The Forth Programmer's Guide to the QVGA Controller

Mosaic
Industries

The Forth Programmer's Guide to the QVGA Controller
by Mosaic Industries, Inc.

Printed in the United States of America

Published by: Mosaic Industries, Inc.
5437 Central Ave. Suite 1
Newark, CA 94560, USA
510-790-8222
www.mosaic-industries.com

Printing History (Revision Notice):

June 2003: Draft v0.5

Not Approved for Life-Support Use

Mosaic's embedded computers, software, and peripherals are intended for use in a wide range of OEM products, but they are not designed, intended or authorized for use as components in life support or medical devices. They are not designed for any application in which the failure of the product could result in personal injury, death or property damage.

Complex software often contains bugs, and electronic components fail. Where a failure may cause serious consequences, it is essential that the product designer protect life and property against such consequences by incorporating redundant backup systems or safety devices appropriate to the level of risk. The buyer agrees that protection against consequences resulting from system failure, of either hardware or software origin, is solely the buyer's responsibility.

ISBN: 7/03

Table of Contents

The Forth Programmer's Guide to the QVGA Controller

How To Use This Book vii

PART 1 GETTING STARTED

Chapter 1: Getting to Know Your QVGA Controller

Introducing the QVGA Controller 3
Real-Time Operating System and Built-In Function Library 3
Choice of Programming Languages 4
Graphical User Interface and GUI Toolkit 5
Memory and Mass Memory 5
Communications 6
Measurement and Control 6

Getting to Know Your Hardware 6
A Tour of Connectors and Switches 8

Configuring QVGA Options 11
Setting the QED DIP Switches 11
Setting the QED Board Jumpers 12
Setting the QVGA Board Jumpers 12
The Remote Mounting Header 12

Chapter 2: Your First Program

Installing the Mosaic IDE 15

Turning on Your QVGA Controller 16
Removing and ReInstalling the Demo Program 17

Using the Mosaic IDE 19
Using the Editor and Compiler 19

An Introduction to Programming in QED-Forth 20
Initializing the Memory Map 23
Output Commands 24
The Stack 24
Defining New Words in QED-Forth 26
A Definition Using Floating Point Math and Local Variables 27
Multitasking 28
Working with Matrices 28
Solving Simultaneous Equations 29
In Case Your Software Crashes 30
The Special Cleanup Mode 31
Saving Your Program in Flash 32

PART 2 PROGRAMMING THE QVGA CONTROLLER

Chapter 3: Making Effective Use of Memory

The QVGA Controller's Memory Map 35

Software Development Using Flash Memory 39

Chapter 4: Programming the Graphical User Interface

The Structure Of The GUI Toolkit 41
A Closer Look At Objects 41
The GUI Toolkit Objects 42

Building Your Application 43
Designing Your User Interface 43
Transferring Your Images to the QVGA Controller 47
Coding Your Application 47
Handling Errors 55
Expanding the GUI Toolkit's Objects and Methods 55

GUI Objects in Detail 55
GUI_TOOLKIT 56
GUI_DISPLAY 58
GUI Screens 58
GUI_TOUCHSCREEN 60
GUI_PEN 62
GUI_BUZZER 63
Graphic Object 64
Font Object 66
Textbox Object 67
Controls 68
Button Object 68
Plot Object 69

PART 3 COMMUNICATIONS MEASUREMENT AND CONTROL

Chapter 5: Digital and Timer-Controlled I/O

Overview of Available Digital I/O 73
Alternate Uses of the Digital I/O Ports 75
Using the Digital I/O Ports on the 68HC11 Chip 76
Using the PIA 81

Using the High Current Drivers 84
Using Uninterruptable Operators 85

Connecting Hardware to the Digital Outputs 86

Chapter 6: Data Acquisition Using Analog to Digital Conversion

Data Acquisition Using the QVGA Controller 91

Using the 8-bit A/D 92

Using the 12-bit A/D 92

Chapter 7: Outputting Voltages with Digital to Analog Conversion

Initializing the DAC 97

Applying a Reference Voltage 97

Outputting a Voltage 98

Getting Greater Resolution 98

DAC Execution Speed 99

Chapter 8: Serial Communications

RS-232 and RS-485 Communications 101
Serial Connectors 101
Serial Protocols 101
Using the Serial Ports 102
Setting Baud Rates 104
Multi-Drop Communications Using RS-232 105

Chapter 9: The Battery-Backed Real-Time Clock

How To Use This Book

Welcome, and thanks for purchasing the QVGA Controller™. This manual provides instructions for using your new embedded computer. The QVGA Controller is a state-of-the-art embedded microcontroller with an advanced operator interface. It's an ideal "brain" for instruments that need a highly visible graphical user interface (GUI), touchscreen control, computational power, I/O, and serial communications – all in one compact package.

Commanded remotely from a PC, or used stand-alone, it provides real-time control of dozens of analog and digital I/O lines. Use it for scientific instruments, machine or process control, or as an advanced operator interface for existing products. Its real-time multitasking operating system facilitates concurrent functioning of its user interface, I/O, and application software. You may use this compact, integrated device as the core hardware, software and user interface for your new products – wherever you need an I/O-rich computer and an advanced user interface.

The QVGA Starter Kit includes everything you need to develop a prototype instrument with an advanced GUI: the QVGA Controller with a monochrome or EL display, battery back-up for 128K RAM memory, a full documentation package, power supply and all cables. For a sleek look you can add a black anodized aluminum bezel. The QVGA Controller can be either flush mounted using the bezel or directly mounted to a panel with a cutout.

Prerequisite Knowledge

The QVGA Controller is intended for use by experienced programmers or any technically minded person up to the challenge of real-time programming. We assume that if you're designing a product requiring an embedded computer, you have experience in the design of the hardware and software needed to customize the QVGA Controller to your product, and an understanding of the basics of writing, compiling and debugging software programs. You should be comfortable programming in either the C or Forth programming languages; you can program the QVGA Controller in either. This manual is geared to the Forth programmer. If your would rather program in C, give us a call and we'll send you the C programmers manual. We recommend the following reference for novice programmers:

- *Starting Forth,* by Leo Brodie

Motorola's *M68HC11 Reference Manual* and *MC68HC11F1 Technical Data Manual* are included with this documentation package as Adobe Acrobat Portable Document Format (*.pdf) files.

How to Use this Documentation

This manual is laid out in several parts, in an order we hope you find most useful. We have invested a lot of effort to make this documentation instructive and helpful. The available software and hardware functions are all described in detail, and many coded examples are presented. For those who are designing "turnkeyed" instruments, we have included a complete turnkeyed application program. This well documented program illustrates how to use dozens of features, including the graphical user interface, interrupts, floating point math, formatted display of calculated results, multitasking, and automatic program startup. The source code is included on your CD-ROM. This sample program can serve as a useful template for a wide variety of applications. This manual contains the following parts:

Part 1,*Getting Started: A Quick Tour of the QVGA Controller,* will familiarize you with the QVGA Controller (Chapter 1) and its programming environment, and get you writing your first program (Chapter 2). These first two chapters guide you through the QVGA Controller's hardware, explain how to establish communications with it, and show you how to compile and run your first program.

After working through the examples of Chapter 2 you will have exercised some of the key hardware and software features of your controller. You might then leaf through the categorized list at the beginning of the Forth Function Glossary in an Appendix to get a feel for the wealth of precoded library functions available for you to use.

Part 2, *Programming the QVGA Controller,* provides everything you need to know to master real-time programming on the QVGA Controller.

Part 3, *Communications, Measurement and Control*, focuses on the QVGA's hardware resources – A/D, D/A, serial communications, timer-controlled I/O, real-time clock and others – and provides examples for using each.

Part 4, *Putting It All Together*, introduces a real-time interactive application, and provides code you can use as a template for your application. It also discusses the nuts and bolts of product integration, mounting, noise considerations and power requirements.

Part 5, *Reference Data*, contains detailed specifications, schematics, dimensions, data sheets and glossaries of Forth functions available for your use.

Conventions Used in This Book

The following conventions are use throughout this manual:

Abbreviations	
A/D	Analog to Digital Converter

Abbreviations

COP	Computer Operating Properly timer
D/A or DAC	Digital to Analog Converter
EEPROM	Electronically Erasable Programmable Read-Only Memory, nonvolatile
EL	Electroluminescent Display
FLASH	Flash Programmable Read-Only Memory, nonvolatile and on-the-fly reprogrammable
GUI	Graphical User Interface, also called an MMI (Man Machine Interface) or OI (Operator Interface)
I/O	Inputs and Outputs
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
PIA	Peripheral Interface Adapter (a chip that provides 24 digital I/O signals)
PROM	Programmable Read-Only Memory, nonvolatile one-time programmable
QED	Quod Erat Demonstrandum, or Quick Easy Design, whichever you prefer
QED-Forth	The name of the QVGA Controller's onboard operating system and interactive resident language.
RAM	Random Access Memory, volatile
RTC	Real-Time Clock
RTOS	Real Time Operating System
SPI	Serial Peripheral Interface, a fast bidirectional synchronous serial interface
SRAM	Static Random Access Memory, volatile
UART	Universal Asynchronous Receiver Transmitter

Throughout this manual the names of code functions and extended code segments are distinguished by their typeface. The following font styles are used:

Typefaces

English Text	Plain text uses a Times New Roman Font
C function names appearing within text	`void InstallMultitasker(void)`
Forth function names appearing within text	`BUILD.STANDARD.TASK`
C code in listings	`#define FULL_SCALE_COUNTS 256`
Forth code in listings	`256 CONSTANT FULL_SCALE_COUNTS`
Commands typed to the QVGA Controller through a terminal	`CFA.FOR main PRIORITY.AUTOSTART↵`
QVGA Controller responses to a terminal program	`ok`

Code function names and listings use a fixed space font. C code uses a font with serifs, QED-Forth code is sans serif. Terminal commands are indented and followed by a back-arrow symbol repre-

senting the enter key on the keyboard, and the QVGA Controller's responses are underlined with a dotted line. Both are indented in the text. Listings of more extensive code examples are set off by indenting and captioning.

Integer numbers are not accompanied by a decimal point; the presence of a decimal point indicates that the number is a floating-point format number. Decimal base numbers are written in standard form while binary numbers are written in hexadecimal (base sixteen, using 0-9 and A-F) and preceded with "**`0x`**", for example, the 16-bit integer 15,604 is represented: **`0x3CF4`**

Obtaining Code Examples and Example Applications

Please check our website periodically at www.mosaic-industries.com, where we'll be posting code examples and example applications, and providing software updates and enhancements.

For Technical Help (or just to chat) Contact Us

We have tested and verified the sample code in this user's guide to the best of our ability, but you may find that some features have changed, or even that we have made a mistake or two. If you find an error, please call us and we'll fix it pronto.

If you are facing a challenging software hurdle, or a hardware problem in interfacing to our products, please don't hesitate to email or call us. We can often help you over the hurdle and save you a lot of time. So contact us by phone or email:

510-790-8222

support@mosaic-industries.com

We provide free technical help to all registered, licensed users.

Part 1

Getting Started

Chapter 1

Chapter 1: Getting to Know Your QVGA Controller

Congratulations on your choice of the QVGA Controller™! This Chapter introduces the various hardware and software features of the QVGA Controller: the graphical user interface (GUI), touchscreen, processor, memory, I/O, serial communications, RTOS and operating system functions.

In this chapter you'll learn:

- ⇨ *All about the operating system and software features of the QVGA Controller;*
- ⇨ *How to connect to your controller; and,*
- ⇨ *How to configure various options on your controller.*

Introducing the QVGA Controller

To serve the needs of real-time control, modern embedded computers must have a set of complementary features including operating system software, device drivers, user interface, and I/O. You'll find the QVGA Controller has a set of hardware and software that work together to simplify your product development cycle while offering new capability to your products. The following subsections discuss the interdependent hardware and software aspects of your controller.

Real-Time Operating System and Built-In Function Library

You wouldn't want to have to load an operating system into your desktop computer each time you turn it on, and the same holds true for embedded computers. Importantly, all of Mosaic's controllers incorporate a full-time, on-board operating system called *QED-Forth*. QED-Forth is an interactive programmable macro language encompassing a real-time operating system (RTOS), object oriented graphical user interface (GUI) toolkit, debugging environment, an assembler and math library for use within the Forth programming language, and a comprehensive set of pre-coded device drivers.

These built-in functions make it easy for you to get the most out of your board's computational and I/O capabilities. You can fully program the QVGA Controller using only the QED-Forth programming language, or you can program it using only the C language – all of the operating system's functions are accessible using either language.

This manual describes how to program your QVGA Controller using the QED-Forth programming language, and how to use the built-in functions. Another manual if available if you

prefer to program in the Control-C programming language. Function glossaries provide an in-depth description of every routine. The QVGA Controller's extensive embedded firmware reduces your time time-to-market – we've precoded hundreds of useful routines so you won't have to.

The RTOS in onboard Flash memory also manages all required hardware initializations and automatically initializes and starts your application code. It provides warning of power failures so you can implement an orderly shutdown, and provides the run-time security feature of a watchdog (COP - computer operating properly) timer.

Programming is a snap using the interactive debugger and multitasking executive. The multitasker allows conceptually different functions of your application run independently in different tasks while all accomplishing their duties in a timely fashion.

Choice of Programming Languages

You can program the QVGA Controller using either the ANSI-standard C language or Mosaic's QED-Forth language. In either language, you can supplement your high-level code with assembly code. Using either language, you have full access to all firmware functions.

The Control-C Programming Environment

Our Control-C™ cross-compiler was written by Fabius Software Systems and customized by Mosaic Industries to facilitate programming the QVGA Controller in C. It is a full ANSI C compiler and macro pre-processor; it supports floating point math, structures and unions, and allows you to program in familiar C syntax. Extensive pre-coded library functions provide easy command of the controller's digital I/O, A/D, D/A, keypad, display, high current drivers, serial ports, memory manager, multitasker, and much more.

Using the Windows™ environment on your PC, you can edit your C program in the supplied TextPad™ editor, and with a single mouse click you automatically compile, assemble and link your program, and generate an ASCII hex file ready for downloading. Clicking in the "Terminal" window and sending the download file to the controller completes the process: you can then type **`main`** from your terminal to execute your program. The interactive development environment also lets you examine and modify variables and array elements, and call individual functions in your program one at a time with arguments of your own choosing. This interactive environment greatly speeds the debugging process!

QED-Forth High Level Language

For those who prefer to program in FORTH, no external compiler is needed. You interact with the QED-Forth operating system (an RTOS, interpreter and compiler, all rolled into one) using your PC as a terminal. When programming in Forth you can use the TextPad text editor supplied as part of the Mosaic IDE (or you can use any other editor you prefer) to write your code and download the source code directly to the controller where it is compiled as it downloads. As we will see, even C programmers benefit greatly by the presence of the QED-Forth operating system, as the built-in Forth language provides a quick and easy way to interactively "talk to" your QED Board while debugging your C programs.

68HC11 Assembly Code

Both Control-C and QED-Forth include complete in-line assemblers that let you freely mix high level and assembly code. This is sometimes useful when creating specialized time-critical functions such as interrupt handlers.

Graphical User Interface and GUI Toolkit

The QVGA Controller features a touchscreen controlled graphical user interface. Combining a high-contrast 5.7" diagonal ¼VGA display and high resolution analog touchscreen, it comes complete with object-oriented menuing software that makes it easy to control your application using buttons, menus, graphs, and bitmapped images.

You can display your own custom graphics on your choice of display: a bright, amber-on-black high-contrast electroluminescent (EL) display, visible from any angle; or a bright white-on-blue cold-cathode fluorescent (CCFL) backlit LCD. Display screens and graphic objects are quickly developed with most Windows paint programs, such as PC Paintbrush, allowing you to create sophisticated displays including your company logo, system diagrams, and icon-based control panels. Real-time data plotting routines are precoded for you, so the user can see what your instrument is doing. Your application's startup screen executes automatically on power-up.

You can use as many screens as you need, each with software configurable buttons and menus. A precoded menu manager simplifies menu-driven control, making it easy to define buttons, menus, icons, and their associated actions. With the touch or release of a button, the menu manager responds, sending an appropriate command to your application program. Onboard software draws the screen graphics and responds to button presses for you, so you can focus on your application rather than display maintenance.

Memory and Mass Memory

384K Flash and 256K RAM are standard on the QVGA Controller. Of the QVGA Controller's 384K of Flash memory, 224K is available for your application program and data storage. Of the 256K of RAM, 253K is available for application program use. Up to 128K of the RAM can be optionally battery backed.

Custom controllers with 768K Flash and 640K SRAM are also available. For those really extensive applications that require lots of memory or removable data storage, the QVGA Controller hosts compact flash cards from 64 Megabytes and up in size.

Like PROM, Flash memory is nonvolatile. Thus it retains its contents even when power is removed, and provides an excellent location for storing program code. Simple write cycles to the device do not modify the memory contents, so the program code is fairly safe even if the processor "gets lost". But Flash memory is also re-programmable, and the Flash programming functions are present right in the QVGA Controller's onboard software library.

Communications

Two serial ports and a fast synchronous serial peripheral interface (SPI) provide plenty of communications capability. A hardware UART drives RS232 or RS485 protocols at up to 19.2 Kbaud, and a software UART provides RS232 at up to 4800 baud. Two serial ports allow you to program through one while your instrument can still communicate with a third party through the other. If you need greater speed or more ports, UART Wildcards plug directly into the QVGA's module bus, each providing two more full-duplex RS232/485 buffered serial communication ports at up to 56Kbaud.

Measurement and Control

You can control dozens of analog and digital I/O lines in real time. In addition to the serial communications channels the QVGA Controller commands:

- 8 channels single-ended or 4 channels differential 12-bit A/D at up to 30kHz sampling rate with unipolar (0 to +5V) or bipolar (-5 to +5V) input;
- 8 channels 8-bit analog inputs (0 to +5V) at up to 100kHz sampling rate;
- 8 channels of cascadable multiplying 8-bit D/A lines with 0 to +3V outputs;
- 26 digital I/O lines including 3 or 4 input capture functions, 4 or 5 output compare functions and pulse accumulator; and,
- 4 open-drain high-current outputs with onboard snubbers to drive 150 mA continuously or 1 amp intermittently.

Pre-coded I/O drivers make it easy to do data acquisition, pulse width modulation, motor control, frequency measurement, data analysis, analog control, PID control, and communications.

Need even more I/O? The QVGA Controller hosts Mosaic's WildcardsTM, small I/O modules for sophisticated and dedicated I/O. Stack up to eight Wildcards for: 16- or 24-bit resolution programmable gain A/D; 12-bit D/A; compact flash mass memory; AC or DC solid state relays; configurable digital I/O; additional RS232, RS422 or RS485; or high-voltage, high-current DC inputs and outputs.

Getting to Know Your Hardware

Your QVGA Controller, diagrammed in Figure 1-1, comprises a touchscreen controlled graphical display and two double sided surface mount boards that integrate the many hardware and software functions in a compact package.

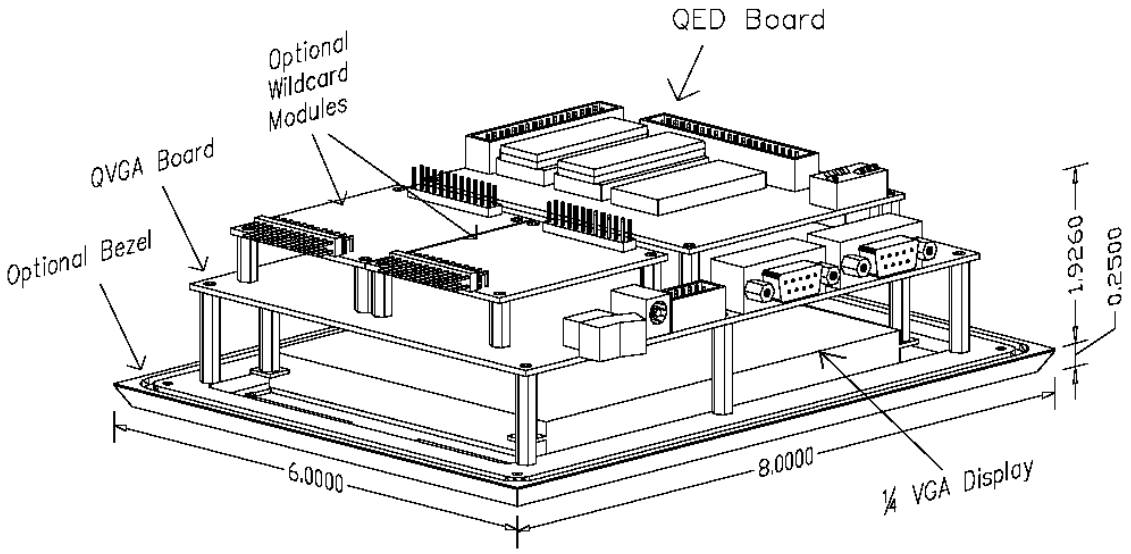

Figure 1-1 The QVGA Controller, display facing down.

Looking at Figure 1-1 , which shows the QVGA Controller oriented with the display facing down, from top to bottom you can see the QVGA Controller has several layers:

- A QED Board (Part No. QED-4-QVGA) sits on top. This double sided, surface mount board is the heart of the QVGA Controller, and uses Motorola's 16 MHz 68HC11F11 microcontroller, reconfigured for an 8 MB address space. The 68HC11 supports 21 interrupts and includes several I/O subsystems including timer-controlled counters, input-capture and output-compare lines, a fast SPI and an 8-channel, 8-bit A/D. The QED Board hosts 128K SRAM (optionally battery-backed) and 256K Flash, and enhances the I/O by providing four high current digital drivers, two serial ports, 12-bit A/D and 8-bit D/A converters.
- The QVGA Board hosts hardware controllers for graphical displays and a touchscreen, contains high efficiency switching regulators to provide regulated, filtered power to other components, and contains additional memory (128K expandable to 512K Flash, and 128K expandable to 512K SRAM).
- The QVGA Board also provides module ports for accommodating up to 8 WildCard I/O expansion modules for just about any kind of I/O you might need. Two modules are shown on the diagram, but you can have any combination of:
  - 16-bit or 24-bit resolution A/D;
  - 12-bit D/A;
  - Isolated AC or DC solid state relays;
  - Compact Flash Cards, 64 Mbyte and up;
  - Logic level, high voltage, and high current digital I/O; or,
  - Additional RS232, RS422 or RS485.

- The ¼ VGA display may be either a bright, amber-on-black, high contrast, all-angle view TFEL electro-luminescent display or a high contrast CCFL white-on-blue monochrome LCD display with software controlled backlight and contrast. For either option the display is 5.7" on a diagonal (3.5" x 4.6 ") and shows 320 x 240 pixels.
- A high resolution transparent analog touchscreen is mounted on the front surface of the display. A software controlled beeper on the QVGA Board provides audible feedback for finger touches.
- An optional bezel simplifies mounting the QVGA Controller on instrument front panels.

QVGA Developer Package

If you purchased a QVGA Controller Developer Package, you should have received the following:

1. A QVGA Controller (Part No. QVGA-Mono or QVGA-EL);
2. A 9-pin PC Serial Cable (Part No. PCC9-232);
3. A 16-24 volt Power Supply (Part No. PS-QVGA);
4. A CD-ROM containing:
   - Mosaic IDE including a TextPad source code editor and the QED-Term terminal program;
   - Program examples; and,
   - Motorola M68HC11 Reference Manual and MC68HC11F1 Technical Data Manual (Part No. MAN-HC11);
   - A users guide, "The Forth Programmer's Guide to The QVGA Controller", and associated glossaries and appendixes.

If you are missing any of these items, please contact us immediately.

A Tour of Connectors and Switches

Figure 1-2 shows a photo of the back of the QVGA Controller, and Figure 1-3 diagrams the positions of important connectors.

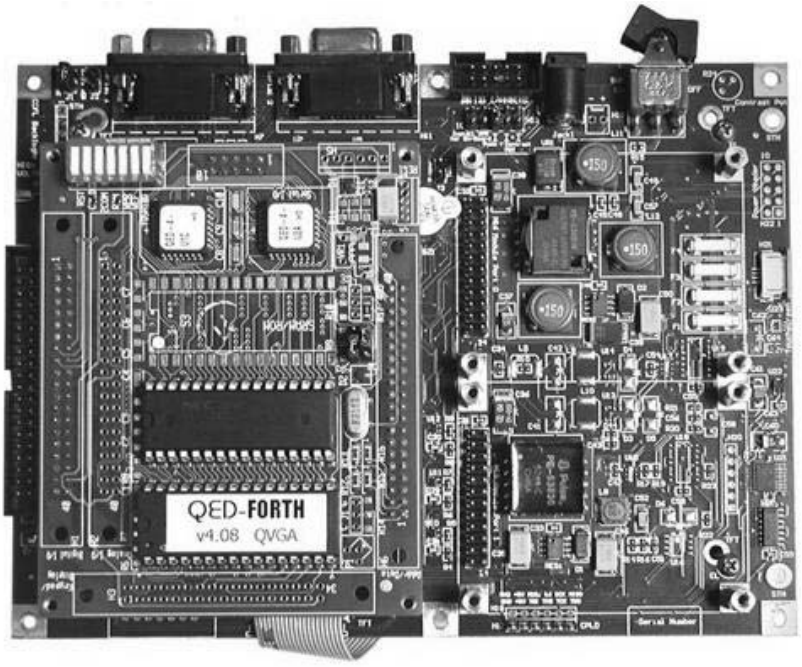

Figure 1-2 The QVGA Controller showing the QED Board mounted on the QVGA Board.

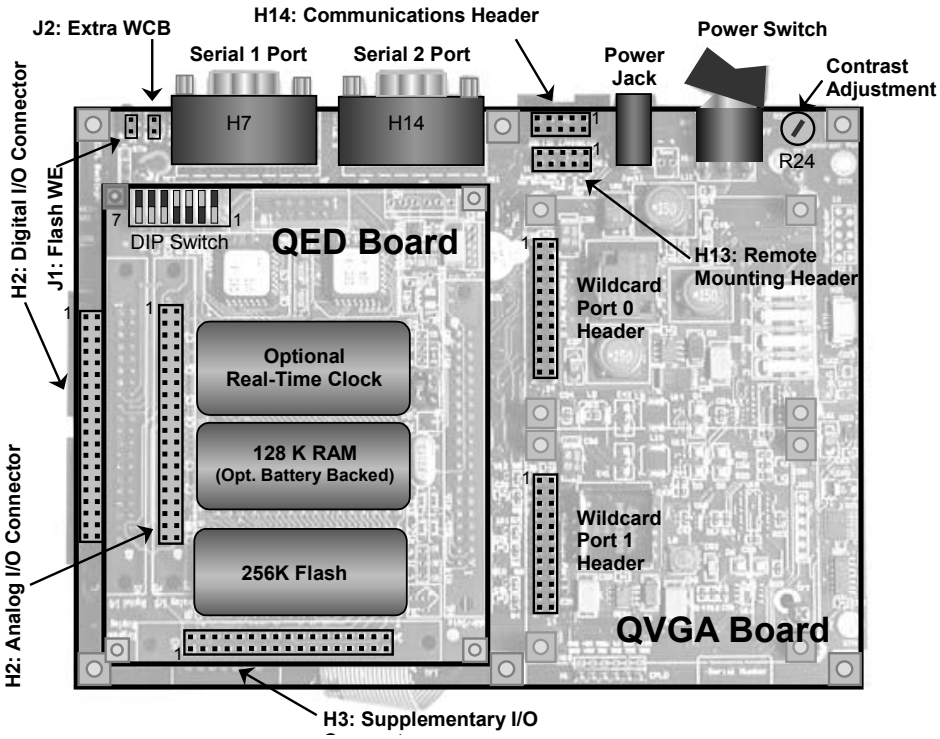

Figure 1-3 Connectors and Switches on the QVGA Controller. In addition to the memory shown on the QED Board, the QVGA Board contains 128 (expandable to 512K) Flash and 128 (expandable to 512K) RAM.

Power Jack and Power Switch

The power jack is located between the communications header and the power switch. The QVGA Controller can be powered by any power supply that can deliver 8 to 24 volts at 5 watts for the QVGA-Mono or 16 to 24 volts at 20 watts for the QVGA-EL. A switch, located between the power jack and the contrast potentiometer, controls power to the QVGA Controller. The switch is in the "off" position when it is depressed away from the power jack as shown in Figure 1-2 and Figure 1-3.

QED Flash Board Memory Sockets

The board contains 256K Flash in socket S1 and 128K of optionally battery-backed RAM in S2.

QED Flash Board Real-Time Calendar Clock Socket

An optional battery-backed Real-Time Clock (RTC, also known as the "smartwatch ") can be plugged into socket S3. The RTC is itself a socket made by Dallas Semiconductor that contains a clock chip, crystal and battery.

Primary and Secondary Serial Ports, Communications Header

The primary serial port can be configured for RS-232 communication up to 19,200 baud. Its 9-pin connector is located between the Flash Write Enable Jumper and the secondary serial port. The secondary serial port, which can be used for peripheral devices such as a printer or a barcode reader, has a maximum baud rate of 4,800. Its 9-pin connector is located between the primary serial port and the Communications Header. To enable the secondary serial port, turn dip switch 5 on the QED Board ON. Otherwise, dip switch 5 should be OFF so that it frees bit 3 of PORTA for use as general-purpose I/O.

Analog I/O Connector

Header H2 on the QED Board provides 24 analog I/O lines, including access to 8 channels of 8-bit A/D, 8 channels of 12-bit A/D, 8 channels of 8-bit D/A and their associated reference voltages.

Digital I/O Connector

Header H2 on the QVGA Board provides 26 uncommitted digital I/O lines.

Supplemental I/O Connector

This connector provides access to four MOSFET high current drivers named HC0-HC3 and their onboard snubber diodes, V+ raw power, and ground. The connector is located on the QED Board and also contains 5 digital outputs and 4 digital inputs.

Piezoelectric Buzzer

A piezoelectric buzzer mounted on the QVGA Board allows audio feedback for software events such as button presses or alarms.

Contrast Pot

The contrast potentiometer (pot) adjusts the contrast of the monochrome LCD display and is located next to the power switch. The contrast of the display is properly set before each unit is shipped. If further adjustment is required, simply turn the pot with a small screwdriver. If the contrast pot needs to be mounted in a different location (on an external panel, for example) you can move it using the Remote Mounting Header as described in the section on Table 1-1 .

Wildcard I/O Expansion Interfaces

Wildcard Port Headers 0 and 1 each accommodate up to four stacked Wildcards I/O expansion modules.

Configuring QVGA Options

Setting the QED DIP Switches

The QED Board has a 7-position DIP switch located prominently on the top of the board. The purpose and default positions of each switch are as follows:

#	Legend	Default State	Effect of Switch in the ON State
1	TOSHIBA	OFF	If the QED board is used as a stand-alone controller this switch configures the display interface for direct connection to a Toshiba graphics display. For proper operation of the QVGA Controller it should always be set to the OFF position.
2	WEF	ON	This switch write-enables the entire Flash memory device in the S1 socket. This switch is useful when using the flash memory for code storage when there is no need to modify its contents during normal operation. When OFF, code is protected against modification, and prevents “flash lockout” crashes. When enabled, the flash device locks out all memory accesses for 10 milliseconds after an “errant” write to the flash – a write without the required “unlocking sequence”. If an application program inadvertently executes a write cycle while the flash is being addressed, the controller will “crash” for at least 10 msec because the program code cannot be read by the processor. This would result in a silent crash that does not return a “QED-Forth V4.x” prompt. Setting Switch 2 OFF prevents this consequence of inadvertent writes. Of course, all application programs should be thoroughly debugged so that there are no “inadvertent writes” to the flash memory. If storing code in Flash, turn this switch ON during development, and OFF once your code is debugged.
3	WPR	ON	Must always be ON.
4	ROM	ON	Switches 3 and 4 together enable Flash in memory socket 1 and RAM in socket 2. Both switches should be ON.
5	2COM	OFF	Enables serial port 2 communications hardware for use as a second RS232 port. (The OFF state configures bit 3 of PORTA for use as general-purpose I/O.) Turn ON if you require use of the second RS232 port, OFF if you need bit 3 of PORTA.
6	COLD	OFF	Puts the QVGA Controller into “special cleanup mode” on the next power-up or reset. To return the QVGA Controller to its pristine, right-from-the-factory condition, turn this switch ON, toggle switch 7, then turn this switch back OFF.

#	Legend	Default State	Effect of Switch in the ON State
7	RST	OFF	Resets the QED Board; return switch to OFF position to resume normal operation.

Setting the QED Board Jumpers

A three-post jumper located between the third socket and H6 configures the primary serial port for RS-232 or RS-485 operation. If a jumper shunt is install across the two pins closest to the crystal (the default configuration), the QVGA Controller will communicate using RS-232 over serial 1. If a jumper shunt is install across the two pins closest to the J3 silk screen, the QVGA Controller will communicate using RS-485 over serial 1.

Setting the QVGA Board Jumpers

Jumpers J1, J2, and J3 on the QVGA Board

The actions of the configurable jumpers on the QVGA board are summarized in Table 1-1 .

Table 1-1 QVGA Controller Jumper Summary.

Jumper	Label	Description
J1	Flash WE	An installed jumper write enables the flash memory on the QVGA Board. This jumper is installed by default.
J2	Extra WCB	An installed jumper allows you to mount a Wildcard Carrier Board onto the QED Board. This jumper maps the modules on Module Port 0 and 1 of the QVGA Board to pages 8 to 15, rather than addressing them at pages 0 to 7. This jumper is not installed by default.
J3	Buzzer Test	If installed a jumper actuates the piezoelectric buzzer. The header pins for this jumper are not installed by default.

The Remote Mounting Header

Header H13, whose location is shown in Figure 1-3 , allows you to remotely mount the power switch, buzzer, and contrast potentiometer. This allows you to move these controls from the back of the controller to a front panel for easy access and increased audibility. The default positions of the jumpers are as follows:

For a QVGA Controller with a monochrome display, jumper shunts are installed across pin pairs 1-2 and 3-4 to enable the on-board contrast potentiometer, and 5-6 to enable the piezoelectric buzzer. For a QVGA Controller with an EL display, a jumper shunt is still installed across pins 5-6 to enable the on-board buzzer, but does not have a contrast pot so pins 1-4 on the Remote Mounting Header are not connected and jumper shunts across these pins do nothing.

Remotely Mounting the Buzzer

You can mount a buzzer to your front panel for increased audibility. To remotely mount a buzzer, remove the jumper shunt across pins 5 and 6 and connect your buzzer across +5V on pin 7 and the /Beeper_ON signal on pin 5.

Remotely Mounting the Contrast Adjustment Potentiometer

If you have a monochrome display, its contrast will vary depending on the ambient temperature. Although we allow you to adjust the contrast in software, if the display is completely washed out, the contrast may have to be adjusted using the contrast adjustment potentiometer. To remotely mount a contrast potentiometer, first remove the jumper shunts across pins 1 and 2 and 3 and 4. Then, connect one side of your potentiometer to Ext_Pot_A on pin 1, the other side to Ext_Pot_B on pin 3, and the wiper to Ext_Pot_A on pin 1.

Remotely Mounting the Power Switch

You can also mount the power switch to your front panel. To remotely mount a power switch, use a toggle switch and mount one side to PWR.in on pin 9 and the other side to Switched.PWR on pin 10.

Chapter 2

Chapter 2: Your First Program

This Chapter will get you started using the Forth language to program your QVGA Controller. It will guide you through the installation of the Mosaic IDE, an integrated editor and terminal, and you'll start up and talk with your controller. You'll also:

- ⇨ *Compile and download your first program using an example multitasking program that performs calculations using floating point math, stores data in arrays in the QVGA Controller's extended memory space, and prints the results to the terminal;*
- ⇨ *Selectively execute any program function using the QVGA Controller's on-board debugging environment (QED-Forth);*
- ⇨ *Use the QVGA Controller's built-in operating system to access extended memory;*
- ⇨ *Use the terminal to interact with a running multitasking application; and,*
- ⇨ *See how easy it is to set up a multitasking application.*

Installing the Mosaic IDE

The Mosaic IDE, which includes a full-featured editor and communications terminal, is provided on an installation CD-ROM. To install it onto your PC, first insert the Installation CD-ROM into your CD Drive. If the installer does not launch automatically, browse to your computer's CD drive using the 'My Computer' icon and double click on 'Setup.exe' to manually launch the installer.

We recommend that you use the default installation directory ("C:\Mosaic\") and choose 'Typical Setup' when asked. If you wish to install into a different directory, you may type in any pathname provided that it does not contain any spaces. The 'Custom' setup option can be used if another version of either TextPad, QED-Term, or an earlier version of the Mosaic IDE has already been installed. However, the Mosaic IDE requires all of its components to work properly. Please call us at (510) 790-8222 if you have any questions.

When the installation is complete, you will need to restart your computer unless you are installing onto a Windows 2000 machine. Be sure to choose 'Yes' when asked to restart – if you don't, the installation may not complete properly. If you choose 'No' and restart later we recommend that to assure a full restart you fully shutdown your computer and restart it; lesser restarts don't always restart fully.

You are now ready to talk with your QVGA Controller!

Turning on Your QVGA Controller

Familiarize yourself with the locations of the power and serial connectors as shown in Chapter 1. After finding them follow these steps to turn on your system and establish communications with it:

1. Examine the settings of the 7 onboard switches. Dip switches 1, 5, 6, and 7 should be turned OFF by depressing the side of the switch towards the edge of the board. Dip switches 2, 3, and 4 should be turned ON. The function of each Dip switch is described in Chapter 1.

2. Connect the female end of the 9-Pin serial communications cable to your computer terminal's serial communications port (typically "COM2") and the male end to the primary serial port on the QVGA Controller.

3. Power up your PC computer or terminal.

4. We recommend that you use QED-Term, the terminal program that comes with the Mosaic IDE. You can start the terminal by double clicking the TextPad editor (the primary application of the Mosaic IDE) and choosing the terminal icon which looks like this:

   

   You can also start the terminal by double clicking the application "QED-Term.exe".

   The terminal starts using a configuration file called QED-Term.ini; this configures communications with the QED Board using COM2 at 9600 baud, no parity, and 1 stop bit. Xon/Xoff flow control is enabled, and the file transfer options are set so that the terminal waits for a line-feed (^J) character before sending each line (this is very important). You can use any other terminal program, but it must be configured with these same settings.

5. You should check the configuration of your Windows communications drivers:

   1. On your PC go to the device manager dialog box by double clicking "System" in the "Control Panel", clicking the "Hardware" tab, and clicking the "Device Manager" button.

   2. In the list of devices open up the list of "Ports" and double click on "Communications Port (COM1)". (If COM1 is tied up with another service, such as a fax/modem, you may want to use COM2 instead.)

   3. You'll now have a dialog box called "Communications Port (COM1) Properties". Click the general tab and make sure you have these settings:

Property	Value
Baud Rate	9600
Data Bits	8
Parity	None
Stop Bits	1
Flow Control	Xon/Xoff

6. Plug the QVGA Controller's power supply into a 110 VAC outlet. European users will need a power transformer for changing European 220 VAC to 110 VAC. Insert the power supply

output plug into the power jack on the QVGA Controller and turn the power switch ON by depressing the side of the switch towards the outer edge of the QVGA Board. After a short pause two things should happen:

1. A demonstration program should appear on the screen; and,
2. If you hit the enter key while the cursor is in your terminal window you should see the QVGA Controller respond with,

   ok

The demo program, and the serial communications response indicate that your QVGA Controller is now working!

The working software demo that is included in each QVGA Controller automatically starts when power is first applied and the power switch is turned ON. The demo shows three screens, accessible by touching *file tabs* at the top of the screen. The first screen shows a typical numeric keypad entry screen, the second provides examples of font sizes, and the third shows random numbers being plotted into a plot window. Touching the file tabs transfers focus from one screen to another.

While the demo is running as one task, another task on the controller responds to commands from the terminal, and responds with the 'ok' prompt to an enter key.

You can play with the demo to see how it works. Observing this demo program run is a good way to verify that all of the hardware is functioning properly and also demonstrates some basic capabilities of the menu control software.

If Something Doesn't Work

If your demo is not running, check that power is being properly applied to the controller. The demo should always run on a new QVGA Controller when power is initially applied. If you see only a blank screen and no message appears on your terminal there's something wrong, so:

1. Verify that power is being properly applied to the controller.
2. Verify that the serial cable is properly connected.
3. Check the terminal configurations of the QED-Term (using the menu item "Settings → Comm"), and recheck the communications properties of the Windows communications port.
4. Email or call us if you are still having trouble.

Removing and ReInstalling the Demo Program

Disabling the Demo Program

When you tire of the demo you can disable it, but don't do that too quickly because once the demo is disabled you can't restart it without sending special commands to the controller. So, feel free to play with the demo for awhile, to see how it uses buttons and menus.

Eventually, you will want to disable the demo. You can disable it by doing a *special cleanup* that places the QVGA Controller in a pristine state with no application program running.

> ***Doing a "Special Cleanup"***
>
> If you ever need to return your QVGA Controller to its factory-new condition, just do a *Special Cleanup*: Turn ON dip switch 6, toggle dip switch 7 ON and then back OFF again, and finally turn OFF dip switch 6. This procedure will remove any application programs and reinitialize all operating system parameters to their factory-new condition.

After you do disable the demo your display should be blank. You can verify that serial communications are working properly by hitting the enter key while your insertion point is in the terminal window,

```
←
```

Whenever you hit the 'Enter' key, as represented by the ← symbol in the line above, you should see an 'ok' prompt.

Now, whenever the QVGA Controller is powered up with your terminal communicating with it, this message,

```
QED-Forth V4.xx
ok
```

where "**V4.xx**" represents the current software version number of your QVGA Controller, should appear, and the 'ok' prompt should repeat each time you press the carriage return. This message indicates that the onboard operating system is ready to receive commands.

Although the demo now no longer automatically starts when the QVGA Controller is turned on, it still resides in Flash memory. The special cleanup has merely erased an *autostart vector* – a special location that tells the controller the starting address of the program to run when it starts up. You can still start the demo program by typing the following into your terminal window,

```
RESTORE ←
main ←
```

in which the ← symbol represents hitting the 'Enter' key on your keyboard. The demo program should now be running. The first command you typed, **RESTORE**, restored pointers to the operating system's list of names of programs. The pointers had been saved in EEPROM by a **SAVE** command. The second line is the name of the program. When you type the name of a program the operating system responds by executing that program.

The GUI demo (**main**) program will continue operating until the board is reset or the power turned off. If you'd like to restore the demo so that it automatically starts whenever the controller is turned on, simply type the following two lines:

```
RESTORE ←
CFA.FOR main PRIORITY.AUTOSTART ←
```

Note that you must type the lines with the spaces between the words just as in the lines above. The case of the characters isn't important.

These commands store the starting address of the **main** routine in the autostart vector. The two words, **CFA.FOR main**, find the *code field address* of the routine **main** and send it to the routine

PRIORITY.AUTOSTART which stores it in the autostart vector. Now, the demo should automatically start up whenever the controller is turned on.

If you download other programs to the QVGA Controller you will overwrite the demo program, which is stored in Flash memory. Also, any execution of the **SAVE** command will overwrite the saved pointers. In either case, to revive the demo you will need to reload it. That is done by using QED-Term to reload the compiled demo file to the controller's Flash memory. For instructions for doing that please see the comments in the folder, "C:\Mosaic\GUI Toolkit\Demo".

Using the Mosaic IDE

Using the Editor and Compiler

The Mosaic IDE has two main components, the TextPad editor, which includes the Control-C compiler, and the QED-Term serial terminal program, both of which you'll find in the default directory "C:\Mosaic\":

- *TextPad* is a fully featured and highly configurable text and program editor. You'll use it to write and compile your code. All of the functions of the C compiler tools are available through the controls in TextPad. You can launch TextPad from the 'Mosaic IDE' group in the 'Programs' section of your Windows 'Start' menu. For convenience, you may want to place a shortcut to it on your desktop or on your Windows Taskbar.
- *QED-Term* is a serial communications terminal that allows you to interactively control your controller over its RS-232 interface. You'll also use it to download your Forth source code for compilation into the memory of the QVGA Controller. *QED-Term* may be launched from the 'Mosaic IDE' group within 'Start→Programs', but it is also available from within TextPad, either from the 'Tools' dropdown menu or by clicking the terminal icon on TextPad's toolbar.

The TextPad Tool Bar 

Along with the standard tools you expect in a text editor you'll find four custom tools available in the toolbar that you'll use to compile and download your programs. Each of these tools is also accessible in the 'Tools' dropdown menu of TextPad. For C programmers the Funnel icon calls the C compiler and assembler only, the Hammer performs a standard build of your program, and the Bricks performs a multi-page memory model build of your program. Forth programmers won't need to use these tools. Both C and Forth programmers will find the Terminal icon useful; it launches the *QED-Term* program. Each of these tools is described in more detail below.

The 'Compile Tool' Finds Syntax Errors in C Programs

The Compile Tool, designated by the "Funnel" icon, invokes the Control-C compiler – Forth programmers do not need to use it because Forth programs are compiled as they are downloaded onto the QVGA Controller.

The 'Make Tool' Compiles and Makes a Downloadable Single-Page C Program

The Make Tool, designated by the "Hammer" icon is also for C programmers. It controls how C programs are linked and stored in memory. Forth programmers do not need to use it.

The 'Multi-Page Make Tool' Compiles and Makes a Downloadable Long C Program

Forth programmers will also not need the Multi-Page Make Tool, designated by the "Bricks" icon, which assists in compiling long C programs.

'QED-Term' Communicates with Your Product

The Terminal icon launches the communications program, QED-Term. When you launch QED-Term for the first time, check the communications settings (Settings→Comm) to verify that the serial port is set correctly for your computer. Even if your QVGA Controller is running a demo program it will still respond to the terminal if the communications settings are correct.

An Introduction to Programming in QED-Forth

The QVGA Controller accepts serial commands from a terminal which is typically a personal computer (PC). QED-Forth's onboard interpreter, compiler, and assembler compiles your program into executable code in the QVGA Controller's memory, and the code can be immediately executed by simply stating the name of a routine. This encourages a productive iterative programming style.

If you wanted, you could program a QVGA Controller application by typing in your application program one line at a time from a terminal. For all but the shortest programs, this would be a tedious process. Fortunately, there is a better way. All you need is Mosaic IDE. The IDE's text editor allows you to create and modify text files and save them on the computer's disk drive. Its terminal program allows you to send and receive characters via the PC's RS232 serial port to communicate with the QVGA Controller. Thus you can rapidly edit your source code into files which are downloaded to the QVGA Controller. The Mosaic IDE allows you to switch between the editor and terminal windows quickly and easily.

How To Send Your Programs to the QED Board

To send a program to the QED Board, edit the "source code" (that is, the QED-Forth commands and any relevant comment statements) into a text file and save the file on the PC's disk. Then use the "file transfer" or "text transfer" feature of the terminal program to send the program file to the QED Board. If you discover errors while debugging the program, simply edit the file and re-transmit all or part of it to the QED Board. If your file contains the **`ANEW`** command (as described in a later Chapter of this manual), the old erroneous version of your program will be automatically forgotten by the QED board when the new version is transmitted to the board.

You have complete flexibility in programming the QED Board. You can send programs to the board using the file transfer method, and you can also use the terminal interactively to compile and execute short commands or manage debugging operations. QED-Term also allows you to record

your debugging session as a text file. This can be a useful technique; you can later edit the recorded file to save the most worthwhile aspects of the debugging session.

You download and develop your programs in RAM. Once they are sufficiently debugged you can then transfer them to Flash memory for permanent storage. QED-Forth programs are generally not relocatable – they must execute from the same addresses they were compiled to. But if they are transferred from RAM to Flash how is this possible? The solution is to swap the RAM and Flash addresses. Programs are downloaded and compiled into RAM using a *download memory map*, transferred to Flash, then the memory map is changed to the *run-time (or standard) memory map* placing the Flash memory at the same addresses that the RAM had been located. The program then executes from the Flash just as it had from the RAM.

Let's go through an example of the steps required to compile and execute a QED-Forth program on the QVGA Controller. We'll use the RAM on memory pages 1-3 and the Flash on pages 4-6 of the standard memory map.

1. First establish the *download memory map* by executing:

   ```
   DOWNLOAD.MAP↵
   ```

   This command can be typed interactively from the terminal program, or can be the first statement in your download file. This swaps the RAM and Flash, placing the RAM on pages 4-6 and the Flash on pages 1-3.

2. Establish memory locations for code and names on pages 4, 5, and 6. One valid way to do this is by executing,

   ```
   4 USE.PAGE
   ```

   which is equivalent to the following set of commands:

   ```
   HEX
   0000 04 DP X!   \ code starts at address 0 on page 4; 20 Kbytes available
   5000 04 NP X!   \ names start at address 5000 on page 4; 12 Kbytes available
   8E00 00 VP X!   \ variables start at address 8E00 in common RAM
   4A00 0F 7FFF 0F IS.HEAP   \ 13.5 Kbyte heap at addresses 4A00-7FFF on page F
   ```

   Typically, the USE.PAGE command or other equivalent commands are the first commands in the source code file. To customize the locations of these memory areas, you can include edited versions of these commands in your source code file. Make sure these commands come *before* the first **ANEW** statement in your code. For example, to locate up to 32 Kbytes of code on page 4, and up to 32 Kbytes of names on page 5, you can execute:

   ```
   HEX
   04 USE.PAGE       \ set default use.page locations
   0000 05 NP X!     \ move names section to page 5, leaving all of page 4 for code.
   ```

   or, equivalently,

   ```
   HEX
   0000 04 DP X!           \ code starts at address 0 on page 4; 32 Kbytes available
   0000 05 NP X!           \ names start at address 0000 on page 5; 32 Kbytes available
   8E00 00 VP X!           \ variables start at address 8E00 in common RAM
   4A00 0F 7FFF 0F IS.HEAP \ 13.5 Kbyte heap at addresses 4A00-7FFF on page F
   ```

3. Send your code to the QED Board. Typically, the first statement of the code is an **ANEW** command which serves as a "forget marker" to simplify re-downloading of code. For example:

```
ANEW  MY.CODE           \ choose any name you want here

: HELLO.WORLD  ( -- )   \ define a new routine called HELLO.WORLD
   CR ."  Hi Everyone!" \ print a message to the terminal
;          \ end the definition

\  < all of your source code goes here >
```

4. After successfully compiling your code, transfer the code and names to flash using the **PAGE.TO.FLASH** routine. For example, if all of your code and names are located on page 4, simply execute:

   ```
   04 PAGE.TO.FLASH   \ transfers page 4 RAM to page 1 in flash
   ```

   If your code is on page 4 and your names are on page 5, you would execute

   ```
   04   PAGE.TO.FLASH \ transfers page 4 RAM to page 1 in flash
   05   PAGE.TO.FLASH \ transfers page 5 RAM to page 2 in flash
   ```

   The routine **PAGE.TO.FLASH** will print an error message if you pass it an illegal page, or if the flash cannot be programmed (for example, if the flash is write protected because DIP switch 2 is OFF). The page parameter passed to the routine must be the source page in RAM where the code has been compiled. **PAGE.TO.FLASH** transfers all 32 Kbytes of the source page to the *parallel* page in flash memory. At this point two copies of your code and names exist in memory: one copy is in RAM, and the other resides in flash. But only the RAM version can be executed properly, because it resides at the same pages at which it was compiled. The next step makes the flash-resident copy of the code executable.

5. Re-establish the standard memory map by executing

   ```
   STANDARD.MAP ←
   ```

   from your terminal. This remaps pages 4, 5, and 6 so that they are addressed in the Flash device, and maps pages 1, 2, and 3 to be addressable in the RAM. At this point, the Flash-resident code (typically on page 4, which is now in Flash) can be executed. The RAM in pages 1, 2, and 3 are available; they can be written over or used by the application program.

6. Run the program by executing any of the names that have been defined. You can also execute an autostart command to cause a specified function to be automatically called upon each restart. To place the autostart vector in EEPROM inside the processor, execute the command:

   ```
   CFA.FOR  <name>   AUTOSTART
   ```

   where <name> is the name of the designated function. To place the autostart vector in Flash memory, execute the command:

   ```
   CFA.FOR  <name>   PRIORITY.AUTOSTART
   ```

   For most applications, the PRIORITY.AUTOSTART option is preferable because it locates the autostart vector with the code in the flash memory device.

7. (OPTIONAL) To download an additional source code file after the prior 6 steps have been executed, you need to transfer the code back to RAM, establish the download map, download the next source code file, re-program the flash, and re-establish the standard map. Assuming you left off after step 6, you would execute the following:

```
NO.AUTOSTART   \ if you installed an autostart, undo it

4 PAGE.TO.RAM  \ move code from page 4 in flash to page 1 in RAM
5 PAGE.TO.RAM  \ (only needed if page 5 used); move page 5 flash to page 2 RAM
DOWNLOAD.MAP  \ get ready for download

<send your source code file here>

4 PAGE.TO.FLASH   \ copy updated page 4 code to flash
5 PAGE.TO.FLASH   \ copy updated page 5 code to flash
STANDARD.MAP      \ code is now executable at pages 4 and 5 in Flash

CFA.FOR <name> PRIORITY.AUTOSTART \ optional
```

FORTH EXAMPLE SESSION

Here's a transcript of an entire session to show how easy it is to compile a Forth program into flash. The program, **HELLO.WORLD**, prints a simple message to the terminal:

```
DOWNLOAD.MAP
4 USE.PAGE

ANEW  MY.CODE

: HELLO.WORLD  ( -- )  CR ." Hi Everyone!" ;

4 PAGE.TO.FLASH
STANDARD.MAP

HELLO.WORLD\ executes the program!
```

You've compiled your program, downloaded it into RAM, transferred it to flash, and executed it. If you want to set up an autostart routine, follow the procedure as explained above, or consult the appropriate chapter of this manual.

Initializing the Memory Map

Before downloading a new program you'll want to make sure that you're starting with a clean slate from a known initialized condition. To do so, type the command,

```
COLD↵
```

followed by a carriage return. QED-Forth executes the command as soon as it receives the carriage return. The **COLD** command tells QED-Forth to perform a cold restart, initializing the system to a known state and causing it to **FORGET** all user-added words. QED-Forth responds with its cold startup message:

```
Coldstart
QED-Forth V4.xx
```

which tells you the version number of the software on your board.

QED-Forth doesn't care whether you use capital letters, small letters, or a combination of both when you type a command or function name. It automatically performs case conversion so that **COLD**, **Cold**, and even **CoLd** are treated as the same command.

To set up a memory map that gives plenty of room for programming type the command

```
4  USE.PAGE↵
```

```
ANEW  MY.ROUTINES↵
```

Be sure to type at least one space between the words as shown above (for example, between the 4 and **USE.PAGE**). The **USE.PAGE** command configures the memory map so that the definitions of new words reside on page 4 of the memory. The **ANEW** command followed by a name of your choice creates a marker which facilitates re-loading your code during program development. It is good policy to execute an **ANEW** command after setting the memory map pointers.

If a bug or error causes the **COLDSTART** message to be printed to your terminal while you are programming, you should re-initialize the memory map with the **USE.PAGE** command.

Output Commands

First, let's have QED-Forth print a statement. At your terminal, type

```
."  Hi There!"↵
```

and QED-Forth responds with

```
Hi There!  ok
```

We have underlined QED-Forth's response for clarity. The command **."** (pronounced dot-quote) tells QED-Forth to print the characters until the next " (quote) character is encountered. The space after the **."** is important because FORTH uses spaces to separate commands. To instruct QED-Forth to print the message on a new line, execute the FORTH command **CR** (carriage return) before the print command, as

```
CR  ."  Hi There!"↵
```

and notice how QED-Forth inserts an extra carriage return/linefeed before printing the message at your terminal.

If you make a mistake while typing a command, just type "backspace" or "delete" to erase as many characters as necessary on the current line. Once you've typed a carriage return, though, QED-Forth executes your command. You can't edit a command that was entered on a previous line. If you type an incorrect command and then type a carriage return, you may receive the "?" error message which means that QED-Forth does not understand the command you typed. If this happens, you can usually just re-type the command line and continue programming. Error messages are discussed in more detail in the QED Software Manual.

The Stack

In FORTH, parameters are passed among routines on a push-down data stack (referred to as "the data stack" or "the parameter stack" or simply "the stack"), and nesting of subroutine calls is handled by a separate "return stack". Most microprocessors are designed to handle stacks, so this scheme results in high efficiency. In addition, stack-based parameter passing is a single organizing principle that unifies and simplifies FORTH syntax.

All arithmetic, logical, I/O, and decision operations remove any required arguments from the stack and leave the results on the stack. This leads to "postfix" notation: the operation is stated after the data or operands are placed on the stack.

To see how this works, put some numbers on the data stack by typing

```
5  7←
```

and QED-Forth responds

```
ok  ( 2 ) \ 5 \ 7
```

QED-Forth is showing a picture of its data stack. The **(2)** means that there are two items on the stack. Each of the items is listed, and items are separated by a \ character, which can be read as "under". So we could describe the stack right now as 5 under 7; the 7 is on top of the stack, and the 5 is under it. If there are more than 5 items on the stack, the stack print displays the number of stack items and the values of the top 5 items.

The stack print that shows what's on the stack is a feature of the debugging environment. To disable the stack print, you could execute **DEBUG OFF**. It is not recommended that you do this, though; it's very helpful to keep track of the items on the data stack while developing your program.

To multiply the numbers that are now on the stack, type the multiply operator which is a ***** character:

```
*←
```

and QED-Forth responds

```
ok  ( 1 ) \ 35
```

The operator removes the two operands 5 and 7 from the stack, multiplies them, and puts the result of the multiplication on the stack. To subtract 5 from the number on the stack, type

```
5 - ←
```

which produces the response

```
ok  ( 1 ) \ 30
```

The - (minus) operator takes the 35 and the 5 from the stack, subtracts, and puts the result on the data stack.

To print the result to the terminal, we could simply type the printing word **.** (dot). A slightly fancier way to do it is:

```
CR  ." The result is "  .
```

which prints the response

```
The result is 30  ok
```

The command **CR** causes the output to be printed on a new line, then the specified message is typed, and then the printing word . removes the 30 from the stack and prints it. The stack is now empty, so QED-Forth does not print a stack picture after the ok.

Notice that throughout this exercise QED-Forth has been interpreting and executing commands immediately. This is because FORTH is "interactive". The results of executing commands can be immediately determined. If they are incorrect, the command can be changed to correct the problem. This leads to a rapid iterative debugging process that speeds program development.

Defining New Words in QED-Forth

Let's define a routine (called a "word" in FORTH) that increments the contents of a variable. To create a variable named **COUNTER** execute

```
VARIABLE  COUNTER←
```

To initialize **COUNTER** to zero, type

```
0  COUNTER  !←
```

This command first puts a 0 on the stack, then stating **COUNTER** leaves the address of **COUNTER** on the stack, and ! (pronounced "store") stores the 0 into the **COUNTER**. To verify the contents of **COUNTER**, type

```
COUNTER  @  . ←
```

and QED-Forth responds with

```
0  ok
```

Here, **COUNTER** leaves its address on the stack, and the **@** (pronounced "fetch") operator removes the address from the stack, fetches the contents of the address, and leaves the contents on the data stack. The printing word **.** (dot) then prints the contents.

To define a word to add 1 to the contents of **COUNTER**, type

```
: INC.COUNTER   ( -- )←
   1   COUNTER  +!  ←
  ;←
```

You have just compiled a definition of a new QED-Forth word. The **:** (colon) tells QED-Forth to append the next name it encounters (in this case, **INC.COUNTER**) to the name area of the dictionary. The comment between parentheses is a "stack picture"; in this case the empty stack picture reminds us that **INC.COUNTER** does not expect any inputs and does not return any outputs on the stack. The remaining commands until the terminating **;** (semicolon) form the body of the definition. The definition puts an increment (1) on the stack, puts the address of **COUNTER** on the stack, and calls **+!** (plus-store) which removes the increment and address, and adds the increment to the current contents of the address, thus increasing the value of **COUNTER** by 1. After receiving the terminating **;** QED-Forth responds with the "ok" prompt indicating that it has compiled (i.e., entered into the dictionary) the definition; it has not executed the definition yet. To verify this, we could type **COUNTER @** . just as we did above. But let's type the equivalent expression

```
COUNTER  ?←
```

and note that QED-Forth responds with

```
0  ok
```

which tells us that the counter is still initialized to 0. Note that the command **?** has been pre-defined in the QED-Forth kernel as:

```
: ?
   @  .
   ;
```

So you see that useful words are built up as combinations of lower level words in FORTH. Useful sequences of commands (such as **@ .**) are defined as higher level words with descriptive names, thus building a lexicon of convenient and powerful words. QED-Forth already has many useful functions; you'll define more to address the unique requirements of your application.

Now we can execute **INC.COUNTER** and again check the contents of **COUNTER** to see how it is affected. Type

```
INC.COUNTER   COUNTER  ?↵
```

and QED-Forth responds

```
1  ok
```

INC.COUNTER increments the value of the variable by 1. This command can be repeated to show that it always increments the variable. We'll use **INC.COUNTER** in the next section to demonstrate the multitasking capability of QED-Forth.

A Definition Using Floating Point Math and Local Variables

Let's define a more interesting word that uses floating point math to sum the inverses of the integers from 1 to 100. The text after the \ character on each line of the definition is commentary that is ignored by QED-Forth.

```
: SUM.OF.INVERSES ( -- )
   \ prints the sum of inverses of integers from 1 to 100
   ZERO  LOCALS{  f&accumulator  }  \ define and initialize local variable
      101  1                        \ put loop limit and index on stack
      DO                           \ start loop
         I  FLOT  1/F              \ calculate next inverse
         f&accumulator  F+         \ add it to accumulator
         TO  f&accumulator         \ update value in accumulator
      LOOP
      CR  ."  The sum of the inverses is "   \ announce answer
      f&accumulator  F.            \ print answer
   ;
```

After entering this definition, we can execute

```
SUM.OF.INVERSES↵
```

and QED-Forth responds

```
The sum of the inverses is  5.185  ok
```

The first line of the word defines a "local variable" named **f&accumulator** and initializes it to 0.0. Because the name of the local begins with "**f&**", QED-Forth recognizes that it is a floating point (as opposed to an integer) quantity. The next line sets up a loop that starts at 1 and stops after 100 is reached. The command I puts the current loop index on the stack, **FLOT** (pronounced float) converts it to a floating point number, **1/F** inverts it, and then **F+** adds it to the accumulator (floating point math operators typically start with the letter **F**). The **TO** instruction updates the contents of the accumulator. After the loop is finished, the print command announces the result, and **F.** (f-dot) prints the final value of the accumulator. Notice how easy it is to program with floating point mathematics, and how the use of the named local variable eliminates stack juggling and makes the definition easy to read.

Multitasking

Let's set up a task whose job it is to continually increment the variable COUNTER while we simultaneously run the QED-Forth interpreter. Type in (or download a text file containing) the following set of commands; their meaning and syntax will be explained in detail in the "Multitasking" chapter in the QED Software Manual.

```
: COUNT.FOREVER   ( -- )
   BEGIN
      INC.COUNTER  PAUSE
   AGAIN
   ;
HEX
9600  0  TASK:  MY.TASK
0\0  0\0  0\0  MY.TASK BUILD.STANDARD.TASK
CFA.FOR COUNT.FOREVER  MY.TASK  ACTIVATE
DECIMAL
```

You are now running two concurrent tasks: one task is the QED-Forth interpreter that you've been using all along to execute your commands, and the second is a task that increments the 16-bit variable named **COUNTER**. To verify this, type **COUNTER ?** and you'll see some random number that changes each time you type the command as the background task continually updates the value.

Let's look at what we've done. First we defined an infinite loop in the word **COUNT.FOREVER** that increments the counter and executes **PAUSE**. **PAUSE** is the task-switching word. The remaining command lines define and build a new task called **MY.TASK** and activate it so that it performs the function **COUNT.FOREVER**. The QED-Forth interpreter is still running, and it executes the **PAUSE** task-switch word while it has spare time; namely, when it is waiting for serial input from the terminal. Each time QED-Forth executes **PAUSE**, **MY.TASK** increments the value of **COUNTER** and itself executes **PAUSE** to return control to the QED-Forth interpreter. Control passes back and forth between the tasks so quickly that it seems as if the two tasks are executing simultaneously. We could add more tasks to the round robin task list, and we could set up a timeslice clock to force more frequent task switching. Those topics are described in the "Multitasking" chapter in the QED Software Manual.

Working with Matrices

QED-Forth makes it easy to work with matrices, which are defined as 2-dimensional arrays of floating point numbers. To create two matrices named **MY.MATRIX** and **DESTINATION**, type

```
MATRIX:  MY.MATRIX↵
MATRIX:  DESTINATION↵
```

To refer to a matrix as a whole, we use the **'** (tick) operator in FORTH; it is the apostrophe character on your keyboard. For example, to dimension the matrix to have 2 rows and 3 columns, type

```
2  3  '  MY.MATRIX  DIMMED↵
```

making sure to separate all of the words by spaces. To load some data into the matrix, type

```
MATRIX  MY.MATRIX  =  1.  2.  3.  4.  5.  6.↵
```

(The decimal points in the numbers flag the numbers as floating point quantities.) Now we can display the contents of the matrix using the matrix display word M.. by typing

```
'  MY.MATRIX  M..↵
```

and QED-Forth responds by printing the matrix as

```
MY.MATRIX = ←
1.   2.   3.   ←
4.   5.   6.    ok←
```

Let's transpose **MY.MATRIX** so that each row becomes a column and each column becomes a row. To place the transpose of **MY.MATRIX** into the matrix **DESTINATION** and then display the result, execute

```
'  MY.MATRIX  '  DESTINATION  TRANSPOSED
'  DESTINATION   M..←
```

and QED-Forth responds by printing the transposed matrix as

```
DESTINATION =
1. 4.
2. 5.
3. 6.    ok
```

There are a host of other pre-programmed matrix transformations; many of them are discussed in the QED Software Manual. For now, let's use matrices to solve a common and important problem.

Solving Simultaneous Equations

Many applications require the solution of simultaneous equations with several unknown quantities. This is easy to accomplish with QED-Forth. Suppose that we need to solve the following equations:

$$1.07\,x1 + 0.19\,x2 + 0.23\,x3 = 2.98$$
$$0.38\,x1 + 1.00\,x2 + 0.74\,x3 = 3.48$$
$$0.12\,x1 + 0.53\,x2 + 1.20\,x3 = 3.70$$

We can express them as the matrix equation:

$$MX = R$$

where M is the coefficient matrix, X is the matrix of unknowns, and R is the matrix of right-hand-side constants (called the residue matrix in linear algebra). That is,

$$M = \begin{bmatrix} 1.07 & 0.19 & 0.23 \\ 0.38 & 1.00 & 0.74 \\ 0.12 & 0.53 & 1.20 \end{bmatrix} \quad X = \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} \quad R = \begin{bmatrix} 2.98 \\ 3.48 \\ 3.70 \end{bmatrix}$$

To solve this matrix equation (that is, to compute the values of X) using QED-Forth, type the following commands. The text after the \ character on each line is commentary that explains what is happening; you need not type the comments.

```
MATRIX: M          \ create coefficient matrix
3 3 ' M DIMMED     \ dimension coefficient matrix
MATRIX M =  1.07 0.19 0.23 0.38 1.00 0.74 0.12 0.53 1.20 \ initialize coefficients
MATRIX: R          \ create residue matrix
3 1 ' R DIMMED     \ dimension residue matrix
```

```
MATRIX R = 2.98 3.48 3.70   \ initialize residue
MATRIX: X          \ create matrix of unknowns
' M  ' R  ' X  SOLVE.EQUATIONS \ solve the equations
' X  M..          \ print the results↵
```

QED-Forth responds,

```
Matrix X =
2.099
0.8259
2.509
ok
```

With these few lines of code, you've solved the set of linear equations with floating point coefficients. Larger sets of equations with many more unknowns are handled with very similar code.

In Case Your Software Crashes

This section demonstrates why you need not worry about "crashes" caused by programming errors. QED-Forth gives you complete control over the board's hardware and software. With all this power, programming "bugs" can sometimes cause the processor to get lost so that it fails to respond to commands or forgets some of the functions that you have defined. These situations are called "crashes". The QED Board is designed so that you can always recover from software-induced crashes. In other words, as long as you have not damaged the electronic hardware on the board, you can restore the QED-Board to a known functional state.

Some software crashes can cause the processor to execute a series of write instructions to various locations in memory. If the code definitions that you are developing reside in unprotected RAM, they may be corrupted by a crash. You can protect your code from accidental corruption using the memory write-protection features described above and some QED-Forth utilities. These development techniques can save a substantial amount of development time by eliminating the need to reload your code after crashes.

For this demonstration we will initialize the QED software and then add a word to the dictionary to represent your application software. After write-protecting and saving the state of the dictionary, we will purposefully crash the board, and then restore the dictionary to its prior state. This demonstrates that you can recover from a crash without having to reload all of your application program.

We begin by initializing the board and setting up an appropriate memory map by typing:

```
COLD
4  USE.PAGE
ANEW CRASH.DEMO ↵
```

The **COLD** command initializes QED-Forth, and **4 USE.PAGE** places the dictionary areas (where new definitions will be compiled) on page 4. The **ANEW** command followed by a name of your choice marks the point in the dictionary where we begin adding new code. It is good policy to execute an **ANEW** command after setting the memory map pointers.

We now add a new function to the dictionary by typing this definition at the terminal:

```
: TEST   CR  .”  Hello world...” ; ←
```

When executed, **TEST** prints a greeting to the terminal. We'll use this definition to represent any custom code you might send to your QED Board while developing your program.

Once the code has been loaded into the QED Board, we execute the command

```
SAVE←
```

by typing it at your terminal. This saves a set of relevant information describing the present state of your dictionary. The information is stored in EEPROM (electrically erasable PROM) inside the 68HC11 chip.

Now that our code has been protected, we can verify that our TEST word works by typing

```
TEST ←
```

at the terminal. It should work. Now let's make a mistake by executing the following expression:

```
FORGET  TEST ←
```

Oops! We just lost the code we worked so hard on (Try to execute **TEST** and see what happens). Fortunately, we **SAVEd** the state of our machine after defining **TEST**. Executing

```
RESTORE ←
```

restores the prior state of the dictionary, and **TEST** is available again (type **TEST** to see for yourself).

That was easy. But what if a more serious bug occurs, for example one that erases the user area where all of QED-Forth's key pointers are kept? To see what happens, execute the following command:

```
HEX 8400 0 100 ERASE ←
```

To recover from this crash you can simply reset the processor by turning the reset switch (DIP switch 7) ON and then OFF again. In this case the processor realizes that the problem that caused the crash was serious, and executes a **COLD** restart, so access to added definitions such as **TEST** will be lost. (Most resets cause a **WARM** restart that automatically preserves the dictionary, as explained in the “Program Development Techniques” chapter of the QED Software Manual.) Once again, you can recover the dictionary including the **TEST** definition by simply executing **RESTORE**.

The Special Cleanup Mode

During software debugging it is possible (though difficult) to crash your QED Board so that even a reset will not re-establish communications. The following example demonstrates this by diabolically revectoring the terminal input and output routines so that no serial communications can occur. Type this command *all on one line* at your terminal:

```
HEX  0\0  UKEY  X!    0\0  UEMIT  X! ←
```

After executing this expression, you will not be able to communicate with your system, even by resetting it. The solution is to enter “Special Cleanup Mode” which resets all critical operating system variables to their original factory-defined settings. This is accomplished by turning DIP switch 6 ON, and then resetting the QED Board by toggling switch 7 or by turning the computer OFF and then ON again.

Resetting or re-powering your QED Board with switch 6 ON causes a COLD reset, initializes all system variables and EEPROM locations to their factory settings, and enters the QED-Forth interpreter. Once a special cleanup has been performed, be sure to turn switch 6 OFF again and execute another reset to return to the normal operating mode. If you are using a graphics display you'll need to re-establish the stored display configuration using the **IS.DISPLAY** routine. The Special Cleanup Mode also sets the default baud rate of the primary serial port to 9600 baud; if you have established a different baud rate as described in the next chapter, you'll need to execute **BAUD1.AT.STARTUP** to re-establish your chosen baud rate.

Now that you can communicate with your QED Board again, execute **RESTORE** to restore access to your dictionary which contains the **TEST** routine.

Saving Your Program in Flash

Any time you execute the **RESTORE** command, the memory map is restored to the same state it was in when SAVE was executed.

Once you have write-protected and saved your memory map, you may wish to define more words in a fresh area of RAM. To do this, simply initialize the pointers that specify where the "dictionary" and "names" entries of new definitions are placed in memory. The following code accomplishs this by moving the dictionary pointer and the names pointer to un-write-protected RAM on page 6:

```
HEX
0000  6  DP  X!          \ set the dictionary pointer
5000  6  NP  X!          \ set the names pointer
ANEW  MORE.ROUTINES
```

You can now add more custom definitions, and they will be compiled in the specified regions on page 6. The "Program Development Techniques" Chapter discusses these issues in detail.

Part 2

Programming the QVGA Controller

Chapter 3

Chapter 3: Making Effective Use of Memory

The QVGA Controller's Memory Map

The QVGA Controller uses a paged memory system to expand the processor's 64Kbyte address space to 8 Megabytes of addressable memory. The top half (32 Kbytes) of the address space (at addresses 0x8000 to 0xFFFF) addresses a common memory page that is always visible (i.e., accessible using standard 16-bit addresses) to any code running, no matter where it resides in the memory space. The bottom half (32 Kbytes) of the address space (at addresses 0x0000 to 0x7FFF) is duplicated many times and addressed through the processor's 16-bit address bus augmented by an 8-bit page address. Together the address and page are held in a 32-bit data type, an *xaddress*.

A subroutine on any page can fetch or store to any address on the same page or in the common memory, or transfer control to another routine there. It "sees" a 64K address space comprising its own page at addresses from 0x0000 to 0x7FFF and the common memory at addresses 0x8000 to 0xFFFF. To address memory on another page, or to call a routine on another page, special memory access routines are used to change the page.

The operating system is designed so that there is very little speed penalty associated with changing pages. The QED-Forth operating system automatically and transparently handles page changes.

Figure 3-1 illustrates the memory map of the QED Board. Briefly, the upper 32K of the 68HC11's address space, the *common memory*, is always accessible without a page change. In the lower 32K of the processor's address space, the operating system creates 256 pages of memory selected by an 8 bit on-chip port, with each page containing 32 Kbytes. The 32K of common memory at addresses 0x8000 to 0xFFFF (the upper half of the processor's memory space) is always accessible without a page change. Up to 256 pages (32K per page) occupy the paged memory at addresses 0x0000 to 0x7FFF. The first 9 pages of the QVGA Controller's 19 pages of installed memory are shown in the figure.

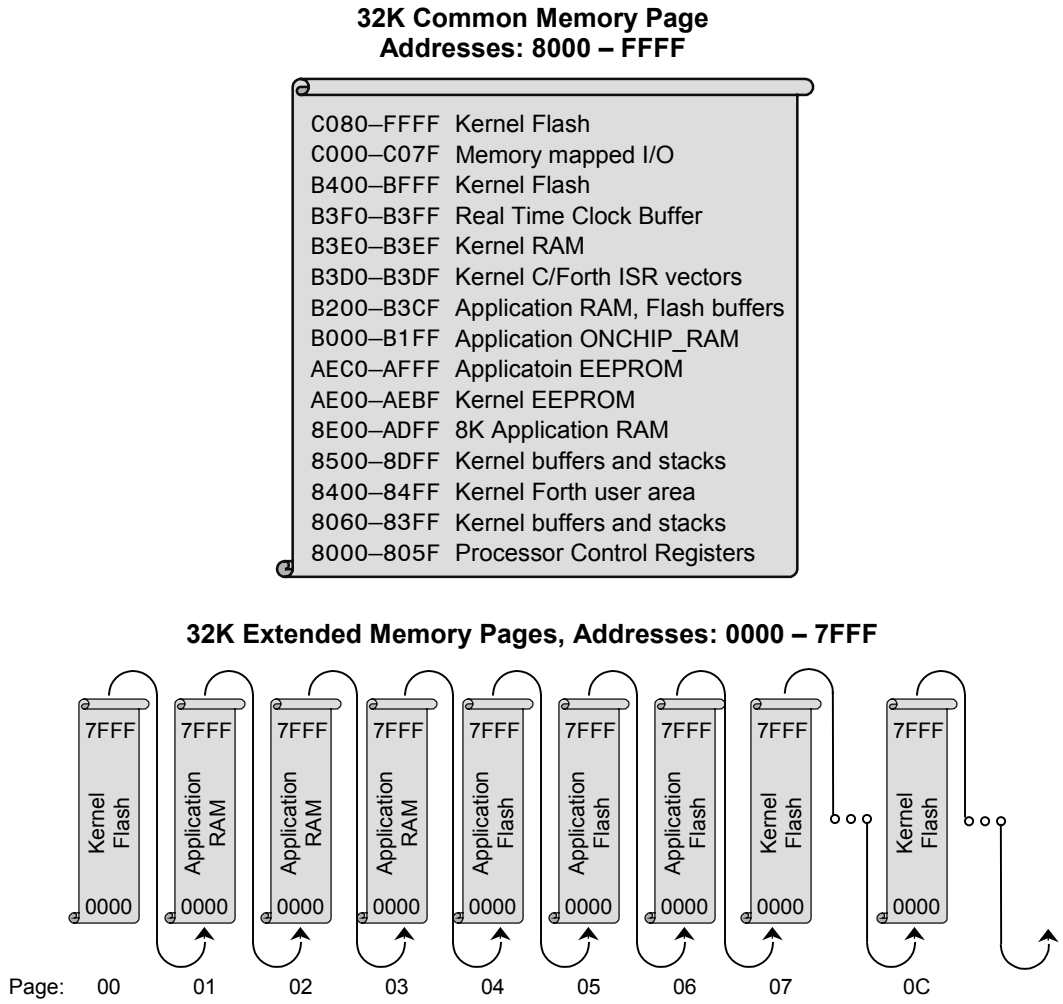

Figure 3-1 The paged memory space of the QVGA Controller.

Common Memory

The *common memory* is addressed at locations 0x8000 – 0xFFFF.

The processor's registers are located at 0x8000 – 0x805F, onboard hardware occupies addresses through 0x80FF, and the operating system reserves memory through location 0x8DFF for user areas, buffers, and stacks. For example, the default user area that runs the interactive Forth interpreter occupies 0x8400 – 0x84FF.

The 8 Kbytes at locations 0x8E00 – 0xADFF are available RAM for the user. The C compiler uses this area for static variables, arrays, task areas, etc. Similarly, the Forth memory map routine USE.PAGE locates the variable area starting at address 8E00 in common memory.

The processor's on-chip EEPROM (Electrically Erasable Programmable Read Only Memory) is located at 0xAE00 – 0xAFFF. Locations 0xAE00 – 0xAEBF are reserved by the operating system for use by the **SAVE** and **RESTORE** utilities, and for interrupt vectors. EEPROM at 0xAEC0 – 0xAFFF is available to your programs.

The 68HC11's 1 Kbyte of on-chip RAM is located at 0xB000 – 0xB3FF. Locations 0xB3F0 – 0xB3FF are reserved for the real-time clock buffers. Locations 0xB3D0-0xB3DF are reserved for support of Forth interrupt service routines called from C-compiled programs. Locations 0xB200 – 0xB3CF are reserved for the flash programming routines. Locations 0xB000 – 0xB1FF are always available to the programmer (this area is named **`ONCHIP_RAM`** in the C linker command file; C programmers can locate data in this area using a **`#pragma`** directive).

Locations 0xB400 – 0xBFFF and 0xC100 – 0xFFFF contain kernel code. A *notch* at 0xC000 – 0xC0FF is not decoded by any onboard devices, and provides a convenient place for the user to memory map I/O that must be accessed quickly (that is, without requiring a page change). Of course, an almost limitless amount of I/O can be mapped onto pages in the QVGA Controller's 8 Megabyte address space.

Paged Memory

Occupying this memory space are 384K Flash and 256K RAM. Of the QVGA Controller's 384K of Flash memory, 224K is available for your application program and data storage. The remainder is used by the QED Forth Kernel for its multitasking operating system, debugger, interactive Forth compiler, assembler, and hundreds of pre-coded device driver functions.

Of the 256K of RAM, 253K is available for application program use. Up to 128K of that can be optionally battery backed. An expanded memory option is available that expands the installed memory to 768K Flash and 640K SRAM.

Table 3-1 illustrates the partitioning of the onboard memory between the operating system (Kernel) and your application functions. Most of the Flash memory is available as two blocks of contiguously addressable memory on pages 4-6 and 70-73. RAM for your application program is also available in the paged memory in two contiguously addressable chunks, filling pages 1-3 and 60-63. There is also 20K available RAM on page 0F, and approximately 9K in the common memory. This 9K is particularly important because it is used to hold C variables and task space for each separate task your application program sets up.

Table 3-1 Partition of Flash and RAM among Kernel and Application Functions

Function	Size	Memory Page	Memory Address	Physical Location
Kernel Flash (160K)				
Kernel	64K	00, 0C	0000 – 7FFF	QED Board S1
Kernel	1K	0E	various	QED Board S1
Kernel	19K	common	B400 – FFFF	QED Board S1
Kernel	12K	0F	0000 – 2FFF	QED Board S1
GUI Toolkit	64K	07, 0D	0000 –7FFF	QED Board S1
Kernel RAM (4K)				
Kernel	3.5K	common	8000 – 8DFF	QED Board S2
Kernel	0.5K	0E	various	QED Board S2
RTC	48 bytes		B3D0– B3FF	68HC11
Kernel EEPROM (192 bytes)				
Kernel	192 bytes	common	AE00 – AEBF	QED Board S2

Function	Size	Memory Page	Memory Address	Physical Location
Application Flash (224K to 608K)				
Application code and graphic images	96K	04 – 06 (01– 03)	0000 – 7FFF	QED Board S1
Application code and graphic images	128K	70 – 73 (60 – 63)	0000 – 7FFF	QVGA Board
Application code and graphic images, available with extended memory option	384K	74 – 7F (64 – 6F)	0000 – 7FFF	QVGA Board
Application RAM (253K to 637K)				
C variables	512 bytes	common	B000 – B1FF	68HC11
C variables, available at runtime but used during download as a Flash write buffer	464 bytes	common	B200 – B3CF	68HC11
C Variables and task area, optionally battery-backed	8K	common	8E00 – ADFF	QED Board S2
Arrays and heap memory, optionally battery-backed	20K	0F	3000 – 7FFF	QED Board S2
Arrays and heap memory, optionally battery-backed	96K	01 – 03 (04 – 06)	0000 – 7FFF	QED Board S2
Arrays and heap memory	128K	60 – 63 (70 – 73)	0000 – 7FFF	QVGA Board
Arrays and heap memory, available with extended memory option	384K	64 – 6F (74 – 7F)	0000 – 7FFF	QVGA Board
Application EEPROM				
EEPROM Variables	320 bytes	common	AEC0 – AFFF	68HC11

Notes:

1. Pages not enclosed in parentheses indicate the *standard*, or run-time memory map; pages in parentheses indicate the addressing of the *download* memory map.
2. The 128K RAM in socket S2 on the QED-Flash Board can be optionally battery-backed.
3. Pages 00, 07, 0C, 0D, 0E and a portion of 0F are reserved to the operating system (the Kernel).
4. Application code is free to reside on pages 01-06, 60-6F, and 70-7F.
5. Addresses from 8000 through FFFF comprise common memory that is visible to code on all pages.

The common memory is also partitioned between the operating system and application program. Table 3-2 shows the addresses used by the operating system, and in boldfaced type those addresses available to the application program.

Table 3-2 Partition of the Common Memory

Address	Size (bytes)	Type	Function
C100 – FFFF	16128	Flash	Kernel – code
C000 – C0FF	256	I/O	Memory mapped I/O
B400 – BFFF	3072	Flash	Kernel – code
B3F0 – B3FF	16	RAM	Kernel – Real Time Clock Buffer
B3E0 – B3EF	16	RAM	Kernel
B3D0 – B3DF	16	RAM	Kernel – C/Forth ISR vectors
B200 – B3CF	**464**	**RAM**	**Application – C variables at runtime, Flash write buffer during program download**
B000 – B1FF	**512**	**ONCHIP_RAM**	**Application – C variables**
AEC0 – AFFF	**320**	**EEPROM**	**Application – nonvolatile storage**
AE00 – AEBF	192	EEPROM	Kernel

Address	Size (bytes)	Type	Function
8E00 – ADFF	**8192**	**RAM**	**Application – C Variables and multitasking task areas, optionally battery-backed**
8500 – 8DFF	2304	RAM	Kernel – buffers and stacks
8400 – 84FF	256	RAM	Kernel – Forth user area
8060 – 83FF	928	RAM	Kernel – buffers and stacks
8000 – 805F	96	RAM	Kernel – processor Control Registers

Shaded entries indicate memory available for application programs.

Addressing Flash

Flash memory is nonvolatile, like PROM. Thus it retains its contents even when power is removed, and provides an excellent location for storing program code. Simple write cycles to the device do not modify the memory contents, so the program code is fairly safe even if the processor "gets lost". But flash memory is also re-programmable, and the flash programming functions are present right in the QVGA Controller's onboard software library. These functions invoke a special memory access sequence to program the flash memory contents "on the fly". This allows you to modify your operating software (for example, to perform system upgrades). You can also store data in the flash device. You can program from 1 byte up to 65,535 bytes with a single function call using the precoded flash programming routine. Programming time is approximately 60 milliseconds per kilobyte.

Six special functions facilitate access to Flash memory. Their function names are:

```
DOWNLOAD.MAP   PAGE.TO.FLASH  PAGE.TO.RAM
STANDARD.MAP   TO.FLASH WHICH.MAP
```

Software Development Using Flash Memory

Automated commands contained in the download file will determine whether flash is present on the QED-FLASH Board, and will automatically establish the download map, load the code into RAM, transfer the code to flash, and re-establish the standard map. If you are curious, you may want to review the following instructions for users of older compilers to see what is going on "under the hood".

Access Routines for Arrays and Matrices

The routines **2ARRAY.STORE** and **2ARRAY.FETCH** have been added to the kernel to enable you to store to and fetch from 2-dimensional arrays or matrices. The unique aspect of these routines is that they properly handle data of different sizes (1 byte, 2 bytes, or 4 bytes) depending upon the way that the array or matrix is dimensioned. For example, if an array named **CHAR.ARRAY** is dimensioned (using the **DIMENSIONED** routine) to hold 1 byte per element, then executing

```
0  0  ' CHAR.ARRAY  2ARRAY.FETCH
```

will return the first 8-bit character in the array. If another array named **DATA.ARRAY** is dimensioned to hold 16-bit data, then executing

```
0 0 ‘ DATA.ARRAY 2ARRAY.FETCH
```

will return the first 2-byte element in the array. These routines help support access to paged memory when programming in Control-C.

Chapter 4

Chapter 4: Programming the Graphical User Interface

The Graphical User Interface (GUI) Toolkit is a suite of programming tools that gives you the ability to build an informative and interactive graphical user interface to monitor and control your instrument. This chapter

- ⇨ *Introduces the structure of the GUI Toolkit;*
- ⇨ *Takes you on a step-by-step guide to building your interactive application; and*
- ⇨ *Provides you with detailed descriptions of each component of the GUI Toolkit.*

The Structure Of The GUI Toolkit

A graphical user interface is created from building blocks such as bitmapped images and ASCII strings. These building blocks, or data, must be organized in an intuitive way so users can easily run your instrument. Object oriented concepts are the key to organizing this data and make it easy for you to design and implement your user interface. Object oriented programming allows you to organize data hierarchically based on objects and manipulate the data using methods. With the GUI Toolkit, it is simple to create elementary objects such as graphics that contain bitmapped image data and textboxes that contain ASCII string data. You can load those objects into other objects such as screens so that they can be shown on the display. You can create yet another type of object, a control, which can execute functions that acquire data from a user or actuate hardware when a user touches the touchscreen. A button is one kind of a control.

A Closer Look At Objects

The two concepts of object oriented programming essential to modern programs with user interfaces are “Objects” and “Events”. An object is an association of *properties* and *methods*, and may respond to *events*. An *event* is an external action that prompts an object into action. GUI Toolkit objects that can respond to events are called *controls*.

Properties

Properties describe an object’s physical attributes like its size, location, or image. Objects contain a data structure that holds its properties. Properties generally don’t do anything by themselves; they are more like read/write variables. However, properties may qualify the object’s behavior when the object does do something or has something done to it.

Some properties may be read only while others may be read/write. GUI Toolkit properties are always 32-bit numbers and are accessed or modified by calling the methods **`Get_Property`** or **`Set_Property`**. **`Set_Property`** requires a reference to an object, its property, and the new value, while **`Get_Property`** requires a reference to an object and its property as parameters and returns the property value.

Methods

Methods are actions the object can do or have done to it. Many methods, like **`Load`**, are overloaded; that is, they are applicable to several different types of objects. For example, you can *load* buttons, graphics, and textboxes into a screen. Overloaded methods allow you to use a single simple syntax with respect to many different objects.

Events

Events are external actions that an object responds to by executing a user defined function called an *event procedure*. Objects automatically recognize a predefined set of events, it is up to you to decide if and how they respond to those events. You specify the event procedure called when the event happens (or *fires*) by storing your function into the appropriate property using **`Set_Property`**. Event procedures are written, for example, to specify the program actions that occur in response to the press of a button.

The GUI Toolkit Objects

The following table lists the objects in the GUI Toolkit.

Object	Description
GUI_TOOLKIT	The central object that contains properties relevant to all other objects.
GUI_DISPLAY	The object tied to the physical display.
GUI_SCREEN0 to GUI_SCREEN5	An object that contains a collection of other functionally related objects such as graphics, textboxes, buttons, and plots.
GUI_TOUCHSCREEN	The object tied to the physical touchscreen.
GUI_PEN	A tool object that is used in conjunction with the Draw method to draw lines and geometrical figures onto screens.
GUI_BUZZER	The object tied to the physical buzzer.
GUI_FONT	The 8-pixel wide by 10-pixel tall Courier New font that is used to render strings in textboxes by default for the GUI Toolkit.
GRAPHIC	An object that contains a single image as a property.
FONT	An object that contains a single image of the 95 ASCII characters that is used to render strings in textboxes.
TEXTBOX	An object that uses a font object to render strings onto a screen.
BUTTON	An object with an associated graphic that responds to a user's touch on the touchscreen.
PLOT	An object used to render numerical data into graphical form onto a screen.

Some of these objects are created when you initialize the GUI Toolkit while others are created by functions that you write. All GUI objects reside in their own task with their own heap and some respond to events. In the next section, we'll show you how to create your user interface and then how to write your application using the GUI Toolkit.

Building Your Application

In this section, we'll explore the GUI Toolkit in the order in which you would typically build your application. We'll show you how to:

- Design and create your user interface on a PC;
- Transfer the images that you created for your user interface from the PC to the QVGA Controller;
- Write your application using the demo program as a reference guide;
- Handle errors; and,
- Expand the capabilities of the GUI Toolkit.

The demo program shows you how to build a multitasking application with three simple screens. The first screen is a numeric keypad that allows you to enter up to 20 digits and render them in a textbox. The second screen displays three different size Courier New fonts ranging from the default font of 8 pixels wide by 10 pixels tall to a large font that is 32 pixels wide by 40 pixels tall. The third screen plots an 8-bit A/D input every 10 seconds. As we describe each part of the GUI Toolkit, we'll use the numeric keypad screen of the demo to illustrate the important concepts.

Designing Your User Interface

Before you start coding your application, it's a good idea to sketch out how you want your user interface to look. Figure 4-1 shows a rough sketch of the first screen of the demo program. Sketching out your user interface before you start programming will help you to think about how you want your application to be organized and how you want to present information to your user.

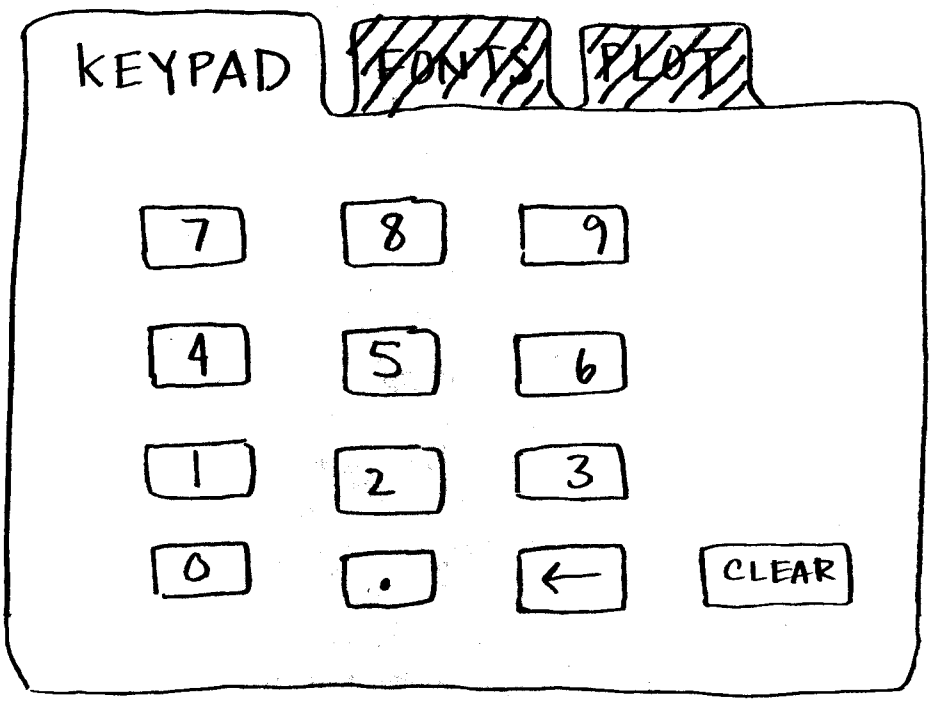

Figure 4-1 The sketch of the first screen of the demo program.

Drawing Your Screens

Once you've sketched out and designed your user interface, you need to create bitmapped images of your interface using an image-editing program. Images may be created using Microsoft Paint, which is included with all versions of Windows. You can also use other image-editing programs such as Corel Draw, Photoshop, and Paint Shop Pro. In this section and throughout this chapter, we provide step-by-step instructions for creating and manipulating images using Photoshop. However, all of the image-editing programs have similar tools and functionality. For example, the pencil tool in Photoshop is simply called the pencil in Paint and the line tool in Photoshop is called the line in Paint. Thus, the steps listed in the examples, although specific to Photoshop, apply to all image-editing programs.

To create an image of a screen from your sketch using Photoshop:

1. Create a new file with a width of 320 pixels and a height of 240 pixels, a resolution of 72 pixels per inch, and using the bitmap image mode. 320x240 is the size of the display and the size of a screen. The resolution is not important because it refers to the number of pixels per inch for the monitor. However, the image mode is important and must be set to bitmap if you are using a monochrome or EL display. This configures the image to have one bit per pixel.

2. Draw all lines using the pencil tool with a brush size of one pixel. The line tool, paint brush tool, and airbrush tool are also useful in drawing shapes and patterns.

3. Draw text using the type tool.

4. Save the image as a windows bitmap. A dialog box may appear informing you that some image data, such as printer settings, cannot be saved with this format. This does not effect the image so you can just click on OK.

The image that we created based on the sketch in Figure 4-1 is shown in Figure 4-2 .

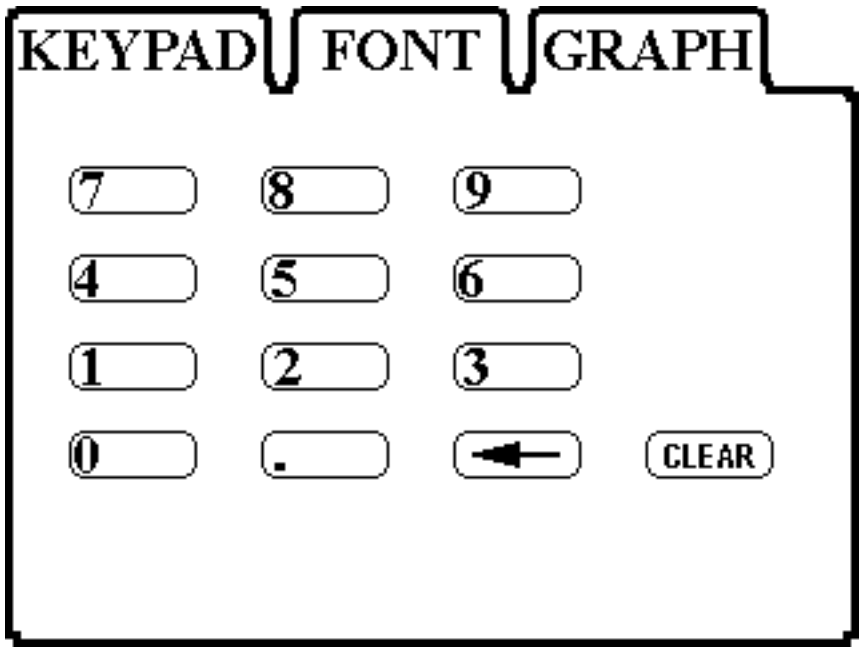

Figure 4-2 The bitmap image of the first screen of the demo program that was created using Adobe Photoshop.

Creating Your Images

Once you have created the screens for your user interface, you need to ungroup them. Ungrouping the screens into smaller images will allow you to save space in Flash memory since you don't want to store images with a lot of white space. It will also allow you to associate the images with individual buttons. To ungroup an image using Photoshop:

1. Select the marquee tool;
2. Left click on the upper left corner of an image you want to ungroup and drag the mouse pointer to the lower right corner of the image. This will create a box with a moving dashed line around the image;
3. Note the location of the upper left corner of the image; this location will be needed when you want to load a graphic containing the image onto a screen;
4. Under the Edit Menu, select the cut option to move the image to the clipboard. The image will disappear from the screen;
5. Under the File Menu, select the create new file option;

6. Name the file, based on the image. The height and width will automatically be set to the height and width of the image that is in the clipboard;
7. Under the Edit Menu; select paste; and,
8. Save the ungrouped image in the Windows bitmap format.

The width of an image must be a multiple of 8 pixels and the image must be positioned on an 8 pixel horizontal grid in a screen. Finer horizontal pixel placement and image widths would require bit shifting of each byte of the image every time it is drawn, significantly slowing performance of the GUI Toolkit. There are no limitations on the height of an image or the placement of an image in the vertical direction.

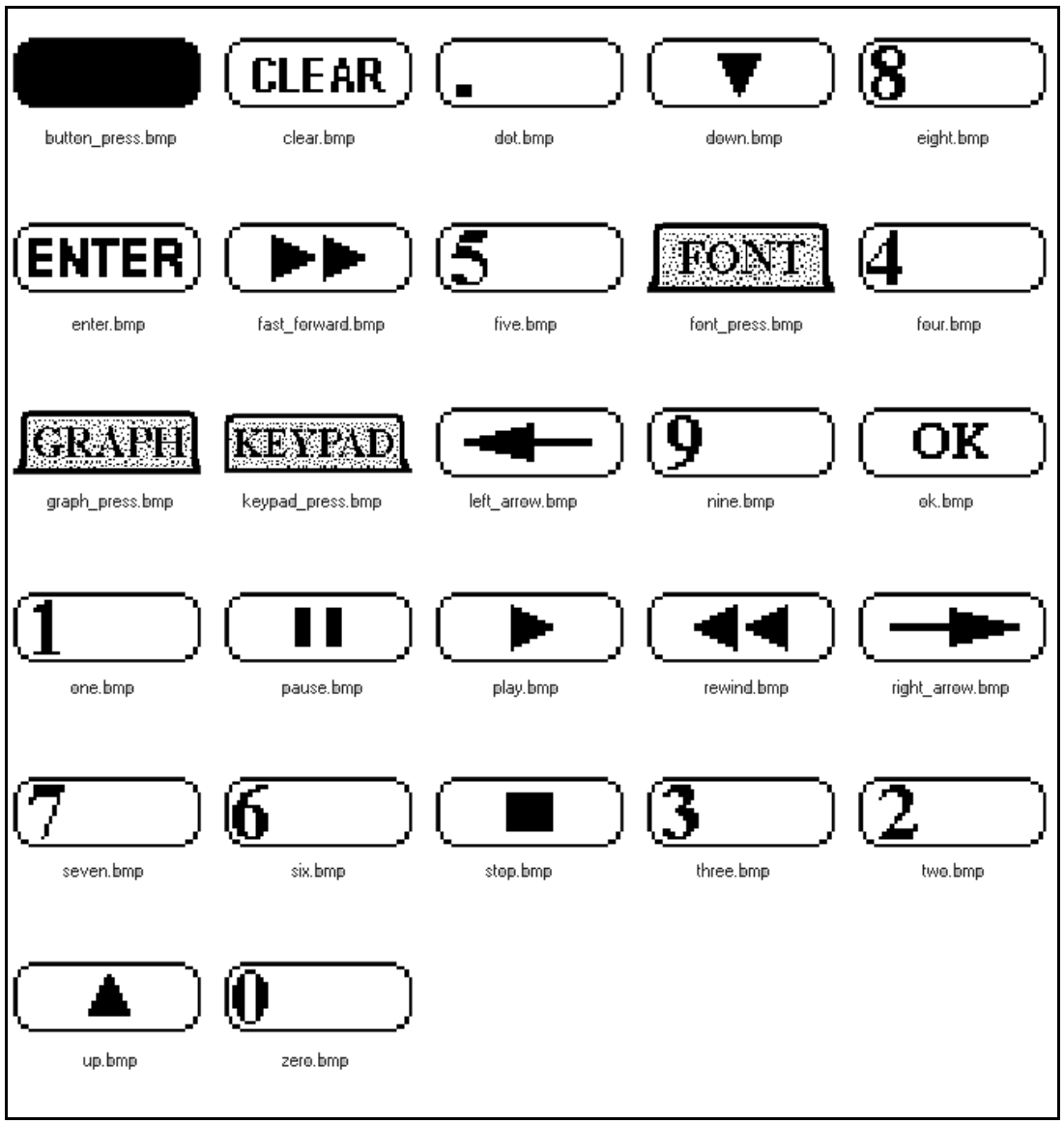

Figure 4-3 The images included with the demo program that you can use in your application.

The demo program also provides a set of images that you can use with your application. The images are shown in Figure 4-3 .

Transferring Your Images to the QVGA Controller

Once you have ungrouped all the images for your user interface, assemble them into a single directory. Images must be processed before you can use them with the GUI Toolkit; Mosaic's Image Conversion Program concatenates images into a single text file called an *Image Data File*. The Image Data File is stored in the same directory as your images and when it is transferred to the QVGA Controller, it stores your image data in RAM and then copies it into Flash. You must transfer the Image Data File using QEDTerm, Mosaic's Terminal Program, to the QVGA Controller before you transfer your application code. For more information on the Image Conversion Program, see the section on the Image Conversion Program .

The Image Header File

An *Image Header File* is also created with the Image Conversion Program and stored in the same directory as your images. The Image Header File associates the names of your images with the location of their image data in Flash. The Image Header File must be included in your application in order for you to create graphic objects.

The following shows the first few lines of the Image Header File used by the demo. The Image Header File defines a few constants, named after the image files down.bmp, button_press.bmp, and one.bmp shown in Figure 4-3 . The constants contain the addresses of the image data in Flash memory on the QVGA Controller. For more information on the memory map of the QVGA Controller see *Chapter 3: Making Effective Use of Memory*.

Listing 4-0 The first few lines of the Image Header File.

```
hex
0000 70 xconstant BOTTOM_EDGE_BMP
0110 70 xconstant BUTTON_PRESS_BMP
.
.
.
5806 70 xconstant ONE_BMP
```

Coding Your Application

Once you have designed and built the user interface for your application and you have transferred the images to the QVGA Controller, you can start writing the code to animate your instrument. All GUI applications need the following components:

- ☑ The pre-loaded GUI Toolkit driver software;
- ☑ The Image Data and Image Header Files from the Image Conversion Program;
- ☑ An initialization routine that initializes the GUI Toolkit and sets up a task to run your application's user interface.
- ☑ Routines that create and configure objects;

- ☑ Event procedures that animate controls; and,
- ☑ Routines that load objects onto screens.

Each of these components are described in the following sections.

The GUI Toolkit Driver Software

The GUI Toolkit driver software is pre-installed on your QVGA Controller. However, you will need to copy the library files (which define constants, structures, and headers for the GUI Toolkit methods) to the same directory as your source code as described below. To reload the driver software, please see section XX.

The GUI Toolkit driver software is provided as a pre-coded modular runtime library, known as a "kernel extension" because it enhances the on-board kernel's capabilities. The library functions are accessible from C and Forth.

The kernel extension for the GUI Toolkit is available from Mosaic Industries on the Demo and Drivers media (diskette or CD). Look in the Drivers directory, in the subdirectory corresponding to your hardware, in the GUI Toolkit folder.

The kernel extension is shipped as a "zipped" file named "packages.zip". Unzipping it (using, for example, winzip or pkzip) extracts the following files:

- readme.txt - Provides summary documentation about the library.
- install.txt - The installation file, to be loaded to COLD-started QED Board.
- library.4th - Forth name headers and utilities; prepend to Forth programs.
- library.c - C callers for all functions in library; #include in C code.
- library.h - C prototypes for all functions; #include in extra C files.

Library.c and library.h are only needed if you are programming in C. Library.4th is only needed if you are programming in Forth. The uses of all of these files are explained below.

We recommend that you move the relevant files to the same directory that contains your application source code; the references to the library files in the demo assume the library files are in the same directory as the source code.

Using the Driver Code with Forth

To use the driver code with Forth, include the following directive as your first line of code in your source code file:

```
#include "library.4th"
```

Library.4th will set up a reasonable memory map and then define the constants, structures, and name headers used by the GUI Toolkit kernel extension. Library.4th leaves the memory map in the download map.

Be sure that your software doesn't initialize the memory management variables DP, VP, or NP, as this could cause memory conflicts. If you wish to change the memory map, edit the memory map commands at the top of the library.4th file itself. For more information on the memory map of the QVGA Controller see section XX. The definitions in library.4th share memory with your Forth code, and are therefore vulnerable to corruption due to a crash while testing. However, the kernel extension runtime code is immune to corruption because it is stored in Flash and it is stored on a page that is not typically accessed by code downloads.

Using the Image Data and Image Header Files

Before writing your application code, be sure to transfer the Image Data File, created with the Image Conversion Program, to the QVGA Controller. Then, be sure to include the Image Header File in your application as shown below:

Listing 4-0 The first few lines of the demo program.

```
#include “library.4th"       \ Define constants, structures, and name heads for the
                              \ GUI Toolkit and set up the memory map.
#include “image_header.4th" \ Load constants that define the location of your images.
```

Writing Your Application’s Initialization Routine

The initialization routine of your application needs to initialize the GUI Toolkit and setup a task to control the GUI Toolkit before any GUI Methods are called.

`Initialize_GUI` **must be called before any other GUI method!**

To initialize the GUI Toolkit, call Initialize_GUI, passing it a heap start address and a heap end address as shown in the demo.

Listing 4-0 Initializing the GUI Toolkit

```
hex
3000 0F xconstant START_GUI_HEAP
6FFF 0F xconstant END_GUI_HEAP
.
.
: GUI_Demo ( -- )
  START_GUI_HEAP END_GUI_HEAP Initialize_GUI
  .
  .
;
```

`Initialize_GUI` performs three actions: it sets up a heap memory structure for the GUI Toolkit; it sets up and activates a task for the GUI Toolkit; and it configures several critical objects, called the GUI Toolkit’s *pre-instantiated* objects.

All GUI objects and their associated properties are stored in a heap accessed only by the GUI Toolkit. This GUI Heap can be located in any contiguous block of RAM (i.e. it may span consecutive pages of RAM) but it is typically placed on page 0x0F from 0x3000 to 0x6FFF as shown in the example above. The size of the heap that is needed by the GUI Toolkit depends on your application. As a benchmark, the demo program uses approximately 4k of heap space. Because all objects are stored in the GUI Heap, it is easy to add, modify, or resize objects without worrying about the specifics of their memory allocation.

The GUI Toolkit runs in a stand-alone task called the GUI Task. The GUI Task executes an endless loop that:

- continuously checks to see if new objects need to be created or existing objects need to be modified;
- polls the touchscreen looking for events;
- services events when they occur; and,
- calls **Pause**.

When a touchscreen event occurs, the GUI Task sends a message indicating that the event procedure of the responsible control should be executed. **Service_GUI_Events** listens for messages from the GUI Task and executes the appropriate event procedure when commanded. **Service_GUI_Events** must be included in the infinite loop of a task that you create (separate from the GUI Toolkit's Task) as shown in the following example. This task should also be in charge of creating and initializing the objects for your application.

Listing 4-0 Servicing GUI Events

```
decimal
\ Define a constant for reserving 1k of space in the variable area for the task.
1024 constant ONE_KILOBYTE
\ Allocate space in the variable area for our task in common memory.
ONE_KILOBYTE v.instance: taskbase
.
.
.
: Build_Task ( -- )
  0\0 0\0 0\0 taskbase build.standard.task
;

: GUI_Monitor ( -- )
  .
  .
  .
  begin
    .
    .
    .
    Service_GUI_Events
    Pause
  again
;

: GUI_Demo ( -- )
  START_GUI_HEAP END_GUI_HEAP Initialize_GUI
  .
  .
  .
  Init_Keypad                           \ create & init objects for the keypad
  Build_Task                            \ build task
  CFA.For GUI_Monitor taskbase Activate   \ activate task
  Start.Timeslicer              \ start the timeslicer for timesliced multitasking
  Pause                         \ give other tasks a chance to run
```

;The GUI Toolkit is a resource whose access must be controlled. Only the GUI Toolkit (running in the GUI Task) polls the touchscreen, writes to the display, creates objects, or modifies objects. We recommend that you use only one task to run the your application's user interface so that only one task interacts with the GUI Toolkit. However, if multiple tasks in your application need to use the GUI Toolkit, you must use the resource variable **gui_resource** to assure that only

GUI Toolkit, you must use the resource variable **gui_resource** to assure that only one task accesses the GUI Toolkit at a time. See section XX for more information on using resource variables or the glossary entry for **gui_resource**.

The GUI Task should have an opportunity to run approximately once every 15 milliseconds. Calling the GUI Task more frequently will not improve performance because the GUI Task polls the processor's free running counter, TCNT, to assure it is not called more than once every 10 milliseconds. Calling the GUI Task less frequently will not cause any errors but may result in sluggish performance. The GUI Task takes approximately one millisecond to execute if no events occur, no new object needs to be created, and no existing object needs to be modified. When an event fires, the time the GUI Task takes to execute is proportional to the number of items that are loaded onto the visible screen. For a screen with 15 objects, it will take approximately 5 milliseconds to pass the event to the responsible control. Additional time is required if the responsible control has an image that has to be redrawn.

The GUI Toolkit's Pre-Instantiated Objects

The GUI Toolkit has a set of special objects that are created and configured only when you initialize the GUI Toolkit. These special objects are called the pre-instantiated objects because they are *instantiated* or created before the GUI Toolkit starts it's task. These objects include the GUI Toolkit itself, the display, six predefined screens, the touchscreen, the pen, the buzzer, and the default font. You can not create additional instances of these objects because they are tied to a specific piece of hardware or unique function of the GUI Toolkit.

The GUI Toolkit is itself an object with properties that influence the operation of the GUI Task and all other objects. The GUI Toolkit Object is called the **GUI TOOLKIT** and has properties that determine how objects such as graphics and controls are loaded into screens, whether touchscreen events are serviced, and the behavior of the GUI Task when an error occurs.

The Display Object is called the **GUI_DISPLAY**, and is tied to the physical display. Properties of the **GUI_DISPLAY** control the behavior of the display such as blanking and inverting. The **GUI_DISPLAY** contains six pre-instantiated screens called **GUI_SCREEN0** to **GUI_SCREEN5**. A property of GUI Screens is their Visibility property. Only one of the GUI Screens may be made visible at a time. When it is made visible, the other GUI Screens are automatically made not visible and the display suddenly shows the visible screen's objects. GUI Screens are the only objects that are shown on the physical display. To show an object like a graphic on the display, the object must be loaded into the visible GUI Screen. The visible screen after **Initialize_GUI** is called is **GUI_SCREEN0**.

The Touchscreen Object is also known as the **GUI_TOUCHSCREEN** and corresponds to the physical touchscreen. The **GUI_TOUCHSCREEN** generates events when a user places a finger on the physical touchscreen (press event), while the user continues pressing the physical touchscreen (held event), and when the user removes their finger (release event). Properties of the **GUI_TOUCHSCREEN** contain the identity and pixel coordinates of the last event.

The Pen Object is called the **GUI_PEN**. This useful tool generates geometric shapes and line drawings on a screen. The **GUI_PEN** has properties that control how the points and lines are drawn so you don't have to pass all the optional parameters to the drawing method each time it is called.

The Buzzer Object is called the **GUI_BUZZER** and it is associated with the piezo electric beeper. Properties of the **GUI_BUZZER** control the length of time of a beep and turn the buzzer on and off.

The Default Font Object is called the **GUI_FONT** and is associated with the 8-pixel wide by 10-pixel tall Courier New font that renders string data in a textbox onto a screen.

Creating New Objects

The GUI Toolkit also has other objects, like graphics and buttons, that you can create using the **New_Object** method. New_Object requires one input parameter that specifies the type of object you want to create; **GRAPHIC**, **FONT**, **TEXTBOX**, **BUTTON**, and **PLOT** are all valid object types. **New_Object** returns an integer that you store into a variable. This variable (which we call an object reference because it contains a reference to the object) is used to set properties of the object and perform actions on the object like loading the object onto a screen.

The following code from the demo, creates a new graphic object and stores its object reference into a variable named **graphicOne**. Later, we'll set the image of the graphic and then configure the "one" button of the keypad to use this graphic as its **DRAW_GRAPHIC**.

Listing 4-0 Creating a New Graphic.

```
variable graphicOne
.
.
.
: Init_Keypad_Images ( -- )
  GRAPHIC New_Object graphicOne !
  .
  .
  .
;
```

Objects can be created but not destroyed. Because objects are stored in the heap, you can only create a finite number of them. Different objects use different amounts of heap space. For example, graphics and fonts only use a few bytes of heap space while plots use hundreds of bytes depending on the size of its buffer.

Setting the Property of Objects

All objects have their properties set to default values when they are created so that you need to set as few properties as possible to have a fully functional object. However, certain objects like graphics require you to set its image before the graphic can be loaded onto a screen or into a button.

In the demo, **Init_Keypad_Images** creates a graphic and then sets it's image property using a constant defined in the Image Header File.

Listing 4-0 Setting the Property of a Graphic.

```
variable graphicOne
.
.
.
: Init_Keypad_Images ( -- )
  GRAPHIC New_Object graphicOne !
  graphicOne IMAGE ONE_BMP Set_Property
  .
```

```
  .
  .
;
```

Creating and Initializing Controls

Now that you're familiar with creating graphic objects, we'll show you how to create a button. To create a button:

1. Use the **New_Object** method to create a button and store the object reference to the button into a variable. By default, the button will beep when it receives a press event. You can change this behavior by setting the **BEEP_ON_PRESS** property to **GUI_FALSE**.
2. Create and define a graphic for the **DRAW_GRAPHIC** property of the button. A button needs at least one graphic to define it's size and active area.
3. Create and define a graphic for the **PRESS_GRAPHIC** property of the button. By defining a **PRESS_GRAPHIC** that is different from the **DRAW_GRAPHIC**, you will provide the user with visual feedback for press events, thus enhancing your user interface.
4. Define the **PRESS_EVENT_PROCEDURE** for the button. Writing event procedures is covered in the next section.
5. Initialize the **DRAW_GRAPHIC**, **PRESS_GRAPHIC**, **RELEASE_GRAPHIC**, and **PRESS_EVENT_PROCEDURE** properties of the button using **Set_Property**. Typically, the **RELEASE_GRAPHIC** is set to the **DRAW_GRAPHIC** to provide visual feedback for release events.

The following code shows the **Init_Keypad_Buttons** function from the demo that illustrates how to create a button that, when pressed, will beep, show a filled-in image of button on the display, and execute an event procedure. When the button is released, the filled-in image of the button is replaced with the **DRAW_GRAPHIC**.

Listing 4-0 Creating and Initializing a Control.

```
variable buttonOne
.
.
.
: Init_Keypad_Buttons ( -- )
\ Init_Keypad_Images must be called before this routine
  BUTTON New_Object buttonOne !
  buttonOne @ DRAW_GRAPHIC graphicOne @ u>d Set_Property
  buttonOne @ PRESS_GRAPHIC graphicButton_Press @ u>d Set_Property
  buttonOne @ RELEASE_GRAPHIC graphicOne @ u>d Set_Property
  buttonOne @ PRESS_EVENT_PROCEDURE cfa.for One_Event_Procedure Set_Property
  .
  .
  .
;
```

Remember that all properties are 32-bits in size so when setting any of the graphic properties of the button, the graphic object reference must be cast into a 32-bit number.

Writing Event Procedures

Event procedures are functions that you write to animate your controls. Event procedures cannot take any parameters or return any values, but they can set variables and call other GUI methods like **`Clear`** and **`Refresh`**. In the demo, the event procedure for the one button of the keypad must write the number one into the textbox of the keypad at the correct position and then refresh the keypad screen so the new number will appear. The following shows the code for the one button's event procedure.

Listing 4-0 Writing the Event Procedure.

```
\ Define a constant to represent the maximum number of numbers we can enter
\ using the keypad.
20 constant BUFFER_LENGTH
\ Declare a variable that contains the next position we're writing into the textbox
variable buffer_pos
.
.
.
\ Add the number one at the current position in the keypad textbox,
\ increment the pointer to the next position, and redraw the
\ textbox containing the new number onto the first screen.
: One_Event_Procedure ( -- )
buffer_pos @ locals{ &pos }
  &pos BUFFER_LENGTH 1 - <
  if
    textboxBuffer @ " 1" 0 &pos String_To_Textbox
    1 buffer_pos +!
    GUI_SCREEN0 Refresh
  endif
;
```

Loading Objects into Screens

The final step in creating a GUI application is loading the objects onto the screen. To load an object onto a screen, use the **`Load`** method. The following code initializes the graphics, buttons, and textbox for the keypad and then loads them onto the first visible screen, **`GUI_SCREEN0`**.

Listing 4-0 Loading a Control into a Screen.

```
: Init_Keypad ( -- )
  Init_Keypad_Images
  Init_Keypad_Buttons
  Init_Keypad_Textbox
  0 buffer_pos ! \ Init pointer to the current position within the keypad textbox
  GUI_SCREEN0 buttonOne 24 126 Load
  .
  .
  .
;
```

We've now shown you how to design and build your user interface, initialize the GUI Toolkit, create and initialize objects, write event procedures, and load objects onto screens. This final section covers how to handle errors that may occur during your application.

Handling Errors

The GUI Toolkit has extensive error handling and reporting abilities. When an error occurs, an error code is logged and an attempt is made to recover from the error. For example, if you try to load a graphic that contains an image which is 24 pixels wide to the coordinates (304,0), 8 pixels of the image will hang off the edge of the screen (since the width of a screen is 320 pixels). This will generate an **X_OUT_OF_RANGE** error, and the graphic will instead be loaded to the right-most position on the screen at (296,0). You can read the error code by using the **Read_Error** method of the GUI Toolkit object. Because only the last error is recorded, you must call Clear_Error after you read the error to prevent servicing the same error more than once. The following code shows how to create a custom error handling and reporting routine:

Listing 4-0 Handling Errors.

```
: Print_Error ( -- )
  Read_Error
  case
    HEAP_FULL of ." Can't create any more new objects" cr endof
    X_OUT_OF_RANGE of ." X coordinate is out of range" cr endof
    \ You would put other error codes and error messages here.
  endcase
  Clear_Error \ Be sure to clear the error after we read it; we don't want
              \ to respond to the same error more than once.
;
```

Multiple errors can occur for each GUI Toolkit method but only one can occur at a time. It is up to you to decide if and when you should check for errors.

A useful debugging tool that comes with the GUI Toolkit is the property **ABORT_ON_ERROR**. If true, an error causes the controller to disable interrupts, stop multitasking, and send a descriptive error message to the primary serial port, Serial 1. The error message identifies the calling method and parameters that triggered the error. **ABORT_ON_ERROR** helps you to quickly identify the source of the error; this is especially useful when developing a complex application. This property should be set to **GUI_FALSE** for the production version of your application.

Expanding the GUI Toolkit's Objects and Methods

We understand that each GUI application you write has unique requirements. Although it is possible for you to implement many things like custom plotting routines or drawing geometrical shapes by writing high level code, there maybe a significant performance advantage in our providing pre-packaged objects to you that perform the desired function. Please contact Mosaic Industries about adding additional objects, properties, and methods to the GUI Toolkit. We are planning to add many features to the toolkit to enhance it's capabilities; the features you are looking for may already exist.

GUI Objects in Detail

There are many kinds of GUI objects. Some are tied tightly to particular hardware (the **GUI_TOUCHSCREEN** and **GUI_DISPLAY**), some collect other objects together (GUI Screens), some respond to events (buttons), some are tools (the **GUI_PEN**), and some simply contain data for display

(graphics and textboxes). Several of the GUI objects are created and configured when **Initialize_GUI** is called. These objects include the **GUI_DISPLAY**, **GUI_TOUCHSCREEN**, **GUI_BUZZER**, **GUI_FONT**, GUI Screens, and **GUI_PEN**. You can not create more of these object types. For other object such as graphics, textboxes, buttons, and plots, you can create as many as you like. All of the GUI Toolkit objects, their 32-bit properties, associated methods, and possible errors are described below.

GUI_TOOLKIT

The GUI Toolkit is itself an object that contains properties that effect all other objects. The GUI Toolkit Object is called **GUI_TOOLKIT**. These are its properties, methods, and possible errors:

Properties

Table 4-1 Properties of the GUI_TOOLKIT**.**

Property	Values	Meaning
HORIZONTAL_SNAP_TO_GRID	**GUI_TRUE**	Read only. Causes all objects loaded onto a screen to be located on the nearest 8-pixel horizontal boundary. This property is always true for speed.
VERICAL_SNAP_TO_GRID	**GUI_TRUE**, **GUI_FALSE**	If true, causes all objects loaded on to a screen to be located on the nearest 8-pixel vertical boundary.
SERVICE_EVENTS	**GUI_TRUE**, **GUI_FALSE**	If false, events are not serviced. All events generated by the touchscreen are ignored.
STANDARD_FONT	**GUI_FONT** or custom font object	Contains the reference to the font object used to render ASCII string data in textboxes. At initialization, **STANDARD_FONT** is set to **GUI_FONT** which is Courier New with a width of 8 pixels and a height of 10 pixels. You can set this property to your own custom font, if desired.
ABORT_ON_ERROR	**GUI_TRUE**, **GUI_FALSE**	If true, disables interrupts, stops multitasking, and sends a detailed message to serial 1 when an error occurs.
AVAILABLE_HEAP_SPACE	Long	Read only. Contains the number of bytes remaining in the GUI Heap.

Methods

Table 4-2 Methods of the GUI_TOOLKIT**.**

Method	Parameters	Return Value	Action
Initialize_GUI	start of the GUI Heap, end of the GUI Heap	none	Initializes the **GUI_TOOLKIT**, **GUI_DISPLAY**, **GUI_TOUCHSCREEN**, **GUI_PEN**, **GUI_FONT**, **GUI_BUZZER**, and GUI Screens. Sets up and activates a task in common memory. Calling any other GUI Toolkit method before calling **Initialize_GUI** will crash the controller.

Method	Parameters	Return Value	Action
Service_GUI_Events	none	none	Executes the event procedures of controls. This method must be included in a task as part of the infinite loop to service events generated from the GUI Task.
Read_Error	none	error code	Reads the last reported error.
Clear_Error	none	none	Clears the last reported error.
New_Object	object type	object reference	Create a new object and return an object reference which is stored in an object variable for later use.
Set_Property	object reference, property, value	none	Set an object's property as a 32-bit value.
Get_Property	object reference, property	value	Get an object's property as a 32-bit value.

Errors

Table 4-3 Errors of the `GUI_TOOLKIT`**.**

Error	Associated Method	Meaning
HEAP_FULL	**Initialize_GUI**, **New_Object**	Heap is full. There is no room to create the GUI Toolkit, create new objects, or resize existing objects.
GUI_EXISTS	**Initialize_GUI**	The GUI Task has already been activated. Do not call **Initialize_GUI** more than once!
DISPLAY_INITIALIZATION_FAILURE	**Initialize_GUI**	There was a problem initializing the display. This error occurs if there was a hardware communications problem talking to the display controller. If the problem persists, contact Mosaic Industries.
INVALID_OBJECT	**New_Object**, **Set_Property**, **Get_Property**	The object reference is not valid.
INVALID_PROPERTY	**Get_Property**, **Set_Property**	The property is not valid.
INVALID_IMAGE	**Set_Property**	There is no valid image at the specified address. This error is returned after attempting to set the **IMAGE** property of a graphic or a font.
INVALID_GRAPHIC	**Set_Property**	The object reference is not a graphic, setting the **DRAW_GRAPHIC**, **PRESS_GRAPHIC**, or **RELEASE_GRAPHIC** property of a button.
INVALID_FONT	**Set_Property**	The object reference is not a font, setting the **STANDARD_FONT** property of the GUI Toolkit or setting the **TEXTBOX_FONT** property of a textbox.

Error	Associated Method	Meaning
BUFFER_SIZE_OUT_OF_RANGE	**Set_Property**	The size of the plot object buffer must be greater than or equal to the width of the plot in pixels
COLUMN_OUT_OF_RANGE	**Set_Property**	The number of columns will create a textbox that is wider than the width of the screen.
ROW_OUT_OF_RANGE	**Set_Property**	The number of rows will create a textbox that is taller than the height of the screen.
WIDTH_OUT_OF_RANGE	**Set_Property**	The width will create a plot that is wider than the width of the screen.
HEIGHT_OUT_OF_RANGE	**Set_Property**	The height will create a plot that is taller than the height of the screen.

GUI_DISPLAY

The **GUI_DISPLAY** corresponds to the physical display. Properties of the **GUI_DISPLAY** and all other GUI objects are accessed or modified using the **GUI_TOOLKIT** methods **Set_Property** and **Get_Property**.

Properties

Table 4-4 Properties of the GUI_DISPLAY**.**

Property	Values	Meaning
WIDTH_IN_PIXELS	320 pixels	Read only. Contains the width of the display in pixels.
HEIGHT_IN_PIXELS	240 pixels	Read only. Contains the height of the display in pixels.
CONTRAST	0 to 31	Changes the contrast voltage of the monochrome display. Saturates at 0 or 31. Initialized to 16.
BACKLIGHT_ON	**GUI_TRUE**, **GUI_FALSE**	If true, turns the backlight on for monochrome displays.
BLANK_DISPLAY	**GUI_TRUE**, **GUI_FALSE**	If true, blanks the display.
INVERT_DISPLAY	**GUI_TRUE**, **GUI_FALSE**	If true, inverts all pixels on the display.

GUI Screens

A screen is a collection of functionally related objects, including textboxes, plots, buttons and graphics that are presented together on the display. Screens are fixed in size to 320 pixels wide by 240 pixels tall. For speed, objects loaded into screens are aligned on 8 pixel horizontal grid. A property of the GUI Toolkit, called **VERTICAL_SNAP_TO_GRID**, allows you to align objects vertically as well. The coordinate system for specifying the location of objects and drawing to a screen is in pixels and the origin is the upper left corner of a screen. The x coordinate represents a displacement from the origin in the horizontal direction and the y coordinate represents a displacement

from the origin in the vertical direction. An increase in x moves the coordinate to the right and an increase in y moves the coordinate down.

Screens whose object images are stored in the display's memory are called GUI Screens. GUI screens have a visible property that when set, causes the selected screen to be shown on the display and makes all other GUI Screens invisible. Only one GUI Screen can be made visible at a time. GUI Screens are the only objects that are shown on the physical display; to show an object like a graphic on the display, the object must first be loaded into a GUI Screen. The visible screen after **`Initialize_GUI`** is called is **`GUI_SCREEN0`**.

Properties

Table 4-5 Properties of GUI Screens.

Property	Values	Meaning
`WIDTH_IN_PIXELS`	320 pixels	Read only. Contains the width of the screen in pixels.
`HEIGHT_IN_PIXELS`	240 pixels	Read only. Contains the height of the screen in pixels.
`VISIBLE`	**`GUI_TRUE`**, **`GUI_FALSE`**	If true, makes the screen visible and sets the last visible screen's property to false.
`NUM_OBJECTS`	Long	Read only. Contains the number of objects contained in the screen.
`IN_DISPLAY`	**`GUI_TRUE`**, **`GUI_FALSE`**	Read only. TRUE if the screen is in the display memory, FALSE otherwise. All GUI screens have this property set to **`GUI_TRUE`**.
`SCREEN_ADDRESS`	Extended Address	Read only. Contains the address of the screen.

Methods

Table 4-6 Methods of GUI Screens.

Method	Parameters	Return Value	Meaning
`Load`	screen reference, object reference, x coordinate, y coordinate	none	Load an object into a screen at the specified x and y coordinates.
`Un_Load`	screen reference, object reference, x coordinate, y coordinate	none	Unload an object from a screen at the specified x and y coordinates.
`Clear`	screen reference	none	Unloads all objects from the screen and erases the screen.
`Refresh`	screen reference	none	Re-renders textboxes and plots in the screen that have their refresh flags set to true. Sets the refresh flags to false once the objects have been refreshed.
`Redraw`	screen reference	none	Re-draws all objects contained in the screen.

Errors

Table 4-7 Errors of GUI Screens.

Error	Associated Method	Meaning
HEAP_FULL	**Load**	Heap is full. There is no room to load an object into a GUI Screen.
INVALID_OBJECT	**Load**, **Un_Load**	The object reference cannot be loaded into a GUI Screen.
INVALID_SCREEN	**Load**, **Un_Load**, **Clear**, **Redraw**, **Refresh**	The object reference is not a GUI Screen.
X_OUT_OF_RANGE	**Load**, **Un_Load**	The x coordinate plus the width of the object is greater than the width of the GUI Screen.
Y_OUT_OF_RANGE	**Load**, **Un_Load**	The y coordinate plus the height of the object is greater than the height of the GUI Screen.

GUI_TOUCHSCREEN

The **GUI_TOUCHSCREEN** corresponds to the physical touchscreen and serves as a container for all the properties of the touchscreen.

Properties

Table 4-8 Properties of the `GUI_TOUCHSCREEN`**.**

Property	Values	Meaning
LAST_EVENT	**NO_EVENT**, **PRESS_EVENT**, **HELD_EVENT**, **RELEASE_EVENT**	Read only. Contains the last touchscreen event that occurred.
CAL_COORDS	Long	Read only. Contains the calibrated touchscreen coordinates of the last touchscreen event. The y coordinate is located in the 16 most significant bits and the x coordinate is located in the 16 least significant bits.
RAW_COORDS	Long	Read only. Contains the uncalibrated touchscreen readings of the last touchscreen event. The y reading is located in the 16 most significant bits and the x coordinate is located in the 16 least significant bits.

Methods

Table 4-9 Methods of the `GUI_TOUCHSCREEN`**.**

Method	Inputs	Outputs	Meaning
`Calibrate`	point1, point2, point3	none	Calibrates the analog touchscreen. See the section on Touchscreen Calibration for more information. Each point is a 32-bit number with the uncalibrated y reading in the 16 most significant bits and the uncalibrated x reading in the 16 least significant bits. The calibration coefficients are stored in Flash memory with the GUI Toolkit.

Errors

Table 4-10 Errors of the `GUI_TOUCHSCREEN`**.**

Error	Associated Method	Meaning
`FLASH_NOT_PROGRAMABLE`	**`Calibrate`**	Can't store the calibration coefficients into Flash memory. Make sure the Flash memory is not write protected.

Touchscreen Calibration

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

The following example code shows how to calibrate the touchscreen. In the example code, there are two routines. The first routine, called **`Get_Raw_Coords`**, waits until there is a press event before beeping the buzzer and reading the raw coordinates of the press event. **`Get_Raw_Coords`** then waits until the user has removed their finger before returning. The second routine, called **`Calibrate_Touchscreen`** clears **`GUI_SCREEN0`**, makes **`GUI_SCREEN0`** the visible screen, prompts the user to press three locations on the touchscreen, captures the user's touches using the **`Get_Raw_Coords`** routine, and finally calls the **`Calibrate`** method.

Listing 4-0 Example Calibration Routine.

```
\ Declare an object variable for the textbox used to hold the calibration
\ messages.  Used in calibrate_gui
variable textboxCalibration
```

```
: Get_Raw_Coords ( -- raw_x \ raw_y )
  begin
    Pause
    GUI_TOUCHSCREEN LAST_EVENT get_property HELD_EVENT d=
  until \ Wait until the user presses the touchscreen before continuing...
  Buzz  \ Provide audio feedback for the press.
  GUI_TOUCHSCREEN RAW_COORDS get_property
  begin
    Pause
    GUI_TOUCHSCREEN LAST_EVENT get_property HELD_EVENT d= not
  until \ Wait until the user releases their finger before continuing...
;
: Calibrate_Touchscreen ( -- )
  \ Calibrates the analog touchscreen by asking the user to press three
  \ points on the display.  Initialize_GUI must be called before calling
  \ this routine.
  GUI_SCREENO Clear \ Make sure the screen is clear.
  GUI_SCREENO VISIBLE GUI_TRUE Set_Property \ Make sure screen is visible.

  \ Display a message to the user, informing them that we need to calibrate
  \ the touchscreen.
  TEXTBOX new_object textboxCalibration !
  textboxCalibration @ ROWS     4 u>d set_property
  textboxCalibration @ COLUMNS 26 u>d set_property
  textboxCalibration @ " Entering Calibration Mode." 0 0 string_to_textbox
  textboxCalibration @ " Please remove finger from " 1 0 string_to_textbox
  textboxCalibration @ " touchscreen and touch the " 2 0 string_to_textbox
  textboxCalibration @ " following points:         " 3 0 string_to_textbox
  GUI_SCREENO textboxCalibration @  0 100 load
  textboxCalibration @ ROWS    1 u>d set_property
  textboxCalibration @ COLUMNS 7 u>d set_property

  textboxCalibration @ " POINT 1" 0 0 string_to_textbox
  GUI_SCREENO textboxCalibration @ 40 15 load       \ Label point 1.
  \ The GUI_PEN, on start up, is configured to draw pixels.  Properties
  \ of the GUI_PEN allow you to also erase pixels, draw lines, and erase
  \ lines.
  35 19 draw 33 19 draw 34 18 draw 34 20 draw       \ Draw point 1.
  get_raw_coords                                    \ Get point 1.

  textboxCalibration @ " POINT 2" 0 0 string_to_textbox
  GUI_SCREENO textboxCalibration @  224 120 load      \ Label point 2.
  291 123 draw 289 123 draw 290 124 draw 290 122 draw \ Draw point 2.
  get_raw_coords                                      \ Get point 2.

  textboxCalibration @ " POINT 3" 0 0 string_to_textbox
  GUI_SCREENO textboxCalibration @ 168 215 load         \ Label point 3.
  163 219 draw 161 219 draw 162 218 draw 162 220 draw   \ Draw point 3.
  get_raw_coords                                        \ Get point 3.
  calibrate
;
```

GUI_PEN

The **GUI_PEN** is a tool used to draw points, lines, and shapes on a screen. You use its **Draw** method to draw to the screen, and its properties control how points or lines are rendered so you don't have to pass all the optional parameters to the **Draw** method.

Properties

Table 4-11 Properties of the `GUI_PEN`**.**

Property	Values	Meaning
PEN_TYPE	**SET**, **UNSET**	Determines whether the pen draws or erases from the screen.
SHAPE	**POINT**, **LINE**	Determines the shape that is drawn or erased from the screen.
TARGET_SCREEN	Screen Reference	Contains the screen the pen is rendering to.
LAST_COORDS	Long	Contains the last coordinates passed to the **Draw** method. The y coordinate is located in the 16 most significant bits and the x coordinate is located in the 16 least significant bits.

Methods

Table 4-12 Methods of the `GUI_PEN`**.**

Method	Inputs	Outputs	Meaning
Draw	x coordinate, y coordinate	none	Draw or erase a point or a line onto a screen. If drawing a line, draws from the last coordinate to the coordinates passed to **Draw**. To draw a line from an arbitrary point rather than the last coordinate, place the new coordinate into **LAST_COORDS**.

Errors

Table 4-13 Errors of the `GUI_PEN`**.**

Error	Associated Method	Meaning
X_OUT_OF_RANGE	**Draw**	The x coordinate is greater than the width of the screen.
Y_OUT_OF_RANGE	**Draw**	The y coordinate is greater than the height of the screen.

GUI_BUZZER

The **GUI_BUZZER** corresponds to the piezo electric beeper and controls the length of the beep when a button is pressed. The buzzer is located on the QVGA Controller PCB. An external buzzer can be attached for increased audibility. Please see Section XX for more information.

Properties

Table 4-14 Properties of the `GUI_BUZZER`**.**

Property	Values	Meaning
BEEP_TIME	Long	Controls the length of time the piezo electric buzzer is on during a call to the **Buzz** method and when a button receives a press event. Only the 16 least significant bits are used. The units of **BEEP_TIME** are in TCNTs or 2 microsecond intervals. The default value is 1000, corresponding to 2 milliseconds.

Property	Values	Meaning
BEEPER_ON	**GUI_TRUE**, **GUI_FALSE**	If true, turns the piezo electric buzzer on.

Methods

Table 4-15 Methods of the GUI_BUZZER**.**

Method	Inputs	Outputs	Meaning
Buzz	none	none	Turns the piezo electric buzzer on for **BEEP_TIME**. This method is automatically called when a button receives a press event.

Graphic Object

A graphic object contains a single image as a property. Images are typically created on a PC but need to be converted and transferred to the QVGA Controller before they can be loaded into a graphic. Images are converted using Mosaic's Image Conversion Program. The Image Conversion Program creates two files, an Image Data File and an Image Header File. The Image Data File contains the converted image data, and the Image Header File associates the location of the image data on the QVGA Controller with a constant named after the file name of the image. Before the GUI Toolkit is initialized and graphics are created, the Image Data File must be transferred to the Controller using QEDTerm. After the Image Data File is transferred and the GUI Toolkit is initialized, a new graphic is created using the **New_Object** method and its image set using the **Set_Property** method. The width and height of the graphic object are automatically set when the **IMAGE** property of the graphic is set to a valid image. Error checking is performed to assure the image is valid.

For the optimal performance of the GUI Toolkit, the width of an image must be a multiple of 8 pixels, and graphics must be loaded onto an 8 pixel horizontal grid in a screen. Finer horizontal pixel placement and image widths would require bit shifting of each byte of the image every time it is drawn, significantly slowing performance of the GUI Toolkit. To enforce the alignment there are two properties of the GUI Toolkit that globally affect the placement of objects, **HORIZONTAL_SNAP_TO_GRID** and **VERTICAL_SNAP_TO_GRID**. **HORIZONTAL_SNAP_TO_GRID** is read-only and is always true where as **VERTICAL_SANP_TO_GRID** may be set to either true or false. Images whose width does not fall on 8 pixel boundaries will not be converted with the Image Conversion Program. The images must be cropped or stretched appropriately using an image-editing program before the Image Conversion Program is used.

Properties

Table 4-16 Properties of Graphic Objects.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 320 pixels	Read only. Contains the width of the image in pixels.

Property	Values	Meaning
HEIGHT_IN_PIXELS	1 to 240 pixels.	Read only. Contains the height of the image in pixels.
IMAGE	Extended Address	Contains a pointer to the image contained in the graphic.

Image Conversion Program

Mosaic's Image Conversion Program allows you to easily transfer images created on a PC to your Panel-Touch QVGA Controller for use with the GUI Toolkit. The image conversion program takes all of the images in a selected directory and concatenates them into a single text file called an Image Data File. The Image Data File is stored in the selected directory and is sent to the Controller using QEDTerm, the terminal program provided by Mosaic Industries. The Image Data File, by default, puts the converted image data into RAM on page 0x60 and then transfers the data into Flash at page 0x70. For more information about the memory map of the QVGA Controller, see section XX.

Another file, called an Image Header File, is created to associate the names of the images (which is just the filename of the image) with their location in memory on the Controller. The Image Header File contains constants named after the file names of the images and must be included in your program to tell the GUI Toolkit where the images are stored. Supported image formats are monochrome, grayscale, color bitmap (*.bmp), and PCX (*.pcx) files with 1, 2, 4, and 8 pixel bit depth. If you have a monochrome or EL display, be sure all of your images are monochrome and you select a bit depth of 1 before converting your images. The following sections describe the user interface and the error messages returned by the Image Conversion Program.

Image Conversion Program Interface

The user interface for Mosaic's Image Converions Program has a main screen, an advanced options screen, and a help screen. On the main screen there are controls that select the directory that contain your image files, specify the type of image file (PCX or BMP) to convert, and select the bit-depth of your image files (1, 2, 4, or 8 bits per pixel). A button labeled "Convert Files Now" starts the conversion process. In the advanced options screen there are controls that select your programming language, desired target memory location for the image data, and filenames for the Image Data and Image Header files. All of the advanced options are set to default values that will work for most applications. The help screen provides additional information for each of the options and controls.

Image Conversion Program Errors

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

- "Error changing to the specified directory". The directory does not exist, it was moved, or deleted. A new directory has to be specified.
- "Error opening an image file, Image Data File, or Image Header File".
- "Not a valid bitmap file (no valid file identifier) or pcx file (no valid manufacturer or encoding)".
- "Bit depth of the image file does not match the specified value".

- "Image width does not fall on an 8-pixel boundary". All images must have a width that is a multiple of eight pixels. This is required to quickly draw the images onto screens. Please crop or stretch the image using any photo or image editing program such as Photoshop or Paint.

Font Object

A font object renders ASCII string data into images that can be displayed on a screen in a textbox. A font object is like a graphic object in that it contains an image, with the bitmaps for the 95 printable ASCII characters starting from ASCII space (0x20 or 32 decimal) to ASCII tilde (0x7E or 126 decimal). Fonts are created on the PC. Each character bitmap must be the same size. The GUI Toolkit provides a Courier New font that is 8 pixels wide by 10 pixels tall and is called the **`GUI_FONT`**. The demo program provides two additional Courier New fonts that are 16 pixels wide by 20 pixels tall and 32 pixels wide by 40 pixels tall. Examples of their use are included with the demo program. While you can't create another instance of the **`GUI_FONT`**, you can create your own custom fonts as described in the next section.

Properties

Table 4-17 Properties of the Font Object.

Property	Values	Meaning
`WIDTH_IN_PIXELS`	8 to 320 pixels	Read only. Contains the width of a single character in pixels.
`HEIGHT_IN_PIXELS`	1 to 240 pixels.	Read only. Contains the height of a single character in pixels.
`IMAGE`	Extended Address	Contains a pointer to the image of ASCII characters contained in the font.

Creating Custom Fonts

You can easily create your own custom fonts on a PC to use with the GUI Toolkit. In this section, we show you how to create a custom font using Adobe Photoshop 4.0.

1. Start Photoshop and create a new image file with dimensions of 760 pixels wide x 10 pixels tall in bitmap mode. This will create a font that is 8 pixels wide and 10 pixels tall (since 760 = 95 ASCII characters x 8 pixels per character).
2. Use the text tool, selecting Courier New as the font, to create the font by typing in all 95 characters in order.
3. Set the font size to 12 pixels with 1 leading pixel and 1 pixel space between characters. The font size in Photoshop is a bit misleading since the 12-pixel font is really 10 pixels tall and 8 pixels wide.
4. Save this font as a bitmap image so you can process the file using the Image Conversion Program.

For a larger Courier New font, say 16 pixels wide by 20 pixels tall, create a 1520 pixel wide x 20 pixel tall image and set Photoshop's font size to 22 with 3 leading pixels and 3 pixel spacing. Any font can be used as long as each character has the same width (for non-fixed spaced fonts, you may

have to manually reposition each character). The width of a font must also be a multiple of 8 pixels. There are no limitations on the height of the font.

Textbox Object

Textboxes are used in conjunction with fonts to render ASCII string data on a screen so it can be shown on the display. Textboxes are useful for displaying information that constantly changes, like the status of a process or the state of an instrument.

When textboxes are created, the font referenced by the **STANDARD_FONT** property of the GUI Toolkit, is stored into the **TEXTBOX_FONT** property of the textbox. If you did not store a custom font into the **STANDARD_FONT** property of the GUI Toolkit, the **GUI_FONT** is stored into the **TEXTBOX_FONT** property of the textbox. The **GUI_FONT** is Courier New and is 8 pixels wide by 10 pixels tall. The initial size of a textbox is 10 columns or characters wide by 2 rows or lines tall.

Properties

Table 4-18 Properties of the Textbox Object.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 320 pixels	Read only. Contains the width of the textbox in pixels. This value is the product of the font width and the number of columns in the textbox.
HEIGHT_IN_PIXELS	10 to 240 pixels	Read only. Contains the height of the textbox in pixels. This value is the product of the font height and the number of rows in the textbox.
COLUMNS	1 to 40 columns	Contains the number of characters per line of the textbox.
ROWS	1 to 24 rows	Contains the number of rows of characters of the textbox.
TEXTBOX_FONT	Font reference	Contains a reference to the font object used to render the ASCII data in the textbox.
BORDER	**GUI_TRUE**, **GUI_FALSE**	If true, draws a one-pixel border around the textbox.

Methods

Table 4-19 Methods of the Textbox Object.

Method	Inputs	Outputs	Meaning
String_To_Textbox	textbox reference, string, row, column	none	Writes an ASCII string into a textbox at the specified row and column.
Textbox_To_String	textbox reference, count, row, column	string	Read count characters from the textbox at the specified row and column into an ASCII string.
Clear	textbox reference	none	Sets all of the characters in the textbox to ASCII space.

Errors

Table 4-20 Errors of the Textbox Object.

Error	Associated Method	Meaning
INVALID_OBJECT	**Clear**	The object reference is not a textbox.
INVALID_TEXTBOX	**Textbox_To_String**, **String_To_Textbox**	The object reference is not a textbox.
STRING_LENGTH_OUT_OF_RANGE	**Textbox_To_String**, **String_To_Textbox**	For **Textbox_To_String**, the length of the string being read is longer than 40 characters or goes past the end of the textbox. For **String_To_Textbox**, the length of the string being written is longer than 40 characters or the string is being written past the end of a line of the textbox.
COLUMN_OUT_OF_RANGE	**Textbox_To_String**, **String_To_Textbox**	The column is greater than the number of columns in the textbox.
ROW_OUT_OF_RANGE	**Textbox_To_String**, **String_To_Textbox**	The row is greater than the number of rows in the textbox.

Controls

Controls are objects that have a graphical representation on a screen and respond to touchscreen events. Controls must have exclusive active regions on a screen, which means that controls can not overlap one another.

Button Object

A Button is the most basic and essential control. A button typically has an image that is drawn to a screen when it is loaded. The coordinates used to load the button into the screen specify the button's top left corner. The image that is drawn to the screen when it is loaded is called the button's **DRAW_GRAPHIC** and it defines the active area of the button. When someone touches the touchscreen anywhere within the area of the **DRAW_GRAPHIC**, the button receives a **PRESS_EVENT**. When the button receives the **PRESS_EVENT**, it could beep the buzzer, draw a different image (called a **PRESS_GRAPHIC**) to the screen at the button's top left corner, and/or execute a user defined function called a **PRESS_EVENT_PROCEDURE**. If the user continues to press the touchscreen within the active area of the button, the button receives a **HELD_EVENT** and will execute the user defined **HELD_EVENT_PROCEDURE** if it is available. When the user removes their finger from the touchscreen or moves off the button, the button receives a **RELEASE_EVENT**. The button, depending on the settings of its properties, may beep the buzzer, draw a third image called a **RELEASE_GRAPHIC** to the button's top left corner, and finally execute a **RELEASE_EVENT_PROCEDURE**. A textbox, called a **BUTTON_TEXTBOX**, may also be associated with a button if you need text in your button. The textbox is rendered to the button's top left corner when the button is loaded, overwriting the **DRAW_GRAPHIC**. If present, the textbox is redrawn when the button receives a release event.

Properties

Table 4-21 Properties of the Button Object.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 320 pixels	Read only. Set based on the width of the draw graphic.
HEIGHT_IN_PIXELS	1 to 240 pixels	Read Only. Set based on the height of the draw graphic.
DRAW_GRAPHIC	Graphic object reference	Graphic shown when the button is drawn. Required to define the active area.
PRESS_GRAHIC	Graphic object reference	Graphic shown when the button is pressed. If zero, no graphic is drawn.
RELEASE_GRAPHIC	Graphic object reference	Graphic shown when the button is released. If zero, no graphic is drawn.
BUTTON_TEXTBOX	Textbox object reference	Contains the Textbox shown when the object is drawn.
BEEP_ON_PRESS	**GUI_TRUE**, **GUI_FALSE**	If true, beep the buzzer when the button is pressed.
BEEP_ON_RELEASE	**GUI_TRUE**, **GUI_FALSE**	If true, beep the buzzer when the button is released.
PRESS_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the button is pressed. If zero, no event procedure is executed.
HELD_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the button is held. If zero, no event procedure is executed.
RELEASE_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the button is released. If zero, no event procedure is executed.

Plot Object

Plot objects are used to render data (such as temperature, voltage, or pressure) into a graphical form onto a screen. Plot objects contain a one-dimensional circular byte array, called the plot buffer, that holds the data. The initial size of a plot is 240 pixels wide by 120 pixels tall, with a 240-byte buffer. The size of the plot buffer is limited by the size of the GUI Heap and must be greater than or equal to the width of the plot in pixels. The data in the plot buffer specifies the y coordinate, as the vertical distance in pixels, measured from the top edge of the plot object. Data is added into the plot buffer sequentially; the position of the data in the plot buffer represents the x coordinate and sequential points are connected by lines. For example, if we enter the two data values 4 and 57 into a new plot object and the plot's upper left corner is at (8,10), a line would be drawn from point (8,14) to (9,67). When the plot buffer is the same size as the plot width and you get to the end of the buffer, the next data point you add will be written into the first position of the buffer and the data will be plotted to the left-edge of the plot (over-writing the previous point on the screen and data in the buffer). When the buffer is larger than the plot width and you get to the end of the plot, the next data point that you add will be written to the next position of the buffer and the data will be plotted to the left-edge of the plot (over-writing only the previous point on the screen). This provides plots that look like oscilloscope traces.

The **Add_Data** method adds a single data value into a plot object. The **Refresh** method must be called after data is added to re-render the plot on the screen. The plot is rendered from left to right. For example, the following code creates a new plot, loads it into **GUI_SCREEN0**, and then repeatedly adds a random data value from 0 to 119 into the plot buffer and refreshes the plot on the screen.

Listing 4-0 Using the Plot Object.

```
variable plotRandom
: Example_Plot ( -- )
  PLOT New_Object plotRandom !
  GUI_SCREEN0 plotRandom @ 40 60 Load
  plotRandom @ random 119 umod Add_Data
  begin
    plotRandom @ random 119 umod Add_Data
    GUI_SCREEN0 Refresh
    pause
  again
;
```

Properties

Table 4-22 Properties of the Plot Object.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 320 pixels	Contains the width of the plot in pixels.
HEIGHT_IN_PIXELS	1 to 240 pixels	Contains the height of the plot in pixels.
BUFFER_SIZE	Long	Sets the size of the circular plot buffer. The plot buffer must be greater than or equal to the width of the plot in pixels.
BORDER	**GUI_TRUE**, **GUI_FALSE**	If true, draws a one pixel border around the plot

Methods

Table 4-23 Methods of the Plot Object.

Method	Inputs	Outputs	Meaning
Add_Data	plot reference, data	none	Adds a single data value into a plot object. Data is added to the next position in the circular plot buffer. The data value represents the distance in pixels as measured from the top edge of the plot.

Errors

Table 4-24 Errors of the Plot Object.

Error	Associated Method	Meaning
DATA_OUT_OF_RANGE	**Add_Data**	The data value is larger than the height of the plot in pixels.
INVALID_PLOT	**Add_Data**	The object reference is not a plot.

Part 3

Communications Measurement and Control

Chapter 5

Chapter 5: Digital and Timer-Controlled I/O

Overview of Available Digital I/O

The QVGA Controller has plenty of digital and analog I/O. Table 5-1 summarizes the analog and digital I/O available.

Table 5-1 All I/O available on the QVGA Controller.

I/O Channels	Type
5	Digital Outputs
4	Digital Inputs
17	Digital Inputs or Outputs
4	Open-Drain High-Current outputs
8	8-bit Analog Inputs
8	12-bit Analog Inputs
8	8-bit DAC Outputs
2	Serial Communications Ports
56	Simultaneously Useable I/O Channels

Many of these 56 I/O lines are digital inputs and outputs. Including the high current drivers the maximum number of digital inputs and outputs is 41 (up to 32 can be configured as inputs, up to 29 as outputs) if none are used for the 8-bit A/D, the RS485, or the secondary serial port. If all of those alternate services are used, thirty (30) fully uncommitted digital I/O lines remain. Of these, up to 21 can be configured as inputs, up to 26 as outputs. Table 5-2 summarizes the *digital* I/O and alternate uese of some of the I/O pins.

Digital inputs and outputs are very useful in data acquisition, monitoring, instrumentation and control applications. A low voltage (approximately 0 Volts) is established on a digital output pin when the processor writes a logical 0 to the corresponding bit in a data register associated with the digital output port. A high voltage (approximately 5 Volts) is established on the digital output pin when the processor writes a 1 to a corresponding bit in the port's data register. This allows software to control external events and processes. For example, an application program can use digital outputs to activate solenoids and turn switches and lights on and off, or to interface the QVGA Controller with a wide variety of digital accessories.

Table 5-2 Available Digital I/O on the QVGA Controller

Signal Type	Available Channels	Hardware Resource	Alternate Uses
Digital inputs or outputs, bytewise configurable	8	PPA	None
Digital outputs	5	PPB 0-4	None
Open-Drain High-Current Outputs	4	PPB 5-7 and PAL	None
Digital inputs	4	PPC 0-3	None
Digital inputs or outputs, nibblewise configurable	3 to 4	PPC 5-7, 4	Pin 4 alternately used for RS485
Digital inputs or outputs, bitwise configurable	6 to 8	CPU Port A	Pins 3-4 alternately used for 2nd serial port, Serial2
Digital inputs	0 to 8	CPU Port E	Alternately used for 8-bit analog inputs
Total	30 to 41		

A digital input allows the processor to monitor the logical state of an external voltage by reading a data register associated with the port. External voltages near 0 Volts connected to a digital input cause the corresponding bit in the port's data register to be read as a logical 0, and external voltages near 5 Volts connected to a digital input are read as a logical 1. Application programs can use digital inputs to read switches and keypads or to interface to digital devices such as A/D converters and real-time clocks.

In addition, there are four high current drivers available. These N-channel MOSFET outputs can sink 150 mA continuously, and up to 1 amp on an intermittent basis. Onboard snubber diodes allow control of inductive loads. They are particularly useful for driving relays, solenoids, and low power stepper motors.

Using digital I/O is very easy: simply configure the output port as input or output as explained below, and then use functions or assignment statements to read from or write to the port. The names of the data and direction registers and all required initialization routines are pre-defined in the header files so you don't have to worry about hexadecimal register addresses in your code.

The following sections describe the available digital I/O ports and how to use them.

The digital I/O signals on the QVGA Controller originate from a Motorolla 68HC11 processor chip and an 82C55A peripheral interface adapter (PIA) chip. The 68HC11 provides two 8 bit ports named PORTA and PORTE, and 4 available bits on PORTD (PD2 through PD5). The PIA supplies three 8 bit digital I/O ports named PPA, PPB, and PPC.

Table 5-3 summarizes the digital input/output available on the QVGA Controller including the names, addresses, and number of signals associated with the digital ports on the 68HC11 and PIA. The "configurable as" column specifies whether the direction of the port may be changed on a bit-by-bit, nibble-by-nibble, or byte basis (or in the case of PORTE, configured as digital or analog input). The final column lists alternate uses (other than standard digital I/O), the signal pins and the

number of signals associated with the alternate uses. Note that a fourth High Current driver output is controlled by the onboard PAL.

Table 5-3 Digital I/O Port Addresses and Configurability.

Port Name	Address (HEX)	I/O Line	Configurable As	Alternate Use	(Signals Used)	
68HC11:						
PORTA	8000	8	Bitwise I/O	Serial2:	PA3 & PA4	(2)
				Pulse accumulator:	PA7	(1)
				Timed inputs:	PA0-3	(3 or 4)
				Timed outputs:	PA3-7	(4 or 5)
PORTD	8008	4	Bitwise I/O	SPI controls AD12 & DAC		(4)
PORTE	800A	8	Bytewise digital or analog input	8 bit A/D:	PE0-7	(8)
PIA:						
PPA	8080	8	Bytewise I/O			
PPB	8081	8	Bytewise I/O	High Current Outs:	PPA5-7	(3)
PPC lower	8082	4	Nibblewise I/O	Keypad Inputs:	PPC0-3	(4)
PPC upper	8082	4	Nibblewise I/O	RS485:	PPC4	(1)

Table 5-4 specifies the named data direction register which controls the input/output direction, or specifies the functions that configure each digital I/O port.

Table 5-4 Digital I/O port data direction registers and configuration functions.

Port Name	Configured By
68HC11:	
PORTA	**PORTA.DIRECTION**
PORTD	**PORTD.DIRECTION**
PORTE	**A/D8.ON** **A/D8.OFF**
PIA:	
PPA	**INIT.PIA**
PPB	**INIT.PIA**
PPC lower	**INIT.PIA**
PPC upper	**INIT.PIA**

Alternate Uses of the Digital I/O Ports

Some of these port signals have alternative uses as summarized in Table 5-3 . The 68HC11's I/O ports and the PIA ports can serve a variety of selectable functions:

PORTA

PORTA may be used as bit-configurable digital input/output. The data direction (input or output) of each bit in PORTA is determined by writing to the DDRA register as described below. If any bits in PORTA are not being used for simple digital I/O, they may be used to implement a variety of counting and timing functions including input captures, output compares, and pulse accumulation. In addition, the QED Board provides an optional software UART that supports a secondary RS232 serial port (called serial2) using pins 3 and 4 of PORTA.

PORTD

PORTD is a 6 bit port that is typically dedicated to serial I/O (PD0 and PD1) and to the Serial Peripheral Interface (PD2-PD5). The SPI is a fast synchronous serial link which is used to communicate with the optional onboard 12 bit analog to digital converter (A/D12) and 8 bit digital to analog converter (DAC). The SPI is turned on by executing InitSPI() or InitAD12andDAC() and is turned off by executing SPIOff(). The SPI is initially off after a reset or restart. If you have ordered a custom board with no 12 bit A/D or DAC, you may use PD2-PD5 as 4 general purpose digital I/O bits whose data direction is set by register DDRD and whose contents are accessible at register PORTD.

PORTE

PORTE provides 8 input lines. They may be used as the analog inputs to the 68HC11's built-in 8 bit A/D converter, or they may be used as general purpose digital inputs if the 8 bit A/D converter is turned off. The AD8On() function turns the 8 bit A/D on, and AD8Off() turns it off. The 8 bit A/D is initially off after a reset or restart.

PPA

PPA has no alternate functions and is available as a general purpose digital input port or output port.

PPB

Three bits of PPB are dedicated to controlling three of the open-drain high-current drivers. The remaining 5 lines are available for your use.

PPC

The upper 4 bits of PPC are available as digital input or output if RS485 communications are not being used. If RS485 communications are used, bit 4 of PPC (that is, the lowest bit in the upper nibble of PPC) controls the direction of the RS485 transceiver, and the upper half of PPC must be configured as an output (see the Glossary entry for **`InitRS485`**).

Using the Digital I/O Ports on the 68HC11 Chip

This section describes how to configure and access the PORTA and PORTE digital ports in the 68HC11 chip on the QED Board.

As you work through the examples in the remaining sections of the chapter, you can use a voltmeter to verify that the outputs are behaving as you expect. You can also connect the input signals through a 1 kOhm resistor to +5V or GND to verify that you can digitally read their values. (The 1 kOhm resistor is just a safety precaution to minimize the chance that you'll "blow out" a port bit by mistakenly connecting an output bit to a supply voltage; even if you make this mistake, the resistor would limit the current to safe levels.)

Digital inputs and outputs are very useful in data acquisition, monitoring, instrumentation and control applications. A low voltage (near 0 Volts) is established on a digital output pin when the processor writes a logical 0 to the corresponding bit in a data register associated with the digital output port. A high voltage (near 5 Volts) is established on the digital output pin when the processor writes a 1 to a corresponding bit in the port's data register. This allows software to control external events and processes. For example, an application program can use digital outputs to activate solenoids and turn switches and lights on and off, or to interface the QVGA Controller with a wide variety of digital accessories such as D/A converters, displays, etc.

A digital input allows the processor to monitor the logical state of an external voltage by reading a data register associated with the port. External voltages near 0 Volts connected to a digital input cause the corresponding bit in the port's data register to be read as a logical 0, and external voltages near 5 Volts connected to a digital input are read as a logical 1. Application programs can use digital inputs to read switches and keypads or to interface to digital devices such as A/D converters and real-time clocks.

Using digital I/O is very easy:

1. Configure the direction of the digital I/O port. This is accomplished by writing to a named data direction port (in the case of **`PORTA`** and **`PORTD`**) to set the directions of individual bits within a port, or by executing an initialization routine such as **`INIT.PIA`**.

2. To change the state of an output port, write to the port's data register (whose address is left on the stack by simply stating the name of the port) using **`C!`**, **`SET.BITS`**, **`CLEAR.BITS`**, or other pre-defined operators.

3. To read the state of an input port, read the port's data register with a C@ command; the result is left on the data stack.

The names of the data and direction registers and all required initialization routines are pre-defined in the QED-Forth kernel so you don't have to hassle with hexadecimal register addresses in your code. The following sections describe the available digital I/O ports and how to use them.

QED-Forth Provides Named Registers and Pre-coded Configuration Routines

The ports are addressed in common memory. The 68HC11 ports are associated with data and direction registers in the processor's 96 byte block of "Control and Status Registers" located at 8000H-805FH; Appendix B summarizes the contents of all of these registers. The PIA ports are associated with data registers addressed at 8080H-8082H and a control register at address 8083H.

QED-Forth names the digital I/O ports, and when the name is executed the 32 bit extended address of the port's data register is left on the stack. This makes it easy to access the port; simply state the

port's name and use the standard byte fetch and store operations **C@** and **C!** to read or write to the port. Individual bits in the digital ports can also be modified with operators such as **SET.BITS**, **CLEAR.BITS**, **TOGGLE.BITS**, and **CHANGE.BITS**.

The names of the digital I/O ports and the respective hexadecimal addresses left on the stack are as follows:

```
PORTA ( -- 8000\0 )
PORTD ( -- 8008\0 )
PORTE ( -- 800A\0 )
PPA   ( -- 8080\0 )
PPB   ( -- 8081\0 )
PPC   ( -- 8082\0 )
```

QED-Forth also makes it easy to configure the data direction (input or output) of the I/O ports. The directions of the individual bits in PORTA and PORTD are controlled by direction registers which are named in QED-Forth. The direction register names and the respective hexadecimal addresses left on the stack are as follows:

```
PORTA.DIRECTION   ( -- 8001\0 )
PORTD.DIRECTION   ( -- 8009\0 )
```

Writing a 1 to a bit in the data direction register sets the corresponding port bit to an output, and writing a 0 configures the bit as an input. The commands **C!**, **SET.BITS**, or **CLEAR.BITS** can be used to modify the contents of the data direction registers. Any combination of input and output bits may be specified for these ports.

The data direction of the PIA ports are set by the routine **INIT.PIA** which is described later in this chapter and in the glossary.

PORTE on the 68HC11 can be configured as an 8 channel 8 bit analog to digital converter by executing **A/D8.ON**, and it reverts to its default condition as an 8 channel digital input port after execution of **A/D8.OFF**.

Setting the Data Direction of PORTA and PORTD

Two named registers control the direction of the bits in **PORTA** and **PORTD**, respectively:

```
PORTA.DIRECTION
PORTD.DIRECTION
```

Writing a 1 to a bit in the direction register sets the corresponding port bit as an output, and writing a 0 to a bit in the direction register sets the corresponding port bit as an input. A one-to-one correspondence exists between bits in the data direction register and its corresponding port. These two ports are configurable on a bit-by-bit basis, so any combination of inputs and outputs can be specified.

For example, to set **PORTA** as all input, execute

```
00  PORTA.DIRECTION C!
```

To set the lower 4 bits of **PORTA** as input and the upper 4 bits as outputs, execute

```
HEX F0 PORTA.DIRECTION C!
```

To set the least significant bit of **PORTA** as an input while leaving the direction of all other bits unchanged, execute

```
01 PORTA.DIRECTION CLEAR.BITS
```

which clears the least significant bit in **PORTA.DIRECTION** to 0.

The direction of **PORTD** is controlled in the same way. Recall that **PORTD** is a 6 bit port, and the two least significant bits are used by the primary serial channel. This leaves the four bits PD2 through PD5 available for digital I/O if they are not used for the SPI. For example, to set the four available bits PD2 through PD5 to all outputs, execute

```
HEX 3C PORTD.DIRECTION C!
```

The command

```
HEX FF PORTD.DIRECTION C!
```

has the exact same effect; the two least significant bits in **PORTD** are not affected by the **PORTD.DIRECTION** register, and the two most significant bits in **PORTD** do not exist.

Configuring PORTE as a Digital Input Port

PORTE is always an input port. After each reset and restart, it is configured as an 8 channel digital input port. Executing

```
A/D8.ON
```

turns on the 8 bit analog converter and configures **PORTE** as an 8 channel 8 bit analog input port (see Chapter 6). Executing

```
A/D8.OFF
```

turns off the 8 bit A/D and configures **PORTE** as an 8 channel digital input port.

(For experts and the curious: **A/D8.OFF** turns the 8 bit analog converter off by clearing the A/D power up bit named **ADPU** in the **OPTION** register; **A/D8.ON** sets the ADPU bit; see MC68HC11F1 Technical Data Manual, p.6-4.)

For Experts: Fast Port Accesses

The following comments may help those who need maximum speed when accessing a port from within a Forth definition.

Because all of the named digital I/O ports are located in common memory, the fast page-less operator **(C@)** can be used to access the ports. For example, the command

```
PORTE DROP (C@)   ( -- byte )
```

returns the same result as the command **PORTE C@**. **(C@)** executes more rapidly than **C@** because it does not change pages during the read operation. But this time savings is mostly offset by having

PORTE place the full extended address including page on the data stack at runtime, and then calling **DROP** to remove the page from the stack. A more efficient method is to instruct the compiler to place only the 16 bit address on the stack at runtime, and then call **(C@)** to read the contents. The following definition shows how this can be accomplished:

```
: READ.PORTE   ( -- )
   [ PORTE DROP ] LITERAL (C@)            \ this is a very fast fetch
   CR ." The contents of PORTE = " .      \ display the result
;
```

The **[** is an immediate word that invokes the execution mode. **PORTE DROP** places the 16 bit address of the port on the data stack, and **]** re-enters the compilation mode. **LITERAL** removes the 16 bit address of the port from the data stack and compiles it as a literal that will be placed on the stack at runtime so that **(C@)** may fetch its contents. The rest of the definition prints the result. This same technique may be used to read, modify, or write to any location in common memory. A wide variety of fast page-less operators are available in the kernel, including **(@)**, **(!)**, **(F@)**, **(F!)**, **(SET.BITS)**, **(CLEAR.BITS)**, **(TOGGLE.BITS)**, and **(CHANGE.BITS)**.

Of course, the fastest way to access the contents of a port in common memory is to use assembly code. The following routine leaves the contents of **PORTE** on the data stack:

```
CODE  FETCH.PORTE.CONTENTS ( -- byte )
   PORTE DROP EXT LDAB  \ B gets contents of portE
   CLRA                 \ zero upper byte of double accumulator
   DEY DEY              \ make room on data stack
   0 IND,Y STD          \ put result on data stack
   RTS
END.CODE
```

Because all of the named port addresses are located in the common memory, it is safe to **DROP** the page and use an assembly coded "load" operation such as **LDAB** to fetch the contents of the port. Note that when assembly coding accesses to locations that are not in common memory, it is best to call the pre-coded memory access routines in the QED-Forth kernel (such as **C@**) which properly handle the page changes.

Port Initialization

The PORTA bits PA0 through PA7 are configured as inputs after a reset or restart, unless the serial2 port is specified as the default startup port (see the glossary entry for SERIAL2.AT.STARTUP). If the secondary serial port is automatically initialized at startup, then PA4 is initialized as the serial output and PA3 is configured as the serial input. The remaining PORTA pins are configured as digital inputs after the reset or restart.

PORTD bits PD2 through PD5 are configured as digital inputs after a reset or restart. PORTE bits PE0 through PE7 are configured as digital inputs after a reset or restart; the default state of the 8 bit A/D converter is OFF.

Summary of Port Access

This chapter describes how to configure and access the 68HC11 digital I/O ports A, D and E and the PIA ports PPA, PPB, and PPC. To use the digital I/O ports, follow these three simple steps.

1. Configure the direction of the digital I/O port.
   - To configure PORTA, write to the PORTA.DIRECTION register using C! or a bit manipulation routine such as SET.BITS or CLEAR.BITS. Writing a 1 to a bit position in PORTA.DIRECTION configures the corresponding port bit as an output, and writing a 0 to a bit position configures the corresponding bit as an input. PORTA is configurable on a bit-by-bit basis.
   - To configure PORTD, write to the PORTD.DIRECTION register. PORTD is a 6 bit port, and the two least significant bits are used by the primary serial channel. This leaves the four bits PD2 through PD5 available for digital I/O if they are not used for the SPI. The available PORTD pins are configurable on a bit-by-bit basis.
   - To configure PORTE for analog input, execute A/D8.ON. To configure PORTE for digital input, execute A/D8.OFF. PORTE is configured as a digital input after a reset or restart.
   - To configure the PIA, place two flags on the stack and execute INIT.PIA. The first flag specifies the direction of PPA, and the second flag (top flag on the stack) specifies the direction of the upper nibble of PPC. A true flag specifies output and a FALSE flag specifies input. INIT.PIA configures PPB as an output and lower PPC as an input to ensure compatibility with the keypad/display drivers.
2. To change the state of an output port, write to the port's data register (whose address is left on the stack by simply stating the name of the port) using C!, SET.BITS, CLEAR.BITS, or other pre-defined operators.
3. To read the state of an input port, read the port's data register with a C@ command; the result is left on the data stack.

Using the PIA

This section describes how to configure and access the available I/O ports of the Peripheral Interface Adapter (PIA) on the QED Board.

PIA Initialization

The QED-Forth word

```
INIT.PIA ( flag1\flag2 -- | flag1 = ppa.output?, flag2 = upper.ppc.output?)
```

writes to the peripheral interface adapter (PIA) configuration register to set the data direction for PPA and the upper 4 bits of PPC. If flag1 is true, **`INIT.PIA`** configures PPA as output, and if flag 1 is false, it configures PPA as input. Likewise, if flag2 is true, **`INIT.PIA`** configures upper PPC as output, and if flag 2 is false, it configures upper PPC as input. **`INIT.PIA`** sets the direction of PPB as output and lower PPC as input to ensure compatibility with the built-in keypad and display interfaces. It clears bit 6 of PPB and sets bit 7 of PPB high so that the display.enable and 12 bit A/D chip select signals are inactive. If the specified input/output configuration of the PIA is the same as the prior configuration, **`INIT.PIA`** does not modify the PIA configuration register, and thus does not change the state of any output pins in PPA or upper PPC. If the specified PIA configuration is dif-

ferent than the prior configuration, **INIT.PIA** writes to the PIA's configuration register and this automatically zeros any outputs in PPA or upper PPC. Consult the PIA data sheet in Appendix C for details of the PIA operation.

Warning!

There may be a short transient ON condition on the three high current outputs during power-up and reset. The state of the PIA chip can not be controlled when it is initially turned on or reset. A consequence of this is that in the interval of time between power-up and the operating system's initialization of the output to an OFF condition, there may be a short transient ON. You may need to take appropriate precautions in critical applications.

Additionally, if the specified PIA configuration is different than the prior configuration, `INIT.PIA` **writes to the PIA's configuration register and this automatically zeros any outputs in PPA or upper PPC, even if the direction of PPA or PPC was not changed!**

If the specified input/output configuration of the PIA is the same as the prior configuration, **INIT.PIA** does not modify the PIA configuration register, and thus does not change the state of any output pins in PPA or upper PPC. But, if the specified PIA configuration is different than the prior configuration, **INIT.PIA** writes to the PIA's configuration register and this automatically zeros any outputs in PPA or upper PPC, even if the direction of PPA or PPC was not changed! Consequently, **INIT.PIA** may disrupt in-progress operations involving the 12 bit A/D, keypad, or display. **INIT.PIA** is called by **INIT.A/D12&DAC**, **INIT.DISPLAY**, and **INIT.RS485**. Consult the PIA data sheet for details of the PIA operation.

When designing your application, the safest course is to perform all required initializations in an autostart routine as soon as the application program begins. This avoids the problem of a late initialization that disturbs an in-progress I/O operation.

Upon each reset or restart, the PIA is configured as follows:

Port	Direction
PPA	input
PPB	output
lower PPC	input
upper PPC	input

Accessing PIA Ports PPA, PPB and PPC

The PIA (ports PPA, PPB, and PPC) uses as special set of routines to input or output values. These routines employ *clock stretching* to slow down the access of the on-board PIA to ensure that its timing criteria are met.

PIA Addressing	Standard Memory Addressing
PIA.C!	C!
PIA.C@	C@
PIA.SET.BITS	SET.BITS
PIA.CLEAR.BITS	CLEAR.BITS
PIA.TOGGLE.BITS	TOGGLE.BITS
PIA.CHANGE.BITS	CHANGE.BITS

For example, to set all the bits of PPA to zero you would execute,

```
0  PPA  PIA.C!
```

The PIA.C! routine performs clock stretching (in other words, inserts wait states) during the access to PPA to guarantee that the timing conditions of the PIA are met. It also disables interrupts for 27 cycles (less than 7 microseconds) during the memory access.

Similarly, PIA port bits can be read, set, cleared, toggled, and changed using the routines PIA.C@, PIA.SET.BITS, PIA.CLEAR.BITS, PIA.TOGGLE.BITS, and PIA.CHANGE.BITS, respectively. Full descriptions for these routines are in the attached Glossary.

Characteristics of the PIA's I/O Ports

The 82C55A peripheral interface adapter (PIA) chip is the industry standard part for providing digital I/O ports. This brief summary is intended to clarify some of the features of this chip.

The PIA is configured by writing to a control register at address 8083H. The pre-defined routine **`INIT.PIA`** described earlier writes to this register. It configures the PIA for simple digital I/O ("mode 0") and sets the data direction of ports PPA and upper PPC according to user-supplied flags. **`INIT.PIA`** configures PPB as an output and the lower half of PPC as an input.

Whenever the PIA is configured by writing to the control register, all outputs are set to the logic low state. This is true whether or not the new configuration is different from the prior configuration. The **`INIT.PIA`** routine tries to minimize the impact of this (often undesirable) "feature" by checking the control register before writing to it. If the PIA configuration requested by the programmer is the same as the existing configuration, the PIA's control register is not modified. In general, it is best to use a static configuration for the PIA; dynamically changing the direction of PIA ports while the application is running can cause troublesome glitches on the PIA output pins.

The PIA has another unusual "feature" called "bus hold circuitry". The PIA tries to maintain specified logic levels on pins that are configured as inputs (rather than the standard approach of leaving the input pins in a "floating" high impedance state). After a reset or change in configuration, the PIA holds all inputs high by sourcing between 50 and 400 microamps into the external load attached to the input pin. If your design requires that the inputs settle to a voltage near ground, you will need to pull the pins low with external pull-down resistors that connect each input pin to ground. The resistors should have a value less than 1 K-ohm; the manufacturer suggests that the pull-down

should be 640 ohms to ensure a logical 0 under worst-case conditions. Port PPA also has bus hold circuitry that can hold an input in the low condition. If your design requires that PPA inputs be held at a logical high state, install external pull-up resistors of less than 3 K-ohms from the PPA input pins to the +5 Volt supply.

PIA output pins have good current drive capabilities. The data sheet states that the chip can maintain an output high voltage of at least 3.0 Volts while sourcing up to 2.5 mA. The manufacturer's technical support staff claims that the typical performance is much better; they say that a typical chip can maintain 3.0 Volts or higher while sourcing up to 20 mA. In the worst case the PIA outputs can sink up to 2 mA while maintaining the output voltage below 0.4 Volts. There is no internal current limiting on the outputs, so your custom circuitry should include current-limiting resistors if significant current will be required from the output pins.

Using the High Current Drivers

Four N-channel MOSFET high current drivers are available at the Supplementary I/O connector. Each driver can sink up to 150 mA continuously, or up to 1 amp on an intermittent basis at voltages as great as 60 V. Onboard snubber diodes suppress inductive kickback, so the drivers can be used to control solenoids, motors, and other inductive loads.

Figure 5-1 shows how to connect a DC load (a DC motor is shown) to the MOSFETs.

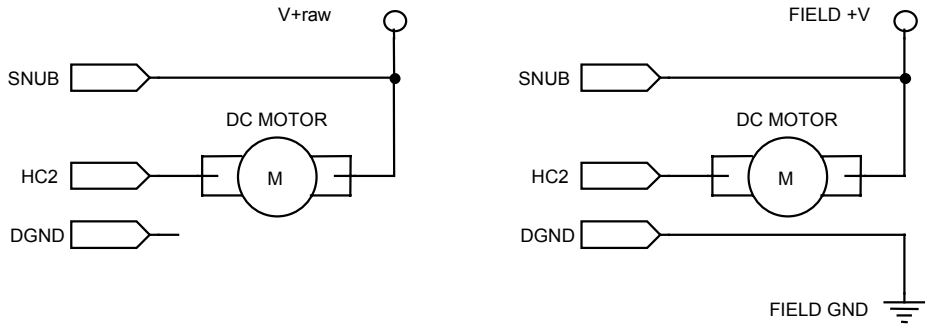

Figure 5-1 Connecting a DC load (for example, a DC motor) to the high current drivers using onboard power (left) and using external power (right).

HC0 is controlled by a PAL signal and HC1 - HC3 are controlled by the PIA port bits PPB5 - PPB7 respectively. NOTE: Upon power-up and reset, the high current MOSFETs, HC1 - HC3, may momentarily sink current until the QED-Forth startup software initializes them. The PAL-controlled HC0 output does not exhibit this transient turn-on behavior at startup.

Two simple functions control these four high current drivers:

```
SET.HIGH.CURRENT
CLEAR.HIGH.CURRENT
```

SET.HIGH.CURRENT accepts a 4-bit mask and turns on the drivers that correspond to "1"s in the mask. **CLEAR.HIGH.CURRENT** accepts a 4-bit mask and turns off the drivers that correspond to "1"s in the mask. Note that "turning on" a driver allows it to sink current, thereby pulling the voltage at

the output pin towards ground. "Turning off" a driver prevents it from sinking current. For example, to turn on all four high current drivers, execute

```
HEX OF SET.HIGH.CURRENT
```

Then, if you execute

```
01  CLEAR.HIGH.CURRENT
```

you will turn off driver number 0 while leaving the state of the other three drivers unchanged.

You can verify the operation of the high current outputs by connecting a pullup resistor (say, 1 kOhm) from the output to +5V and measuring the output voltage with a voltmeter. Note that the output voltage goes LOW when the output is ON (sinking current to ground), and the voltage goes HIGH when the output is OFF (no current flowing in the load resistor).

Using Uninterruptable Operators

The Importance of Uninterruptable Operators

Sometimes it is necessary to set, clear, toggle, or change one or more bits in a port while leaving other bits unaffected. QED-Forth provides convenient read/modify/write routines called **SET.BITS**, **CLEAR.BITS**, **TOGGLE.BITS**, and **CHANGE.BITS** to accomplish these functions. The corresponding fast page-less operators named **(SET.BITS)**, **(CLEAR.BITS)**, **(TOGGLE.BITS)**, and **(CHANGE.BITS)** can also be used to modify the contents of addresses in common memory. The glossary entries provide detailed descriptions of these operations.

For example, if upper PPC has been configured as an output using **INIT.PIA**, the top 4 bits in PPC can be cleared to zeros by executing

```
HEX F0 PPC CLEAR.BITS
```

F0 is a bit mask with the top 4 bits equal to ones; this tells **CLEAR.BITS** that only the top 4 bits should be cleared. The bottom 4 bits of PPC are unaffected.

SET.BITS, **CLEAR.BITS**, **TOGGLE.BITS**, and **CHANGE.BITS** (and the corresponding page-less operators) globally disable interrupts just before reading the memory contents and restore the prior state of the interrupt flag (enabled or disabled) after writing to the specified address. This makes these routines robust with respect to interrupts and timesliced multitasking when two or more concurrently executing routines are modifying bits in the same memory location.

The following scenario illustrates the importance of these uninterruptable operators when more than one task or interrupt routine is writing to a memory location. Let's assume that two different tasks are controlling the bits of the upper nibble of PPC. Assume that **TASK1** is controlling the state of bit 4 in **PPC** (perhaps to set the direction of the RS485 transceiver), and **TASK2** controls bit 7 in **PPC**. Let's assume that bit 4 is low when **TASK2** tries to execute the following sequence:

```
HEX
PPC C@  80 OR
PPC C!
```

TASK2 is merely trying to set the top bit in PPC to 1, but this sequence of commands may have unintended consequences. Assume that the timeslicer interrupt is serviced just after the OR instruction and transfers control to **TASK1**. **TASK1** may change the state of bit 4 to a 1. When control is then transferred back to TASK2, the remaining command **PPC C!** is executed. Unfortunately, this **C!** command erroneously sets bit 4 back to the low state! **TASK2** was interrupted after it read the state of **PPC** but before it had a chance to write the new contents, so it undoes the change that **TASK1** made in the state of **PPC** bit 4.

The uninterruptable read/modify/write routines avoid this problem by disabling interrupts for ten to sixteen cycles (5 to 8 microseconds at an 8 MHz crystal speed). This prevents the corruption of the contents when different tasks or interrupts share access to a single location.

Similar problems can arise when one task writes to a floating point or other 4-byte variable, and a second task needs to read the saved value. The data that is read may be invalid if the read or the write is interrupted between the time of the writing/reading of the first 16 bits and the writing/reading of the second 16 bits. For this reason a set of uninterruptable operators denoted by the **|** ("bar") character are in the kernel. These are **|2@|**, **|F@|**, **|X@|**, **|2!|**, **|F!|**, and **|X!|**. Consult the "Multitasking" chapter in the Software Manual for a more detailed discussion of this topic.

Connecting Hardware to the Digital Outputs

On the QVGA Controller the processor's ports A and D are available for you to connect to external devices. You can use them to directly drive LEDs, relays, transistors, opto-isolators or other digital logic devices. But please be careful -- whenever these outputs are connected to external devices you must stay within the voltage and current capabilities of the output pins. Because the MC68HC11 reference manuals don't specify the electrical capability of these ports very well we provide some additional information here.

Electrical Characteristics of the 68HC11's I/O Ports

The electrical characteristics of the 68HC11F1's digital I/O signals are specified in detail in section 13.4 of the MC68HC11F1 Technical Data Manual. This table lists the "DC Electrical Characteristics" of the processor.

Pins on the 68HC11 configured as digital inputs report a logical "high" if the input voltage is greater than 0.7 times the supply voltage, or 3.5 Volts. They report a logical "low" if the input voltage is less than 0.2 times the supply voltage, or 1.0 Volt. Input voltages between 1.0 and 3.5 Volts may be read as either high or low.

Pins on the 68HC11 configured as digital outputs in the "high" state can maintain the output voltage within 0.8 Volts of the positive supply if 0.8 mA or less is being drawn from the pin. If less than 10 microamps is being drawn, the output high voltage is within 0.1 Volt of the positive supply. In the low state, the digital outputs can keep the voltage below 0.4 Volts while sinking up to 1.6 mA of current. Load circuitry that requires significant current to be sourced or sunk by the digital output

should include external resistors to ensure that the absolute maximum rating of 25 mA per output pin is never exceeded.

Protecting the Input and Output Pins

These output pins are very useful because they can directly drive a wide range of devices. Nevertheless, any circuitry connected to the processor should take care to:

- Prevent excessive voltage levels at the pin; and,
- Prevent excessively great currents.

We'll address each of these concerns in turn.

Preventing Excessive Voltages

Excessive voltages are prevented by ensuring that voltages of less than a diode drop below V_{SS} (-0.6 V) or greater than a diode drop above V_{DD} (V_{DD}+0.6 V) are never applied to the processor. For some applications, particularly when driving inductive loads such as relays, you may need to provide Schottkey diode clamps between the pin and V_{DD} and between the pin and ground. All pins on the processor have inherent diode clamps to the processor's ground voltage, V_{SS}, but it is best not to rely on these; if there is the possibility of the output pin being driven to a negative voltage level it is better to prevent excessive power dissipation in the processor package by externally clamping the voltage to ground (V_{SS}) with a Schottkey diode. Processor ports A and D also have inherent diode clamps to the chip's +5V supply voltage, V_{DD}, but it is likewise better not to rely on these; instead external Schottkey clamps to V_{DD} should be used.

Preventing Excessive Currents

The current into or out of any pin on the MC68HC11 should also be limited to prevent damage. The specified maximum current is 25 mA into or out of any one pin at a time, although these pins can typically withstand transients of more than 100 mA at room temperature. In driving more than one pin at a time it is necessary only to stay within the processor's maximum power dissipation. Unfortunately, Motorola doesn't specify what this maximum is, but we recommend that you don't exceed a total of 100 mW for all processor I/O pins combined. The chip's total power dissipation is the sum of its internal power (which varies from device to device so much that it can only be determined by actually measuring it, but which is specified at less than 150 mW) and the power associated with the I/O pins. Pin currents must be limited using external resistors.

Output Pin V-I Characteristics

The output pins of the MC68HC11 microcontroller are similar in electrical characteristics to the SN54/74HC digital logic family. They are capable of sourcing or sinking up to 25 mA per output. They are guaranteed to source up to 0.8 mA while providing a valid logic high and to sink up to 1.6 mA at a valid logic low, although they generally do much better as shown in the following figures. A valid logic high level is between $V_{OH} = V_{DD}$ and $V_{OH} = V_{DD}$ - 0.8 V, and a valid low level is between $V_{OL} = V_{SS}$ = 0 V and $V_{OL} = V_{SS}$ + 0.4 V. As the output sources or sinks current, the V_{OL} and V_{OH} levels rise or fall respectively. It is often useful to know just how much to expect the V_{OL} and V_{OH} levels to degrade with current. For currents of less than 10 mA the voltage change is linear

with current; that is, the output can be modeled as a voltage source of either zero or five volts and an equivalent series resistor of about 40 ohms. At greater output currents the equivalent series resistance increases until at the greatest specified current for any one pin, 25 ma., the equivalent series resistance is 60 ohms. At this current the voltage degradation of the V_{OL} or V_{OH} is 1.5 volts. Figure 5-2 and Figure 5-3 illustrate this variation.

These figures can be used when choosing component values for particular drive circuits. For example suppose we wish to use a pin of Port A or D to drive a light-emitting diode. On circuit configuration would be to place the LED in series with a resistor and connect them between an output pin and ground. The resistor functions as a current limiter, and the luminous intensity of the LED depends on the amount of forward current. From its data sheet we note that its forward voltage at a current of 10 mA is specified to be 2.2 V. What should the resistor value be? We can calculate it as,

Eqn. 5-1 $\mathbf{R = (V_{OH} - 2.2\ V\ ) / 10\ mA}$

Consulting the V_{OH} vs I curve for the output pin we find that at 10 mA V_{OH} = 4.4 V. We therefore need a resistance of 220 ohms.

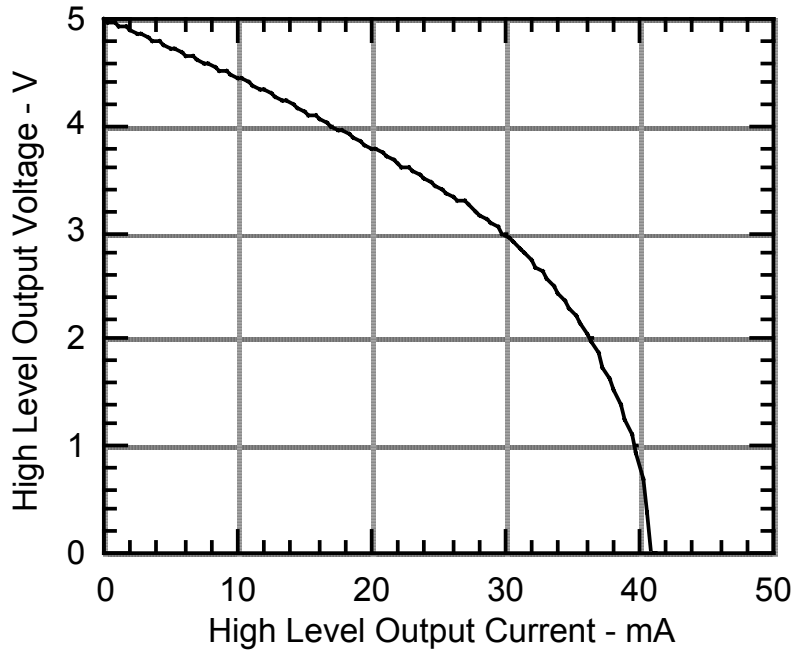

Figure 5-2 Degradation of the Port A or Port D output high voltage with current. The maximum current allowed for continuous operation is 25 ma.

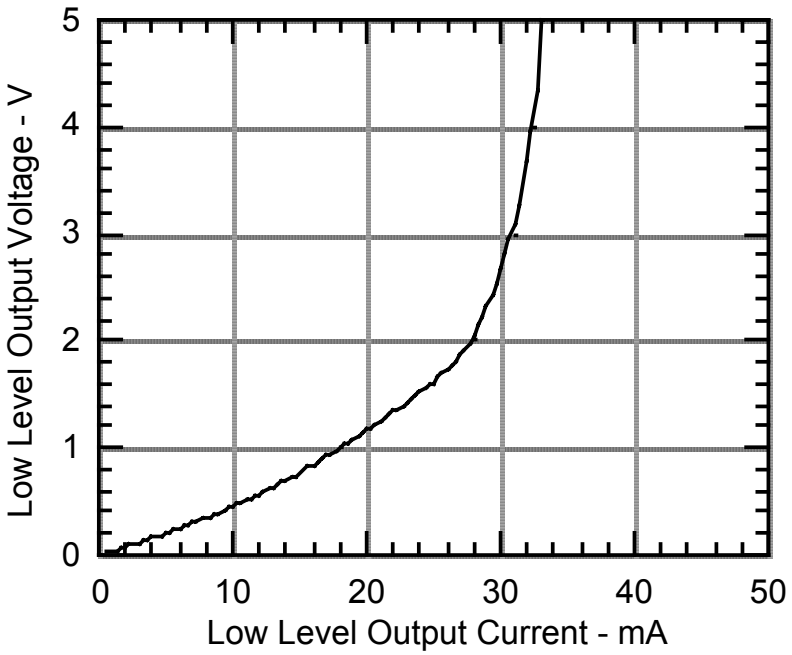

Figure 5-3 Degradation of the Port A or Port D output low voltage with current. The maximum current allowed for continuous operation is 25 ma.

Chapter 6

Chapter 6: Data Acquisition Using Analog to Digital Conversion

This chapter shows how easy it is to use the 24 channels of analog input/output on the QVGA Controller. The QVGA Controller includes a high performance 8- channel 12-bit analog to digital (A/D) converter. The 12-bit A/D can be configured to convert 8 channels of single-ended inputs or 4 channels of differential signal inputs, or combinations of single-ended and differential signals. The input voltages can be unipolar, with a range from 0 to 5 volts, or bi-polar, with a nominal range from -5 to +5 volts. This voltage range can be adjusted using low and high reference voltages. The demonstration code for this chapter in the ANALOGIO.C file also shows how to use C arrays and FORTH_ARRAYS as buffers, and describes how to store semi-permanent information in the QVGA Controller's Electrically Erasable PROM (EEPROM)..

Data Acquisition Using the QVGA Controller

Many instrument applications require monitoring of analog signals and generation of analog control voltages. Analog to digital (A/D) converters and digital to analog converters (DACs) can perform these functions. An A/D converter samples analog signals and converts them to digital values that can be stored, processed, or displayed. A DAC generates an analog voltage whose value is proportional to a specified digital number.

The resolution of an A/D or DAC is specified in bits. For example, an *8-bit* DAC can output any of 2^8 or 256 distinct analog levels; 256 is the maximum number of levels representable by an 8 bit binary number. Similarly, a *12-bit* A/D converts an analog signal into one of 4096 discrete digital numbers.

The QVGA Controller hosts both analog to digital and digital to analog converters to address a wide variety of instrumentation and control applications. It includes three 8-channel analog I/O devices:

- An 8-channel 8-bit analog to digital (A/D) converter that is built into the 68HC11 processor chip converts unipolar signals with a nominal 0 to +5 volt range;
- An 8-channel 12-bit A/D converter; and,
- An 8-channel 8-bit digital to analog converter (DAC), discussed in the next Chapter.

The 16 analog inputs and 8 analog outputs are brought out to a 40 pin analog connector (H2) on the QED-Flash Board (the smaller of the two boards of the QVGA Controller); the connector diagrams in an Appendix specify the pin assignments.

The 8 bit A/D Converter

The 8 bit A/D converter in the processor converts unipolar signals with a nominal 0 to +5 volt range, and conversion results are returned in registers in the 68HC11. The analog inputs are connected to the PORTE pins on the processor; these can be used as digital inputs if the 8 bit A/D is disabled.

The 12 bit A/D Converter

The 12 bit A/D converter is interfaced to the processor via the fast serial peripheral interface (SPI). Its chip select is a memory-mapped signal generated by the onboard logic. The 12 bit A/D can be configured to convert 8 channels of single ended inputs or 4 channels of differential signal inputs, or combinations of single-ended and differential signals. The input voltages can be unipolar, with a range from 0 to 5 volts, or bipolar, with a nominal range from -5 to +5 volts. This voltage range can be adjusted via the low and high reference voltages named VRL and VRH.

Using the 8-bit A/D

Initializing the 8 Bit A/D

The 8 bit A/D is inside the 68HC11 processor chip, and is accessed via **PORTE**. This 8 bit port can be used as either an 8 channel A/D or as an octal digital input port. When the QED Board is first powered up, the 8 bit A/D converter is off and **PORTE** is configured as a digital input port. To turn on the A/D converter, a program must call the function:

```
A/D8.ON
```

Another function named **A/D8.OFF** is available to turn off the 8 bit A/D so that PORTE once again acts as a digital input port.

In an autostarting application you should include **A/D8.ON** in your start-up code.

Hardware Connections

To sample an analog voltage, attach a voltage with a value between zero and +5 Volts to the 8 bit A/D channel 0 input named **AN0**. For example, try connecting the 1.5Vref signal (pin 37 on the Analog I/O connector) to the **AN0** input (pin 10 on the Analog I/O connector). See tbe Appendix for connector diagrams.

Using the 12-bit A/D

Initializing the 12 Bit A/D and DAC

The 12-bit A/D is interfaced to the 68HC11F1 via the high speed serial peripheral interface (SPI). The function:

```
INIT.A/D12&DAC
```

initializes the SPI to be compatible with operation of the 12-bit A/D and the digital to analog converter (DAC).

Specifying the Type of Conversion

The function

```
A/D12.SAMPLE   ( flag\channel# -- result )
```

expects a configuration flag and a channel number as inputs, performs a single conversion using the 12 bit A/D, and returns the conversion result. The meaning of the flag is as follows:

Flag value	Type of conversion
-1	single ended, unipolar
0	differential, unipolar
1	single ended, bipolar
2	differential, bipolar

To perform a single-ended, unipolar conversion on channel 0 and print the result to your terminal, execute,

```
-1  0  A/D12.SAMPLE  .←
```

Additional QED-Forth routines such as **A/D12.MULTIPLE** facilitate speedy multiple conversions; see their glossary entries for more details. In sum, the 12 bit A/D converter is an easy to use peripheral that greatly enhances the analog input capabilities of the QVGA Controller.

Hardware Connections

To sample an analog voltage, attach a ground-referenced voltage in the range between zero and +5 Volts to the 12 bit A/D channel 0 input named **12AN0**. For example, try connecting the 1.5Vref signal (pin 37 on the Analog I/O connector) to the **12AN0** input (pin 18 on the Analog I/O connector). See the Appendix for connector diagrams, and see Figure 1-3 in Chapter 1 to identify the Analog I/O connector on the QVGA Controller.

If the input voltage equals VRL (the low reference voltage, typically at 0 Volts), the result will equal 0. If the input is within 1 bit of VRH (the high reference voltage, typically at 5.0 Volts), the result will equal decimal 4095. If the input is exactly half of (VRH - VRL), the result will equal decimal 2048. If the input is 1.5 volts, the result should be approximately 1230 counts.

To provide a default 0 to 5.0 Volt conversion range, the VRL and VRH references are tied to AGND and +5VAN through 1 kohm resistors; while this is fine for casual testing as we're doing now, you'll want to make the following connections for your actual design project:

To achieve full 12 bit accuracy, you must connect the low reference VRL signal and the high reference VRH signal to low-impedance supplies such as Analog Ground (AGND) and Analog +5V

(+5VAN), respectively. This is accomplished by connecting VRL (pin 1) to AGND (pin 19), and connecting VRH (pin 2) to +5VAN (pin 20) on the Analog I/O connector of the QED Board.

Multiple 12 Bit A/D Conversions

These Forth drivers control multiple conversions by the 8-bit and 12-bit A/D converters:

```
(A/D8.MULTIPLE)      ( xaddr\u1\#samples\channel# -- )
A/D8.MULTIPLE        ( xaddr\u1\#samples\channel# -- )
(A/D12.MULTIPLE)     ( xaddr\u1\#samples\flag\channel# -- )
A/D12.MULTIPLE       ( xaddr\u1\#samples\flag\channel# -- )
```

The stack pictures include an unsigned parameter "u1" that controls the interval between successive samples. The revised routines and their stack pictures are as follows:

The xaddr specifies the starting address where the multiple samples are to be stored. The #samples and channel# parameters have obvious meanings (note that # is pronounced "number"). The "u1" parameter sets the sampling interval; setting u=0 delivers the fastest sampling rate and u1 = 65,535 delivers the slowest sampling rate. The sampling rate equals the base sampling rate as specified below plus 2.5 microseconds times "u1".

Each routine has a latency (the time between calling the routine and taking the first sample) and a sampling period (the time between successive samples). The times vary depending on whether the storage xaddr is in common memory (above 8000 hex) or in paged memory. The following table summarizes the relevant times in microseconds:

Routine Name	Xaddr in Common Memory		Xaddr in Paged Memory	
	Latency (µsec)	Sampling Period (µsec)	Latency (µsec)	Sampling Period (µsec)
(A/D8.MULTIPLE)	16	10 + 2.5*u	11	32.5 + 2.5*u1
A/D8.MULTIPLE	86	10 + 2.5*u	81	32.5 + 2.5*u1
(A/D12.MULTIPLE)	58	27.5 + 2.5*u	68	50 + 2.5*u1
A/D12.MULTIPLE	128	27.5 + 2.5*u	138	50 + 2.5*u1

Of course, the operation of interrupt routines including the timeslicer may increase the specified sampling times. If the system can be structured so that no interrupts are active during sampling, these routines offer very precise control over the sampling frequency of the A/D converters.

In addition, the "flag" in the stack picture of the routines

```
A/D12.SAMPLE       ( flag\channel# -- result )
(A/D12.SAMPLE)     ( flag\channel# -- result )
```

```
A/D12.MULTIPLE     ( xaddr\u1\#samples\flag\channel# -- )
(A/D12.MULTIPLE)   ( xaddr\u1\#samples\flag\channel# -- )
```

has the following meaning:

Flag value	Type of conversion
-1	single ended, unipolar
0	differential, unipolar
1	single ended, bipolar
2	differential, bipolar

The latter two flag values instruct the software to interpret the input voltage as a bipolar signal that can swing both above and below ground.

Bipolar Conversions

You can instruct the 12-bit A/D to interpret the input as a unipolar signal in the range 0-5 volts, or a bipolar signal in the range -5 to +5 volts. Even without an external negative supply, voltages as low as -4.0 or -4.5 V can be converted. This is achieved by using a negative voltage generator on the Maxim RS-232 chip. This negative voltage is connected to the V- supply input of the 12 bit A/D chip, and is also accessible at pin 39 on the Analog I/O connector, labeled "Analog Bus V-". Drawing current out of this "Analog Bus V-" pin to power external circuitry is not recommended, as the RS-232 chip cannot source much current, the output impedance is hundreds of ohms, and the voltage is poorly regulated. Instead, to achieve clean rail-to-rail -5.0 V to +5.0 V conversions, connect an external -5 V supply to "Analog Bus V-" at pin 39 on the Analog I/O connector.

The drivers for the 12-bit A/D converter are:

```
(A/D12.SAMPLE)     A/D12.SAMPLE
(A/D12.MULTIPLE)   A/D12.MULTIPLE
```

The configuration flag passed to these routines lets you select unipolar or bipolar conversion as discussed above, in addition to single-ended (ground-referenced) or differential conversions.

Chapter 7

Chapter 7: Outputting Voltages with Digital to Analog Conversion

The onboard 8 channel 8 bit digital to analog converter (DAC) is interfaced to the processor via the fast serial peripheral interface (SPI), and its chip select is a memory-mapped signal generated by the onboard logic. The octal DAC allows the precise output of analog voltages ranging from 0V to 3V. Each of the 8 DACs have an "input" voltage (named Vinx where x is the DAC channel number) that can be connected to a stable voltage reference or a voltage to be multiplied, for example, a time-varying input representing an audio signal. If the input is connected to the steady 1.5 Volt reference which is available on the QED Board, the output of each channel ranges from 0 to 3 Volts in increments of 11.7 mV per digital count. Alternatively, to implement digital volume control you could connect a time-varying sound signal to the input of a DAC, and connect the DAC output to a speaker. Then the volume would be proportional to the digital code that you write to the DAC.

The DAC inputs and outputs are located on the Analog I/O connector; see the appropriate Appendix for the connector pinouts.

Initializing the DAC

Like the 12 bit A/D, the 8 bit DAC is interfaced to the 68HC11F1 via the high speed serial peripheral interface (SPI). The DAC is initialized by the same function that initializes the 12-bit A/D converter:

```
INIT.A/D12&DAC
```

INIT.A/D12&DAC sets the eight DAC outputs to zero. You can call this function from within your code, or simply type,

```
INIT.A/D12&DAC↵
```

at your terminal. After you execute this command the DAC is ready to communicate with the 68HC11 via the SPI.

Applying a Reference Voltage

To use the DAC, you'll first need to apply a valid input/reference voltage. To create a standard voltage output DAC with the maximum 0 to 3 volt output range, you can use the 1.5Vref voltage

provided on the QVGA Controller to drive the Vin of each of the DAC channels that you want to use. For example, let's connect 1.5Vref (pin 37 on the Analog I/O connector) to Vin1 (pin 21 on the Analog I/O connector).

This 1.5 volt reference may be used to provide a stable reference voltage for any or all of the digital-to-analog converters on the QVGA Controller, and can also provide a reference voltage for external circuitry.

Outputting a Voltage

Setting the 8-channel 8-bit DAC is very easy. To output a voltage, simply call the function:

```
>DAC ( byte\n -- | byte=data, n=channel# )
```

which expects a count in the range 0 to 255, and a channel in the range 1 to 8. After calling the >DAC (pronounced "to DAC") function the multiplying DAC then outputs a voltage that is proportional to the specified count and to the analog input voltage. The analog input voltage must be between 0 and 1.5 volts, and the output voltage equals:

Eqn. 7-1 Vout = 2 * Vin * count / 256

Using the 1.5V reference, we can interactively type from the terminal:

```
DECIMAL 255 1 >DAC ←
```

to create the full-scale output of the DAC which should be just under 3 volts. The output voltage appears at Vout1 on pin 22 of the Analog I/O connector. To output a voltage of one half of the full scale range execute

```
DECIMAL 128 1 >DAC ←
```

Since 128 is half of the full 8 bit digital range, a voltage of 1.5 Volts (one half of the full 3 volt analog range) appears at the output of DAC1. You can connect a digital voltmeter to the DAC1 output (pin 22 on the Analog I/O connector) to verify that the DAC is working as expected.

The other DACs may be controlled in a similar manner. Each must have an input voltage reference that establishes that DAC's mid-range value.

Getting Greater Resolution

The hardware manual also describes how to combine two DACs to construct a high-resolution 12-bit DAC; this technique is implemented for you on the QED Analog Conditioning Board which is available from Mosaic Industries.

The 8 channel 8 bit DAC brings to the QED Board a very exciting capability. The DACs are easy to use and flexible. Moreover, pairs of DACs can be combined to yield over 11 bits of resolution, as provided on the QED Analog Conditioning Board. The "Analog to Digital and Digital to Analog Conversion" Chapter in the QED Hardware Manual provides additional insight into the breadth of capability of the digital to analog converter.

DAC Execution Speed

The DAC driver **(>DAC)** executes in approximately 46 microseconds, and the multitasking version **>DAC** executes in 131 microseconds. The QVGA Controller maps the chip enable of the onboard D/A converter to page 14 (0x0E); the page addressing is responsible for appx. 6 microseconds of the execution time.

Chapter 8

Chapter 8: Serial Communications

RS-232 and RS-485 Communications

The QVGA Controller has two serial communications ports: the primary serial port called Serial1 and the secondary serial port called Serial2. The Serial1 port is implemented with the 68HC11's on-chip hardware UART (*Universal Asynchronous Receiver/Transmitter*). Serial2 is implemented by a software UART in the controller's QED-Forth Kernel that uses two of the processor's PortA I/O pins to generate a serial communications channel.

A UART translates the bit-by-bit data on the serial cable into bytes of data that can be interpreted by the QED-Forth Kernel or by your application program.

The secondary channel is very useful for debugging application programs that communicate with other computers or I/O via the primary channel. Since both channels can operate simultaneously and independently, debugging can be performed while the application program is communicating via its primary channel. The dual communications channels also provide an easy way to link systems that communicate using different serial protocols.

Serial Connectors

The primary and secondary serial communications ports are accessible through the QVGA Controller's 10 pin, dual row Communications Connector (H14) and through the individual Serial1 and Serial2 connectors (H7 and H11 in Figure 1-3). These standard 9-pin serial connectors are located on the top edge of the QVGA Controller. A serial communications cable is supplied with all QVGA Starter Kits.

Serial Protocols

There are several *protocols* that govern the format of exchanged data, with the RS232 protocol used primarily by personal computers, and the RS485 protocol used in industrial control systems.

RS232

RS232 is by far the most common protocol. It is supported by virtually all personal computers, and is the default protocol for both of the QVGA Controller's serial ports. RS232 allows both commu-

nicating parties to transmit and receive data at the same time; this is referred to as *full duplex* communications. The QVGA Controller offers a unique addition to full duplex RS232: it can place its RS232 transmitter into a high impedance *silent* mode under software control. This allows you to configure *full duplex multi-drop networks* in which a single master can sequentially address one of many slaves and start a full-duplex exchange of data.

RS485

RS485 is another protocol supported by the primary serial port on the QVGA Controller. It is a *half duplex* protocol, meaning that only one party at a time may transmit data. Unlike the standard RS232 protocol, RS485 allows many communicating parties to share the same 3-wire communications cable. Thus RS485 is the standard protocol of choice when *multi-drop* communications are required.

Using the Serial Ports

Using the primary serial port is easy. In fact, you have been using it all along as you worked through the examples in this document. All high level routines call the following low level revectorable serial primitives to access the currently active serial port:

```
EMIT ( char -- )     \ outputs the specified char to the serial port
KEY  ( -- char )     \ waits for and returns the next input char
```

Because all of the serial I/O routines on the QED Board are revectorable, it is very easy to change the serial port in use without modifying any high level code.

Switching the Default Serial Port

The secondary serial port is implemented by a software UART that controls two pins on PortA. Pin 3 of PortA is the Serial2 input, and pin 4 of PortA is the Serial2 output. To switch to the secondary serial port running at 1200 baud, simply flip DIP switch #5 to the ON position to enable the secondary serial port hardware, and type from the terminal the following QED-Forth commands:

```
DECIMAL ↵
1200  BAUD2↵
USE.SERIAL2↵
```

You can operate the port at any baud rate up to 4800 baud; just specify the rate you want before the **BAUD2** command. Now select the "Communications" item in the "Settings" menu of the Terminal program, and click on 1200 baud (or whatever baud rate you selected in the command above). Move the serial cable from the "Serial Port 1" connector to the "Serial Port 2" connector at the QVGA Controller. Typing a carriage return at the terminal should now produce the familiar "**ok**" response via the Serial2 port.

For those of you interested in the details, here's how it works: The low-level serial driver routines named KEY and EMIT are revectorable routines that can be redirected to use either of the serial ports. By interactively executing the QED-Forth function

```
USE.SERIAL2↵
```

before calling main, we revectored these serial primitives to use the Serial2 port.

To return to using the primary serial port, simply type from the active terminal the QED-Forth command:

```
USE.SERIAL1←
```

which transfers control back to serial port 1 running at the prior established baud rate (typically 9600 baud). A hardware reset (toggling DIP switch #7) has the same effect. If you do this now, remember to move the QVGA Controller's serial connector back to Serial Port 1, and to change the terminal's baud rate back to 9600 baud using the "Communications" item under the terminal's "Settings" menu.

If you always want the QED Board to start up using the secondary serial port as the default serial communications link, you can type at your terminal:

```
1200  SERIAL2.AT.STARTUP←
```

where 1200 is the baud rate that you choose; you can specify any standard baud rate up to 4800 baud. The complementary routine is:

```
SERIAL1.AT.STARTUP
```

which makes the primary serial port the default startup serial link. We recommend that you keep the faster Serial1 port as the default serial link as you work through the exercises in this book.

In summary, the code provided for implementing the second serial port is very flexible and can be used to support dual concurrent communications ports. Data translation between different machines can be performed with ease, and applications that communicate via the primary serial port can be debugged using the secondary channel.

Timing Considerations and Multitasking

In multitasking systems using both serial ports Serial1 and Serial2, the application code should include one of the commands

```
RELEASE.ALWAYS  SERIAL.ACCESS !
RELEASE.NEVER   SERIAL.ACCESS !
```

before building the tasks. This prevents contention that can occur if the default **RELEASE.AFTER.LINE** option is installed in the **SERIAL.ACCESS** user variable.

The primary serial port, Serial1, is supported by the 68HC11's on-chip hardware UART, and does not require interrupts to work properly. On the other hand, the secondary serial port (Serial2) is implemented using hardware pins PA3 (input) and PA4 (output), and is controlled by the associated interrupts IC4/OC5 and OC4, respectively. The QVGA Controller's kernel software contains a complete set of high level driver routines for the Serial2 port, and these functions are summarized in the QED-Forth Glossary.

The maximum Serial2 communications rate is 4800 baud. Because the software UART is interrupt based, competing interrupts that prevent timely servicing of the Serial2 interrupts can cause communications errors on the secondary serial channel. For example, at 4800 baud (bits per second), each bit lasts about 200 microseconds (µs), and if communications are full duplex (e.g., if the QVGA Controller echoes each incoming character), then there is a serial interrupt every 100 µs or

so. In the middle of a character, each interrupt service routine takes about 35 µs. At the end of a received character, the service routine takes about 45 µs. At the start of a transmitted character, the service routine takes about 65 µs. Thus, as a rough approximation, operating at 4800 baud full duplex requires about 40 to 50% of the 6811's CPU time (that is, an average of approximately 40 to 50 µs service time every 100 µs).

If you are running Serial2 at 4800 baud, the rest of your application must be able to function properly using the remaining portion of the CPU time. Moreover, if Serial2 is running full duplex at 4800 baud, any other interrupt service routine that takes longer than 100 µs is likely to cause a problem. If an interrupt service routine takes longer than 200 µs, then an entire serial bit will be missed, causing a communications error. Also, several non-serial interrupts can *stack up*; if they have higher priority than the serial interrupts, they will be serviced before the Serial2 interrupt routine, and again a serial input or output bit may be lost.

Routines that temporarily disable interrupts for significant periods of time can also interfere with the Serial2 port. The QED-Forth Glossary contains a list of functions that temporarily disable interrupts, and the glossary entries give further information regarding how long interrupts are disabled. In most cases the times are less than 25 µs which does not pose a problem. However, note that the READ.WATCH and SET.WATCH functions disable interrupts for about one millisecond (msec.), and the functions that write to EEPROM disable interrupts for 20 msec. per programmed byte. Be sure to account for these effects when designing your application.

We have built sophisticated instruments using the QVGA Controller that operate very reliably using multiple interrupts in addition to the software UART. If your application requires use of the secondary serial port as well as other interrupt routines, the key is to keep the interrupt service routines short and fast. You might also consider operating the secondary serial port at a lower baud rate to relax the timing constraints.

Setting Baud Rates

The rate of data transmission is expressed in bits per second, or *baud*. The primary serial channel can operate at standard speeds up to 19200 baud and can be configured for either RS232 (the default) or RS485 operation. The Serial2 channel is always configured for RS232 communications, and can sustain baud rates up to 4800 baud.

The routines

```
SERIAL1.AT.STARTUP
SERIAL2.AT.STARTUP
```

make it easy to establish a standard baud rate at which the board will communicate each time it starts up. Although the maximum standard baud rate of the primary serial port is 19200 baud, non-standard baud rates of over 80 Kbaud can be attained by the 68HC11's on-chip UART and the on-board RS232 driver. The maximum sustainable baud rate on the secondary serial port is 4800 baud.

While the default baud rate of the primary serial port is 9600 baud, you can speed your communications and download times appreciably by switching to a faster baud rate. It's very simple. Just type at your terminal:

```
DECIMAL←
19200 BAUD1.AT.STARTUP←
```

The new baud rate takes effect upon the next reset or restart. Thus you can type the command

```
WARM←
```

or reset the board using DIP switch number 7, or turn the system off and on again. In each case, your QVGA Controller will be communicating at 19200 baud, and this baud rate will remain in effect until another **BAUD1.AT.STARTUP** command is executed, or until you invoke the *Special Cleanup Mode* as described earlier.

To reconfigure QED-Term to communicate at 19200 baud, select the "Communications..." item under the "Settings" menu and click on 19200 in the "Baud Rate" section. If you want to use the fast 19200 baud rate as the standard from now on, you can save the new terminal settings by using the Terminal's "Save..." or "Save as..." selection under the "File" menu to save the settings as a **TERMINAL.TRM** file.

Multi-Drop Communications Using RS-232

The RS-232 driver (Maxim Part No. MAX242) can be *tri-stated* under software control. This allows standard point-to-point full duplex communications, as well as a multi-drop configuration with one *master* (a single QVGA Controller or a desktop computer) and multiple QVGA Controller *slaves*. To implement this multi-drop scheme, each slave keeps its RS-232 transmitter *silent* until it is addressed by the master and is given permission to transmit. The advantage of such a multi-drop RS-232 network is that the communications are *full duplex*, with each communicating party capable of simultaneous transmission and reception of data. The software routines

```
RS232.TRANSMIT
RS232.SILENT
```

control the dual RS232 transmitters on the board.

Connecting a standard full duplex link RS232 between two computers is the same as with a standard RS232 link, with the TxD (transmitter output) of each computer connected to the RxD (receiver input) of the other computer. A serial cable accomplishs this connection.

In a multi-drop configuration, the TxD of the master is connected to the RxD of each slave, and the RxD of the master is connected to the TxD of each slave. Figure 8-1 shows a typical multi-drop communications network. The QVGA Controller makes it easy to configure a multi-drop network of QVGA Controllers controlled by a single master desktop computer.

To properly operate the network each slave computer executes **RS232.SILENT** at startup; thus all of the slave transmitters remain silent individually addressed by the master. When a slave is addressed, it executes **RS232.TRANSMIT** at which point full duplex (two-way) communications commences between the master and the selected slave. When the communication is complete the slave silences its transmitter so that the master can talk to a different slave on the network.

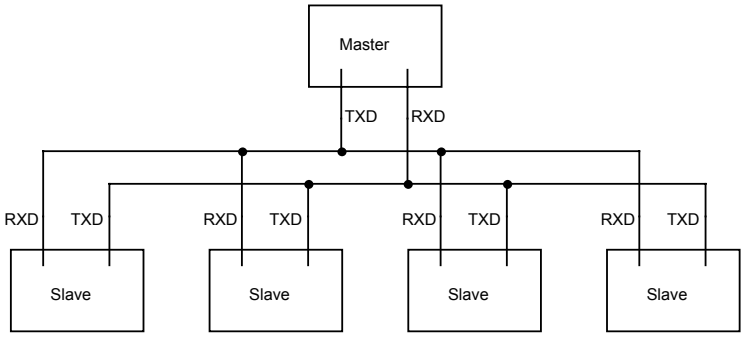

Figure 8-1 Typical RS-232 multi-drop communications network.

Chapter 9

Chapter 9: The Battery-Backed Real-Time Clock

The battery backed real time clock is set and read using the functions **READ.WATCH** and **SET.WATCH**.

Interrupts are disabled during the entire operation of the functions **READ.WATCH** and **SET.WATCH**, so that these two routines are fully re-entrant in QED-Forth multitasking applications.

With a 16 MHz crystal on the QED Board, **READ.WATCH** disables interrupts for approximately 0.8 milliseconds (msec), and **SET.WATCH** disables interrupts for approximately 1.0 msec. All of the input and output parameters of **SET.WATCH** and **READ.WATCH** are passed on the Forth data stack. Because they are re-entrant they they can be simultaneously called from multiple tasks in a single QED-Forth application without producing any conflicts.