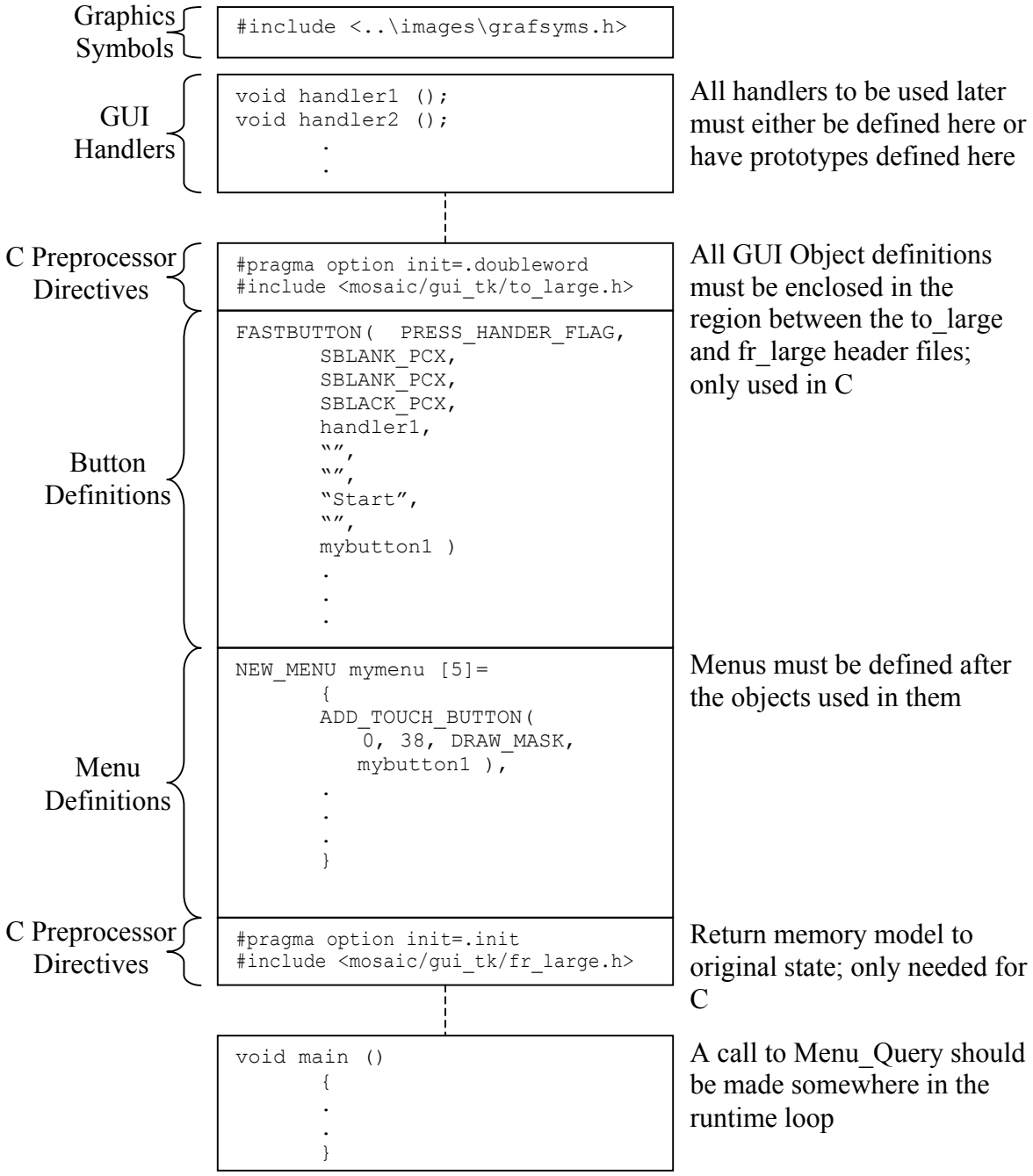


Figure 7. C simple program construction

Most applications, however, 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 in *Coding Your Application → For C Programmers*. Note that 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.

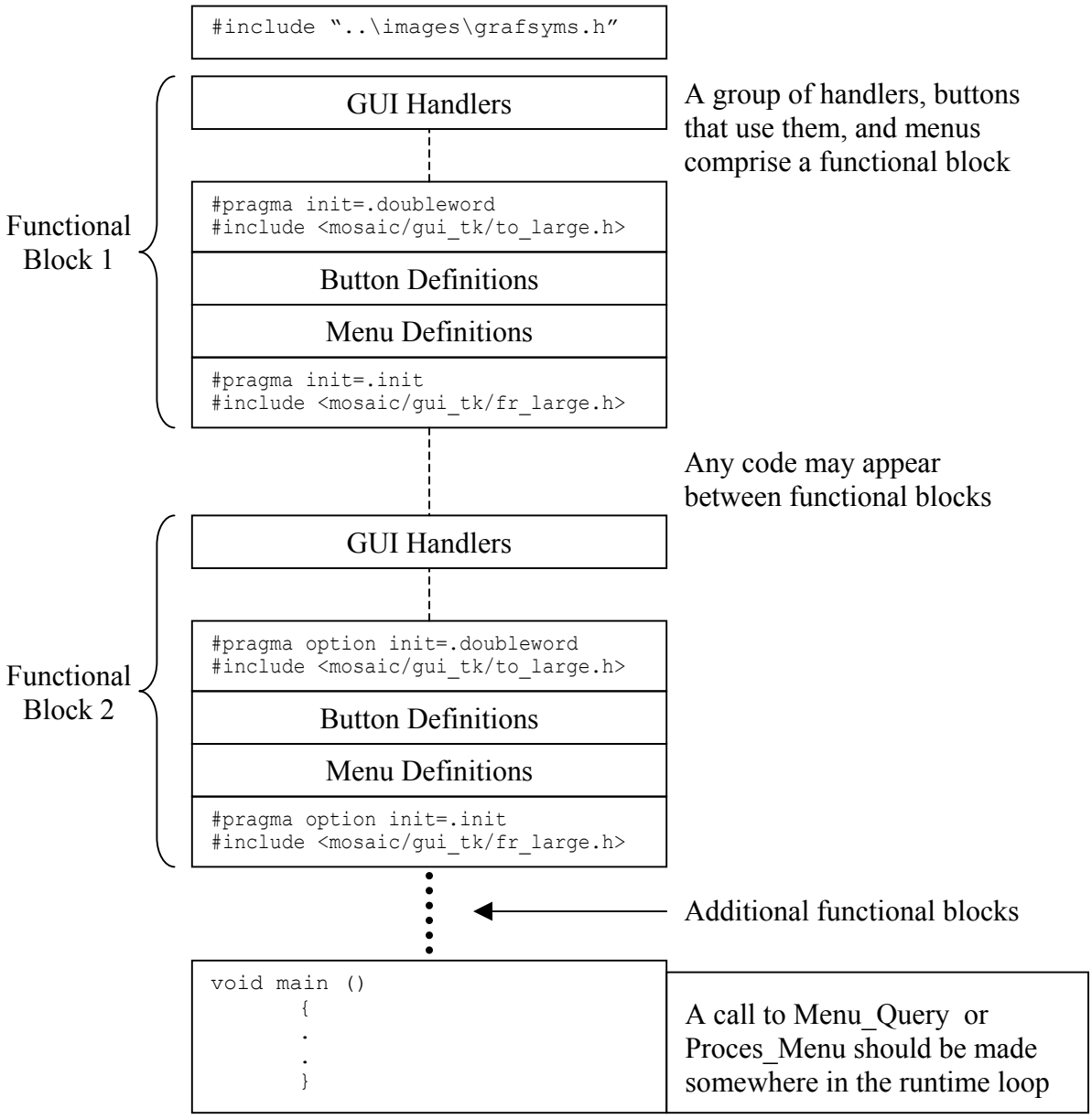


Figure 8. C complex program construction

Notes on Updated Drivers

The QED Board'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 Menu_Query. The touchscreen may be read using the built-in kernel functions, AskKeypad(), AskKeypress(), and Keypad(), or in Forth, ?keypad, ?keypress, and keypad. However, it should not be necessary to call these keypad drivers directly in most cases.

Old function name	Replacement	Comment
UpdateDisplay	Update_Graphics 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 Clear_Text	
Is_Display	Config_Display	
InitDisplay	Init_Display	See Config_Display in the GUI Toolkit *Glossary of Functions*
DisplayOptions	Set_Display_State Set_Cursor_State	
IsDisplayAddress	Obsolete	Part of Update_Here_With
DISPLAY_HEAP	tvars.display_heap_top tvars.display_heap_bottom	See tvars in GUI Toolkit *Glossary of Terms*
LinesPerDisplay CharsPerDisplayLine	tvars.graphic_rows tvars.text_rows tvars.graphic_cols tvars.text_cols	See tvars in GUI Toolkit *Glossary of Terms*
CharToDisplay 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 *QED Forth/C Glossary*
PutCursor	none	Positions the cursor. See its entry in the *QED 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 QED Board to span across multiple 32 Kbyte pages of memory. Consult the "Getting Started with the QED Board Using the Control C Programming Language" manual for a complete description of multipage compilation. 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, grafsyms.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.txt which is downloaded to the QED Board.

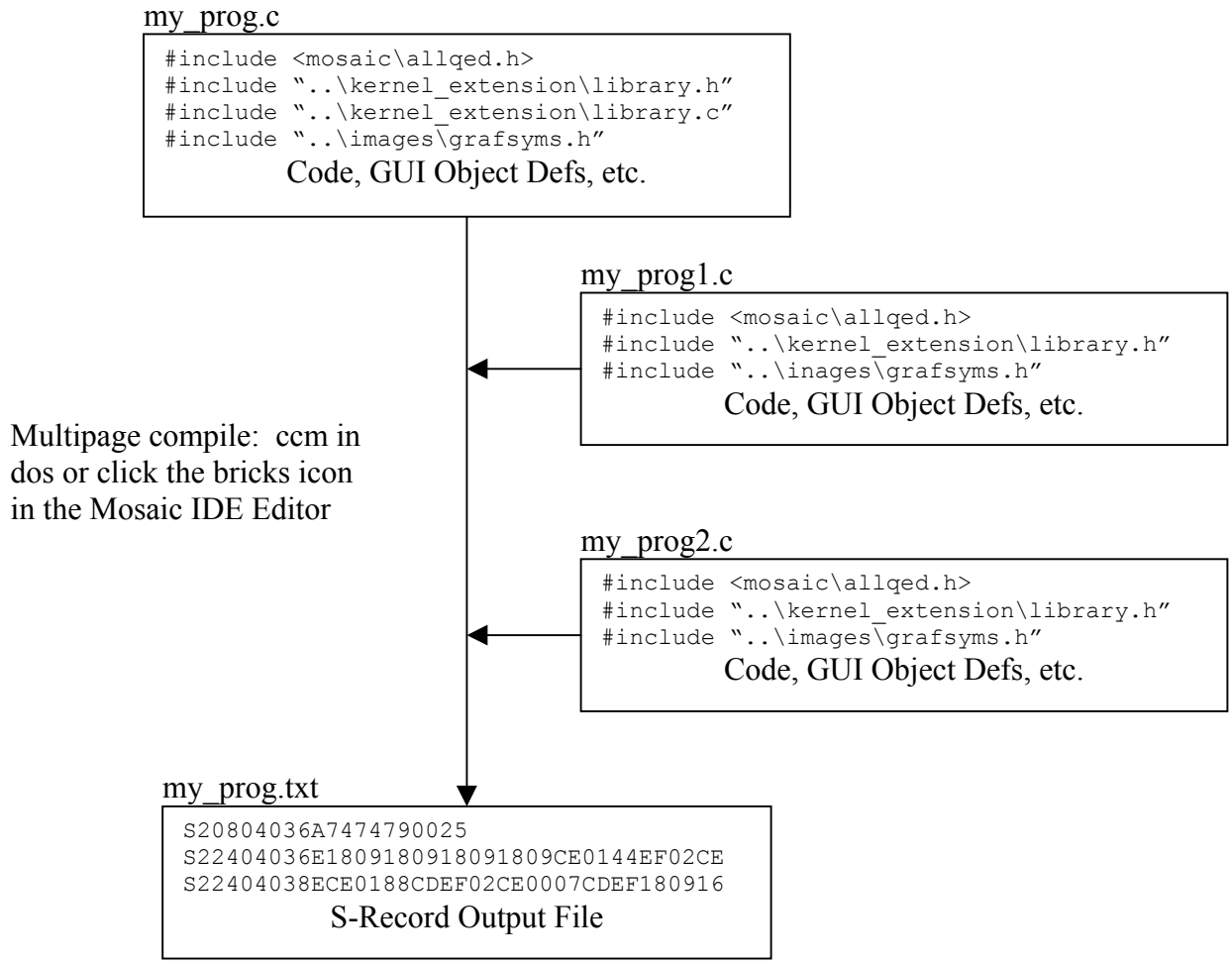


Figure 9. Multipage C programming diagram

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 Panel-Touch 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 QED Board, 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 *Glossary of Terms.* 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.

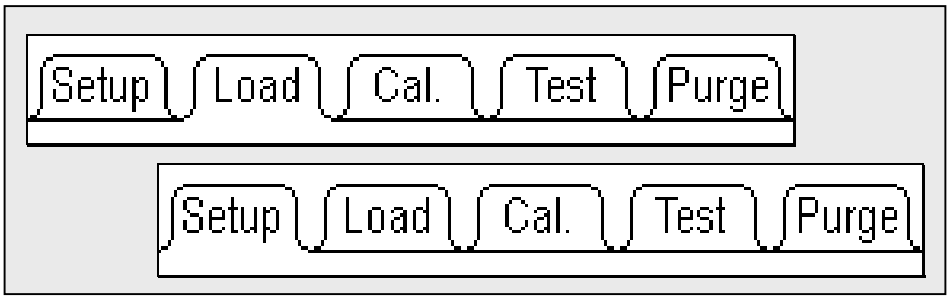


Figure 10. File tab selectors

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 *Glossary of Functions*). 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.

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

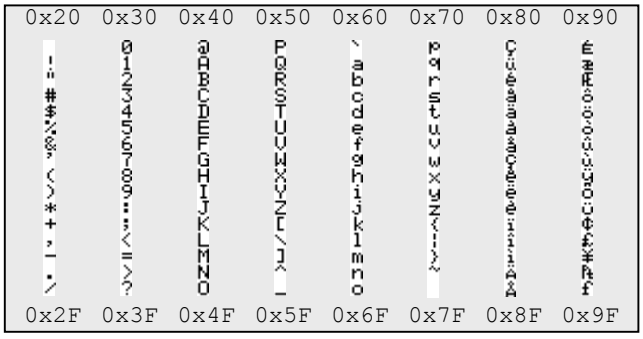


Figure 11. LCD Built in ROM font for text mode

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? Yes 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 GUI_Toolkit_PTC\examples\C\modal\modal.c in your IDE workspace. 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 areusing other drivers or 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
// into the my_kernel_extensions directory of your workspace directory
// #include "..\..\my_kernel_extensions\library.h"
// #include "..\..\my_kernel_extensions\library.c"

#include "PCX\GRAFSYMS.H"   // NOTE: DOWNLOAD .\PCX\GRAFIX.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
information
    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 the 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
pointer

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 the
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 the touchscreen hardware
and vars

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

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

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 GUI_Toolkit_PTC\examples\C\fonts\custchar.c in your IDE workspace directory. The graphics used are in the \examples\C\fonts\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 areusing other drivers or 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
// into the my_kernel_extensions directory of your workspace directory
// #include "..\..\my_kernel_extensions\library.h"
// #include "..\..\my_kernel_extensions\library.c"

#include "pcx\grafsyms.h"   // The graphics symbol file generated from
                            // the pcx converter
                            // The font graphics in grafix.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 the
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 the area starting at
1800.
```

```
    // Also, the TC6963C has a 0x20 ascii offset.  That is, all ascii
codes
    // must have 0x20 subtracted from them before being sent to the
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
```

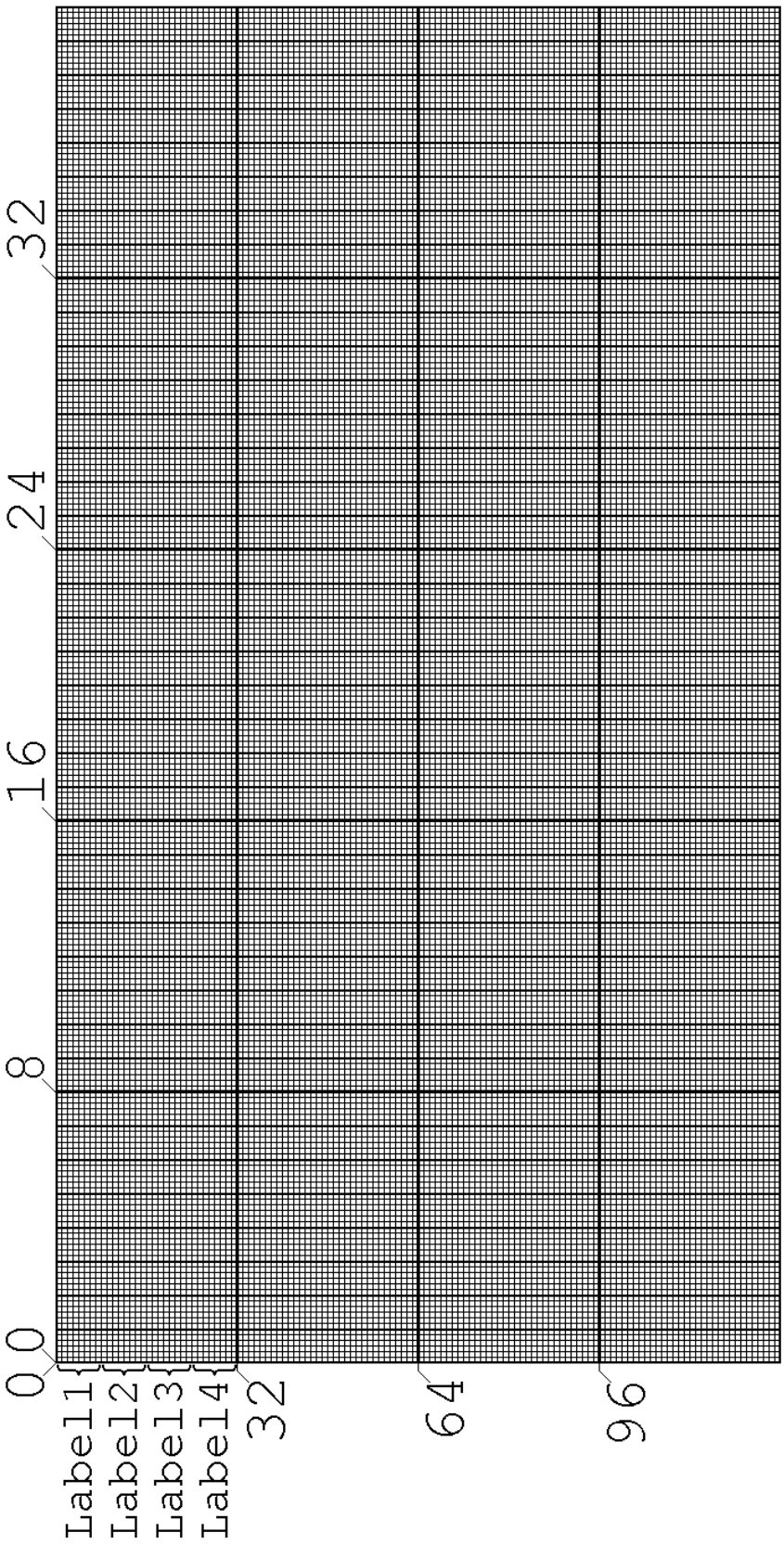
```
    init_lcd();                          // initialize the display
    installfonts();                      // upload the fonts to the
display

    StringToDisplay("Testing the 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 the 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);       // the 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 page contains 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.



LCD Worksheet

Appendix 4 – Example Code

C Language Example Program

```
// June 10, 1999          (C)Mosaic-Industries, Inc. 1999 (510) 790-1255
//                       written by: Jeremy Wade; translation & update: MGD

// ********************************************************
// ******          Example user interface
// ******  This program uses the Gui Software Toolkit to produce a user interface
// ******  on a SmarTouch Controller or a touchscreen/graphics display connected
// ******  to a QED board.  The Gui Software Toolkit must already be installed on
// ******  the board.
// ******  SEE THE BOTTOM OF THIS FILE FOR AN OVERVIEW OF THE DEMO APPLICATION
// ******  AND HINTS ON INSTALLING THE DEMO SOFTWARE.

// ******  SEE THE ACCOMPANYING HOW_TO.TXT FILE FOR ADDITIONAL INSTALLATION
// ******  AND "HOW TO DO IT" INFORMATION.


#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 areusing other drivers or 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
// into the my_kernel_extensions directory of your workspace directory
// #include "..\..\my_kernel_extensions\library.h"
// #include "..\..\my_kernel_extensions\library.c"


// ****  Graphics symbols - Generated by PCX2QED.exe utility program
#include "..\images\GRAFSYMS.H"

// General program constants and variable definitions:

#define BOTTOM_OF_HEAP 0X0F4800L      // Init constants defining the heap
#define TOP_OF_HEAP    0X0F7FFFL

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.

// ************************************************************
// ****
// ****    Beginning of keypad module
// ****

    // The keypad module consists of a set of alphanumeric buttons that
    // comprise 2 menus.  Num_keypad_menu is a floating/integer numerical
    // data input screen. Alnum_keypad_menu is a alphanumeric data input screen.

// ********************************************************
// **
// **    Keypad variables and constants
// **

// The following 6 variables must be initialized before using either keypad menu.
```

```
static uint keypad_text_col;    // Col for text string being edited by the keypad
static uint keypad_text_row;    // Row for text string being edited by the keypad
static void (*enter_handler_cfa)();  // variable that holds pointer to a function.
                                // which is the handler for the keypad.
                                // DEFAULT = no_op(), a do-nothing function.
static uint kpbuffer_length;    // Holds the max length of kpbuffer.  This is
                                // also the number of spaces that will be printed
                                // over the string to refresh it.
static uint kpbuffer_count ;    // Holds the current number of chars in kpbuffer.

static char** kpbuffer;         // Holds the addr of the string to be edited
                                // by the keypad.
    // kpbuffer holds a pointer to the base of the result string
    // created by the alphanumeric button presses.


static int shift_offset;        // Keeps track of the
                                // use of the shift key for the 0 key.
                                // Points into text string to indicate current
                                // char to be selected by the button.

static int alpha_offset;        // Keeps track of the
                                // use of the shift key for the 1-9 keys.
                                // Points into text string to indicate current
                                // char to be selected by the button.
                                // Constrained to equal 0 in numeric mode.

static int current_case;        // The current case of the letters in the alpha
                                // numeric keypad.

static int alpha;               // Boolean to indicate whether we allow alpha
                                // mode.  It is modified by the 2 top level menu
                                // init functions, init_alnum_keypad and
                                // init_num_keypad.

// define ten string buffers;
// max string length = button width + 1{for null terminate}.

char num0text[ BUTTON_WIDTH+1 ];        // String pointed to by 0 button
           // note that button 0 text is not displayed; see init_num0_text().
char num1text[ BUTTON_WIDTH+1 ];        // String pointed to by 1 button
char num2text[ BUTTON_WIDTH+1 ];        // String pointed to by 2 button
char num3text[ BUTTON_WIDTH+1 ];        // String pointed to by 3 button
char num4text[ BUTTON_WIDTH+1 ];        // String pointed to by 4 button
char num5text[ BUTTON_WIDTH+1 ];        // String pointed to by 5 button
char num6text[ BUTTON_WIDTH+1 ];        // String pointed to by 6 button
char num7text[ BUTTON_WIDTH+1 ];        // String pointed to by 7 button
char num8text[ BUTTON_WIDTH+1 ];        // String pointed to by 8 button
char num9text[ BUTTON_WIDTH+1 ];        // String pointed to by 9 button
char ptr_text[ BUTTON_WIDTH+1 ];        // Contains ^ ptr; used by num 1-9 buttons
char ptr0_text[ BUTTON_WIDTH+1 ];       // String pointed to by num0 button

// for high level handlers:
#define MAIN_TITLE_MAXLEN 23   // Max length for the main title
#define MAX_FLOW_LENGTH   11   // Sets max length of the numeric string
#define MAX_POWER_LENGTH  10   // Sets max length of the numeric string

#define MIN_FLOW  0.1
#define MAX_FLOW  1000.0

static int   pump_power;              // 0-100 % pump power
static float flow_limit;              // The floating point L/min of the pump
```

```
static char tmpbuff[MAX(MAX_POWER_LENGTH,MAX_FLOW_LENGTH)+1];
                                        // a temporary buffer for editing the power

static char main_title[MAIN_TITLE_MAXLEN];  // Declare the title buffer


// **
// **    End of keypad variables and constants
// **
// ************************************************************


// ************************************************************
// **
// **    Keypad Handler and support code
// **


// declare prototypes here for handler functions that must be referenced before
// they are defined; the functions should not accept nor return any parameters:
extern _Q void kp_text_refresh( void );
extern _Q void s_title_handler( void );
extern _Q void s_flow_handler( void );
extern _Q void s_power_handler( void );
extern _Q void increase_handler( void );
extern _Q void decrease_handler( void );

void no_op( void )
// A do-nothing function; we use it to initialize enter_handler_cfa
{
}

_Q int char_count( char* stringname )
// counts the number of characters in a string before the terminating null.
{
  int numchars;
  for( numchars=0; *stringname++; numchars++ )
    ;
  return numchars;
}


_Q void blank_keypad_chars( void )
// Places null text in all the text fields for the buttons
{
  num1text[0]=0;
  num2text[0]=0;
  num3text[0]=0;
  num4text[0]=0;
  num5text[0]=0;
  num6text[0]=0;
  num7text[0]=0;
  num8text[0]=0;
  num9text[0]=0;
}

_Q void set_lowercase( void )
// Initializes the letters in the keypad to their lower case versions.
{
  current_case = FALSE;                     // Set the current case var
  strcpy( num1text, "   qz " );
  strcpy( num2text, "   abc" );
```

```
  strcpy( num3text, "   def" );
  strcpy( num4text, "   ghi" );
  strcpy( num5text, "   jkl" );
  strcpy( num6text, "   mno" );
  strcpy( num7text, "   prs" );
  strcpy( num8text, "   tuv" );
  strcpy( num9text, "   wxy" );
}


_Q void set_uppercase( void )
// Initializes the letters in the keypad to their upper case versions.
{
  current_case = TRUE;                      // Set the current case var
  strcpy( num1text, "   QZ " );
  strcpy( num2text, "   ABC" );
  strcpy( num3text, "   DEF" );
  strcpy( num4text, "   GHI" );
  strcpy( num5text, "   JKL" );
  strcpy( num6text, "   MNO" );
  strcpy( num7text, "   PRS" );
  strcpy( num8text, "   TUV" );
  strcpy( num9text, "   WXY" );
}


_Q void set_char_pointer( void )
    // Positions the ^ pointer underneath the alpha strings in each alpha
    // button.  The value in alpha_offset is used to locate the ^.
{
  char* which_letter;       // holds string containing ^ at proper position
  switch( shift_offset )
  {
    case 0:
        which_letter = "      " ;        // Point to no letter
        break;
    case 3:
        which_letter = "   ^  " ;        // Point to first letter
        break;
    case 4:
        which_letter = "    ^ " ;        // Point to second letter
        break;
    case 5:
        which_letter = "     ^" ;        // Point to third letter
        break;
  }
  if(alpha)
    strcpy( ptr_text, which_letter );     // Store in ptr_text if mode = alpha
  else
    ptr_text[0] = 0;                      // Nothing displayed in numeric mode
  strcpy( ptr0_text, which_letter );      // Always store in ptr0_text
}

_Q void init_num0_text( void )
// Initializes the text label for the 0 button of the keypad.
// this button is handled separately from the others:
// it displays a graphic for readability, and the text initialized here
// is not displayed; this text IS used to build up the number string.
{
  strcpy( num0text, "   +-E" );
}
```

```
_Q void short_beep( void )
{
  SetHighCurrent(0x01);
  MicrosecDelay(2000);
  ClearHighCurrent(0x01);
}


_Q void num_shift_handler( void )
// Handles the pressing of the shift key.  Pressing the shift key
// causes the ^ indicator to rotate through the alternate character positions.
// Note that in numeric mode, only the 0 key has alternate characters.
{
  short_beep();
    switch( shift_offset )        // adjust the value of shift_offset
    {
      case 0:
        shift_offset=3;           // If it is 0, take it to 3 {1st char}
        break;
      case 3:
        shift_offset=4;           // If it is 3, take it to 4 {2nd char}
        break;
      case 4:
        shift_offset=5;           // If it is 4, take it to 5 {3rd char}
        break;
      case 5:                     // If it is 5, decide what to do...
        if(alpha)
        {   if(current_case)      // If caps, go to lower and start over with num
            {   set_lowercase();
                shift_offset=0;   // Roll over: back to the number mode
            }
            else                  // else if we're in lower case now...
            {   set_uppercase();
                shift_offset=3;   // Roll over to 1st char and go to caps
            }
        }
        else                      // if it is 5 in numeric mode...
            shift_offset=0;       // take it to 0 {select the number}
        break;
    }                             // end of switch statement
  if(alpha)
        alpha_offset = shift_offset;
  set_char_pointer();
  kp_text_refresh();            // Refresh the button text since it has changed
  Update_Text( TVARS );
}


_Q void refresh_string_display( void )
// Refresh the alphanumeric string on the display
// as specified by the current contents pointed to by kpbuffer.
{ xaddr buffer_base =        // row, column in garray are swapped intentionally
    ARRAYMEMBER( keypad_text_col, keypad_text_row, (FORTH_ARRAY*) GARRAY_XPFA);
  FillMany( buffer_base, kpbuffer_length, 0 );   // Clear string area on screen
  STRING_TO_DISPLAY( kpbuffer, keypad_text_row, keypad_text_col);
                               // Write the string at *kpbuffer to the display
  Update_Text( TVARS );         // Update the text layer
  PutCursor( keypad_text_row, keypad_text_col + kpbuffer_count ); // Reposition
}


_Q void num_ent_handler( void )
// Handles the "enter" button press.  The intended result of pressing
```

```
// the enter button is typically context sensitive (that is, it may vary
// depending on how the programmer intends to use the keypad).
// Consequently, we simply execute another handler via the ram-based function
// pointer named enter_handler_cfa that may be changed at runtime.
// (A "cfa" is a "code field address", another name for a "function pointer".)
// For example, see init_kp_vars() and enter_handler() later in this file.
// NOTE that the use of THIS_PAGE means that enter_handler()
// must be defined IN THIS FILE to ensure that the handler is located
// on the same page in memory as this num_ent_handler calling function.
// Also note that if the keypad is used without first initializing
// this variable, the board will probably crash when the enter button is pressed.
{
  short_beep();
  Execute( enter_handler_cfa, THIS_PAGE );  // call the specified function
}

_Q void num_del_handler( void )
// Handles the pressing of the delete button (shown as a <- ) on the number pad.
{
  if(kpbuffer_count)            // Only proceed if string is not empty
  {
    short_beep();               // If the cursor moves, beep
    (kpbuffer)[ kpbuffer_count-1 ] = 0;     // store terminating null
    kpbuffer_count--;           // decrement the count
    refresh_string_display();   // Refresh the string on the display
  }
}


_Q void num_handler( char ascii_value )
// This is the action routine called when a number button is pressed.
// Appends n to the end of the string at *kpbuffer, increments its count, beeps.
{
  // Comment out the following 2 lines if you do not want to revert to number
  // mode after each keystroke:
  alpha_offset = shift_offset = 0; // This resets the shift mode
  set_char_pointer();              // Regenerate the '^' pointer

  kp_text_refresh();            // Refresh the button labels
  if(kpbuffer_count >= kpbuffer_length)  // Don't overflow allotted string space
    return;                     // The buffer is full.  Reject the number.
  short_beep();                 // If we are able to print the char, beep
  (kpbuffer)[ kpbuffer_count ] = ascii_value;  // store new char in buffer
  kpbuffer_count++;             // Adjust the string count
  (kpbuffer)[ kpbuffer_count ] = 0;       // store terminating null
  refresh_string_display();     // Refresh the string on the display
}


_Q void num_0_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(shift_offset)
    num_handler( *(num0text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '0' );
}

_Q void num_1_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num1text + shift_offset));  // char from button's text string
```

```
  else                        // offset == 0 means we're in numeric mode
    num_handler( '1' );
}

_Q void num_2_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num2text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '2' );
}


_Q void num_3_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num3text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '3' );
}


_Q void num_4_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num4text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '4' );
}


_Q void num_5_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num5text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '5' );
}


_Q void num_6_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num6text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '6' );
}


_Q void num_7_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num7text + shift_offset));  // char from button's text string
  else                        // offset == 0 means we're in numeric mode
    num_handler( '7' );
}
```

```
_Q void num_8_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num8text + shift_offset));  // char from button's text string
  else                      // offset == 0 means we're in numeric mode
    num_handler( '8' );
}


_Q void num_9_handler( void )
// adds the selected ascii character to *kpbuffer and beeps
{
  if(alpha_offset)
    num_handler( *(num9text + shift_offset));  // char from button's text string
  else                      // offset == 0 means we're in numeric mode
    num_handler( '9' );
}


_Q void num_dec_handler( void )
// handles press of '.' (decimal point/period) button.
{
  num_handler( '.' );           // Tell num_handler about the button pressed
}


// **
// **    End of Keypad Handlers and support code
// **
// ************************************************************


// ************************************************************
// **
// **    Begin button definitions
// **


#pragma option init=.doubleword    // put initialized structs in ROM area!
#include </mosaic/gui_tk/to_large.h>

// Buttons for the keypad menu:
// Each FASTBUTTON macro statement initializes a structure in ROM to indicate
// the action flags, graphics, and text associated with the button.
// "FASTBUTTON" indicates that these are "direct draw" buttons;
// that is, when the button is pressed, the "on state" of the button
// (typically indicated by "blacking out" the button area")
// is drawn directly to the display, bypassing the graphics array in
// the QED RAM.  Similarly, when the button is released, another direct draw
// to the screen is executed.  Only the initial draw is done via the
// GARRAY in the QED RAM.
// Because the display of the press and release graphics are transitory
// and because the final button graphical display after press and release
// is (typically) the same as before the press/release actions,
// the direct draw speeds response time without compromising any other
// performance aspects.
// The graphic elements are denoted by _PCX names, and are
// #included via the "grafsyms.h" file.
// Note that each button is assigned a unique name, such as num0_button,
// which is the last parameter in the macro's parameter list.
// These names are in turn used by the NEW_MENU macro to assemble the
```

```
// buttons into one or more menus later in this file.

FASTBUTTON(                     // "Number 0" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM0_PCX,                       // Draw graphic
NUM0_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_0_handler,                  // The function to execute when pressed
"",                             // No text for the first line
"",                             // No text for the second line
ptr0_text,                      // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num0_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 1" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM1_PCX,                       // Draw graphic
NUM1_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_1_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num1text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num1_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 2" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM2_PCX,                       // Draw graphic
NUM2_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_2_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num2text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num2_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 3" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM3_PCX,                       // Draw graphic
NUM3_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_3_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num3text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num3_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 4" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM4_PCX,                       // Draw graphic
```

```
NUM4_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_4_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num4text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num4_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 5" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM5_PCX,                       // Draw graphic
NUM5_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_5_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num5text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num5_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 6" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM6_PCX,                       // Draw graphic
NUM6_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_6_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num6text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num6_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 7" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM7_PCX,                       // Draw graphic
NUM7_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_7_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num7text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num7_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 8" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM8_PCX,                       // Draw graphic
NUM8_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_8_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num8text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
```

```
num8_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "Number 9" button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG |                   //     this is a repeating key
DRAW_TEXT_FLAG,                 //     print text for this button
NUM9_PCX,                       // Draw graphic
NUM9_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
num_9_handler,                  // The function to execute when pressed
"",                             // No text for the first line
num9text,                       // The string to be printed inside this button
ptr_text,                       // The text string that contains the ^ pointer
"",                             // No text for the 4th line
num9_button );                  // Instantiate new button with above parameters

FASTBUTTON(                     // "." (decimal point/period) button
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG,                    //     this is a repeating key
NUMDEC_PCX,                     // Draw graphic
NUMDEC_PCX,                     // Release graphic
SBLACK_PCX,                     // Press graphic
num_dec_handler,                // The function to execute when pressed
"",                             // No text for this button
"",
"",
"",
numdec_button );                // Instantiate new button with above parameters

FASTBUTTON(                     // "Delete" button, drawn as <--
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG,                    //     this is a repeating key
NUMDEL_PCX,                     // Draw graphic
NUMDEL_PCX,                     // Release graphic
SBLACK_PCX,                     // Press graphic
num_del_handler,                // The function to execute when pressed
"",                             // No text for this button
"",
"",
"",
numdel_button );                // Instantiate new button with above parameters


FASTBUTTON(                     // "Enter" button
PRESS_HANDLER_FLAG,             // Draw graphic flag: execute action upon press
NUMENT_PCX,                     // Draw graphic
NUMENT_PCX,                     // Release graphic
SBLACK_PCX,                     // Press graphic
num_ent_handler,                // The function to execute when pressed
"",                             // No text for this button
"",
"",
"",
nument_button );                // Instantiate new button with above parameters

FASTBUTTON(
PRESS_HANDLER_FLAG |            // Draw graphic flag: execute action upon press
REPEAT_FLAG,                    //     this is a repeating key
SHIFT_PCX,                      // Draw graphic
SHIFT_PCX,                      // Release graphic
SBLACK_PCX,                     // Press graphic
num_shift_handler,              // The function to execute when pressed
"",                             // No text for this button
```

```
"",
"",
"",
numshift_button );              // Instantiate new button with above parameters


// **
// **        End of keypad menu buttons
// **
// ************************************************************

// ************************************************************
// **
// **        Keypad menu definitions
// **

// Here, we define the alphanumeric keypad menu; it allows the user to
// create or modify a string comprising both letters and numbers.
// The NEW_MENU macro initializes an array of structs in ROM that references
// each of the buttons in the menu.  The declared size of the array
// (which is [14] in this example) MUST equal the total number of declared
// buttons and graphics in the menu (in other words, the array size equals
// the number of ADD_ macros in the NEW_MENU initializer list). COUNT CAREFULLY!
// The buttons were defined and named above using the FASTBUTTON() macro.
// Each named button is added to the NEW_MENU
// by the ADD_TOUCH_BUTTON() macro, whose parameters are the upper left
// column and row location, the draw mask, and the button name, respectively.
// Note that NO SEMICOLON IS PLACED AFTER THE CALL TO ADD_TOUCH_BUTTON().
// RATHER, A COMMA IS PLACED AFTER EACH CALL EXCEPT THE LAST ONE IN THE LIST.
// This is because the macro is part of an initializer list, not a set of
// statements.
// A SEMICOLON FOLLOWS THE TERMINATING } OF THE NEW_MENU INITIALIZER STATEMENT.
// The row position for buttons that contain text has been carefully chosen
// so that the text is properly positioned inside the button;
// the standard text  has a height of 8 pixel lines (7 for the character,
// and 1 for the blank bottom pixel line that separates one line of text
// from the next.)

// Following each NEW_MENU() macro is a BUILD_MENU()
// macro that creates a FORTH_CONST_ARRAY struct; this is necessary
// for the underlying touchscreen software to properly access the data.
// Note that the menu name used by NEW_MENU and BUILD_MENU
// is modified by BUILD_MENU, which adds _menu to the name.
// For example, alnum_keypad is referred to as alnum_keypad_menu
// in all subsequent code.


//                (  Col, Row,  DRAW_MASK,  ButtonName   )

NEW_MENU alnum_keypad[14] =
{           // add buttons to the menu by initializing array of structs
// first column:
  ADD_TOUCH_BUTTON(  0,   38,  DRAW_MASK, numshift_button ),  // shift key
  ADD_TOUCH_BUTTON(  0,   70,  DRAW_MASK, numdec_button ),    // Decimal point
  ADD_TOUCH_BUTTON(  0,   102, DRAW_MASK, num0_button ),      // Numeral 0
// second column:
  ADD_TOUCH_BUTTON(  8,   38,  DRAW_MASK,  num7_button ),     // Numeral 7
  ADD_TOUCH_BUTTON(  8,   70,  DRAW_MASK,  num4_button ),     // Numeral 4
  ADD_TOUCH_BUTTON(  8,   102, DRAW_MASK,  num1_button ),     // Numeral 1
// third column:
  ADD_TOUCH_BUTTON(  16,  38,  DRAW_MASK,  num8_button ),     // Numeral 8
  ADD_TOUCH_BUTTON(  16,  70,  DRAW_MASK,  num5_button ),     // Numeral 5
  ADD_TOUCH_BUTTON(  16,  102, DRAW_MASK,  num2_button ),     // Numeral 2
```

```
// forth column:
  ADD_TOUCH_BUTTON(  24,  38,  DRAW_MASK,  num9_button ),     // Numeral 9
  ADD_TOUCH_BUTTON(  24,  70,  DRAW_MASK,  num6_button ),     // Numeral 6
  ADD_TOUCH_BUTTON(  24,  102, DRAW_MASK,  num3_button ),     // Numeral 3
// fifth column:
  ADD_TOUCH_BUTTON(  32,  70,  DRAW_MASK,  numdel_button ),   // Delete Key
  ADD_TOUCH_BUTTON(  32,  102, DRAW_MASK,  nument_button )    // Enter Key
};

BUILD_MENU( alnum_keypad, 14);
// allocates and initializes FORTH_CONST_ARRAY struct for the menu
// named alnum_keypad_menu which has 14 buttons(or graphics) added.
// The number of elements added (14 in this case) MUST MATCH
// the array size specified in the corresponding NEW_MENU declaration above.
// NOTE: because this macro adds _menu to the name;
// we refer to this menu as alnum_keypad_menu in the code below.




NEW_MENU num_keypad[14] =
{      // add buttons to the menu by initializing array of structs
//                   Col Row   DRAW_MASK  ButtonName
// first column
  ADD_TOUCH_BUTTON(  0,   06,  DRAW_MASK, num7_button ),     // Numeral 7
  ADD_TOUCH_BUTTON(  0,   38,  DRAW_MASK, num4_button ),     // Numeral 4
  ADD_TOUCH_BUTTON(  0,   70,  DRAW_MASK, num1_button ),     // Numeral 1
  ADD_TOUCH_BUTTON(  0,   102, DRAW_MASK, num0_button ),     // Numeral 0
// second column
  ADD_TOUCH_BUTTON(  8,   06,  DRAW_MASK, num8_button ),     // Numeral 8
  ADD_TOUCH_BUTTON(  8,   38,  DRAW_MASK, num5_button ),     // Numeral 5
  ADD_TOUCH_BUTTON(  8,   70,  DRAW_MASK, num2_button ),     // Numeral 2
  ADD_TOUCH_BUTTON(  8,   102, DRAW_MASK, numdec_button ),   // Decimal point
// third column
  ADD_TOUCH_BUTTON(  16,  06,  DRAW_MASK, num9_button ),     // Numeral 9
  ADD_TOUCH_BUTTON(  16,  38,  DRAW_MASK, num6_button ),     // Numeral 6
  ADD_TOUCH_BUTTON(  16,  70,  DRAW_MASK, num3_button ),     // Numeral 3
  ADD_TOUCH_BUTTON(  16,  102, DRAW_MASK, nument_button ),   // Enter Key
// forth column
  ADD_TOUCH_BUTTON(  24,  70,  DRAW_MASK, numshift_button ), // shift key
  ADD_TOUCH_BUTTON(  24,  102, DRAW_MASK, numdel_button )    // Delete key
};

BUILD_MENU( num_keypad, 14);
// allocates and initializes FORTH_CONST_ARRAY struct for the menu
// named num_keypad_menu which has 14 buttons(or graphics) added.
// The number of elements added (14 in this case) MUST MATCH
// the array size specified in the corresponding NEW_MENU declaration above.
// NOTE: because this macro adds _menu to the name;
// we refer to this menu as num_keypad_menu in the code below.


#include </mosaic/gui_tk/fr_large.h>

#pragma option init=.init   // return the initialized variable area to RAM;
                            // we're done defining buttons and menus for a while.


// **
// **        End of keypad menu definitions
```

```
// **
// ************************************************************


// ************************************************************
// **
// **        Top init code and redefinitions
// **

_Q void kp_text_refresh( void )
// redraw the text in each button of either alnum_keypad_menu (if in alpha mode)
// or of num_keypad_menu (if in numeric mode).
// Do_Menu() calls the handler, passing the DRAW_TEXTONLY_ACTION action flag,
// for each element (that is, each button) in the menu.
  {
  if(alpha)
    Do_Menu( 0, 0, TVARS, DRAW_TEXTONLY_ACTION, alnum_keypad_menu);
  else
    Do_Menu( 0, 0, TVARS, DRAW_TEXTONLY_ACTION, num_keypad_menu);

  }


_Q void init_alnum_keypad( void )
// This function places the alphanumeric menu named alnum_keypad_menu
// on the screen and initializes all associated variables
// except for the kpbuffer text string pointer and count,
// which must be initialized prior to calling this function.
// This allows editing of any string.  Since this string will be written to
// by the keypad functions, it must be located in modifiable RAM.
{
  set_lowercase();                // Load the button labels with lowercase
letters
  init_num0_text();               // Init the text string for the 0 key
  shift_offset = alpha_offset = 0; // point to numeric selections
  ptr_text[0] = 0;                // Initialize text pointer string to empty
  ptr0_text[0] = 0;               // Initialize num0 pointer string to empty
  alpha = TRUE;                   // alpha mode enabled
  Init_Menu( 0, 0, 0, TVARS, alnum_keypad_menu); // draw & install menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  Update_Graphics( TVARS );         // Update the graphics layer
  Update_Text( TVARS );             // Update the text layer
  Set_Cursor_State( TRUE, TRUE);    // Make the cursor visible
  refresh_string_display();         // Set cursor position and print the string
}


_Q void uninit_alnum_keypad( void )
// This function removes the alnum_keypad_menu from the screen and keymap
// array.  unint_menu erases all of the graphics and text that the menu drew
// and removes the buttons from the keymap array.
// Then Update_Graphics updates the display.
{
  Uninit_Menu( 0, 0, 0, TVARS, alnum_keypad_menu); // erase & remove menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  Update_Graphics( TVARS );          // Refresh the graphics layer
  Set_Cursor_State( FALSE, FALSE );  // Turn the cursor back off
  Clear_Text( TVARS );               // Clear all text from the display
}
```

```
_Q void init_num_keypad( void )
// This function places the numeric-only menu num_keypad_menu on the screen and
// initializes all associated variables except for the kpbuffer text string
// pointer and count, which must be initialized prior to calling this function.
// This allows editing of any string.  Since this string will be written to
// by the keypad functions, it must be located in modifiable RAM.
{
  alpha = FALSE;                    // alpha mode disabled
  blank_keypad_chars();             // Place zero length strings in num 1-9 labels
  init_num0_text();                 // Init the text string for the 0 key
  shift_offset = alpha_offset = 0; // point to numeric selections in numeric mode
  ptr_text[0] = 0;                  // Initialize text pointer string to empty
  ptr0_text[0] = 0;                 // Initialize num0 pointer string to empty
  Init_Menu( 0, 0, 0, TVARS, num_keypad_menu ); // draw & install menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  Update_Graphics( TVARS );          // Update the graphics layer
  Set_Cursor_State( TRUE, TRUE);     // Make the cursor visible
  refresh_string_display();          // Set cursor position and print the string
}

_Q void uninit_num_keypad( void )
// This function removes the num_keypad_menu from the screen and keymap
// array.  Calls Uninit_Menu to erase all of the graphics and text
// that the menu drew and remove the buttons from the keymap array.
// This function then updates the display.
{
  Uninit_Menu(0, 0, 0, TVARS, num_keypad_menu); // erase & remove menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  Update_Graphics( TVARS );           // Update the graphics layer
  Set_Cursor_State( FALSE, FALSE );  // Turn off the cursor visibility
  Clear_Text( TVARS );                // Clear all text from the display
}

_Q void init_kp_vars( void )
// Initializes all keypad related variables to 'safe' values.  This function
// should be called as part of the startup init code in the application.
{
  enter_handler_cfa = no_op;     // initialize handler
  keypad_text_col = 10;          // Default col location of text string
  keypad_text_row = 0;           // Default row location of text string
  kpbuffer_length = kpbuffer_count = 0;    // no contents
}

// **
// **        Top init code
// **
// ************************************************************


// ****
// ****    End of keypad module
// ****
// ********************************************************************



// ********************************************************************
// ****
// ****    Main menu module
// ****
```

```
// ********************************************************
// **
// **    Main screen Handlers and support code
// **




_Q void sht_dn_handler( void )
// handler for "SHUTDOWN" button; simply prints to the serial port.
{
  short_beep();
  printf( "Shut down button has been pressed\n" );
}

_Q void config_handler ( void )
// handler for "Config" button; simply prints to the serial port.
{
  short_beep();
  printf( "Config button has been pressed\n" );
}

_Q void stats_handler ( void )
// handler for "Stats" button; simply prints to the serial port.
{
  short_beep();
  printf( "Stats button has been pressed\n" );
}

_Q void exit_handler ( void )
// handler for "Exit" button; calls WARM to restart the program.
{
  short_beep();
  Warm();
}


// In this example, the "Exit" button executes a WARM restart.
// If the priority.autostart or autostart is installed, pressing the "Exit"
// button will restart the entire application.
// If no autostart is installed, pressing "Exit" returns to the QED-Forth monitor.

// **
// **    End of Main screen Handlers and support code
// **
// ********************************************************




// ************************************************************
// **
// **        Main screen button definitions
// **

#pragma option init=.doubleword    // put initialized structs in ROM area!
#include </mosaic/gui_tk/to_large.h>


// Buttons for the main screen

FASTBUTTON(                      // "ShutDown" button
```

```
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
SHT_DN_PCX,                     // Draw graphic
SHT_DN_PCX,                     // Release graphic
DBL_BLK_PCX,                    // Press graphic
sht_dn_handler,                 // The function to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
sht_dn_button);                 // Instantiate new button with above parameters

FASTBUTTON(                     // "Config" button
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
CONFIG_PCX,                     // Draw graphic
CONFIG_PCX,                     // Release graphic
SBLACK_PCX,                     // Press graphic
config_handler,                 // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
config_button);                 // Instantiate new button with above parameters

FASTBUTTON(                     // "Stats" button
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
STATS_PCX,                      // Draw graphic
STATS_PCX,                      // Release graphic
SBLACK_PCX,                     // Press graphic
stats_handler,                  // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
stats_button);                  // Instantiate new button with above parameters


FASTBUTTON(                     // "Exit" button
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
EXIT_PCX,                       // Draw graphic
EXIT_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
exit_handler,                   // Exit button executes a warm restart!!
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
exit_button);                   // Instantiate new button with above parameters


FASTBUTTON(                     // "Set Title" button
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
STITLE_PCX,                     // Draw graphic
STITLE_PCX,                     // Release graphic
LBLACK_PCX,                     // Press graphic
s_title_handler,                 // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
s_title_button);                // Instantiate new button with above parameters


FASTBUTTON(                     // "Power" button
```

```
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
SPOWER_PCX,                     // Draw graphic
SPOWER_PCX,                     // Release graphic
LBLACK_PCX,                     // Press graphic
s_power_handler,                // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
s_power_button);                // Instantiate new button with above parameters


FASTBUTTON(                     // "Set Flow" button
PRESS_HANDLER_FLAG,             // flag indicating when to execute the action
SFLOW_PCX,                      // Draw graphic
SFLOW_PCX,                      // Release graphic
LBLACK_PCX,                     // Press graphic
s_flow_handler,                 // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
s_flow_button);                 // Instantiate new button with above parameters


FASTBUTTON(                     // "Inc" button (for power adjustment)
PRESS_HANDLER_FLAG |            // flag indicating when to execute the action
REPEAT_FLAG,                    // flag indicating that this is a repeating key
INCB_PCX,                       // Draw graphic
INCB_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
increase_handler,               // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
increase_button);               // Instantiate new button with above parameters


FASTBUTTON(                     // "Dec" button (for power adjustment)
PRESS_HANDLER_FLAG |            // flag indicating when to execute the action
REPEAT_FLAG,                    // flag indicating that this is a repeating key
DECB_PCX,                       // Draw graphic
DECB_PCX,                       // Release graphic
SBLACK_PCX,                     // Press graphic
decrease_handler,               // The code to execute when pressed
"",                             // This button does not have text
"",                             // .
"",                             // .
"",                             // This button does not have text
decrease_button);               // Instantiate new button with above parameters



// **
// **        End of main screen button definitions
// **
// ************************************************************

// ************************************************************
// **
// **        Main screen menu definition
// **
```

```
// ADD_TOUCH_BUTTON and ADD_GRAPHIC parameter list:
//                   Col Row   DRAW_MASK    Button/graphic_Name
// ADD_BUTTON parameter list:
//     keymap_offset Col Row   ACTION_MASK  Button_Name

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);
// allocates and initializes FORTH_CONST_ARRAY struct named mmain_menu
// The number of elements added (12 in this case) MUST MATCH
// the array size specified in the corresponding NEW_MENU declaration above.
// NOTE: because this macro adds _menu to the name;
// we refer to this menu as mmain_menu in the code below.


#include </mosaic/gui_tk/fr_large.h>
#pragma option init=.init       // return the initialized variable area to RAM;
                                // we're done defining buttons and menus.

// **
// **        Main screen menu definition
// **
// ************************************************************

// ************************************************************
// **
// **        Top level init code
// **

_Q void update_main_text( void )
// Updates the display of the pump parameters on the main menu screen.
// For real time monitoring of an actual system, this routine might be executed
// as part of a loop in a separate task for periodic update of displayed data.
{
  char tempstring[2*MAX_FLOW_LENGTH];           // make buffer amply large
  STRING_TO_DISPLAY( "    %", 10, 25);          // Blank old value of pump_power
  sprintf( tempstring, "%3d", pump_power );     // pump_power -> 3digit string
  STRING_TO_DISPLAY( tempstring, 10, 25 );      // Display the pump power level
  sprintf( tempstring, "%10.2f", flow_limit ); // fieldwidth=10;2 digits after dp
  STRING_TO_DISPLAY( tempstring, 4, 24 );       // Display the flow limit
}
```

```
_Q void init_main_menu( void )
// This function places the mmain menu on the screen and loads the keymap
// array with the menu's buttons.
// Note that for this example, some displayed parameters are fixed dummy values,
// such as the input and output pressure, and pump RPM and temperature.
{
  Init_Menu( 0, 0, 0, TVARS, mmain_menu ); // draw & install menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  STRING_TO_DISPLAY( main_title, 0, 16);     // Display title
  STRING_TO_DISPLAY( "Flow Limit:                L/min", 4, 8); // flow units
  STRING_TO_DISPLAY( "Input pressure:        314 PSI", 5, 8);   // in pres units
  STRING_TO_DISPLAY( "Output pressure:       102 PSI", 6, 8);   // out pres units
  STRING_TO_DISPLAY( "Pump RPM:   1812  RPM", 13, 8);           // RPM units
  STRING_TO_DISPLAY( "Pump Temp:    92  C", 14, 8);             // Temp units
  update_main_text();              // Fill in the initial values
  Update_Text( TVARS );            // Update the text layer
  Update_Graphics( TVARS );        // Update the graphics layer
}


_Q void uninit_main_menu( void )
// This function removes the mmain_menu from the screen and keymap
// array.  Unint_Menu erases all of the graphics and text that the menu drew
// and removes the buttons from the keymap array.
// Then Update_Graphics updates the display.
{
  Uninit_Menu( 0, 0, 0, TVARS, mmain_menu ); // erase & remove menu
  //       (keymap_offset,col,row,tvars,menu_addr,menu_page) doesn't update
display
  Update_Graphics( TVARS );         // Update the graphics layer
  Clear_Text( TVARS );              // Clear all text from the display
}


_Q void kp_edittitle_exit( void )
// This function is used by the enter button of the keypad menu to
// restore the main menu after the title string has been edited.
{
  uninit_alnum_keypad();                      // uninitialize the keypad menu
  init_main_menu();                           // Initialize the main menu
  Update_Text( TVARS );                       // Update the text
  Update_Graphics( TVARS );                   // Update the graphics
  enter_handler_cfa = no_op;                 // Although this variable should...
}   // be initialized at each calling of the keypad menu, it is good to reset it.


_Q void kp_setflow_exit( void )
// This function is used by the enter button of the keypad menu to
// restore the the main menu after the flow rate string has been edited.
{
  float flow_limit_save = flow_limit;
  sscanf( kpbuffer, "%f", &flow_limit); // Attempt to convert the string as fp#
  if( (flow_limit < (MIN_FLOW)) || (flow_limit > (MAX_FLOW)) )
      flow_limit = flow_limit_save;      // If out of range, restore prior value
  uninit_num_keypad();                          // uninitialize the keypad menu
  init_main_menu();                             // Initialize the main menu
  enter_handler_cfa = no_op;
  // Although the enter_handler_cfa variable should be initialized at each
  // calling of the keypad menu, it is good to reset it.
  update_main_text();                           // Update the text value display
  Update_Text( TVARS );                         // Update the text display layer
```

```
  Update_Graphics( TVARS );                     // Update the graphics layer
}


_Q void kp_setpower_exit( void )
// This function is used by the enter button of the keypad menu to
// restore the the main menu after the power string has been edited.
{
  int pump_power_save = pump_power;
  sscanf( kpbuffer, "%d", &pump_power); // Attempt to convert string as int
  if( (pump_power < 0) || (pump_power > 100) )
      pump_power = pump_power_save;      // If out of range, restore prior value
    // note: increase_handler and decrease_handler clamp 0<= pump_power <=100
  uninit_num_keypad();                          // uninitialize the keypad menu
  init_main_menu();                             // Initialize the main menu
  enter_handler_cfa = no_op;
  // Although the enter_handler_cfa variable should be initialized at each
  // calling of the keypad menu, it is good to reset it.
  update_main_text();                           // Update the text value display
  Update_Text( TVARS );                         // Update the text display layer
  Update_Graphics( TVARS );                     // Update the graphics layer
}


_Q void s_title_handler( void )
{
  short_beep();
  kpbuffer = &main_title;                // kpbuffer points to title for editing
  kpbuffer_count = char_count(main_title); // find terminating null, calc count
  kpbuffer_length = MAIN_TITLE_MAXLEN;   // prevent buffer overrun.
  keypad_text_col = 16;                  // Col location of text string
  keypad_text_row = 3;                   // Row location of text string
  enter_handler_cfa = kp_edittitle_exit; // brings back main menu when enter
pressed
  uninit_main_menu();                    // Clear the current menu
  init_alnum_keypad();                   // init the alnum menu
                                         // Print a little help for the user:
  STRING_TO_DISPLAY( "Edit the main screen title field below", 0, 0);
  Update_Text( TVARS );                         // Update the text display layer
  Update_Graphics( TVARS );                     // Update the graphics layer
}

_Q void s_flow_handler( void )
{
  short_beep();
    // Before we call keypad menu, set up its environment:
  sprintf( tmpbuff, "%.2f", flow_limit );     // starting fp value to buffer
  kpbuffer = &tmpbuff;                          // Set keypad to edit tmpbuff
  kpbuffer_count = char_count(tmpbuff);         // find final null, calc count
  kpbuffer_length = MAX_FLOW_LENGTH;            // Set max length of the string
  keypad_text_col = 25;                         // Col location of text string
  keypad_text_row = 7;                          // Row location of text string
  enter_handler_cfa = kp_setflow_exit;
        // kp_setflow_exit saves the modified string and reverts to main menu.
        // Now that everything is set up, we can change menus:
  uninit_main_menu();                             // Clear the current menu
  init_num_keypad();                              // init the alnum menu
  STRING_TO_DISPLAY( "Edit the value", 0, 25 );    // User prompt
  STRING_TO_DISPLAY( "for exchanger", 1, 25 );
  STRING_TO_DISPLAY( "flow limit:", 2, 25 );
  STRING_TO_DISPLAY( "0.1<flow<1000", 3, 25 );
  Update_Text( TVARS );                         // Update the text display layer
  Update_Graphics( TVARS );                     // Update the graphics layer
```

```
}


_Q void s_power_handler( void )
{
  short_beep();
    // Before we call keypad menu, set up its environment:
  sprintf( tmpbuff, "%d", pump_power );        // starting int value to buffer
  kpbuffer = &tmpbuff;                         // Set keypad to edit tmpbuff
  kpbuffer_count = char_count(tmpbuff);        // find terminating null, calc
count
  kpbuffer_length = MAX_POWER_LENGTH;          // Set max length of the string
  keypad_text_col = 25;                        // Col location of text string
  keypad_text_row = 7;                         // Row location of text string
  enter_handler_cfa = kp_setpower_exit;
        // kp_setpower_exit saves the modified string and reverts to main menu.
        // Now that everything is set up, we can change menus:
  uninit_main_menu();                             // Clear the current menu
  init_num_keypad();                              // init the alnum menu
  STRING_TO_DISPLAY( "Edit the value", 0, 25 );    // User prompt
  STRING_TO_DISPLAY( "for pump power", 1, 25 );
  STRING_TO_DISPLAY( "percentage", 2, 25 );
  Update_Text( TVARS );                          // Update the text display layer
  Update_Graphics( TVARS );                      // Update the graphics layer
}

_Q void increase_handler( void )
{
  if(pump_power < 100)                            // Get the pump power %
  {  pump_power++;                                // If less than 100, increment
     short_beep();                                // Generate a click
     update_main_text();                          // Update the text value display
     Update_Text( TVARS );                        // Update the text display layer
  }
}


_Q void decrease_handler( void )
{
  if(pump_power > 0)                              // Get the pump power %
  {  pump_power--;                                // If more than 0, decrement
     short_beep();                                // Generate a click
     update_main_text();                          // Update the text value display
     Update_Text( TVARS );                        // Update the text display layer
  }
}


// **
// **        End of top level init code and redefinitions
// **
// ************************************************************


// ****
// ****      End of main menu module
// ****
// ************************************************************


// ************************************************************
// ****
// ****      Startup code
```

```
// ****


_Q void init_all( void )
// initializes all variables.  Sets initial default values of
// displayed flow_limit and pump_power.
{
  IsHeap( BOTTOM_OF_HEAP, TOP_OF_HEAP);   // Set up the heap
  init_kp_vars();                         // Initialize the keypad variables
  strcpy( main_title, "Heat Exchanger Unit 2");   // Init main menu title string
  flow_limit = 4.20;            // init the flow_limit variable
  pump_power = 53;              // init the pump_power variable
  StartTimeslicer();            // timeslicer must be on for repeating buttons
  Std_Display( TVARS );         // Config the display hardware
  Init_Display( TVARS );        // Initialize the display hardware
  Init_Touch( TVARS );          // Initialize the touchscreen hardware and vars
}


_Q void touch_mon( void )
// Infinite loop for getting screen touches.
// The PauseOnKey() statement is for development and demo use only;
// it allows you to quit the program by typing a "." (period)
// and then pressing a touchpad button; this reverts to the QED-Forth monitor
// at your terminal.  If you wanted to, you could then type
//  NO.AUTOSTART
// to ensure that the application did not start again.
// PauseOnKey() is not recommended for use in an actual application.
{
  while(1)                // infinite loop runs application
  { Menu_Query( TVARS );  // Get a single touch from the user
    PauseOnKey();         // Comment out this line in a real application!

  }
}


void main( void )
// This is the top level startup function. Call it and everything starts running.
{
  init_all();
  init_main_menu();               // Display the alnum menu
  touch_mon();                    // Loop forever while scanning the touchscreen
}


// Type the following code from your terminal if you want the
// application to automatically start upon each powerup or reset:

// CFA.FOR MAIN PRIORITY.AUTOSTART


// ****
// ****      End of startup code
// ****
// ************************************************************


// ******
// ******    End of example user interface
// ******
// ************************************************************
```

```
/* ****** START OF DOCUMENTATION SECTION ******


****** Overview of the Touchscreen Demo Application *******

PURPOSE OF THE DEMO PROGRAM

This application was designed to show how a user interface is put
together, and also to provide a guide to coding useful touchscreen
functions such as an alphanumeric keypad, a numeric keypad, and
up/down buttons to adjust a parameter.  The application integrates text with
graphics that were drawn on a desktop PC to create
an intuitive context-sensitive user interface.

For instructions on how to load the demo program if it is not
already present in your SmarTouch Controller, see the next section titled
"Loading the Demo Onto the Smartouch Controller", or read the "HOW_TO.txt" file.


TOP FOUR BUTTONS ON THE MAIN MENU

When the application starts, you'll see on your touchscreen a display with 4
buttons across the top, and 5 more buttons on the lower half of the screen.
The 3 buttons in the upper left of the screen labeled "Shutdown", "Config", and
"Stats" simply beep (to provide audio feedback)
and send a message to the serial port when pressed.
The messages are visible if you connect a terminal running at 9600 baud
to the Serial1 port on the SmarTouch Controller.  The "Exit" button at
the top right of the screen performs the action "WARM" which restarts the
software application if the priority.autostart or autostart has been configured;
otherwise, WARM calls the QED-Forth monitor.

USING THE NUMERIC KEYPAD TO SET THE FLOW LIMIT

Just below the "Shutdown" button is a "Flow Limit:" label and an associated
numeric field with units of "L/min" (Liters per minute).  The default initial
flow is 4.2.  To modify this value, press the "Set Flow" button.
You'll hear a beep and see a full-screen numeric keypad
that allows you to modify the flow.

The numeric keypad works in an intuitive manner.  There are ten number buttons
(0 to 9), a decimal point button, and three control buttons (shift, enter, and
delete).  Pressing any non-control button adds to the string and moves the cursor
to
the right; holding a button down adds multiple instances of the number to the
string.  The maximum length of the numeric string (set in the application software
in this file) is 10 characters, and attempts to add more characters beyond
the limit are ignored. The <-- button is a "delete" key that erases the last
character before the cursor and moves the cursor to the left.  Holding down
the delete key erases multiple characters (until there are no characters
left in the string, at which time the key does not beep or respond).

The buttons numbered 1 through 9 add their number to the string when pressed.
The 0 button adds a 0 when pressed, unless the shift key has been pressed
to advance the ^ pointer to one of the three alternate meanings for that key.
The alternates are "+" (plus), "-" (minus), and 10^x (10 raised to a power;
this causes an "E" to be inserted into the string, indicating a
scientific notation exponent is to follow).

After the user is satisfied with the string, pressing the "Enter" key
exits the numeric keypad and returns to the main menu.  If the numeric
string is convertible to a valid floating point number in the allowed range,
```

```
the new flow limit is displayed on the main menu.
The software rounds the number as appropriate
to display the value in the allotted space on the screen. If the user specifies
an invalid or out of range numeric string, the string is ignored and the prior
value of flow is displayed on the main menu.

USING THE ALPHANUMERIC KEYPAD TO SET THE TITLE

At the top of the main menu, just to the right of the large "Shutdown" button,
is a title string.  The default title is "Heat Exchanger Unit 2".
Pressing the "Set Title" button in the middle of the screen results in a
beep (for audible feedback), and displays an alphanumeric keypad
that allows you to modify the title.

The alphanumeric keypad was designed to work in an intuitive manner.
Ten numeric buttons each display a number (0 to 9) plus 3 alternate text
characters; the decimal point/period button does not have any alternate text.
Like the numeric keypad, the three control buttons are Shift, Delete (<--), and
Enter.

Pressing any non-control button adds to the string and moves the cursor to the
right; holding a button down adds multiple instances of the selected number
to the string.  The maximum length of the numeric string
is set in the application software in this file, and attempts to add characters
beyond the limit are ignored. The <-- button is a "delete" key that erases the
last character before the cursor and moves the cursor to the left.  Holding down
the delete key erases multiple characters (until there are no characters
left in the string, at which time the key does not beep or respond).

Initially, or after a prior non-control button press,
the buttons numbered 0 through 9 add their number to the string when pressed.
Pressing the shift key advances the ^ pointer to one of the three alternate
text characters for that button.  Pressing the shift key one to three times
selects the displayed characters; on the fourth press, the case (upper or lower)
of the displayed alternate text is reversed; this allows the user
to insert capital or lower case letters into the title string.

The alternates for the zero button are described in the previous section.
The alternate text characters for buttons 2 through 9 are arranged the same
as on a standard telephone keypad.  The alternates for key number 1 include the
characters that are not on a telephone, namely q, z, and blank (space).

In this implementation, each keypress resets the shift state to the numeric mode.
This can be changed by altering a bit of code in the num_handler() function.

After the user is satisfied with the string, pressing the "Enter" key beeps,
exits the numeric keypad, and returns to the main menu, displaying the new title.

USING THE NUMERIC KEYPAD TO SET THE POWER

Just to the right of the "Set Title" button is the "Power" button that
displays the power percentage.  Pressing this button brings up a numeric
keypad that lets you modify the power percentage in a way analogous
to setting the flow as described above.

USING THE INCREMENT AND DECREMENT BUTTONS TO SET THE POWER

The numeric value displayed in the "Power" button
can also be altered by pressing the "Inc" and "Dec" buttons. This illustrates
how easy it is to use a pair of buttons to select a parameter value.
Simply press the Inc or Dec button and watch the displayed power change.
The software clamps the value to the range 0 to 100.
```

```
CONCLUSION

This demonstration software provides a sample of the capabilities of the
graphical user interface software on the QED Board.  It shows how
to place buttons on the screen, create menus as assemblies of buttons
and graphics, and simultaneously manage text and graphics to create
an intuitive user interface.  This software can be used as
a template to guide the design of your customized user interface software.

Please read the file "HOW_TO.txt" for a discussion of the overall
procedure for implementing a graphical user interface.


************ LOADING THE DEMO ONTO THE SMARTOUCH CONTROLLER ************

This demonstration program is pre-loaded into new SmarTouch Controllers
that are shipped from Mosaic Industries.  If the program is already loaded
on your QED Board, you should see the main screen as described in the previous
section ("Overview of the Touchscreen Demo Application")  when your
SmarTouch Controller is turned on.  If you see the multi-button menu,
you're all set! You can skip this section.

If your board does not have this software installed, follow these simple steps
to load the Graphical User Interface (GUI) Toolkit software and the
demonstration program:

From your 9600 baud terminal communicating with the QED Board as explained
in the "Getting Started" manual, type
    COLD

You should see the "Coldstart" and "QED-Forth V4.01" messages.
If you do not see any text, try the factory cleanup procedure using
DIP swiches 6 and 7 as explained in "Getting Started".  If you are still
having trouble, or if the displayed Version number is not V4.01, call
Mosaic Industries.

Use your PC terminal program (for example, QEDTERM) to send the file named
    GTKLOAD.TXT
to the QED Board.  This file contains a Motorola hex memory image
plus flash programming commands that transiently compile code on pages 4 and 6,
and then move it to flash in page 7, and set up an AUTOSTART vector so that
the program will start automatically each time the SmarTouch Controller
is turned on.  The file's memory image contains the compiled GUI Toolkit,
plus the graphics and compiled code for this GUI_DEMO.C file.

To regain control of your board for other uses, simply do the factory
cleanup (flip dip switch 6 on, reset the board, flip switch 6 off, and reset
again).  Because the application and GUI firmware is stored "out of the way"
on page 7, you can freely use your board for development without corrupting
the GUI firmware, graphics images, or the demo application code itself
(as long as you don't explicitly program the flash on page 7 using TO.FLASH).

If the demo code is loaded on your QED Board, but you've erased the
autostart vector by performing the factory cleanup operation,
you can execute the demo program by typing the following at your terminal:
HEX  7FF4  7  X@  EXECUTE

In addition, you can restore the demo's autostart vector by typing
HEX  7FF4  7  X@  AUTOSTART
or
HEX  7FF4  7  X@  PRIORITY.AUTOSTART

This is possible because the xaddress of the demo program's
```

```
main() function is saved at location 7FF4 on page 7.

****** END OF DOCUMENTATION SECTION ******
*/
```

Forth Language Example Program

```
\ June 3, 1999          (C)Mosaic-Industries, Inc. (510) 790-1255
\                                              Jeremy Wade 1999

\ Revised 9/18/2003 for Kernel Extension compatibility

\ ********************************************************
\ ******          Example user interface
\ ******  This program uses the Gui Software Toolkit to produce a user
interface
\ ******  on a SmarTouch Controller or a touchscreen/graphics display
connected
\ ******  to a QED board.  The Gui Software Toolkit must already be
installed on
\ ******  the board and the Forth headers (gui_tk.4th) must be part of
the
\ ******  current vocabulary.

#include "../Kernel_Extension/library.4th"

anew gui_tk_demo                \ Re-cue the pointers

\ ********************************************************
\ ****
\ ****  Graphics symbols - Generated by pcx2qed
\ ****      The graphics symbols are usually generated from pcx2qed and
pasted
\ ****      into this area.
\ ****

hex

din 072000  xconstant INCB_PCX
din 0720a6  xconstant DECB_PCX
din 07214c  xconstant SBLANK_PCX
din 0721f2  xconstant SBLACK_PCX
din 072298  xconstant LBLANK_PCX
din 07238e  xconstant LBLACK_PCX
din 072484  xconstant NUM9_PCX
din 07252a  xconstant NUM1_PCX
din 0725d0  xconstant NUM2_PCX
din 072676  xconstant NUM3_PCX
din 07271c  xconstant NUM4_PCX
din 0727c2  xconstant NUM5_PCX
din 072868  xconstant NUM6_PCX
din 07290e  xconstant NUM7_PCX
din 0729b4  xconstant NUM8_PCX
din 072a5a  xconstant NUM0_PCX
din 072b00  xconstant STITLE_PCX
din 072bf6  xconstant NUMDEC_PCX
din 072c9c  xconstant NUMDEL_PCX
din 072d42  xconstant NUMENT_PCX
din 072de8  xconstant S_BUTTON_PCX
din 072e8e  xconstant EXIT_PCX
din 072f34  xconstant PUMP_PCX
din 0731e3  xconstant SFLOW_PCX
din 0732d9  xconstant CONFIG_PCX
din 07337f  xconstant SHT_DN_PCX
din 073575  xconstant STATS_PCX
din 07361b  xconstant DBL_BLK_PCX
din 073811  xconstant SHIFT_PCX
```

```
din 0738b7  xconstant SPOWER_PCX
din 0739ad  xconstant PWRLABEL_PCX

\ ****
\ ****    End Graphics symbols
\ ****
\ ************************************************************

\ General program constants and variable definitions follow

hex

4800 0F xconstant BOTTOM_OF_HEAP        \ Init constants defining the
heap
7FFF 0F xconstant TOP_OF_HEAP
gui_vars v.instance: tvars              \ Create an instance of the
touchscreen
                                        \ variable structure.  All
programs
                                        \ that use the Gui Toolkit must
declare
                                        \ an instance of this structure.

\ ****************************************************************
\ ****
\ ****    Beginning of keypad module
\ ****

    \ The keypad module consists of a set of alphanumeric buttons that
    \ comprise 2 menus.  Num_keypad_menu is a floating/integer numerical
    \ data input screen.  Alnum_keypad_menu is a alphanumeric data input
    \ screen.

\ ************************************************************
\ **
\ **    Keypad variables and constants
\ **

decimal

    \ The following 5 variables must be initialized before using either
keypad
    \ menu.
integer: keypad_text_col        \ Col for text string being edited by
the keypad
integer: keypad_text_row        \ Row for text string being edited by
the keypad
xaddr:   enter_handler_xcfa     \ The xcfa executed by the enter button
                                \ handler for the keypad
integer: kpbuffer_length        \ Holds the max length of kpbuffer.
This is
                                \ also the number of spaces that will be
printed
                                \ over the string to refresh it.
xaddr:   kpbuffer               \ Holds the xaddr of the string to be
edited
                                \ by the keypad


integer: shift_offset           \ Self fetching variable to keep track
of the
                                \ use of the shift key for the 0 key
```

```
integer: alpha_offset           \ Self fetching variable to keep track
of the
                                \ use of the shift key for the 1-9 keys

integer: current_case           \ The current case of the letters in the
alpha
                                \ numeric keypad

integer: alpha                  \ Boolean to indicate whether we allow
alpha
                                \ mode.  It is set by the 2 top level
menu
                                \ init functions, init_alnum_keypad and
                                \ init_num_keypad.

BUTTON_WIDTH 1+ v.instance: num0text    \ String pointed to by 0 button
BUTTON_WIDTH 1+ v.instance: num1text    \ String pointed to by 1 button
BUTTON_WIDTH 1+ v.instance: num2text    \ String pointed to by 2 button
BUTTON_WIDTH 1+ v.instance: num3text    \ String pointed to by 3 button
BUTTON_WIDTH 1+ v.instance: num4text    \ String pointed to by 4 button
BUTTON_WIDTH 1+ v.instance: num5text    \ String pointed to by 5 button
BUTTON_WIDTH 1+ v.instance: num6text    \ String pointed to by 6 button
BUTTON_WIDTH 1+ v.instance: num7text    \ String pointed to by 7 button
BUTTON_WIDTH 1+ v.instance: num8text    \ String pointed to by 8 button
BUTTON_WIDTH 1+ v.instance: num9text    \ String pointed to by 9 button
BUTTON_WIDTH 1+ v.instance: ptr_text    \ String pointed to by num 1-9
buttons
BUTTON_WIDTH 1+ v.instance: ptr0_text   \ String pointed to by num0
button

\ **
\ **    End of keypad variables and constants
\ **
\ ************************************************************


\ ************************************************************
\ **
\ **    Keypad Handler and support code
\ **

decimal

: blank_keypad_chars ( -- )

    \ Places null text in all the text fields for the buttons

  " " xdup c@ 1+ num1text rot cmove
  " " xdup c@ 1+ num2text rot cmove
  " " xdup c@ 1+ num3text rot cmove
  " " xdup c@ 1+ num4text rot cmove
  " " xdup c@ 1+ num5text rot cmove
  " " xdup c@ 1+ num6text rot cmove
  " " xdup c@ 1+ num7text rot cmove
  " " xdup c@ 1+ num8text rot cmove
  " " xdup c@ 1+ num9text rot cmove
  ;


: set_lowercase ( -- )

    \ Initializes the letters in the keypad to their lower case
versions.
```

```
  false to current_case                     \ Set the current case var
  "    qz " xdup c@ 1+ num1text rot cmove
  "    abc" xdup c@ 1+ num2text rot cmove
  "    def" xdup c@ 1+ num3text rot cmove
  "    ghi" xdup c@ 1+ num4text rot cmove
  "    jkl" xdup c@ 1+ num5text rot cmove
  "    mno" xdup c@ 1+ num6text rot cmove
  "    prs" xdup c@ 1+ num7text rot cmove
  "    tuv" xdup c@ 1+ num8text rot cmove
  "    wxy" xdup c@ 1+ num9text rot cmove
  ;


: set_uppercase ( -- )

    \ Initializes the letters in the keypad to their upper case
versions.

  true to current_case                      \ Set the current case var
  "    QZ " xdup c@ 1+ num1text rot cmove
  "    ABC" xdup c@ 1+ num2text rot cmove
  "    DEF" xdup c@ 1+ num3text rot cmove
  "    GHI" xdup c@ 1+ num4text rot cmove
  "    JKL" xdup c@ 1+ num5text rot cmove
  "    MNO" xdup c@ 1+ num6text rot cmove
  "    PRS" xdup c@ 1+ num7text rot cmove
  "    TUV" xdup c@ 1+ num8text rot cmove
  "    WXY" xdup c@ 1+ num9text rot cmove
  ;


: set_char_pointer ( -- )

    \ Positions the ^ pointer underneath the alpha strings in each alpha
    \ button.  The value in alpha_offset is used to locate the ^.

  shift_offset
  case
    0 of "       " endof        \ Point to no letter
    4 of "    ^  " endof        \ Point to first letter
    5 of "     ^ " endof        \ Point to second letter
    6 of "      ^" endof        \ Point to third letter
  endcase

  alpha
  if
    xdup
    xdup c@ 1+ ptr_text rot cmove \ Store appropriate version of the
ptr_text
  else
    0 ptr_text c!
  endif
  xdup c@ 1+ ptr0_text rot cmove \ Store appropriate version of the
ptr0_text
  ;


: init_num0_text ( -- )

    \ Initializes the text label for the 0 button of the keypad.

  "    +-E" xdup c@ 1+ num0text rot cmove
```

```
  ;


: short_beep ( -- )
  1 set.high.current
  2000 microsec.delay
  1 clear.high.current
  ;


: kp_text_refresh ( -- )

    \ This is just a dummy place holder.  Functions that must be defined
prior
    \ to the definition of menus will call this function which will be
    \ redefined later to reference the menus that have not yet been
defined.

  no.op
  ;


: num_shift_handler ( -- )

    \ Handles the pressing of the shift shift key.  Pressing the shift
key
    \ causes the last character printed to rotate through alternate
chars
    \ if it is a number key.

  short_beep

  alpha
  if
    shift_offset                  \ We want to conditionally adjust the
value
    case                          \ of the alpha_offset variable
      0 of 4 endof                \ If it is 0, take it to 4
      4 of 5 endof                \ If it is 4, take it to 5
      5 of 6 endof                \ If it is 5, take it to 6
      6 of                        \ If it is 6, we have to decide what
to do...
        current_case              \ If caps, go to lower and start over
with num
        if
          set_lowercase 0         \ Roll over back to the number mode
        else
          set_uppercase 4         \ Roll over and go to caps
        endif
      endof
    endcase

    dup
    to shift_offset               \ Save the new value
    to alpha_offset               \ Save a copy in alpha_offset
  else
    shift_offset                  \ We want to conditionally adjust the
value
    case                          \ of the alpha_offset variable
      0 of 4 endof                \ If it is 0, take it to 4
      4 of 5 endof                \ If it is 4, take it to 5
      5 of 6 endof                \ If it is 5, take it to 6
      6 of 0 endof                \ If it is 6, take it to 0
```

```
    endcase

    to shift_offset               \ Save the new value
  endif

  set_char_pointer
  kp_text_refresh               \ Refresh the text of the buttons since
it has
  tvars update_text             \ changed.
  ;


: refresh_string_display ( -- )

    \ Refresh the display of the string on the display

  keypad_text_col keypad_text_row garray.xpfa []   ( -- xaddr )
  kpbuffer_length erase         \ Clear out the string area on the
screen

  kpbuffer                      \ We need the x$addr
  keypad_text_row
  keypad_text_col $>display     \ Write the string at kpbuffer to the
display

  tvars update_text             \ Update the text layer

  keypad_text_row
  keypad_text_col kpbuffer c@ +
  put.cursor                    \ Reposition the cursor
  ;


: num_ent_handler ( -- )

    \ Handles the enter button press.  The intended result of pressing
    \ the enter button may vary depending on how the programmer intends
to
    \ use the keypad.  Consequently, we simply execute another handler
    \ that may be configured at runtime.

  short_beep

  enter_handler_xcfa execute    \ Execute the xcfa stored in the self
fetching
                                \ variable when the enter button is
pressed.
                                \ If the keypad is used without
initializing
                                \ this variable, the board will crash
when
                                \ the enter button is pressed.
  ;


: num_del_handler ( -- )

    \ Handles the pressing of the delete button on the number pad.

  kpbuffer c@                   \ We need the current count of the
string
  dup                           \ Make a copy for later
  0= if                         \ Only proceed if string is not empty
```

```
    drop                          \ String already empty.  Drop and exit.
    exit
  endif
  short_beep                      \ If the cursor moves, beep

  1-                              \ decrement the count
  kpbuffer c!                     \ store it as the new count
  refresh_string_display          \ Refresh the string on the display
  ;


: num_handler ( char -- )

    \ Appends n to the end of the string kpbuffer and increments its
count.
    \ This function is called by the number

  \ Comment out the following 3 lines if you do not want the text mode
  \ to be cleared at each keystroke.
  0 to alpha_offset               \ This resets the shift mode
  0 to shift_offset               \ ....
  set_char_pointer                \ Regenerate the '^' pointer

  kp_text_refresh                 \ Refresh the button labels

  kpbuffer c@                     \ Fetch the current length of the string
  dup                             \ We'll need a copy of that
  kpbuffer_length >=              \ Do not overflow the alloted string
space
  if
    2drop                         \ The buffer is full.  Reject the number
    exit
  endif

  short_beep                      \ If we are able to print the char, beep
        ( char\count -- )
  1+                              \ Adjust the string count
    dup                           \ Make a copy of the adjusted string
count
    kpbuffer c!                   \ Store the new count
  kpbuffer rot xn+                \ Offset the kpbuffer base address by
the count
  c!                              \ Store the char in that location
  refresh_string_display          \ Refresh the string on the display
  ;


: num_0_handler ( -- )
  shift_offset not if
    ascii 0
  else
    num0text shift_offset xn+ c@
  endif
  num_handler                     \ Tell num_handler about the number
pressed
  ;

: num_1_handler ( -- )
  alpha_offset not if
    ascii 1
  else
    num1text alpha_offset xn+ c@
  endif
```

```
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_2_handler ( -- )
  alpha_offset not if
    ascii 2
  else
    num2text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_3_handler ( -- )
  alpha_offset not if
    ascii 3
  else
    num3text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_4_handler ( -- )
  alpha_offset not if
    ascii 4
  else
    num4text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_5_handler ( -- )
  alpha_offset not if
    ascii 5
  else
    num5text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_6_handler ( -- )
  alpha_offset not if
    ascii 6
  else
    num6text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_7_handler ( -- )
  alpha_offset not if
    ascii 7
  else
    num7text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;
```

```
: num_8_handler ( -- )
  alpha_offset not if
    ascii 8
  else
    num8text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_9_handler ( -- )
  alpha_offset not if
    ascii 9
  else
    num9text alpha_offset xn+ c@
  endif
  num_handler                   \ Tell num_handler about the number
pressed
  ;

: num_dec_handler ( -- )
  ascii . num_handler           \ Tell num_handler about the button
pressed
  ;

\ **
\ **    End of Keypad Handlers and support code
\ **
\ ************************************************************


\ ************************************************************
\ **
\ **    Begin button definitions
\ **

\ Buttons for the keypad menu

hex

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM0_PCX                        \ Draw graphic
NUM0_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_0_handler           \ The code to execute when pressed
" "                             \ No text for the first line
" "                             \ No text for the second line
ptr0_text                       \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num0_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
```

```
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM1_PCX                        \ Draw graphic
NUM1_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_1_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num1text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num1_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM2_PCX                        \ Draw graphic
NUM2_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_2_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num2text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num2_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM3_PCX                        \ Draw graphic
NUM3_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_3_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num3text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num3_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM4_PCX                        \ Draw graphic
NUM4_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_4_handler           \ The code to execute when pressed
```

```
" "                             \ No text for the first line
num4text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num4_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM5_PCX                        \ Draw graphic
NUM5_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_5_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num5text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num5_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM6_PCX                        \ Draw graphic
NUM6_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_6_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num6text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num6_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM7_PCX                        \ Draw graphic
NUM7_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_7_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num7text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
```

```
fastbutton num7_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM8_PCX                        \ Draw graphic
NUM8_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_8_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num8text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num8_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DRAW_TEXT_FLAG or               \ flag indicating to print text for this
button
NUM9_PCX                        \ Draw graphic
NUM9_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_9_handler           \ The code to execute when pressed
" "                             \ No text for the first line
num9text                        \ The string to be printed inside this
button
ptr_text                        \ The text string that contains the ^
pointer
" "                             \ No text for the 4th line
fastbutton num9_button          \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
NUMDEC_PCX                      \ Draw graphic
NUMDEC_PCX                      \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for num_dec_handler         \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton numdec_button        \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
NUMDEL_PCX                      \ Draw graphic
NUMDEL_PCX                      \ Release graphic
SBLACK_PCX                      \ Press graphic
```

```
cfa.for num_del_handler          \ The code to execute when pressed
" "                              \ This button does not have text
" "                              \ .
" "                              \ .
" "                              \ This button does not have text
fastbutton numdel_button         \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG               \ flag indicating when to execute the
action
NUMENT_PCX                       \ Draw graphic
NUMENT_PCX                       \ Release graphic
SBLACK_PCX                       \ Press graphic
cfa.for num_ent_handler          \ The code to execute when pressed
" "                              \ This button does not have text
" "                              \ .
" "                              \ .
" "                              \ This button does not have text
fastbutton nument_button         \ Instantiate new button with above
parameters

PRESS_HANDLER_FLAG               \ flag indicating when to execute the
action
REPEAT_FLAG or                   \ flag indicating that this is a
repeating key
shift_PCX                        \ Draw graphic
shift_PCX                        \ Release graphic
SBLACK_PCX                       \ Press graphic
cfa.for num_shift_handler        \ The code to execute when pressed
" "                              \ This button does not have text
" "                              \ .
" "                              \ .
" "                              \ This button does not have text
fastbutton numshift_button       \ Instantiate new button with above
parameters



\ **
\ **        End of keypad menu buttons
\ **
\ ************************************************************

\ ************************************************************
\ **
\ **        Keypad menu definitions
\ **

decimal

\ Here, we define the alpha numeric keypad menu.  The 2 numbers on the
left
\ of each entry define the row and column location of the button.  Since
text
\ has a granularity of 8 pixel lines, the row position for buttons that
contain
\ text has been carefully choosen so that the text is properly
positioned
\ inside the button.

\ Col Row       DRAW_MASK ButtonName      touch_button

new_menu: alnum_keypad_menu
```

```
\ first column
  0   38        DRAW_MASK numshift_button add_touch_button  \ shift key
  0   70        DRAW_MASK numdec_button   add_touch_button  \ Decimal
point
  0   102       DRAW_MASK num0_button     add_touch_button  \ Numeral 0

\ second column
  8   38        DRAW_MASK num7_button     add_touch_button  \ Numeral 7
  8   70        DRAW_MASK num4_button     add_touch_button  \ Numeral 4
  8   102       DRAW_MASK num1_button     add_touch_button  \ Numeral 1

\ third column
  16  38        DRAW_MASK num8_button     add_touch_button  \ Numeral 8
  16  70        DRAW_MASK num5_button     add_touch_button  \ Numeral 5
  16  102       DRAW_MASK num2_button     add_touch_button  \ Numeral 2

\ forth column
  24  38        DRAW_MASK num9_button     add_touch_button  \ Numeral 9
  24  70        DRAW_MASK num6_button     add_touch_button  \ Numeral 6
  24  102       DRAW_MASK num3_button     add_touch_button  \ Numeral 3

\ fifth column
  32  70        DRAW_MASK numdel_button   add_touch_button  \ Delete Key
  32  102       DRAW_MASK nument_button   add_touch_button  \ Enter Key

build_menu


new_menu: num_keypad_menu

\ first column
  0   06        DRAW_MASK num7_button     add_touch_button  \ Numeral 7
  0   38        DRAW_MASK num4_button     add_touch_button  \ Numeral 4
  0   70        DRAW_MASK num1_button     add_touch_button  \ Numeral 1
  0   102       DRAW_MASK num0_button     add_touch_button  \ Numeral 0

\ second column
  8   06        DRAW_MASK num8_button     add_touch_button  \ Numeral 8
  8   38        DRAW_MASK num5_button     add_touch_button  \ Numeral 5
  8   70        DRAW_MASK num2_button     add_touch_button  \ Numeral 2
  8   102       DRAW_MASK numdec_button   add_touch_button  \ Decimal
point

\ third column
  16  06        DRAW_MASK num9_button     add_touch_button  \ Numeral 9
  16  38        DRAW_MASK num6_button     add_touch_button  \ Numeral 6
  16  70        DRAW_MASK num3_button     add_touch_button  \ Numeral 3
  16  102       DRAW_MASK nument_button   add_touch_button  \ Enter Key

\ forth column
  24   70       DRAW_MASK numshift_button add_touch_button  \ shift key
  24  102       DRAW_MASK numdel_button   add_touch_button  \ Delete key

build_menu

\ **
\ **        End of keypad menu definitions
\ **
\ ************************************************************


\ ************************************************************
```

```
\ **
\ **        Top init code and redefinitions
\ **

\ Here we redefine the word alnum_text_refresh.  It was needed earlier,
\ but couldn't be defined until now since it references the menu
\ alnum_keypad_menu.

hex

unique.msg @                                    \ Save old value of
unique.msg
unique.msg off                                  \ Disable non-unique
message
here cfa.for kp_text_refresh redefine           \ Point the previously
defined
                                                \ place holder here
: kp_text_refresh ( -- )
  0 0 tvars DRAW_TEXTONLY_ACTION                \ This action causes all
text
                                                \ in buttons in the menu
to be
                                                \ redrawn.
  alpha                                         \ See which kind of
keypad
  if                                            \ We should update
    alnum_keypad_menu
  else
    num_keypad_menu
  endif
  do_menu
  ;
unique.msg !                                    \ Restore the state of
the
                                                \ unique messages.


: init_alnum_keypad ( -- )

    \ This function places the menu alnum_keypad_menu on the screen and
    \ initializes all associated variables except for the kpbuffer text
string
    \ pointer.  It must be initialized prior to calling this function.
    \ This allows editing of any string.  Since this string will be
written to
    \ by the keypad functions, it must be located in writable RAM.

  set_lowercase                 \ Load the button labels with lowercase
letters
  init_num0_text                \ Init the text string for the 0 key

  0 to alpha_offset             \ Set the alpha state to numerical mode
  0 to shift_offset             \ Set the shift state to numerical mode

  0 ptr_text c!                 \ Initialize the text pointer string
  0 ptr0_text c!                \ Initialize the num0 pointer string

  true to alpha                 \ Set boolean to indicate alpha mode
enabled
  0 0 0 tvars alnum_keypad_menu
  init_menu                     \ Initialize the menu to the screen and
keymap
  tvars update_graphics         \ Update the graphics layer
```

```
  tvars update_text             \ Update the text layer
  true true set_cursor_state    \ Make the cursor visible
  refresh_string_display        \ Set the cursor position and print the
string
  ;


: uninit_alnum_keypad ( -- )

    \ This function removes the alnum_keypad_menu from the screen and
keymap
    \ array.  It also updates the display when this action is finished.

  0 0 0 tvars alnum_keypad_menu \ Uninitialize the menu.  This erases
all of
  uninit_menu                   \ the graphics and text that the menu
drew and
                                \ removes the buttons from the keymap
array.
  tvars update_graphics         \ Refresh the graphics layer
  false false set_cursor_state  \ Turn the cursor back off
  tvars clear_text              \ Clear all text from the display
  ;


: init_num_keypad ( -- )

    \ This function places the menu num_keypad_menu on the screen and
    \ initializes all associated variables except for the kpbuffer text
string
    \ pointer.  It must be initialized prior to calling this function.
    \ This allows editing of any string.  Since this string will be
written to
    \ by the keypad functions, it must be located in writable RAM.

  false to alpha                \ Set boolean to indicate alpha mode
disabled

  blank_keypad_chars            \ Place zero length strings in num 1-9
labels
  init_num0_text                \ Init the text string for the 0 key

  0 to alpha_offset             \ Set the alpha state to numerical mode
  0 to shift_offset             \ Set the shift state to numerical mode

  0 ptr_text c!                 \ Initialize the text pointer string
  0 ptr0_text c!                \ Initialize the num0 pointer string

  0 0 0 tvars num_keypad_menu
  init_menu                     \ Initialize the menu to the screen and
keymap
  tvars update_graphics         \ Update the graphics layer
  true true set_cursor_state    \ Make the cursor visible
  refresh_string_display        \ Set the cursor position and print the
string
  ;


: uninit_num_keypad ( -- )

    \ This function removes the num_keypad_menu from the screen and
keymap
    \ array.  It also updates the display when this action is finished.
```

```
  0 0 0 tvars num_keypad_menu   \ Uninitialize the menu.  This erases
all of
  uninit_menu                   \ the graphics and text that the menu
drew and
                                \ removes the buttons from the keymap
array.
  tvars update_graphics         \ Update the graphics layer
  false false set_cursor_state  \ Turn off the cursor visibility
  tvars clear_text              \ Clear all text from the display
  ;


decimal
: init_kp_vars ( -- )

    \ Initializes all keypad related variables to 'safe' values.  This
function
    \ should be called as part of the startup init code in the
application.
    \ It sets up all the keypad menu's variables to reasonable values.

  cfa.for no.op to enter_handler_xcfa
                                \ Initialize the enter handler xcfa

  10 to keypad_text_col         \ Default col location of text string
  0  to keypad_text_row         \ Default row location of text string

  0 to kpbuffer_length          \ Set to 0 initially
  ;




\ **
\ **        Top init code and redefinitions
\ **
\ ************************************************************


\ ****
\ ****    End of keypad module
\ ****
\ ************************************************************



\ ************************************************************
\ ****
\ ****    Main menu module
\ ****


\ ********************************************************
\ **
\ **    Main screen Handlers and support code
\ **

decimal

23 constant MAIN_TITLE_MAXLEN       \ Max length for the main title

integer: pump_power                 \ 0-100 % pump power
```

```
real:   flow_limit                  \ The floating point L/min of the
pump

12 v.instance: tmpbuff              \ Make a tmp buffer for editing the
power
MAIN_TITLE_MAXLEN v.instance: main_title
                                    \ Declare the title buffer

: sht_dn_handler ( -- )

  short_beep

  ." Shut down button has been pressed" cr
  ;

: config_handler ( -- )

  short_beep

  ." Config button has been pressed" cr
  ;

: stats_handler ( -- )

  short_beep

  ." Stats button has been pressed" cr
  ;

: s_title_handler ( -- )
  no.op                         \ Deferred for later redefinition
  ;

: s_flow_handler ( -- )
  no.op                         \ Deferred for later redefinition
  ;

: s_power_handler ( -- )
  no.op                         \ Deferred for later redefinition
  ;

: decrease_handler ( -- )
  no.op                         \ Deferred for later redefinition
  ;

: increase_handler ( -- )
  no.op                         \ Deferred for later redefinition
  ;


\ **
\ **    End of Main screen Handlers and support code
\ **
\ ************************************************************




\ ************************************************************
\ **
\ **        Main screen button definitions
\ **
```

```
\ Buttons for the main screen

PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
SHT_DN_PCX                      \ Draw graphic
SHT_DN_PCX                      \ Release graphic
DBL_BLK_PCX                     \ Press graphic
cfa.for sht_dn_handler          \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton sht_dn_button        \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
CONFIG_PCX                      \ Draw graphic
CONFIG_PCX                      \ Release graphic
SBLACK_PCX                        \ Press graphic
cfa.for config_handler          \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton config_button        \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
STATS_PCX                       \ Draw graphic
STATS_PCX                       \ Release graphic
SBLACK_PCX                        \ Press graphic
cfa.for stats_handler           \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton stats_button         \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
EXIT_PCX                        \ Draw graphic
EXIT_PCX                        \ Release graphic
SBLACK_PCX                        \ Press graphic
cfa.for warm                    \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton exit_button          \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
STITLE_PCX                      \ Draw graphic
STITLE_PCX                      \ Release graphic
```

```
LBLACK_PCX                      \ Press graphic
cfa.for s_title_handler         \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton s_title_button       \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
SPOWER_PCX                      \ Draw graphic
SPOWER_PCX                      \ Release graphic
LBLACK_PCX                      \ Press graphic
cfa.for s_power_handler         \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton s_power_button       \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
SFLOW_PCX                       \ Draw graphic
SFLOW_PCX                       \ Release graphic
LBLACK_PCX                      \ Press graphic
cfa.for s_flow_handler          \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton s_flow_button        \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
INCB_PCX                        \ Draw graphic
INCB_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for increase_handler        \ The code to execute when pressed
" "                             \ This button does not have text
" "                             \ .
" "                             \ .
" "                             \ This button does not have text
fastbutton increase_button      \ Instantiate new button with above
parameters


PRESS_HANDLER_FLAG              \ flag indicating when to execute the
action
REPEAT_FLAG or                  \ flag indicating that this is a
repeating key
DECB_PCX                        \ Draw graphic
DECB_PCX                        \ Release graphic
SBLACK_PCX                      \ Press graphic
cfa.for decrease_handler        \ The code to execute when pressed
" "                             \ This button does not have text
```

```
" "                                 \ .
" "                                 \ .
" "                                 \ This button does not have text
fastbutton decrease_button          \ Instantiate new button with above
parameters


\ **
\ **        End of main screen button definitions
\ **
\ ************************************************************

\ ************************************************************
\ **
\ **        Main screen menu definition
\ **

decimal

new_menu: main_menu

\ first and second columns
  01  33        DRAW_MASK pump_pcx        add_graphic       \ The pump
diagram
   0   0        DRAW_MASK sht_dn_button   add_touch_button  \ Shutdown
button
4  0   0        0         sht_dn_button   add_button        \
Doublewidth


\ Second Column
  8    66       DRAW_MASK s_flow_button   add_touch_button  \ Set flow
button


\ third column
  16   12       DRAW_MASK config_button   add_touch_button  \ Config
button
  16   66       DRAW_MASK s_title_button  add_touch_button  \ Set Title
button


\ forth column
  24   12       DRAW_MASK stats_button    add_touch_button  \ Stats
button
  24   66       DRAW_MASK s_power_button  add_touch_button  \ Set Title
button
\  32   34      DRAW_MASK s_title_button  add_touch_button  \ Set Title
button

\ fifth column
  32  12        DRAW_MASK exit_button     add_touch_button  \ Exit
button

  32  72        DRAW_MASK increase_button add_touch_button  \ Increase
button
  32  90        DRAW_MASK pwrlabel_pcx    add_graphic       \ Label for
inc/dec
  32 104        DRAW_MASK decrease_button add_touch_button  \ Decrease
button

build_menu
```

```
\ **
\ **        Main screen menu definition
\ **
\ ************************************************************

\ ************************************************************
\ **
\ **        Top level init code and redefinitions
\ **


decimal

: update_main_text ( -- )

    \ Updates the display of the pump parameters on the main menu
screen.
    \ For real time monitoring a real system, this routine might be
executed
    \ as part of a loop in a separate task for periodic update of data.

  "     %" 10 25 $>display                    \ Blank old value of
pump_power
  pump_power u>d <# #s #>                     \ Convert pump_power to
a
                                              \ string that can be
displayed
  >r 1xn- 10 28 r> - $>display                \ Display the pump power
level


  flow_limit f>floating$ drop                 \ Drop the true flag
left over
  xdup c@ >r 4 34 r> - $>display              \ Compute the
destination col
                                              \ based on the length
  ;


: init_main_menu ( -- )

    \ This function places the main menu on the screen and loads the
keymap
    \ array with the menu's buttons.

  0 0 0 tvars main_menu         \ ( -- offset\col\row\tvars\menu xpfa )
  init_menu                     \ Initialize the menu to the screen and
keymap

  main_title                            0   16   $>display  \ Display
title
  " Flow Limit:                 L/min"   4   8    $>display  \ flow units
  " Input pressure:        314 PSI"     5   8    $>display  \ in pres
units
  " Output pressure:       102 PSI"     6   8    $>display  \ out pres
units

  " Pump RPM:   1812  RPM"              13   8   $>display  \ RPM units
  " Pump Temp:    92  C"                14   8   $>display  \ Temp units

  update_main_text              \ Fill in the initial values
  tvars update_text             \ Update the text layer
  tvars update_graphics         \ Update the graphics layer
```

```
  ;


: uninit_main_menu ( -- )

    \ This function removes the main menu from the screen and keymap
    \ array.  It also updates the display when this action is finished.

  0 0 0 tvars main_menu         \ Uninitialize the menu.  This erases
all of
  uninit_menu                   \ the graphics and text that the menu
drew and
                                \ removes the buttons from the keymap
array.
  tvars update_graphics         \ Update the graphics layer
  tvars clear_text              \ Clear all text from the display
  ;


: kp_edittitle_exit ( -- )

    \ This function is used by the enter button of the keypad menu to
    \ restore the the main menu after the title string has been edited.

  uninit_alnum_keypad                           \ uninitialize the
keypad menu
  init_main_menu                                \ Initialize the main
menu
  tvars update_text                             \ Update the text
  tvars update_graphics                         \ Update the graphics
  cfa.for no.op to enter_handler_xcfa           \ Although this variable
should
                                                \ be initialized at each
                                                \ calling of the keypad
menu,
                                                \ it is good to reset
it.
  ;


: kp_setflow_exit ( -- )

    \ This function is used by the enter button of the keypad menu to
    \ restore the the main menu after the flow rate string has been
edited.


  bl kpbuffer xdup c@ 1+ xn+ c!                 \ Append a space to the
end of
                                                \ the string
  kpbuffer $>f                                  \ Attempt to convert the
string
  if                                            \ If successful, then
save the
    to flow_limit                               \ number to flow_limit
  endif


  uninit_num_keypad                             \ uninitialize the
keypad menu
  init_main_menu                                \ Initialize the main
menu
```

```
  cfa.for no.op to enter_handler_xcfa           \ Although this variable
should
                                                \ be initialized at each
                                                \ calling of the keypad
menu,
                                                \ it is good to reset
it.

  update_main_text                              \ Update the text value
display
  tvars update_text                             \ Update the text
display layer
  tvars update_graphics                         \ Update the graphics
layer
  ;


: kp_setpower_exit ( -- )

    \ This function is used by the enter button of the keypad menu to
    \ restore the the main menu after the flow rate string has been
edited.


  bl kpbuffer xdup c@ 1+ xn+ c!                    \ Append a space to
the end of
                                                \ the string
  kpbuffer $>f                                  \ Attempt to convert the
string
  if                                            \ If successful, then
save the
    fixx                                        \ Round to nearest
integer
    dup dup 0 >= swap 100 <= and                \ Make sure it is valid
    if
      to pump_power                             \ save number to
power_level
    else                                        \ If pump_power is
illegal,
      drop                                      \ throw it away
    endif
  endif

  uninit_num_keypad                             \ uninitialize the
keypad menu
  init_main_menu                                \ Initialize the main
menu

  cfa.for no.op to enter_handler_xcfa           \ Although this variable
should
                                                \ be initialized at each
                                                \ calling of the keypad
menu,
                                                \ it is good to reset
it.

  update_main_text                              \ Update the text value
display
  tvars update_text                             \ Update the text
display layer
  tvars update_graphics                         \ Update the graphics
layer
  ;
```

```
unique.msg @                                    \ Save old value of
unique.msg
unique.msg off                                  \ Disable non-unique
message



here cfa.for s_title_handler redefine           \ Point place holder
here
: s_title_handler ( -- )

  short_beep

  main_title to kpbuffer                        \ Move the title to the
                                                \ kpbuffer for editing

  MAIN_TITLE_MAXLEN to kpbuffer_length          \ Store the max length
of the
                                                \ string.  This is
important
                                                \ to prevent memory
corruption.

  16 to keypad_text_col                         \ Col location of text
string
  3  to keypad_text_row                        \ Row location of text
string

  cfa.for kp_edittitle_exit to enter_handler_xcfa
                                                \ This lets the main
menu be
                                                \ brought back when the
enter
                                                \ key is pressed on the
keypad

  uninit_main_menu                              \ Clear the current menu
  init_alnum_keypad                             \ init the alnum menu

        \ Print a little help for the user
  " Edit the main screen title field below" 0 0 $>display

  tvars update_text                             \ Update the text
  tvars update_graphics                         \ Update the graphics
  ;


here cfa.for s_flow_handler redefine            \ Point place holder
here
: s_flow_handler ( -- )

  short_beep

    \ Before we actually call the keypad menu, we must set up its
environment.

  flow_limit f>floating$ drop                   \ Turn floating point
number
  xdup c@ >r                                    \ into a string to edit
it.
  tmpbuff r> 1+ cmove                           \ Move it to the tmpbuff
```

```
  tmpbuff to kpbuffer                              \ Set keypad to edit
tmpbuff



  10 to kpbuffer_length          \ Set max length of the string

  25 to keypad_text_col                            \ Col location of text
string
  7  to keypad_text_row                            \ Row location of text
string

  cfa.for kp_setflow_exit to enter_handler_xcfa
                                                   \ This lets the main
menu be
                                                   \ brought back when the
enter
                                                   \ key is pressed on the
keypad
                                                   \ This code also saves
the
                                                   \ modified string.

    \ Now that everything is set up, we can change menus.

  uninit_main_menu                                 \ Clear the current menu
  init_num_keypad                                  \ init the alnum menu

  " Edit the value"     0 25 $>display             \ User prompt
  " for exchanger"      1 25 $>display
  " flow limit"         2 25 $>display

  tvars update_text                                \ Update the text
  tvars update_graphics                            \ Update the graphics
  ;


here cfa.for s_power_handler redefine              \ Point place holder
here
: s_power_handler ( -- )

  short_beep

    \ Before we actually call the keypad menu, we must set up its
environment.

  pump_power u>d <# #s #> >r 1xn-                  \ Turn integer into a
string
  tmpbuff r> 1+ cmove                              \ Copy the string into
tmpbuff
  tmpbuff to kpbuffer                              \ Now edit the buffer

  10 to kpbuffer_length                            \ Set max length of the
string

  25 to keypad_text_col                            \ Col location of text
string
  7  to keypad_text_row                            \ Row location of text
string

  cfa.for kp_setpower_exit to enter_handler_xcfa
                                                   \ This lets the main
menu be
```

```
                                                \ brought back when the
enter
                                                \ key is pressed on the
keypad
                                                \ This code also saves
the
                                                \ modified string.

    \ Now that everything is set up, we can change menus.

  uninit_main_menu                              \ Clear the current menu
  init_num_keypad                               \ init the alnum menu

  " Edit the value"     0 25 $>display          \ User prompt
  " for pump power"     1 25 $>display
  " percentage"         2 25 $>display

  tvars update_text                             \ Update the text
  tvars update_graphics                         \ Update the graphics
  ;


here cfa.for increase_handler redefine          \ Point place holder
here
: increase_handler ( -- )


  pump_power dup                                \ Get the pump power %
  100 <
  if
    1+                                          \ If less than 100, inc
it
    to pump_power                               \ Save it
    short_beep                                  \ Generate a click
    update_main_text                            \ Update the text value
display
    tvars update_text                           \ Update the text
display layer
  else
    drop
  endif
  ;


here cfa.for decrease_handler redefine          \ Point place holder
here
: decrease_handler ( -- )

  pump_power dup                                \ Get the pump power %
  0 >
  if
    1-                                          \ If more than 0, inc it
    to pump_power                               \ Save it
    short_beep                                  \ Generate a click
    update_main_text                            \ Update the text value
display
    tvars update_text                           \ Update the text
display layer
  else
    drop
  endif
  ;
```

```
unique.msg !                                    \ Restore the state of
the
                                                \ unique messages.
\ **
\ **        End of top level init code and redefinitions
\ **
\ ************************************************************


\ ****
\ ****      End of main menu module
\ ****
\ ************************************************************


\ ************************************************************
\ ****
\ ****      Startup code
\ ****

decimal

: init_all ( -- )

  fp.defaults                   \ Set the floating control vars to
defaults
  no.spaces on                  \ For floating strings, do not pad with
spaces

  BOTTOM_OF_HEAP TOP_OF_HEAP
  is.heap                       \ Set up the heap

  init_kp_vars                  \ Initialize the keypad variables


  " Heat Exchanger Unit 2" xdup c@ 1+ main_title rot cmove
                                \ Init the title main menu title string

  4.20 to flow_limit            \ init the flow_limit variable
  53 to pump_power              \ init the pump_power variable

  start.timeslicer              \ The timeslicer must be running for
repeating
                                \ buttons to work
  tvars std_display             \ Config the display hardware
  tvars init_display            \ Initialize the display hardware
  tvars init_touch              \ Initialize the touchscreen hardware
and vars
  ;


: touch_mon ( -- )

    \ Loop for getting screen touches.

  begin
    tvars menu_query          \ Get a single touch from the user
    pause.on.key                \ See if a key has been pressed
  again                         \ Do it all over again
  ;
```

```
: start ( -- )

    \ This is the top level startup function.  Call it and everything
starts
    \ running.

  init_all
  init_main_menu                \ Display the alnum menu
  touch_mon                     \ Loop forever while scanning the
touchscreen
  ;

\ ****
\ ****      End of startup code
\ ****
\ ***************************************************************


\ ******
\ ******    End of example user interface
\ ******
\ ***************************************************************
```

Appendix 5 – GUI Toolkit Glossary

V1.6 RELEASE 9/16/2003 KERNEL EXTENSION

Glossary of Terms

Action flag

A flag that is passed to an object handler with an object to determine the behavior of the object. Some actions are used by several object types and others have meaning only to one specific type of object. The actions are described in the Glossary of Functions under the object handlers for which they are used. See Do_Graphic and Do_Button for contextual explanations of the actions.

Array_pf struct

Forth array parameter field structure describing the geometry of a forth array. The actual array data is contained in the heap pointed to by a heap handle. Consult the QED Software manual for more details on arrays.

CFORTH_ARRAY structure			
Offset	**Type**	**Element name**	**Description**
0x0000	xaddr	handle	Xaddress of the pointer (heap xhandle) to the beginning of the data
0x0004	addr	cur_heap	16 bit addr of top of heap (same page as xhandle)
0x0006	uint	bytes_per_element	# of bytes in each element
0x0008	uint	num_dimensions	# of dimensions in array
0x000A	uint	num_cols	# of columns (1st dimension)
0x000C	uint	num_rows	# of rows (2nd dimension)
0x000E	uint	num_pages	# of pages (3rd dimension)
0x0010	uint	num_books	# of books (4th dimension)

Button location

An integer value describing one of the 20 touch sensitive areas of the touchscreen panel. This quantity is used to index the keymap array when buttons are pressed. Below is a map of the button locations on the touchscreen.

0	4	8	12	16
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19

Button object

A data structure of type BUTTON that describes a button. The button object handler, Do_Button, accepts the xaddress of this structure, an action flag, and a col/row location.

BUTTON structure			
Offset	**Type**	**Element name**	**Description**
0x0000	uint	flags	A set of bitmapped flags describing button's behavior
0x0002	xaddr	graphic_handler_xcfa	The xaddress of Do_Graphic
0x0006	xaddr	draw_graphic	Xaddress of graphic object drawn for DRAW_ACTION
0x000A	xaddr	release_graphic	Xaddress of graphic object drawn for RELEASE_ACTION
0x000E	xaddr	press_graphic	Xaddress of graphic object drawn for PRESS_ACTION
0x0012	xaddr	press_handler	Xaddress of user code executed for PRESS_ACTION
0x0016	xaddr	release_handler	Xaddress of user code executed for RELEASE_ACTION
0x001A	xaddr	label1	Modified xaddress of text string for line 1 of the button
0x001E	xaddr	label2	Modified xaddress of text string for line 2 of the button
0x0022	xaddr	label3	Modified xaddress of text string for line 3 of the button
0x0026	xaddr	label4	Modified xaddress of text string for line 4 of the button

The flags in the above structure determine the specific behavior of the button. All other elements are irrelevant if unused. The flags, which are constants that may be ORed together, determine which elements will be required. Button defining macros configure common flags. If a bit is set, then the flag is true. If it is cleared, then it is false. The possible flags are:

DRAW_GRAPHIC_FLAG

Indicates that there is a valid xaddress for a graphic object in the draw_graphic field to be drawn for the DRAW_ACTION.

RELEASE_GRAPHIC_FLAG

Indicates that there is a valid xaddress for a graphic object in the release_graphic field to be drawn for the RELEASE_ACTION.

PRESS_GRAPHIC_FLAG

Indicates that there is a valid xaddress for a graphic object in the press_graphic field to be drawn for the PRESS_ACTION.

DIR_DRAW_GRAPHIC_FLAG

When drawing the draw_graphic object, use direct to screen drawing.

DIR_RELEASE_GRAPHIC_FLAG

When drawing the release_graphic object, use direct to screen drawing.

DIR_PRESS_GRAPHIC_FLAG

When drawing the press_graphic object, use direct to screen drawing.

DRAW_TEXT_FLAG
Print the text strings whose xaddress are stored in label1, label2, label3, and label4 in the button.

PRESS_HANDLER_FLAG
Execute the code at the xaddress stored in the press_handler field when the button is given a PRESS_ACTION.

RELEASE_HANDLER_FLAG
Execute the code at the xaddress stored in the release_handler field when the button is given a RELEASE_ACTION.

REPEAT_FLAG
If the button is pressed and held, start executing the press_handler repeatedly. If the press handler is not enabled by having the PRESS_HANDLER_FLAG set, then the REPEAT_FLAG will have no effect.

TEXT_UPDATE_PRESS_FLAG
Call Update_Text when the button is given a PRESS_ACTION.

TEXT_UPDATE_RELEASE_FLAG
Call Update_Text when the button is given a RELEASE_ACTION.

GRAPHICS_UPDATE_PRESS_FLAG
Call Update_Graphics when the button is given a PRESS_ACTION.

GRAPHICS_UPDATE_RELEASE_FLAG
Call Update_Graphics when the button is given a RELEASE_ACTION.

C_STYLE_TEXT_FLAG
Interpret strings in label fields as C style strings instead of Forth.

The macros used to create buttons are **FASTBUTTON**, **NORMBUTTON**, and **BLANKBUTTON**. They are described in the *Glossary of Functions*.

Col, Row
Used to describe position on the LCD screen. When used in reference to graphics, unless otherwise stated, col is a unit of 6 horizontal pixels and row is a unit of 1 vertical pixel. When used in reference to text, unless otherwise stated, col is a unit of 1 character width, 6 pixels, and row is a unit of 1 character line, 8 pixels.

Dualmode
Refers to the technique of operating the LCD display simultaneously in text and graphics modes. The QED board has internal display routines that control the text layer of the display while the dualmode driver extensions contained in the GUI Toolkit extend the capabilities to allow graphics mode to be used in tandem with text mode.

Graphic object
Image to be displayed on the LCD display handled by Do_Graphic. Graphic objects are constant 2 dimensional forth arrays generated by the pcx2qed converter program. The pcx2qed converter program also generates a header file comprising constants that refer to the xaddresses of the graphic objects. See Do_Graphic.

Graphics array
An array in RAM that contains a shadow of the graphics layer on the display. The graphics array is a forth array dimensioned in the display heap area. The tvars struct

contains the graphics array's parameter field (array_pf struct). The location of the display heap is also specified in the tvars struct. DRAW_ACTION, REDRAW_ACTION, or ERASE_ACTION cause objects to write data to this array. Subsequently calling Update_Graphics sends the graphics array data to the display. Direct screen writes completely bypass this array. If Update_Graphics is called after a direct screen write, then any data not mirrored in the graphics array is overwritten. Init_Display dimensions and fills the graphics array with the background_fill member of tvars.

Keymap array

A forth array whose parameter field is stored in the tvars struct. This is an array of structures of type KEYMAP_ENTRY. It has the number of elements equal to the number of touch sensitive areas on the touchscreen or keypad. For the standard Panel-Touch controller or a 4x5 keypad, there are 20 elements. When a the touchscreen area or keypad key is pressed or released, the corresponding element of the keymap array is examined to determine if anything should done in response. The menu manager looks to see if that element contains a valid button object, and if so the button object's handler is executed. The parameters passed to the handler are the location of the button in the menu, the button object's xaddress, and the PRESS_ACTION, RELEASE_ACTION, or REPEAT_ACTION. If that element contains 0x00000000 in the object field, then the button press and the subsequent release are ignored.

KEYMAP_ENTRY structure			
Offset	**Type**	**Element name**	**Description**
0x0000	uint	row	The relative or absolute row position for the button graphics*
0x0002	uint	col	The relative or absolute col position for the button graphics*
0x0004	xaddr	object	The xaddress of the object structure
0x0008	xaddr	obj_handler	The xaddress of the object handler

* When this structure is used as part of the keymap, the row and col are absolute. When it is used as part of a menu (see MENU_ENTRY struct), the row and col are relative displacements from the upper left corner of the menu.

Menu

An array of structures, each of type MENU_ENTRY, that serves as a grouping of button and graphic objects. This array also contains location information about each object. A typical menu might contain several buttons and perhaps a logo or other graphic objects. Multiple menus may be displayed at the same time. Conflicts will occur if the same location is used by two menus at the same time however. Each element of the array is of type MENU_ENTRY. Macros simplify the creation of menus. See NEW_MENU, ADD_BUTTON, ADD_GRAPHIC, and BUILD_MENU for information on creating menus. See Init_Menu, Uninit_Menu, Do_Menu for information on using menus. Each element of a menu is of the following form:

MENU_ENTRY structure			
Offset	**Type**	**Element name**	**Description**
0x0000	KEYMAP_ENTRY	keymap_entry	A sub-structure of type keymap_entry that will be 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.
0x000E	uint	button	The relative keymap index of a button object. If a nonzero number is passed to Init_Menu for the button offset, then it is added to the relative keymap index to form an absolute keymap index.

Menu manager

The software routine that scans the keypad or touchscreen hardware and makes calls to objects stored in the keymap array. The standard menu manager used in the GUI Toolkit is called Menu_Query, but any routine that can collect user input and make the appropriate calls to the objects stored in a menu can act as a menu manager.

Modified xaddress

See xaddress.

Object handler

The function that processes an object and its parameters and initiates its behavior. Do_Graphic and Do_Button are the object handlers for graphic and button objects respectively. These object handlers may be called from C as described in the Glossary of Functions section, but they are also required as part of certain structures such as BUTTON and KEYMAP_ENTRY. When programming in C, use the following syntax for placing the 24 bit xaddress of these handlers in the structures when building them manually.

TO_XADDR(DO_GRAPHIC_ADDR, DO_GRAPHIC_PAGE)
TO_XADDR(DO_BUTTON_ADDR, DO_BUTTON_PAGE)

The macros that assist in button and menu creation eliminate the need to build those structures manually. Consequently, there is rarely a need to explicitly specify the xaddresses. See Do_Graphic and Do_Button.

Text array

An array in RAM that contains a shadow of the text layer on the display. The text array is a forth array dimensioned in the display heap area. Its parameter is field stored at the address location returned by GARRAY_XPFA which is always in common RAM. The forth equivalent garray.xpfa returns an xaddress with 0x00 as the page. Although for historical reasons the name implies that it is used for graphics, the GUI Toolkit only uses this array for storing text. This location is the standard

location for the display array parameter field on the QED board. The elements of this array contain ASCII data that has been shifted down by 0x20. The Toshiba TC6963 chip on the display uses this modified ASCII method for storing text. StringToDisplay (forth: $>display) requires that the text array to be located here. Init_Display dimensions and clears the text array by filling it with 0x00 (ASCII space shifted by 0x20).

Tvars struct

A structure that contains all the global variables used by the GUI Toolkit. The user's program must contain an instance of this structure of type GUI_VARS. Throughout this document and in all of the example routines, the instance of this structure is named tvars. The macro, TVARS is a replacement for (tvars, THIS_PAGE). Many of the GUI Toolkit's functions require the base address of this structure as one of the arguments. Below is a description of the elements of this structure.

GUI_VARS structure			
Offset	**Type**	**Element name**	**Description**
0x0000	xaddr	display_heap_top	Xaddress of top (last byte) of display heap
0x0004	xaddr	display_heap_bottom	Xaddress of bottom (first byte) of display heap
0x0008	int	background_fill	Background fill byte for graphic layer. The lower 6 bits describe a 6 pixel horizontal field.
0x000A	array_pf	graphics_garray	18 byte long array_pf sub-structure for the graphics layer array. This array is dimensioned and initialized in the display heap area by Init_Display.
0x001C	xaddr	display_resource	Display resource variable for access control
0x0020	addr	gr_home_addr	Address of graphics area on the display controller
0x0022	addr	text_home_addr	Address of text area on the display controller
0x0024	uint	graphic_rows	# of pixel lines on the display
0x0026	uint	graphic_cols	# of cols on the display (6 pixels/byte) 1 col=6 pixels
0x0028	uint	text_rows	# of text lines on the display
0x002A	uint	text_cols	# of text columns on the display
0x002C	array_pf	keymap_array	18 byte long array_pf sub-structure for the keymap array. This array_pf struct is initialized and the array is dimensioned and initialized to 0xFF by Init_Touch in the current heap when Init_Touch is called.
0x003E	uint	current_row*	The row of the button being pressed
0x0040	uint	current_col*	The col of the button being pressed
0x0042	uint	current_keynum*	The button number of the button being pressed
0x0044	xaddr	current_button*	The xaddress of the button object being pressed
0x0048	uint	repeat_delay	# of timeslicer counts to wait before repeating
0x004A	uint	repeat_period	# of timeslicer counts to wait between repetitions

* These quantities are never read by the GUI Toolkit routines. They are written to by the menu manager to provide a means for the user's handler code to implement location sensitive behavior. For example, a handler could use these variables to implement a modal set of selector buttons so that when one button is pressed, it stays in the 'pressed' position while releasing the previously pressed button.

Xaddress

A 24 bit number consisting of a 16 bit address and an 8 bit page. Xaddresses occupy 32 bit fields. C functions that use pointers only return 16 bit addresses. The macros used in the GUI Toolkit pad out the 16 bit addresses as needed to accommodate the functions that do require full xaddresses. A modified xaddress is used to describe strings of two possible types, forth style and C style. If the upper 8 bits of the xaddress of a string are set to 0xFF, then the xaddress will be interpreted as a C style null terminated string. If the upper 8 bits are set to 0x00, then the string will be interpreted as a forth style counted string.

Glossary of Functions

Many of the functions in this glossary are not needed for typical GUI based applications. They are provided to allow advanced programmers to use the tools in a more manual and flexible way. These are the functions that are likely to be used in any application.

ADD_BUTTON	Adds button to menu in a menu definition
ADD_GRAPHIC	Adds graphic to menu in a menu definition
ADD_TOUCH_BUTTON	Adds touchscreen button to a menu
BUILD_MENU	Terminates the creation of a menu
Clear_Graphics	Clears the graphics layer
Clear_Text	Clears the text layer
Do_Graphic	Handles graphic object actions
FASTBUTTON	Creates a new button that is fast drawing
Init_Display	Initializes the LCD display hardware
Init_Menu	Displays and activates a menu
Init_Touch	Initializes touchscreen environment vars
Menu_Process	Processes a button press passed to it *New function
Menu_Query	Polls the user input hardware for a keypress
NEW_MENU	Begins a new menu definition
Set_Cursor_State	Sets flashing and visibility cursor attributes
Std_Display	Configures the LCD for typical defaults
Uninit_Menu	Erases a menu from screen and nullifies it
Update_Graphics	Updates the graphics layer on the LCD
Update_Text	Updates the text layer on the LCD
Update_Text_And_Graphics	Updates text and graphics layers

#include <mosaic/gui_tk/to_large.h> Must specify this before each set of button or menu definitions if programming in C

#include <mosaic/gui_tk/fr_large.h> Must specify this at the end of each set of button or menu definitions if programming in C

The Forth and C header files have many constants defined in them. Constants shown in italics are only included in the Forth header file. Most of these are not useful in most applications, but are provided to allow advanced programmers to have better access to the GUI Toolkit configuration information. A complete listing of the constants in use by the GUI Toolkit is provided below. If any of these names is used in the user's Forth code, a non-unique warning will be issued. In C programs, a preprocessor error is issued. The constants in this list that are generally useful are described in detail in their relevant glossary entries and manual sections.

Display hardware constants

**PRIOR_CURSOR_STATE*	Address of a display driver control variable
GRAPHICS_DATA_ADDR	Hardware address of display data port
GRAPHICS_CMD_ADDR	Hardware address of display command port
AWSET_CMD	Command for AutoWrite Set
AWRESET_CMD	Command for AutoWrite Release
SET_TX_HOME_CMD	Command to set text layer home display address
SET_TX_AREA_CMD	Command to set text layer width
SET_GR_HOME_CMD	Command to set graphics layer home address
SET_GR_AREA_CMD	Command to set graphics layer width
MODE_CMD	Command to set display controller mode
DISPLAY_MODE_CMD	Command to set display visibility mode
CURSOR_BLINK	Bit flag to set cursor blink state
CURSOR_ON	Bit flag to set cursor visibility
TEXT_MODE	Bit flag to enable/disable text mode
GRAPHICS_MODE	Bit flag to enable/disable graphics mode
OR_TEXT	Bit flag to set text to be ORed with graphics
EXOR_TEXT	Bit flag to set text to be XORed with graphics
AND_TEXT	Bit flag to set text to be ANDed with graphics
LCD_TEXT_ADDR	Default text home layer display address
LCD_GRAPHIC_ADDR	Default graphics layer home display address
LINES_PER_CHAR	Number of pixel rows per text mode character
GRAPHICS_COLUMNS	Number of columns in bytes (6 pixels)
GRAPHICS_ROWS	Number of pixel rows on the graphics layer
TEXT_COLUMNS	Number of text columns on text layer
TEXT_ROWS	Number of lines of text on text layer
DISPLAY_HEAP_TOP	Xaddress of display heap top
DISPLAY_HEAP_BOTTOM	Xaddress of display heap bottom
DEF_BACKGROUND_FILL	Default fill byte for graphic layer background

Graphic object action flags

GRAPHICS_MASK	Action mask for graphic objects
DRAW_ACTION	Draw action flag
DIR_DRAW_ACTION	Direct Draw action flag
REDRAW_ACTION	Redraw action flag
ERASE_ACTION	Erase action flag
DIR_ERASE_ACTION	Direct erase action flag

Button object action flags

BUTTON_MASK	Action mask for button objects
PRESS_REPEAT_ACTION	Press while repeating action flag
PRESS_ACTION	Press action flag
REL_ACTION	Release action flag
DRAW_TEXTONLY_ACTION	Draw only text portion action flag
ERASE_TEXTONLY_ACTION	Erase only text portion action flag

Button object configuration flags

DRAW_GRAPHIC_FLAG	Button uses draw graphic
RELEASE_GRAPHIC_FLAG	Button uses release graphic
PRESS_GRAPHIC_FLAG	Button uses press graphic

DIR_DRAW_GRAPHIC_FLAG	Button directly draws draw graphic
DIR_RELEASE_GRAPHIC_FLAG	Button directly draws release graphic
DIR_PRESS_GRAPHIC_FLAG	Button directly draws press graphic
DRAW_TEXT_FLAG	Button has text labels
PRESS_HANDLER_FLAG	Button has a press handler
RELEASE_HANDLER_FLAG	Button has a release handler
REPEAT_FLAG	Button is repeating
TEXT_UPDATE_PRESS_FLAG	Update_Text called when button is pressed
TEXT_UPDATE_RELEASE_FLAG	Update_Text called when button is released
GRAPHICS_UPDATE_PRESS_FLAG	Update_Graphics called when button is pressed
GRAPHICS_UPDATE_RELEASE_FLAG	Update_Graphics called when button is released
C_STYLE_TEXT_FLAG	Use C style interpretation of label strings

Menu object action flags

MENU_MASK	Action mask for menu objects (obsolete)
INIT_ACTION	Init action flag (obsolete)
UNINIT_ACTION	Uninit action flag (obsolete)

Menu object entry configuration flags

BUTTON_NULL	Disable menu entry slot
BUTTON_NONLOCAL	Disable entry insertion into keymap
DEFAULT_REPEAT_PERIOD	Default repeat period for repeating buttons
DEFAULT_REPEAT_DELAY	Default repeat delay for repeating buttons

Constants used by the object building macros

BUTTON_COLS	Number of button columns on the touchscreen
BUTTON_ROWS	Number of button rows on the touchscreen
BUTTON_WIDTH	Number of columns comprising a button
BUTTON_HEIGHT	Number of pixel rows comprising a button
GRAPHICS_MASK	Action mask for all drawing actions
FASTBUTTON_FLAGS	Button configuration flags for fastbuttons
NORMBUTTON_FLAGS	Button configuration flags for normbuttons

Forth: ADD_BUTTON (button_location\ col\ row\ action_mask\ button_obj --)
C: ADD_BUTTON(uint button_location, uint col, uint row, uint action_mask, BUTTON * button_obj)

Adds a button object to a menu. Inside a menu definition, ADD_BUTTON is a macro that inserts all the necessary information for a button object based on the relative screen position specified by col and row and the address of the graphic object. The button_location describes the relative button number used for this button. If you are adding a touchscreen button, the macro ADD_TOUCH_BUTTON may be used to eliminate the need for specifying the button_location. ADD_TOUCH_BUTTON computes the button_location automatically based on the col and row given. See Init_Menu for more details on how the button number is used. The action_mask is a mask used to control which actions may be passed to the object when Do_Menu is called. For most applications, DRAW_MASK should be used. All of the actions are single bit flags thus several actions can be ORed together to form an action mask. DRAW_MASK is a constant that ORs DRAW_ACTION, ERASE_ACTION, DIR_DRAW_ACTION, REDRAW_ACTION, DIR_ERASE_ACTION, DRAW_TEXTONLY_ACTION, and ERASE_TEXTONLY_ACTION. Here is an example of the usage:

Forth:

```
NEW_MENU: mymenu_menu
...
12  3  32  DRAW_MASK  mybutton1  ADD_BUTTON
...
...
BUILD_MENU
```

C:

```
NEW_MENU mymenu[4]=
{
... ,
ADD_BUTTON(12,  3, 32, DRAW_MASK, mybutton1),
... ,
...
};
BUILD_MENU( mymenu, 4);
```

Also see the example in the glossary entry for NEW_MENU.

Forth: ADD_GRAPHIC(col\ row\ action_mask\ graphic_obj_xaddr\ --)
C: ADD_GRAPHIC(uint col, uint row, uint action_mask, xaddr graphic_obj_xaddr)

Adds a graphic object to a menu. Inside a menu definition, ADD_GRAPHIC is a macro that inserts all the necessary information for a graphic object based on the relative screen position specified by col and row and the address of the graphic object. The action_mask is a mask used to control which actions may be passed to the object when Do_Menu is called. For most applications, DRAW_MASK should be used. All of the actions are single bit flags thus several actions can be ORed together to form an action mask. DRAW_MASK is a constant that ORs DRAW_ACTION, ERASE_ACTION, DIR_DRAW_ACTION, REDRAW_ACTION, DIR_ERASE_ACTION, DRAW_TEXTONLY_ACTION, and

ERASE_TEXTONLY_ACTION. Since a graphic is simply a static image, it has no touchscreen button associated with it. Here is an example of the usage:

Forth:

```
NEW_MENU: mymenu_menu
...
3  32  DRAW_MASK     MY_LOGO  ADD_GRAPHIC
...
...
BUILD_MENU
```

C:

```
NEW_MENU mymenu[4]=
{
... ,
ADD_GRAPHIC( 3, 32,  DRAW_MASK,  MY_LOGO),
... ,
...
};
BUILD_MENU( mymenu, 4);
```

Also see the example in the glossary entry for NEW_MENU.

Forth: ADD_TOUCH_BUTTON(col\ row\ action_mask\ button_obj --)
C: ADD_TOUCH_BUTTON(uint col, uint row, uint action_mask, BUTTON * button_obj)

Places an initializing data into a menu for a button. Inside a menu definition, ADD_TOUCH_BUTTON is a macro that inserts all the necessary information for a button object based on the relative screen position specified by col and row and the address of the graphic object. The button_location is computed automatically based on the col and row given since there is a direct relationship between the touchscreen and the col and row position. The action_mask is a mask used to control which actions may be passed to the object when Do_Menu is called. For most applications, DRAW_MASK should be used. All of the actions are single bit flags thus several actions can be ORed together to form an action mask. DRAW_MASK is a constant that ORs DRAW_ACTION, ERASE_ACTION, DIR_DRAW_ACTION, REDRAW_ACTION, DIR_ERASE_ACTION, DRAW_TEXTONLY_ACTION, ERASE_TEXTONLY_ACTION. Here is an example of the usage:

Forth:

```
NEW_MENU: mymenu_menu
...
3  32  DRAW_MASK  mybutton1  ADD_TOUCH_BUTTON
...
...
BUILD_MENU
```

C:

```
NEW_MENU mymenu[4]=
{
... ,
ADD_TOUCH_BUTTON(3, 32, DRAW_MASK, mybutton1),
... ,
...
};
BUILD_MENU( mymenu, 4);
```

Also see the example in the glossary entry for NEW_MENU.

Forth: BLANKBUTTON(flags\ draw_graphic_xaddr\ release_graphic_xaddr\ press_graphic_xaddr\ press_handler\ release_handler\ label1\ label2\ label3\ label4 <name> --)

C: BLANKBUTTON(uint flags, xaddr draw_graphic_xaddr, xaddr release_graphic_xaddr, xaddr press_graphic_xaddr, (void *) press_handler, (void *) release_handler, char * label1, char * label2, char * label3, char * label4,<name>)

Creates a new button object. This macro integrates the creation and initialization of the BUTTON structure. BLANKBUTTON is a lower level macro than its more used cousins, FASTBUTTON and NORMBUTTON. It simply automates the creation of the object. The specified value for the flags is stored in the flags field of the button. The other parameters are stored in their respective fields in new button. The name given to the new button is specified by <name>. Here is an example:

Forth:

```
DRAW_GRAPHIC_FLAG                \ It has a draw graphic
RELEASE_GRAPHIC_FLAG or          \ It has a release graphic
PRESS_GRAPHIC_FLAG or            \ It has a press graphic
DRAW_TEXT_FLAG or                \ It has text
PRESS_HANDLER_FLAG or            \ It has a press handler
GRAPHICS_UPDATE_PRESS_FLAG or    \ Call Update_Graphics on press
GRAPHICS_UPDATE_RELEASE_FLAG or  \ Call Update_Graphics on release
LBLANK_PCX                       \ Graphic for DRAW_ACTION
LBLANK_PCX                       \ Graphic for RELEASE_ACTION
LBLACK_PCX                       \ Graphic for PRESS_ACTION
cfa.for myfunction         \ The code xaddress for press handler
0\0                              \ Dummy value for release handler
" "                              \ Line 1 label
" Start"                        \ Line 2 label
" Pump"                         \ Line 3 label
" "                              \ Line 4 label
BLANKBUTTON mybutton1            \ Instantiate the new button
```

C:

```
BLANKBUTTON(
DRAW_GRAPHIC_FLAG |        // Has draw graphic
RELEASE_GRAPHIC_FLAG |           // It has a release graphic
PRESS_GRAPHIC_FLAG |             // It has a press graphic
DRAW_TEXT_FLAG |                 // It has text
PRESS_HANDLER_FLAG |             // It has a press handler
GRAPHICS_UPDATE_PRESS_FLAG |     // Call Update_Graphics on press
GRAPHICS_UPDATE_RELEASE_FLAG,    // Call Update_Graphics on release
LBLANK_PCX,                      // Graphic for DRAW_ACTION
LBLANK_PCX,                      // Graphic for RELEASE_ACTION
LBLACK_PCX,                      // Graphic for PRESS_ACTION
myfunction,                      // The code address for press handler
0,                               // Dummy value for release handler
"",                              // Line 1 label
"Start",                         // Line 2 label
"Pump",                          // Line 3 label
"",                              // Line 4 label
mybutton1);                      // Instantiate the new button
```

Forth: BUILD_MENU

C: BUILD_MENU(arrayname, num_elements)

See NEW_MENU.

Forth: Button_Draw (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Draw(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the DRAW_ACTION of a button. See **Do_Button.**

Forth: Button_Draw_Textonly (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Draw_Textonly(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the DRAW_TEXTONLY_ACTION of a button. See **Do_Button.**

Forth: Button_Erase_Textonly (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Erase_Textonly(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the ERASE_TEXTONLY_ACTION of a button. See **Do_Button.**

Forth: Button_Press (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Press(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the PRESS_ACTION of a button. See **Do_Button.**

Forth: Button_Release (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Release(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the RELEASE_ACTION of a button. See **Do_Button.**

Forth: Button_Repeat (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Button_Repeat(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, BUTTON * button_xaddr)
Direct call to that carries out the REPEAT_ACTION of a button. See **Do_Button.**

Forth: Clear_Graphics (tvars_addr\ tvars_page --)
C: void Clear_Graphics(GUI_VARS * tvars_addr, page tvars_page)
Clears the graphics array by filling it with background_fill, a member of the tvars struct. Clear_Graphics then calls Update_Graphics so that the graphics layer on the display is cleared.

Forth: Clear_Text (tvars_addr\ tvars_page --)
C: void Clear_Text(GUI_VARS * tvars_addr, page tvars_page)
Clears the text array by filling it with spaces. ASCII values in this array are shifted down by 0x20, a requirement of the TC6963 display controller. Clear_Text then calls Update_Text so that the text layer on the display is cleared.

Forth: colrow_to_button (row\ col -- button number)
C: COLROW_TO_BUTTON(col,row)
Converts from column and row coordinates to a button number (0-19). This macro is used by ADD_TOUCH_BUTTON to convert the specified graphical positional information to a button number corresponding to the touchscreen button location.

Forth: Config_Display (graphics_cols\ graphics_rows\ graphics_start\ background_fill\ text_cols\ text_rows\ text_start\ heap_bottom\ heap_top\ tvars_addr\ tvars_page --)
C: void Config_Display(uint graphics_cols, uint graphics_rows, addr graphics_start, uchar background_fill, uint text_cols, uint text_rows, addr text_start, xaddr heap_bottom, xaddr heap_top, GUI_VARS * tvars_addr, page tvars_page)
Fills the variables that control the display initialization with configuration parameters. Below is a summary of the parameters.

graphic_cols, graphic_rows -- the col, row size of the graphics array
graphics_start -- the starting address inside the LCD display for the graphics data
background_fill -- the background fill byte used by the Clear_Graphics function
text_cols, text_rows -- the col, row size of the text array
text_start -- the starting address inside the display for the text data
heap_bottom -- the xaddress of the first byte of the display heap to be used
heap_top -- the xaddress of the last byte of the display heap to be used
tvars_addr, tvars_page -- the address of the structure tvars

Although this function fills the variables that control the initialization of the display, it does NOT initialize the display. That must be done by Init_Display. Config_Display simply initializes the variables needed by Init_Display. Std_Display calls this function to set up the display according to a generic set of defaults. If you are using a Panel-Touch Controller, use Std_Display. See Std_Display for the default values.

Forth: Direct_Draw_Graphic (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Direct_Draw_Graphic(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, FORTH_CONST_ARRAY * graphic_xaddr)
Direct call to that carries out the DIR_DRAW_ACTION of a graphic. See **Do_Graphic.**

Forth: Direct_Erase_Graphic (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Direct_Erase_Graphic(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, FORTH_CONST_ARRAY * graphic_xaddr)
Direct call to that carries out the DIR_ERASE_ACTION of a graphic. See **Do_Graphic.**

Forth: Do_Button (col\ row\ tvars_addr\ tvars_page\ action\ xpfa --)
C: void Do_Button(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, uint action, BUTTON * button_xaddr)

Action handler for all button objects. Col and row specify the location of the upper left corner of the image. Tvars_addr and tvars_page is the global structure that contains control variables used by the GUI Toolkit. This function's behavior is determined by the action passed to it. The actions are described below. The button_xaddr refers to a button object's xaddress. The button objects are structures of type BUTTON that contain xaddresses of three graphic objects, the draw graphic, release graphic, and press graphic. The button structure also contains a bitmapped set of switches that further shape the behavior of the button as well as xaddresses for the user code to be executed upon press, release, or both depending on which bitmapped flags are set. Here is how Do_Button is called:

```
C: Do_Button ( <col>, <row>, <tvars>, <action>, <button_xaddr> );
Forth: <col> <row> <tvars> <action> <button_xaddr> Do_Button
```

Buttons are used as parts of menus. The menu manager is responsible for calling this function to react to a touchscreen or keypad button press/release detection. Generally, your application would not call this function directly, but you can use this function to simulate button a press/release. This function can be used to produce the exact same effect as actually pressing or releasing the button. The action passed to Do_Button is one of the following predefined constants:

DRAW_ACTION
Draws the draw graphic member of the button structure to the graphics array with col, row as the upper left corner. You must subsequently call Update_Graphics to make the image appear on the LCD display unless the DIR_DRAW_GRAPHIC_FLAG is set in which case the graphic will be drawn directly to the display. See DIR_DRAW_ACTION under Do_Graphic.

DIR_DRAW_ACTION
For button objects, this action is equivalent to DRAW_ACTION. In order for a button to be directly drawn to the display, the flag DIR_DRAW_GRAPHIC_FLAG must be set. See DRAW_ACTION.

REDRAW_ACTION
For button objects, does the same thing as DRAW_ACTION.

ERASE_ACTION
Writes the tvars background_fill byte to the graphics array in the area previously occupied by the button object at specified screen location. If the DIR_DRAW_GRAPHIC_FLAG is set, then the button is directly erased from the display. If that flag is not set, then you must then call Update_Graphics to make the change evident on the screen.

DIR_ERASE_ACTION
For button objects, this action is equivalent to ERASE_ACTION. In order for a button to be directly erased from the display, the flag DIR_DRAW_GRAPHIC_FLAG must be set. See ERASE_ACTION.

DRAW_TEXTONLY_ACTION
If the button has text labels, then this action causes them to be printed to the display. You must call Update_Text for this to become apparent on the screen.

ERASE_TEXTONLY_ACTION
If the button has text labels, then this action causes them to be erased from the display. You must call Update_Text for this to become apparent on the screen.

PRESS_ACTION
Executes the user code press_handler and draws the release graphic depending on the value of the flags. If the PRESS_HANDLER_FLAG is set, then the code in the press_handler field of the button structure is executed. If the PRESS_GRAPHIC_FLAG is set, then the graphic object for the press_graphic field is drawn. If the TEXT_UPDATE_PRESS_FLAG is set, then Update_Text is called. If the GRAPHIC_UPDATE_PRESS_FLAG is set, then Update_Graphics is called. If none of those flags is set, then this action has no effect.

RELEASE_ACTION
Executes the user code release_handler and draws the release graphic depending on the value of the flags. If the RELEASE_HANDLER_FLAG is set, then the code in the release_handler field of the button structure is executed. If the RELEASE_GRAPHIC_FLAG is set, then the graphic object for the release_graphic field is drawn. If the TEXT_UPDATE_RELEASE_FLAG is set, then Update_Text is called. If the GRAPHIC_UPDATE_RELEASE_FLAG is set, then Update_Graphics is called. If none of those flags is set, then this action has no effect.

REPEAT_ACTION
Executes the user code press_handler without drawing the press graphic. If the REPEAT_FLAG or PRESS_HANDLER flags are not set, this action has no effect. When a button is repeating, it is a waste of processor time to redraw the same graphic for each repetition.

Forth: Do_Graphic (col\ row\ tvars_addr\ tvars_page\ action\ graphic_xaddr --)
C: void Do_Graphic (uint col, int row, GUI_VARS * tvars_addr, page tvars_page, uint action, FORTH_CONST_ARRAY * graphic_xaddr)
Action handler for all graphics objects. Col and row describe the position of the upper left corner of the object in absolute coordinates measured from the upper left corner of the display. This function's behavior is determined by the action flag passed to it. The actions are described below. The graphic_xaddr refers to a graphics object. Typically, such objects are created by the pcx2qed program which converts a pcx graphic image on a PC to a block of data that can be loaded into the QED board. Pcx2qed also provides a symbol listing of constants named based on the filename of the graphic on the PC. This symbol listing may be #included in a C file, or pasted into a forth file. The language in which the constants are defined can be set by a flag passed to the pcx2qed utility. The constant referring to the graphic objects address takes the place of the graphic_xaddr. For example, if the original image was named logo.pcx on the PC, then its name as a graphic object would be LOGO_PCX. Do_Graphic would then be called as follows:

```
C:  Do_Graphic ( <col>, <row>, <tvars>, <action>, LOGO_PCX );
Forth: <col> <row> <tvars> <action> LOGO_PCX Do_Graphic
```

The actions passed to Do_Graphic can be one of the following predefined constants:

DRAW_ACTION
Draws the graphic object to the graphics array with col, row as the upper left corner. You must subsequently call Update_Graphics to make the image appear on the LCD display.

DIR_DRAW_ACTION
Draws the graphics object directly to the LCD display bypassing the graphics array. This is useful for fast screen updates and animation. Since DIR_DRAW_ACTION draws directly to the screen, and not to the graphics array, calling Update_Graphics will overwrite anything placed on the screen by DIR_DRAW_ACTION.

REDRAW_ACTION
For graphic objects, does the same thing as DRAW_ACTION.

ERASE_ACTION
Writes the tvars background_fill byte to the graphics array in the area previously occupied by the graphic object at specified screen location. You must then call Update_Graphics to make the change evident on the screen.

DIR_ERASE_ACTION
Writes the tvars background_fill byte directly to the LCD display over the area previously occupied by the graphic object at specified screen location. See DIR_DRAW_ACTION.

Forth: Do_Menu (col\ row\ tvars_addr\ tvars_page\ action\ menu_xaddr --)
C: void Do_Menu(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, uint action, MENU * menu_xaddr)
Counts through each element of the menu executing each object with the specified action. A menu may consist of graphic or button objects. For example, calling Do_Menu with the action DRAW_ACTION would execute each object in the menu with that action. The relative col and row of the objects is stored in the menu. The col and row passed to Do_Menu is the desired position on the display of the upper left corner of the entire menu. The col and row passed to Do_Menu will be added to the col and row of the objects stored in the menu to get the absolute locations of the actual objects contained in the menu. That new col and row will then be passed to the object along with the action flag. Most commonly, this function is used to draw or redraw a menu to the screen. Each element of the menu array has an action mask which is ANDed with the action flag passed to Do_Menu before the object in the menu is executed. If the ANDed result is zero, the object is skipped. This allows certain objects in a menu to have some action flags disabled. See the following example:

```
C:  Do_Menu ( <col>, <row>, <tvars>, <action>, <menu_xaddr> );
Forth: <col> <row> <tvars> <action> <menu_xaddr> Do_Menu
```

Forth: Draw_Graphic (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Draw_Graphic(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, FORTH_CONST_ARRAY * graphic_xaddr)
Direct call to that carries out the DRAW_ACTION of a graphic. See **Do_Graphic.**

Forth: Erase_Graphic (col\ row\ tvars_addr\ tvars_page\ xpfa --)
C: void Erase_Graphic(uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, FORTH_CONST_ARRAY * graphic_xaddr)
Direct call to that carries out the ERASE_ACTION of a graphic. See **Do_Graphic.**

Forth: FASTBUTTON(flags\ draw_graphic_xaddr\ release_graphic_xaddr \press_graphic_xaddr \ handler\ label1\ label2\ label3\ label4 <name> --)
C: FASTBUTTON(uint flags, xaddr draw_graphic_xaddr, xaddr release_graphic_xaddr xaddr press_graphic_xaddr, (void *) handler, char * label1, char * label2, char * label3, char * label4,<name>)
Works in exactly the same way as NORMBUTTON, but with a different set of default flags. FASTBUTTON has the default flags, DRAW_GRAPHIC_FLAG, RELEASE_GRAPHIC_FLAG, PRESS_GRAPHIC_FLAG, DIR_PRESS_GRAPHIC_FLAG, and DIR_RELEASE_GRAPHIC_FLAG. This type of button uses the direct screen drawing for the pressed and released graphics, and standard graphics array drawing for the initial drawing of the buttons. This technique is quite effective for maximizing the responsiveness of the user interface while still loosely following the paradigm of using a graphics array. It eliminates the need to update the entire screen when only a small portion the size of a button is changing. When using direct to screen drawing, you should not specify the GRAPHICS_UPDATE_PRESS_FLAG or GRAPHICS_UPDATE_RELEASE_FLAG since updating the display will overwrite the directly drawn graphics. See **graphics objects** in the *Glossary of Terms* for more information. Also see **NORMBUTTON** in the *Glossary of Functions*.

Forth: Init_Display (tvars_addr\ tvars_page --)
C: void Init_Display(GUI_VARS * tvars_addr, page tvars_page)
High level function that initializes the graphics hardware. This function sets up a Toshiba TC6963C according to the variables in the tvars struct initialized by Config_Display. Init_Display dimensions the graphics and text arrays to the appropriate sizes in the display heap and enables the display hardware. Init_Display zeros the display_resource variable in the tvars struct.

Forth: Init_Menu (offset\ col\ row\ tvars_addr\ tvars_page\ menu xpfa --)
C: void Init_Menu(uint offset, uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, MENU * menu_xaddr)
Draws and installs a menu at the given offsets. Offsets for the screen position, col and row, are applied to the relative locations of the objects contained in the menu. The resulting absolute screen locations are used to draw the objects to the screen. Init_Menu then calls Menu_Install. Each element of the menu has a relative button number associated with it which is added to the offset to get an absolute button

number. The resulting absolute button number is the keymap array index in which the button object reference is stored by Menu_Install. This function is equivalent to passing the DRAW_ACTION flag to Do_Menu followed by a call to Menu_Install. When changing from one menu to another, you should call Uninit_Menu for the old menu before calling Init_Menu for the next menu. Afterwards, you must update the display since Uninit_Menu and Init_Menu do not automatically update the display. See Menu_Install, Uninit_Menu, and Do_Menu.

Forth: Init_Touch (tvars_addr\ tvars_page --)
C: void Init_Touch(GUI_VARS * tvars_addr, page tvars_page)
Initializes the touchscreen variables keymap_array, repeat_delay, and repeat_period in the tvars struct. It dimensions the keymap array for a 20 button touchscreen/keypad in the current heap. All the elements are then filled with 0x00. There must be a valid heap with enough room for the keymap array prior to calling this Init_Touch. Repeat_delay and repeat_period are initialized to 80 and 10 timeslice counts respectively. This assumes that the timeslicer period is set to its default value of 5 mS. If you change the timeslice period, may be necessary to adjust the repeat_delay and repeat_period variables to maintain desired operation. This function should be called as part of the start up initialization.

Forth: Menu_Install (offset\ col\ row\ tvars_addr\ tvars_page\ menu_xaddr --)
C: void Menu_Install(uint offset, uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, MENU * menu_xaddr)
Copies each item from the menu object's array to the keymap array. Col, row, and offset determine the position of the menu on the display. Each element of the menu has a relative button number associated with it which is added to the offset to get an absolute button number. The resulting absolute button number is the keymap array index in which the button object reference is stored. Offsets for the screen position location (col and row) are applied to the relative locations of the buttons contained in the menu. The resulting absolute screen locations are stored in the keymap array. Menu_Install does not draw the menu, but only places its buttons in the keymap so that the menu manager will be able to access them. This function is called by Init_Menu. See Menu_Query, Menu_Remove, and Init_Menu.

Forth: Menu_Process (button number \ tvars_addr \ tvars_page --)
C: void Menu_Process(int button, GUI_VARS * tvars_addr, page tvars_page)
This routine is very similar to Menu_Query. Instead of polling the hardware and then reacting as Menu_Query does, this function accepts a button number as collected by the calling environment using one of the built-in kernel functions. Menu_Process then processes the button exactly as Menu_Query does, and if the button is being held down, then Menu_Process blocks until it is released. This routine effectively serves as a non-blocking version of Menu_Query. See Menu_Query. The following examples show how to implement Menu_Query yourself using Menu_Process and the builtin kernel driver for the keypad:

C:

```
void My_Menu_Query ( GUI_VARS * tvars_addr, page tvars_page )
{
```

```
  int this_press=-1;      // Init to default
  while (this_press==-1)  // Keep looping until a button is pressed
  {
    this_press = ScanKeypress();
  }
  Menu_Process( this_press, TVARS );  // we have a button press!  Act on it
}
```

Forth:

```
: my_menu_query ( tvars -- )
  locals{ x&tvars }
  begin
    ?keypress         \ ( [ button number \ true ] or [ false ] -- )
    if                \ ( tvars \ button number -- )
      x&tvars menu_process  \ Process the button number received
      true            \ cause an exit
    else
      false           \ cause the loop to continue
    endif
  until               \ loop if nothing yet
  ;
```

Forth: Menu_Query (tvars_addr\ tvars_page --)
C: void Menu_Query(GUI_VARS * tvars_addr, page tvars_page)

This routine serves as a runtime menu manager for monitoring the user input hardware (touchscreen or keypad). It waits for a keypad or touchscreen press. When a button is pressed, held, or released the touchscreen/keypad hardware driver returns a button number which is used as an index to the keymap array. Menu_Query then examines the indexed element of the keymap array and invokes the PRESS_ACTION, REPEAT_ACTION, or RELEASE_ACTION to the object whose xaddress is stored in the indexed keymap array element. Menu_Query does not loop. After one press/release cycle, it exits. If the button has the REPEAT_FLAG set, then Menu_Query invokes the REPEAT_ACTION to the button repeatedly according to the repeat times, represented in timeslicer counts, stored in the repeat_period and repeat_delay variables in the tvars struct. The timeslicer must be running for buttons to repeat. Menu_Query is usually used inside a loop that iterates for each press/release cycle. Menu_Query calls Pause while awaiting a button press. Any user handlers associated with the buttons will run under the same task as Menu_Query since Menu_Query executes the code.

Forth: Menu_Remove (offset\ tvars_addr\ tvars_page\ menu_addr\ menu_page --)
C: void Menu_Remove(uint offset, GUI_VARS * tvars_addr, page tvars_page, MENU * menu_xaddr)

Removes a menu from the keymap array. Each element of the menu has a relative button number that is added to the offset to determine which button areas of the keymap array are occupied by the objects of the menu. Those areas are marked as unused by filling them with 0x00. Menu_Remove does not erase the menu from the screen. Uninit_Menu calls Menu_Remove. See **Uninit_Menu** and **Menu_Install** in the *Glossary of Functions*.

Forth: NEW_MENU: and BUILD_MENU

C: NEW_MENU and BUILD_MENU(arrayname, num_elements)

Declares a new menu array. There are syntactical differences in how this macro is used between C and forth. In C, NEW_MENU is actually a synonym for MENU_ENTRY, a structure type. Here is an example:

Forth:

```
NEW_MENU: mymenu_menu
\ Below, the Col and Row are the screen positions of the upper left
\ corners of the objects relative to the upper left corner of the menu.
\ The upper left corner of the menu on the screen is determined by the
\ values passed to Init_Menu.
\  Col Row   Action mask  Object xaddress     Object adding macro

0      38     DRAW_MASK    numshift_button     ADD_TOUCH_BUTTON
8      70     DRAW_MASK    numdec_button       ADD_TOUCH_BUTTON
24     102    DRAW_MASK    num0_button         ADD_TOUCH_BUTTON
3      32     DRAW_MASK    MY_LOGO             ADD_GRAPHIC
32     0      DRAW_MASK    mybutton1           ADD_TOUCH_BUTTON
BUILD_MENU
```

C:

```
NEW_MENU mymenu[5]=
{
// Below, the Col and Row are the screen positions of the upper left
// corners of the objects relative to the upper left corner of the menu.
// The upper left corner of the menu on the screen is determined by the
// values passed to Init_Menu.
// Object adding macro     (Col, Row, Action mask, Object xaddress)

ADD_TOUCH_BUTTON(  0,   38,  DRAW_MASK, numshift_button ),
ADD_TOUCH_BUTTON(  8,   70,  DRAW_MASK, numdec_button ),
ADD_TOUCH_BUTTON(  24, 102, DRAW_MASK, num0_button ),
ADD_GRAPHIC(           3,   32,  DRAW_MASK, MY_LOGO),
ADD_TOUCH_BUTTON(  32,  0, mybutton1)
};
BUILD_MENU( mymenu, 5);
```

The menus built above are identical. When programming in forth, NEW_MENU acts as a defining word that uses the stack to pass information to BUILD_MENU which instantiates the menu. This should be done at compile time, not inside a colon definition. Since C doesn't have the ability for two functions to communicate at compile time, it is important to restate the number of elements in the call to BUILD_MENU. The name of the final menu pointer in both cases is mymenu_menu. In C, a C style array called mymenu must first be created which is then used to create a forth array parameter field called mymenu_menu. BUILD_MENU automatically appends the suffix, _menu to the base name. This modified name is the name by which the menu should be referred in calls to menu related functions, not mymenu. In forth, the entire data structure is built at once and only has one name.

Forth: NORMBUTTON(flags\ draw_graphic_xaddr\ release_graphic_xaddr \press_graphic_xaddr \ handler\ label1\ label2\ label3\ label4 <name> --)

C: NORMBUTTON(uint flags, xaddr draw_graphic_xaddr, xaddr release_graphic_xaddr, xaddr press_graphic_xaddr, (void *) handler, char * label1, char * label2, char * label3, char * label4,<name>)

Creates a new button object. This macro integrates the creation and initialization of the BUTTON structure with some useful defaults. NORMBUTTON builds a button that uses all three graphics (draw, release, and press). By default, only DRAW_GRAPHIC_FLAG, RELEASE_GRAPHIC_FLAG, and PRESS_GRAPHIC_FLAG are set. Additional flags should be specified to further shape the button's behavior. These flags are ORed with the default flags. For the handler to be executed, you must specify PRESS_HANDLER_FLAG or RELEASE_HANDLER_FLAG. If both PRESS_HANDLER_FLAG and RELEASE_HANDLER_FLAG are specified, then the handler is executed twice, once when the button is pressed, and again when it is released. Other flags that might be useful are REPEAT_FLAG to make the button repeat or DRAW_TEXT_FLAG if the label text is to be printed in the button. The other parameters are stored in their respective fields in new button. The name given to the new button is specified by <name>. Here is an example:

Forth:

```
DRAW_TEXT_FLAG or                \ It has text
PRESS_HANDLER_FLAG or            \ It has a press handler
GRAPHICS_UPDATE_PRESS_FLAG or    \ Call Update_Graphics on press
GRAPHICS_UPDATE_RELEASE_FLAG or  \ Call Update_Graphics on release
LBLANK_PCX                       \ Graphic for DRAW_ACTION
LBLANK_PCX                       \ Graphic for RELEASE_ACTION
LBLACK_PCX                       \ Graphic for PRESS_ACTION
cfa.for myfunction               \ The code xaddress for press handler
" "                              \ Line 1 label
" Start"                         \ Line 2 label
" Pump"                          \ Line 3 label
" "                              \ Line 4 label
NORMBUTTON mybutton1             \ Instantiate the new button
```

C:

```
NORMBUTTON(
DRAW_TEXT_FLAG |                 // It has text
PRESS_HANDLER_FLAG |             // It has a press handler
GRAPHICS_UPDATE_PRESS_FLAG |     // Call Update_Graphics on press
GRAPHICS_UPDATE_RELEASE_FLAG,    // Call Update_Graphics on release
LBLANK_PCX,                      // Graphic for DRAW_ACTION
LBLANK_PCX,                      // Graphic for RELEASE_ACTION
LBLACK_PCX,                      // Graphic for PRESS_ACTION
myfunction,                      // The code address for press handler
"",                              // Line 1 label
"Start",                         // Line 2 label
"Pump",                          // Line 3 label
"",                              // Line 4 label
mybutton1);                      // Instantiate the new button
```

Forth: Set_Cursor_State (isvisible\ isflashing --)
C: void Set_Cursor_State(boolean isvisible, boolean isflashing)

Calls Set_Display_Mode to sets the state of the cursor using the 2 flags. Isvisible is true if the cursor is visible and false if not. Isflashing is true for flashing and false for non-flashing. See PutCursor (forth: put.cursor) in the main QED glossary.

Forth: Set_Display_Mode (mode --)
C: void Set_Display_Mode(uint mode)

Sets the mode byte of the display controller. This word uses the lower 4 bits of mode to determine the operating mode. Details about the meaning of the mode flag can be found in the datasheet for the TC6963 controller, but it is handled for you by Set_Cursor_State and Set_Display_State which call Set_Display_Mode.

Forth: Set_Display_State (graphics\ text --)
C: void Set_Display_State(boolean graphics, boolean text)

Calls Set_Display_Mode to enable or disable graphics or text according to the 2 flags. Graphics is true to indicate that the graphics layer is enabled and text is true to indicate that the text layer is enabled. Init_Display sets this automatically based on the display configuration specified by Config_Display. Either text or graphics may be written to the display even when that layer has been disabled with this function. The layer may then be re-enabled to make visible the data currently stored in the display. This may be useful for blanking the screen during an update. Most applications don't need this ability.

Forth: Set_Gr_Area (columns --)
C: void Set_Gr_Area(uint columns)

Sets the width for graphics in the TC6963 display controller. The graphics area value should be equal to the number of graphics columns. This is not the same as the number of pixels of display width, but the number of bytes required to represent a line of graphics. The 240x128 display is configured for 6 bits per byte meaning that the number of columns is 240/6 or 40, the same as text. Init_Display sets this automatically.

Forth: Set_Gr_Home_Addr (address --)
C: void Set_Gr_Home_Addr(addr address)
Sets the home address for graphics in the TC6963 display controller. Only the rarest of circumstances require altering the display's internal memory configuration. The address specified will become the starting address inside the display module for the graphics data. It is set automatically by Init_Display.

Forth: Set_Text_Area (columns --)
C: void Set_Text_Area(uint columns)
Sets the width for text in the TC6963 display controller. The text area value should be equal to the number of character columns. Characters are 6 pixels wide meaning that there are 240/6 or 40 text columns on the display. Init_Display sets this automatically.

Forth: Set_Text_Home_Addr (address --)
C: void Set_Text_Home_Addr(addr address)
Sets the home address for text in the TC6963 display controller. Only the rarest of circumstances would require altering the display's internal memory configuration. The address specified will become the starting address inside the display module for the text data. It is set automatically by Init_Display.

Forth: Set_Text_Mode (modebyte --)
C: void Set_Text_Mode(uchar modebyte)
Sets the text attribute bits. Only the lower 4 bits used, and the other bits are ignored. Bit 3 is 0 for character generator ROM mode and 1 for character generator RAM mode. Unless you are using custom fonts uploaded to the display, this bit should be 0. If it is set to 1, then the cg offset pointer, a register in the TC6963 display controller chip, must be set to point to the base address of the character table in the display's RAM. An example of how to do this may be found in the fonts directory of the distribution of the GUI Toolkit package. Bits 2, 1, and 0 determine the display mode for text.

Constant	Bit2	Bit1	Bit0	Description
OR_TEXT	0	0	0	Text is ORed with graphics (default)
EXOR_TEXT	0	0	1	Text is EXORed with graphics
AND_TEXT	0	1	1	Text is ANDed with graphics
	1	0	0	Text is in special attribute mode. This specialized mode is not usable with graphics mode and is not discussed here. See TC6963 Datasheet for more info.

Forth: Std_Display (tvars_addr\ tvars_page --)
C: void Std_Display(GUI_VARS * tvars_addr, page tvars_page)
Sets the display configuration information in the tvars struct to default values for the 240x128 display used on the Panel-Touch controller. When using such a display,

simply calling this function prior to calling Init_Display will eliminate the need to use Config_Display which can be unwieldy. This function calls Config_Display with the following parameters. Don't forget to call Init_Display after calling Std_Display.

graphic_cols, graphic_rows -- 40, 128
graphics_start -- 0x0280
background_fill -- 0
text_cols, text_rows -- 40, 16
text_start -- 0x0000
heap_bottom -- 0x0F47FF
heap_top -- 0x0F3000

See **Init_Display** and **Config_Display** in the *Glossary of Functions*.

Forth: Uninit_Menu (offset\ col\ row\ tvars_addr\ tvars_page\ menu xpfa --)
C: void Uninit_Menu(uint offset, uint col, uint row, GUI_VARS * tvars_addr, page tvars_page, MENU * menu_xaddr)
Erases and uninstalls the menu at the given offsets. Offsets for the screen position, col and row, are applied to the relative locations of the objects contained in the menu. The resulting absolute screen locations are used to erase the objects from the screen. Uninit_Menu then calls Menu_Remove. Each element of the menu has a relative button number associated with it which is added to the offset to get an absolute button number. The resulting absolute button number is the keymap array index in which the button object reference is deleted by Menu_Remove. This function is equivalent to calling Do_Menu with the ERASE_ACTION flag followed by a call to Menu_Remove. When changing from one menu to another, you should call Uninit_Menu for the old menu before calling Init_Menu for the next menu. Afterwards, you must update the display since Uninit_Menu and Init_Menu do not automatically update the display. See Menu_Remove, Init_Menu, and Do_Menu

Forth: Update_Graphics (tvars_addr\ tvars_page --)
C: void Update_Graphics (GUI_VARS * tvars_addr, page tvars_page)
Calls Update_Here_With to send the entire contents of the graphics array to the LCD display. Call this function after modifying the graphics array to update the display. The contents of the graphics array are transferred to the memory address inside the display specified by Gr_Home_Addr in the tvars struct. Any graphics that were drawn directly to the LCD bypassing the graphics array will be overwritten when Update_Graphics is called.

Forth: Update_Here_With (address\ graphics_resource_addr\ graphics_resource_page\ garray_xaddr --)
C: void Update_Here_With(addr address, addr * graphics_resource_addr, page graphics_resource_page, FORTH_ARRAY * garray_xaddr)
Directly copies the contents of the 2 dimensional array pointed to by garray_xaddr to the display starting at address in the display's memory. This function honors the resource variable pointed to by graphics_resource_addr and graphics_resource_page.

The display resource must be available or this function will hold up execution until it can take control of the display to perform the update. Update_Here_With is a low level function that shouldn't be needed in most circumstances. It is called by Update_Text and Update_Graphics. Update_Here_With is useful for loading special areas of the display memory with data such as custom fonts. See Update_Text and Update_Graphics.

Forth: Update_Text (tvars_addr\ tvars_page --)
C: void Update_Text(GUI_VARS * tvars_addr, page tvars_page)
Calls Update_Here_With to send the entire contents of the text array to the LCD display. Call this function after modifying the text array to update the display. The contents of the text array are transferred to the memory address inside the display specified by Text_Home_Addr in the tvars struct.

Forth: Update_Text_And_Graphics (tvars_addr\ tvars_page --)
C: void Update_Text_And_Graphics(GUI_VARS * tvars_addr, page tvars_page)
Sends the contents of the text and graphics arrays to the LCD display. This function is the equivalent of calling Update_Text and Update_Graphics. See Update_Text and Update_Graphics