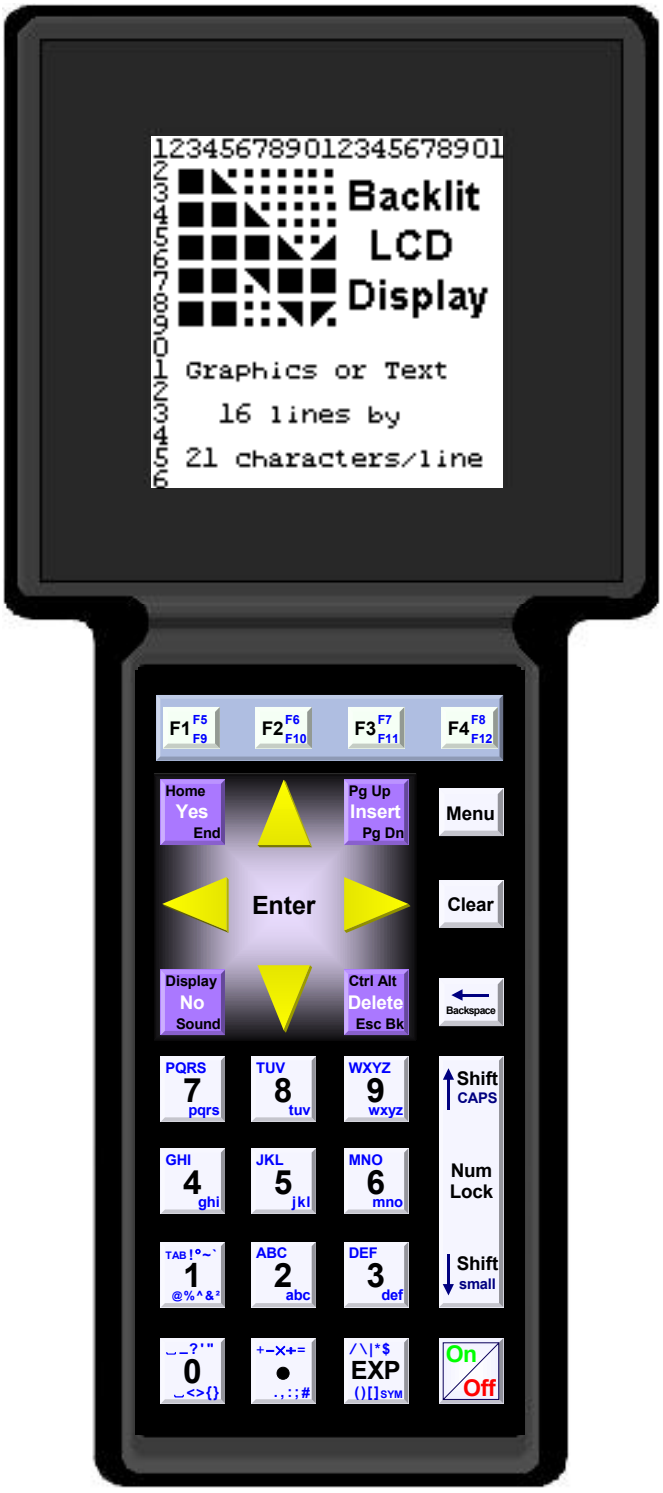


The C Programmer's Guide to the Mosaic Handheld

The C Programmer's Guide to the Mosaic Handheld
by Mosaic Industries, Inc.

Copyright © 2004 Mosaic Industries, Inc. All rights reserved.
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):

March 2004: Draft v0.1

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.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and Mosaic Industries, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. For example, Microsoft, Microsoft Windows, and Visual Basic are registered trademarks of Microsoft Corporation.

While every precaution has been taken in the preparation of this manual, Mosaic assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

The C Programmer's Guide to the Mosaic Handheld

How To Use This Book viii

Getting to Know Your Handheld Instrument 1

Introducing the Mosaic Handheld 1

Programming the Handheld 3
Choice of Programming Languages 4

Hardware Functionality 5
Graphical User Interface: Keypad, LCD Display, Backlight 5
Battery Power for Portability 5
68HC11 Processor 6
Memory and Mass Memory 7
Measurement and Control 7
Communications 8
Real-Time Clock 8
Personality Board and Extensible I/O 8

Getting to Know Your Hardware 9
A Tour of the Processor Board 11
A Tour of the Personality Board 14
A Tour of the Power Board 15

Powering Your Handheld 17

Controlling Power 17
Automatic Turn-On 17
Manual Turn On and Off 17
Programmable Shut-Down 17
Continuously ON 18

Using Battery Power 18
Replacing the Batteries 18
Charging the Batteries 18
Battery Lifetime 19

Tethered Operation 20

Using the Power Adapter 20

Power Metering 20

Your First Program 23

Installing the Mosaic IDE and Control-C Compiler 23

Turning on Your Handheld 24

Using the Mosaic IDE 26
Using the Editor and Compiler 26

Your First C Program 28
Compiling a Program 28
Downloading and Running the Program 28
Interactively Debugging Your Program 30
An Introduction to Extended Memory 33
An Introduction to Multitasking 36
Summary 41

The IDE: Writing, Compiling, Downloading and Debugging Programs 43

Writing Programs 43
Using the Editor/Compiler 43
Sylistic Conventions 44
Accessing the Standard (Kernel) Library Functions 46
Initializing Variables 47

Compiling Programs 47
Compiling Multiple Source Code Files 47

Using the Interactive Debugger 48
Overview of the Forth Language and Programming Environment 48
Displaying the Values of Static Variables 51
Use Type Keywords To Interactively Call C Functions 53
Displaying the Values of FORTH_ARRAY Elements 54
Assigning Values to Static Variables and FORTH_ARRAY Elements 55
Under the Hood of the QED-Forth Interactive Debugger 56
Other Useful QED-Forth Functions 58

The Handheld Kernel vs. Prior Kernels 59

Making Effective Use of Memory 61

The Handheld's Memory Map 61
Addressing Memory in C 67

Software Development Using Flash Memory 67

Locating Nonvolatile Data in EEPROM 69

Using C Arrays and Forth (Kernel) Arrays 70
Storing Data Acquisition Results in C Arrays and Forth Arrays 70

Real Time Programming 77

The Timeslicer and Task Switching 77

Using Interrupt Service Routines (ISRs) 78
Interrupt Recognition and Servicing 78
External Hardware Interrupts /IRQ and /XIRQ 84
Routines that Temporarily Disable Interrupts 85
Writing Interrupt Service Routines 85
An Example: Periodically Calling a Specified Function 88
Calling Kernel Functions From Within ISRs 91

Failure and Run-Time Error Recovery 93

Getting Started and Getting Stopped – Restarts and Resets 93
Resets versus Restarts 94

The COP Watchdog Timer and Clock Monitor 96
The Clock Monitor 98

Processor Operating Modes 98
Low Power Modes 98
Operating Modes of the 68HC11F1 CPU 99

Programming the Graphical User Interface 101

The Structure Of The GUI Toolkit 101
A Closer Look At Objects 101
The GUI Toolkit Objects 102

Building Your Application 103
Designing Your User Interface 103
Transferring Your Images to the Handheld Controller 106
Coding Your Application 107
Handling Errors 115
Expanding the GUI Toolkit's Objects and Methods 116

GUI Objects in Detail 116
GUI_TOOLKIT 116
GUI_DISPLAY 118
GUI Screens 119
GUI Keypads 121
GUI_PEN 123
GUI_BUZZER 123
GUI_BATTERY 124
GUI_TIMER 125
Graphic Object 126
Font Object 127
Textbox Object 128
Controls 130
Action Key Object 130
Data Entry Key Object 131
Shift Key Object 132
Plot Object 132
Screen Object 134

Customizing the Keypad Overlay 136

Digital and Timer-Controlled I/O 139

Overview of Available Digital I/O 139
Using the Digital I/O Ports on the 68HC11 Chip 141
Using Uninterruptable Operators 143

Connecting Hardware to the Digital Outputs 144

The Processor's Output Compare Functions 147
Pulse and PWM Generation Techniques 149

Data Acquisition Using Analog to Digital Conversion 155

Data Acquisition Using the Handheld 155
Examining the Demonstration Program 156

Fundamentals of Analog to Digital Conversion 156
Determining the Resolution of an A/D Converter 156
Converting an A/D Count into Its Equivalent Voltage Reading 157

Using the 8-bit A/D 157

Serial Communications 161

RS-232 and RS-485 Communications 161
Serial Protocols 162
Serial Connectors and Configuration Options 162
Using the Serial Ports 164
Setting Baud Rates 168
Multi-Drop Communications Using RS-485 168

Synchronous Serial Peripheral Interface (SPI) 169
SPI Bus Pins 169
SPI Network Connections 170
Configuring the SPI 171
Summary 175

The Battery-Backed Real-Time Clock 177

Customizing Your Handheld Instrument 181

Using the Personality Board to Customize I/O 181

Routing I/O Into the Instrument 182

Mounting Additional Components 183

Appendix A: GUI Function Reference 197

Categorized List of All Objects, Properties, & Methods 197
Object Types (constants used by New_Object) 197
Errors (constants returned by Read_Error) 197
Properties Of Objects (constants used by Set_Property & Get_Property) 197
Pre-Instantiated Object Reference Constants 199
Battery Constants 199
Keypad Event Constants (contained in the KEY_EVENT and PRIOR_KEY_EVENT property) 199

Keypad Type Constants (contained in the KEY_TYPE and PRIOR_KEY_TYPE property) 199
Keypad Shift State Constants (contained in the SHIFT_STATE property) 199
GUI Methods 199

Glossary 200

How To Use This Book

Welcome, and thanks for purchasing the Mosaic Handheld™. This manual provides instructions for using your new handheld embedded computer platform. The Handheld packs a keypad and graphics/character display, programmable computer, up to 1 megabyte of memory, communications, analog and digital I/O, battery pack, and an expansion I/O bus into a rugged handheld enclosure. This platform is perfect for prototyping and manufacturing handheld instruments and dramatically cuts the cost of data acquisition and control from a handheld user interface. It is ideal for machine automation, industrial control, robotics, handheld data acquisition, and scientific instrumentation.

The I/O-rich Handheld hosts a 16 MHz Motorola 68HC11F1 microprocessor, 512K Flash (expandable to 1 MB) and 128K Battery Backed RAM (expandable to 512K), and 320 bytes of EEPROM. On-board I/O includes 8 digital I/O lines with counter/timer capabilities, 8 analog inputs, a fast synchronous SPI serial interface, and dual RS232/485 ports. A real-time clock tracks the calendar and time of day and battery-backs the 128K RAM.

A convenient prototyping board allows you to integrate application-specific circuitry including sensors and actuators. The Handheld has plenty of room for your own circuitry and up to 4 I/O expansion modules called WildCards™ that you can mix and match depending on your application, allowing you to have virtually any kind of I/O in a convenient handheld form factor.

WildCards implement a wide variety of communications, data acquisition and control capabilities. Available WildCards include octal 12-bit D/A and 16-bit A/D converters, a 24-bit resolution analog data acquisition subsystem, a Compact Flash card mass memory interface, fast buffered RS232/485 dual UARTs, high voltage/high current isolated I/O, and AC or DC solid state relays. You can select the WildCards that meet your needs to configure a cost-effective customized controller for your application.

The Handheld also includes a nickel-metal-hydride battery pack and self-contained charger, with plenty of power for several days of intermittent use. The battery pack may be charged overnight or while the instrument is in use.

The Handheld is easily programmable in C, Forth or Assembly using any PC. Built-in programming tools include an interactive debugger, a multitasking executive, and comprehensive device-driver libraries.

Prerequisite Knowledge

The Mosaic Handheld 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 Mosaic Handheld 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 Mosaic Handheld in either. This manual is geared to the C programmer. If your would rather program in Forth, give us a call and we'll send you the Forth programmers manual. We recommend the following references for novice programmers:

- *The C Programming Language*, by Kernighan and Ritchie
- *C: A Reference Manual*, by Harbison and Steel

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 Mosaic Handheld,* will familiarize you with the Mosaic Handheld (Chapters 1 and 2) and its programming environment, and get you writing your first program (Chapter 3). These first two chapters guide you through the Mosaic Handheld's hardware, explain how to establish communications with it, tell you how to install your compiler, and show you how to compile and run your first program.

After working through the examples of Chapter 3 you will have exercised some of the key hardware and software features of your instrument. You might then leaf through the categorized list at the beginning of the C Function Glossary to get a feel for the wealth of precoded library functions available for you to use.

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

Part 3, *Communications, Measurement and Control*, focuses on the Handheld's hardware resources – A/D, 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, pin-outs, and schematics.

Conventions Used in This Book

The following conventions are use throughout this manual:

Abbreviations	
A/D	Analog to Digital Converter
CCFL	Cold Cathode Fluorescent, a display backlight that uses a small fluorescent light bulb
CF	Compact Flash, a nonvolatile memory with fast random access for reading and slower block writing.
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 Mosaic Handheld'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 Mosaic Handheld through a terminal	`CFA.FOR main PRIORITY.AUTOSTART↵`
Mosaic Handheld 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 representing the enter key on the keyboard, and the Mosaic Handheld'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 equivalents of decimal 0, 65,535 and 15,604 are represented as `0x0000`, `0xFFFF`, and `0x3CF4` respectively.

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.

Chapter 1

Getting to Know Your Handheld Instrument

Congratulations on your choice of the Mosaic Handheld™, a customizable handheld computing, data acquisition and communications platform that is ideal for machine automation, industrial control, robotics, data acquisition, and scientific instrumentation. This Chapter introduces the various software and hardware features of the Handheld: the enclosure, battery management, keypad and display, real time operating system, choice of programming languages, processor, memory, serial communications, real-time clock, and I/O.

In this chapter you'll learn:

- ⇨ *All about the operating system and software features of the Mosaic Handheld;*
- ⇨ *How to use the hardware features of your instrument; and,*
- ⇨ *How to configure various options.*

Introducing the Mosaic Handheld

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, expandable I/O, and enclosure. You'll find the Mosaic Handheld has a set of hardware and software that work together to simplify your product development cycle while bringing new capability to your portable products.

The Handheld packs a programmable computer, plenty of memory, communications, analog and digital I/O, battery power, 128x128 pixel graphic display, 32 button keypad, and an expansion I/O bus into a rugged handheld enclosure. The ABS plastic enclosure is small enough to fit comfortably in your hand but large enough to hold custom circuitry, sensors, and actuators.

Figure 1-1 diagrams the Handheld, showing its primary components. In addition to its keypad and graphics/text display it contains:

- A Processor Board with serial communications, and analog and digital I/O;
- A Personality Board that allows you to filter and route I/O signals and add custom circuitry or components; and,
- A Power Board containing high efficiency switcher power supplies, battery charger, and battery measurement circuitry.

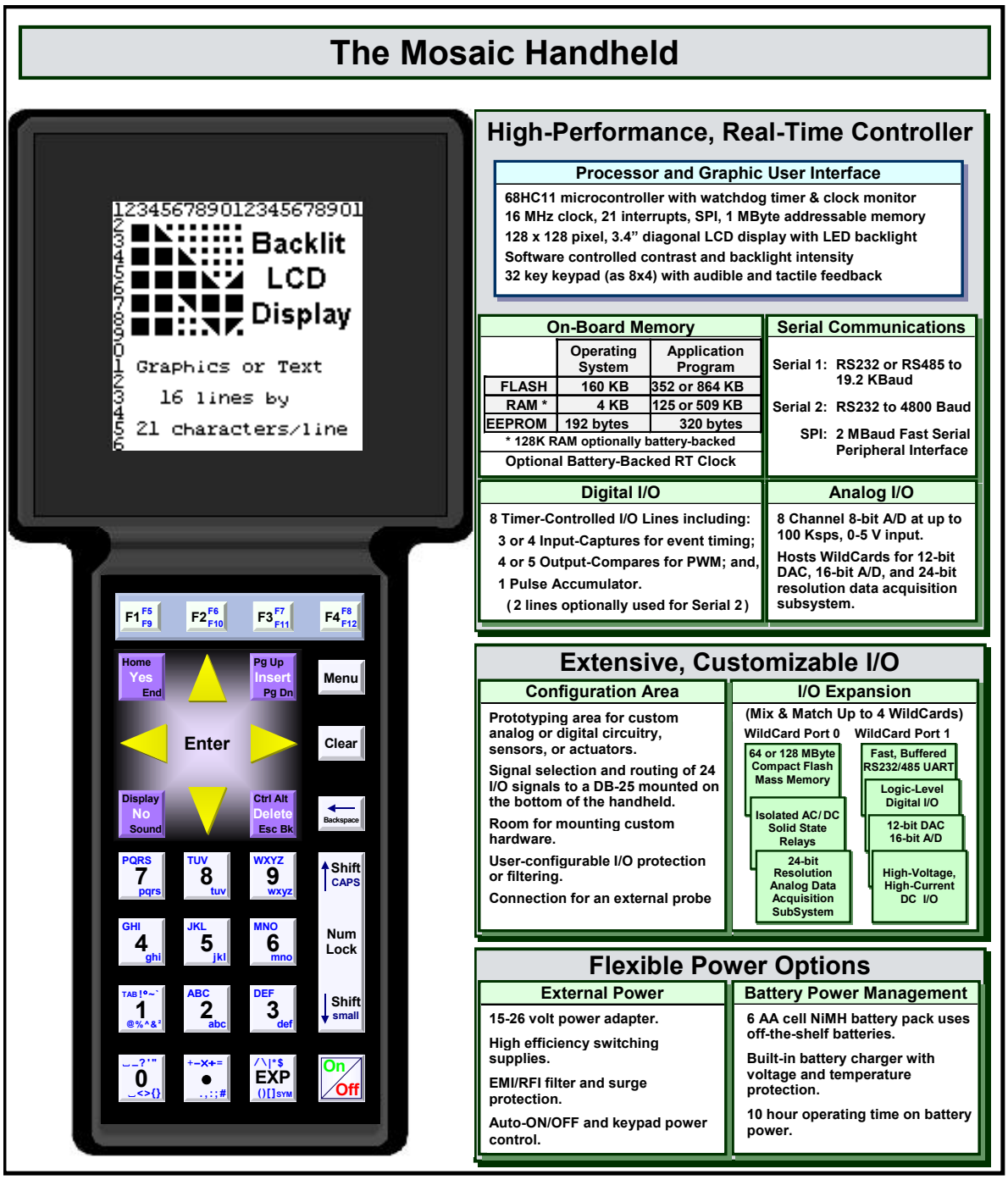


Figure 1-1 The Handheld comprises a graphics/text display, keypad, processor board (or motherboard) with native I/O and communications, personality board for customization and I/O expansion, and power board with built-in battery charger.

The Processor Board hosts a 16 MHz Motorola 68HC11F1 microprocessor, 512K Flash and 128K RAM (expandable to 1 MB Flash and 512K RAM), and 320 bytes of EEPROM. On-board I/O includes 8 digital I/O lines with counter/timer capabilities, 8 analog inputs, a fast synchronous SPI

serial interface, and dual RS232/485 ports. An optional real-time clock tracks the calendar and time of day and battery backs the 128K RAM.

The processor is easily programmable in C, Forth or Assembly using any PC. Built-in programming tools include an interactive debugger, a multitasking executive, comprehensive device-driver libraries, and a toolkit to simplify displaying graphics and creating your user interface.

The Personality Board allows you to filter and protect the I/O lines and to route them to a customizable DB25 connector on the bottom of the enclosure. The Personality Board also hosts up to 4 I/O expansion modules called WildCards™ that you can mix and match depending on your application. The easy to use WildCards stack onto 24-pin WildCard connectors on the Personality PCB. WildCards implement a wide variety of communications, data acquisition and control capabilities. Available WildCards include an octal 12-bit D/A and 16-bit A/D converters, a 24-bit resolution analog data acquisition subsystem, Compact Flash card mass memory interface, fast buffered RS232/485 dual UART, high voltage/high current isolated I/O, and AC or DC solid state relays. You can select the WildCards that meet your needs to configure a cost-effective customized instrument for your application.

In addition to powering the Handheld, the Power Board contains battery management circuitry for six rechargeable 2100mAH nickel-metal hydride (NiMH) batteries. They can supply power for up to 10 hours and recharge in only two hours.

The following subsections discuss the interdependent hardware and software aspects of your instrument.

Programming the Handheld

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 Mosaic Handheld 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 Mosaic Handheld using the Control-C programming language, and how to use the built-in functions. Function glossaries provide an in-depth description of every routine. If you prefer to program in the QED-Forth programming language full function glossaries are available for that language too. The Mosaic Handheld'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 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 to run independently in different tasks while accomplishing their duties in a timely fashion.

Choice of Programming Languages

You can program the Mosaic Handheld 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 Mosaic Handheld 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, serial ports, memory manager, multitasker, and much more.

Using the Windows™ environment on your PC, you can edit your C program in the supplied Mosaic IDE, 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 Mosaic IDE (or you can use any other editor you prefer) to write your code and download the source code directly to the handheld 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.

Hardware Functionality

The block diagram of Figure 1-1 provides a cogent summary of the hardware capabilities of the Mosaic Handheld. Each of the hardware modules shown is described in the following sections.

Graphical User Interface: Keypad, LCD Display, Backlight

The Handheld features a keypad and LCD display for showing text and/or graphic images. Combining a high-contrast 3.4" diagonal display and 4 column by 8 row keypad, it comes with software that makes it easy to control your application using menus, graphs, and bitmapped images.

You can display your own custom graphics. 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. Your application's startup screen executes automatically on power-up.

You can use as many screens as you need, each with software configurable menus. Onboard software draws the screen graphics and responds to button presses for you, so you can focus on your application rather than display maintenance.

The GUI Toolkit is described in a separate document titled *Handheld GUI Toolkit Manual*.

Dedicated ON/OFF Button

The lower right button of the keypad turns the instrument on and off. A press of the button turns the instrument on, and holding it down in excess of three seconds turns it off.

Battery Power for Portability

The Handheld may be powered one of three ways:

- with a power adapter of 15-30 VDC;
- by six rechargeable NiMH 2100mAH batteries; or,
- through the DB25 connector on the bottom of the unit.

When operated on its batteries it lasts over ten hours with the backlight turned off, and six hours with it turned on. If it is fully loaded with four WildCards and the backlight is continuously on, the batteries last four hours. Whenever the power adapter is plugged in the batteries automatically charge. They fast-charge to approximately 80% of full charge within 90 minutes, then trickle charge the remainder over five hours.

The Handheld includes several high-efficiency switching regulators and extensive power line filtering for EMI/RFI isolation. In tethered operation, it may be powered by applying any unregulated or regulated DC input voltage between 15 and 30 volts. In either case, the onboard circuitry is isolated by electromagnetic interference (EMI) suppression circuitry. This improves the reliability of the electronics in harsh industrial environments.

When using an AC adapter, the power jack is located on the right side of the enclosure near the bottom. The Handheld can be powered by any power supply that can deliver 15 to 30 volts at 20 watts. Of this, 17 watts is required for fast charging and all other instrument operations require only 3 watts. Consequently, if batteries are not used only 3 watts is required to power the instrument.

68HC11 Processor

Motorola's 68HC11F1 programmable microcontroller provides the core capabilities of the Handheld. This chip integrates a central processing unit (CPU), communications, analog and digital I/O, timing capabilities, and memory. In this document we will refer to the microcontroller chip as the 68HC11F1, the 68HC11, or simply the HC11.

The processor includes:

- An 8 bit timer-controlled digital I/O port called PORTA. The timer functions include "input captures" that facilitate accurate measurement of pulse widths, "output compares" that make it easy to generate pulse trains and pulse-width modulated waveforms, and a pulse accumulator. These functions are backed up by interrupts that simplify real-time response to external events. Two of the time lines, PA0 and PA1, may be used for detecting keypad button presses if desired, or the keypad can be scanned without using these timer lines.
- An 8 channel input port (called PORTE) on the processor that can be configured to read either analog or digital inputs. Analog inputs are converted to an 8 bit digital number by a built-in analog to digital (A/D) converter.
- A built-in serial communications interface (SCI). The SCI is an asynchronous interface, meaning that there is no clock transmitted with the data. Using the SCI, the Mosaic Handheld can transmit and receive data at standard baud rates to 19200 baud, and at non-standard intra-character rates as high as 250 Kbaud.
- A serial peripheral interface (SPI) capable of data transfer rates up to 2 megabits per second. Many useful devices communicate via such a synchronous (clocked) serial interface. Multiple peripheral devices can communicate on this interface as long as each device has a unique chip select signal.
- A watchdog timer that can be configured to reset the processor if the application program fails to execute properly.
- A clock monitor available to reset the processor if the clock fails.
- External interrupt request pins and a variety of interrupt functions for quick response to real-time events.
- 512 bytes of nonvolatile, electrically erasable PROM (EEPROM). Of this, 192 bytes are used by the operating system and 320 bytes are available for your application. EEPROM provides a convenient way to store calibration constants and other information that must be periodically updated.
- A 16 bit address bus and an 8 bit data bus. While the 68HC11's native address space is 64 Kilobytes (equal to 65,536 bytes, and often written as 64K), the Mosaic Handheld expands the

addressable memory space to 2 Megabytes by effectively adding 5 "page" bits to the address bus. The memory architecture is optimized for fast execution, and the onboard operating system makes page changing transparent to the user. Once you set up your starting memory map, the operating system handles all of the details of accessing memory.

Learning More about the 68HC11

The Mosaic Handheld uses a version of the 68HC11 microcontroller called the 68HC11F1. Motorola's *M68HC11 Reference Manual* and *MC68HC11F1 Technical Data Manual* are included with this documentation package as Adobe Acrobat Portable Document Format (*.pdf) files.

The *M68HC11 Reference Manual* thoroughly describes how to use microcontrollers of the 68HC11 variety, but it does not describe some of the enhanced features that appear in the F1 version. The *MC68HC11F1 Technical Data Manual* provides details of the operation of the F1.

Memory and Mass Memory

The Handheld includes 512K Flash memory, 128K RAM (or 1 MB Flash and 512K RAM with the expanded memory option), 512 bytes of EEPROM and an additional 1K RAM (on the processor).

Of the 512K of Flash memory, 448K is available for your application program and data storage; the remaining 64K is used by the operating system. Of the 129K (513K, with expanded memory option) of RAM, 125K (509K) is available for application program use.

Like PROM, Flash memory is nonvolatile. That is, 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 safe even if the processor "gets lost" and overwrites memory. But Flash memory is also re-programmable, and the Flash programming functions are present right in the Handheld's onboard software library.

Compact Flash

You can also add 64 MB, 128 MB, or more of removable mass memory by plugging in a Compact Flash WildCard module. Pre-coded software supports a standard DOS- and Windows-style "FAT" file system, allowing files to be created on a PC and read by the Handheld, or vice versa. Compact Flash cards are easily inserted into a slot on the side of the Handheld. CF memory is now a commodity, so CF modules are easily obtained at low cost with ever increasing capacity.

For more information, consult the *Compact Flash WildCard User's Manual*.

Measurement and Control

Before adding WildCards, the Mosaic Handheld provides a total of 17 I/O channels, distributed among digital, analog and serial communications functions as shown in Table 1-1.

Table 1-1 I/O available on the Mosaic Handheld

I/O		Type
Digital	6	Timer-controlled inputs or outputs including 3 input-capture, 3 output-compare, and a pulse accumulator. (Two additional lines are available if the second serial port is not needed.)
Analog	8	8-bit, 0-5 V analog inputs at up to 100kHz sampling rate
Serial	1	Serial 1: RS232/485 hardware UART at up to 19.2 KBaud
	1	Serial 2: RS232 software UART at up to 4800 Baud
	1	Synchronous Serial Peripheral Interface at 2 MBaud
	17	**Total I/O channels**

For each of these I/O lines, 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. You may add additional I/O by mounting up to four WildCards within the Handheld enclosure.

Communications

Two serial ports and a fast synchronous serial peripheral interface (SPI) provide plenty of communications capability. A hardware UART with single character buffer drives RS232 or RS485 protocols at up to 19.2 Kbaud (the default serial baud rate), and a software UART provides RS232 at up to 4800 baud. Onboard serial interface chips generate the logic levels necessary to implement either the RS232 or RS485 protocol. Two serial ports allow you to program through one while your instrument can still communicate with a third party through the other.

The RS485 communications port may be optionally terminated with a resistor or RC network. Jumpers on the processor board allow for the insertion of RS485 network bias resistors.

If you need greater speed or more ports, UART Wildcards plug directly into the Handheld's Wildcard module bus, each providing two more full-duplex RS232/422/485 buffered serial communication ports at up to 56 Kbaud.

Real-Time Clock

A battery-backed real time clock (RTC) may be optionally included. If so, the RTC's battery also backs-up the 128K RAM. Note however that if the expanded memory option is chosen, providing 512K of RAM, then the RAM can only be battery-backed if the rechargeable batteries are installed.

Personality Board and Extensible I/O

Need even more I/O? A Personality Board provides at platform for you to add custom off-the-shelf I/O modules or your own dedicated circuitry and components. You can also do point-to-point wiring on the Personality Board to bring any I/O to the Handheld's bottom-mounted DB25 connector or to a probe mounted on the top of the Handheld.

WildCard I/O Modules

The Handheld hosts Mosaic's WildCards™, small I/O modules for sophisticated and dedicated I/O. Up to four WildCards can be stacked directly on the Handheld's Personality Board. Available WildCards include:

- Compact flash mass memory;
- Data Acquisition Subsystem with a 24-bit resolution programmable gain A/D;
- 16-bit A/D and 12-bit D/A;
- Optically isolated AC or DC solid state relays;
- Configurable general-purpose digital I/O;
- Fast, buffered RS232, RS422 or RS485 communications interfaces; or,
- High-voltage, high-current DC inputs and outputs.

Customizing Your Hardware

You may want to bring signals into to your handheld instrument. You can configure the DB-25 connector in the Handheld's base with cross-point wiring on the personality board. In a special area of the personality board you can include filtering or protection circuitry for any of the I/O lines and choose which I/O lines to run to the DB-25 connector, or to a specialized probe connector.

The Handheld's personality board also includes a prototyping area for you to add your own specialized circuitry. It includes mounting holes and power rails with ample room for you to add all kinds of sensors and actuators.

Getting to Know Your Hardware

Handheld Starter Kit

If you purchased a Mosaic Handheld Starter Kit, you should have received the following. If you are missing any of these items, please contact us immediately:

1. A Mosaic Handheld with
   - -MM option – memory expanded to 1 MB Flash and 512K RAM (battery backed by the NiMH battery pack),
   - -RB option – real-time clock, and
   - -BT option – high capacity NiMH batteries;
2. Serial cables including
   - 9-pin DB-9 PC serial cable (Part No. PCC9-232),
   - Dual 9-pin DB-9 to 10-pin IDC cable (Part No. QED-COM-CABLE-9), and,

- 25-pin DB-25M to 9-pin DB-9F serial adapter cable for connection to the Handheld's DB-25F (Part No. HHADAP-9F25M);

3. A 40-pin ribbon cable to connect the Handheld's processor and power boards when the unit is opened (HH-40-CABLE);
4. A 25V, 20W power adapter (Part No.PS-HH-24V);
5. A CD-ROM containing:
   - The Mosaic IDE and the Control-C Compiler, integrated within the Mosaic IDE;
   - Documentation;
   - Program examples; and,
   - Motorola M68HC11 Reference Manual and MC68HC11F1 Technical Data Manual (Part No. MAN-HC11);
6. A User's Guide on how to use the Kernel Extension Manager for additional Wildcard Drivers.

Opening and Closing the Handheld

Opening the Handheld is required to access the serial communications header, install new WildCards, or customize the Personality Board.

Before attempting to open the enclosure, be sure all cables are removed from the enclosure including the power cable and any DB-25 tether, and that the Handheld is *off*. If the Handheld is *on*, holding down the ON/OFF key for longer than three seconds will turn it *off*. To open the enclosure, turn the Handheld over so that the display is face down (but be careful not to scratch it). Remove the six screws that fasten the battery compartment cover using a Phillips screwdriver and remove the battery compartment cover. The battery compartment is located on the bottom of the Handheld on the back of its handle. There are four additional screws located under the battery compartment cover in addition to the four holes on the top of the Handheld. Remove these eight screws with a Phillips screwdriver to open the enclosure.

You can then gently pull apart the two half shells. You'll notice that the processor board, mounted on the top shell just under the keypad, and the power board, mounted on the bottom shell just over the battery compartment, had been connected together with a single high-density connector.

Once open, you can operate the Handheld by connecting a cable between the high-density connectors on the processor and power boards. A 40-pin ribbon cable is supplied for that purpose. When using it, be sure that the marked side of the ribbon cable connects to the pin 1 end of both connectors. You can also connect the 10-pin serial cable to the processor board's serial header, shown in Figure 1-2.

Connecting Cables for Development

Once the enclosure is open, the handheld separates into two pieces. The top shell contains the Processor Board (including CPU, memory, and logic), Personality Board, CF WildCard or other installed WildCards, Display, and Keypad. The bottom shell contains the Power Board and battery compartment. To establish serial communication with the Handheld, connect the DB-9 serial cable to your

PC. Next, connect the Handheld Serial Cable to the DB-9 Serial Cable and the dual row IDC connector to the shrouded 10-pin serial header H4 on the Processor Board. You can also connect you serial line to the external DB25 connector whether the Handheld is open or closed.

Carefully connect the male end of the 40-pin ribbon cable to the female connector on the Processor Board. When inserting the male connector gently insert it dead center. If it is angled too far from vertical it is very easy to permanently damage the connector on the Processor Board. Also, be sure you connect pin 1 of the cable, as indicated by the red stripe, to pin 1 of the header, as indicated by the silk screen on the Processor Board PCB. Be careful! The connector and the header are not shrouded or polarized, and plugging in the cable incorrectly may damage the Processor Board, any connected WildCards, or the Power Board. Next, connect the female end of the 40-pin ribbon cable to the male connector on the Power Board. Finally, plug in the power supply into the power jack (located on the side of the Power Board).

Turning on and Using the Handheld

With the Handheld open, you can still turn it on by pressing the *ON/OFF* button. With your serial cables connected and the PC running the Mosaic Terminal program you should see the QED-Forth "ok" prompt on the PC screen.

Reassembling the Enclosure

To reassemble the Handheld first turn it off and unplug power, then remove the interconnecting cable and serial cable. Carefully fit the two halves together. If the halves are kept parallel as they are fit together the connector between the Processor Board and the Power Board will mate properly. Once the two halves are firmly pressed together you can reinstall the eight screws through the Handheld's back, then replace the battery cover and reinstall its six screws.

Why is there tape on the keypad?

You can customize the graphic used over your keypad. The keypad includes a clear plastic overlay which forms a pocket into which you can place a paper graphic. When shipped from the factory, the keypad is loosely attached with tape to the enclosure; the adhesive on the back of the keypad is not used to securely fasten the keypad to the enclosure. This allows you to pull up the keypad and install your own custom graphic. Once you have finalized your keypad design, you can remove the backing on the keypad (exposing the adhesive) and permanently attach the keypad to the enclosure. If you need help setting up your serial port or the Mosaic Terminal for communications, please refer to Chapter 3, *Your First Program*.

A Tour of the Processor Board

Figure 1-2 shows a photo of the Processor Board, and diagrams the positions of the headers and jumpers.

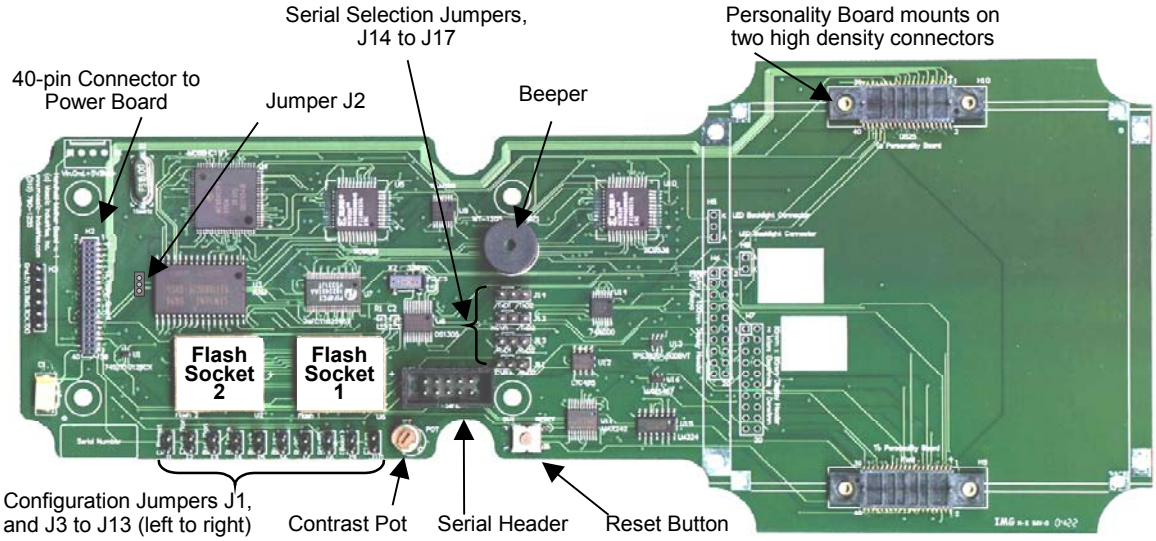


Figure 1-2 Headers and Jumpers on the Handheld's Processor Board.

The Communications Header Hosts Primary and Secondary Serial Ports

The primary serial port, designated Serial 1, can be configured for RS232 communication at standard rates up to 19,200 baud and nonstandard rates to 250,000 baud. The secondary serial port, Serial 2, which can be used for peripheral devices such as a printer or a barcode reader, has a maximum baud rate of 4800 baud. Both serial ports bring out their transmit and receive pins to the processor board's communications header, H4. The primary serial port can be jumper configured for RS485 operation; in that case, the XCVR± pins are used. On the Personality Board the serial 1 lines can also be jumpered to the bottom-mounted DB25 connector.

H4: Communications Header

Signal	Pins	Signal
/TxD1	1 – 2	/RxD1
GND	3 – 4	GND
RS485 XCVR-	5 – 6	RS485 XCVR+
/TxD2	7 – 8	/RxD2
GND	9 – 10	+5V

Piezoelectric Beeper

A piezoelectric beeper mounted on the Processor Board allows audio feedback for software events such as button presses or alarms. Chapter 8, *Using the Keypad and Display*, discusses using the beeper.

Contrast Pot

The contrast potentiometer (pot) adjusts the contrast of the monochrome LCD display and is located next to the Configuration Jumpers. 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. The con-

trast is also software adjustable. Please consult the demo program for an example of programmatically changing the contrast.

Configuring Jumpers on the Processor Board

Table 1-2 summarizes the purpose and default settings of each of the jumpers on the board.

Table 1-2 Processor Board Jumpers

Label	Default Setting	Description
J1 C Protect	OFF	Write protects memory page 0x0C. Useful for "bullet proofing" a kernel extension to facilitate firmware upgrades.
J2 NiMH / Li RAM BB	NiMH	Chooses whether the RAM is battery-backed by the NiMH battery pack or the real-time clock's (RTC) Li battery. If a 512K RAM is installed, NiMH should be chosen because the RAM draws too much current for the Li battery; if a 128K RAM is installed and the RTC is installed, the Li-ion option may be chosen to backup the RAM even when the NiMH battery pack is disconnected.
J3 Chrg Mon	ON	Enables monitoring the battery current through A/D input line PE6. Indicates fast charge, trickle charge, and current consumption while discharging.
J4 Bat Mon	ON	Enables monitoring the battery voltage through A/D input line PE7.
J5 R Term	OFF	If J8 is installed, terminates the RS485 lines with 120Ω. If J5 is not installed while J8 is installed, terminates with an RC combination of 120Ω and 0.1 μF. RC termination is typically used in low power applications where purely resistive termination would consume too much current.
J6 Bias+	OFF	Pull-up biases the XCVR+ RS485 line with 1KΩ. Typically used to enhance noise immunity when the transmitters are inactive. Must be used with J7.
J7 Bias-	OFF	Pull-down biases the XCVR- RS485 line with 1KΩ. Typically used to enhance noise immunity when the transmitters are inactive. Must be used with J6.
J8 R / RC	OFF	Used with J5, enables R or RC termination of the RS485 lines. If not installed the lines are unterminated and J5 has no effect.
J9 RS485	OFF	Enables RS485 communication via Serial 1.
J10 (Boot)	OFF	Puts the 68HC11 into bootstrap mode for special diagnostics.
J11 Clean	OFF	Invokes *special cleanup mode* on the next power-up or reset. To return the Handheld to its pristine, right-from-the-factory condition, install J11, install the reset jumper (J12) or switch power OFF then ON, then remove both jumpers.
J12 Reset	OFF	Forces a processor reset; remove it to restore processor operation.
J13 Off Detect	OFF	Connects processor Port A pin PA0 to the ON/OFF key so that PA0 can be used to detect an ON/OFF keypress, or to programmatically turn OFF the Handheld.
J14 XCVR+_/RxD2	XCVR+	Selects whether XCVR+ or /RxD2 is connected to the Personality Board.
J15 XCVR-_/TxD2	XCVR-	Selects whether XCVR- or /TxD2 is connected to the Personality Board.
J16 /RxD1_/RxD2	/RxD1	Selects whether /RxD1 or /RxD2 is connected to the Personality Board.
J17 /TxD1_/TxD2	/TxD1	Selects whether /TxD1 or /TxD2 is connected to the Personality Board.

A Tour of the Personality Board

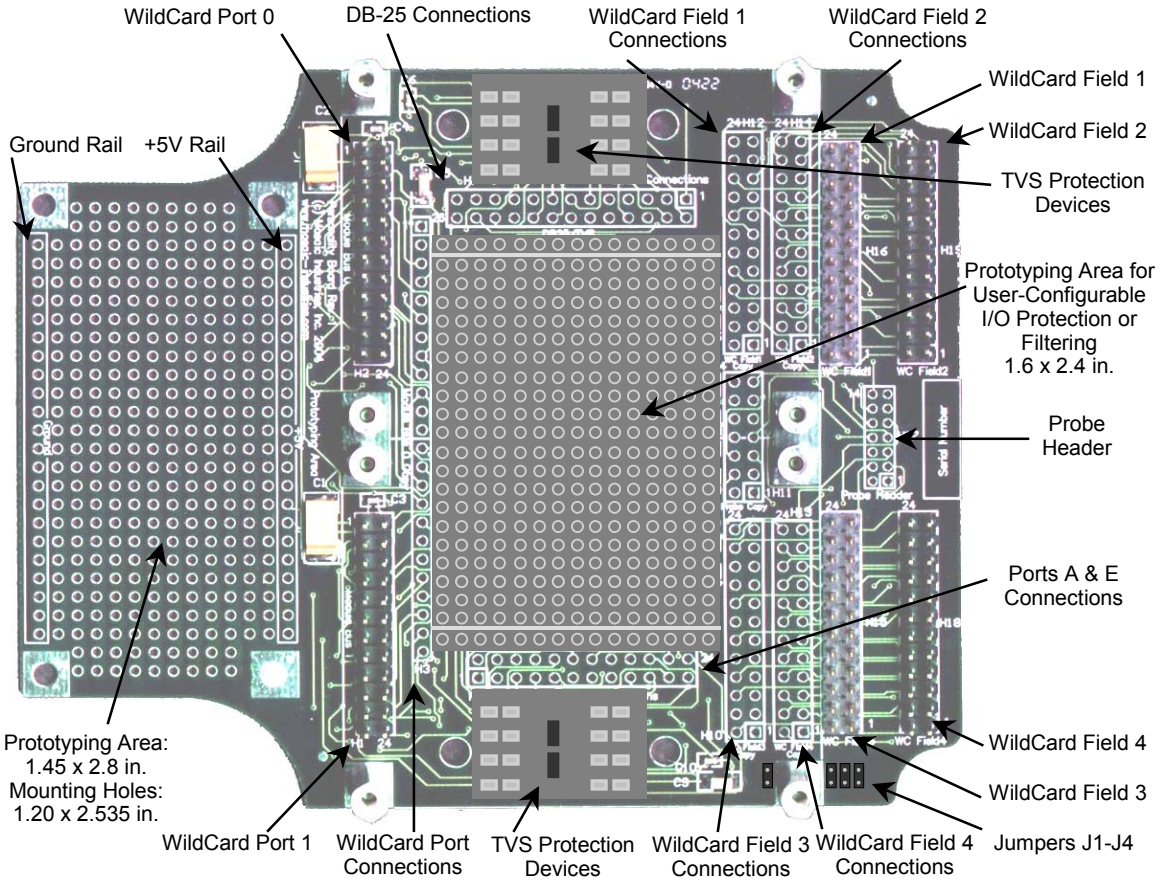


Figure 1-3 Headers and Jumpers on the Handheld's Personality Board.

Figure 1-3 shows prototyping areas on the left and in the middle for you to place custom circuitry if needed. Up to four WildCards stack on the Personality Board and ribbon cables connect their field headers to corresponding headers on the board. All headers are mirrored with their own sea of holes for point-to-point wiring. Any of the I/O signals from the WildCards or the Processor Board can be wired directly to the Handheld's DB-25 connector, or they can be wired through filters, protection circuitry, or your own circuitry. On the center right of Figure 1-3 of you can see an area for a probe. If you like, you can mount a probe on the center top of your Handheld and bring the signals through this connection.

There are four jumpers on the Personality Board, described in Table 1-3, useful while you are developing new software with the enclosure closed. Place jumpers at J1, J2 and J3 to route the serial line to the external DB25 connector, and place a jumper at J4 to route the cold reset line to the external connector.

Table 1-3 Personality Board Jumpers.

Label	Default Setting	Description
J1 DB25 Serial	ON	Connects serial line /TXD1 or 2 (as chosen by J17 on the Processor Board) to the DB25 pin 3. Place jumpers on J1, J2 and J3 to bring the serial line out to the DB25 connector. Useful for developing software with the case closed.
J2 DB25 Serial	ON	Connects serial line /RXD1 or 2 (as chosen by J16 on the Processor Board) to the DB25 pin 4. Place jumpers on J1, J2 and J3 to bring the serial line out to the DB25 connector. Useful for developing software with the case closed.
J3 DB25 Serial	ON	Connects serial ground to the DB25 pin 5. Place jumpers on J1, J2 and J3 to bring the serial line out to the DB25 connector. Useful for developing software with the case closed.
J4 Cold	ON	Connects the cold reset line to the DB25 pin 6 so that the Handheld can be reset externally. Useful for developing on the Handheld with the case closed.

Wildcard I/O Expansion Interface

The Personality Board's WildCard Port Headers allow you to mount up to four WildCards simultaneously. Port 0 allows two WildCards at module addresses 01, 02, or 03. The Handheld's display and keypad use module number 0 so that kernel-resident software drivers can be used. Port 1 allows two WildCards at module addresses 04-07. The 24-pin stackable go-through WildCard headers provide the following signals:

H2 and H3: WildCard Port Headers

Signal	Pins	Signal
GND	1 – 2	+5V
/IRQ	3 – 4	V+Raw
SEL1_XMIT-	5 – 6	SEL0_XMIT+
MOSI_XCV-	7 – 8	MISO_XCV+
/RESET	9 – 10	SCK
/MOD.CS	11 – 12	16 MHz
E	13 – 14	R/W
/OE	15 – 16	/WE
AD7	17 – 18	AD6
AD5	19 – 20	AD4
AD3	21 – 22	AD2
AD1	23 – 24	AD0

A Tour of the Power Board

The Power Board contains high efficiency switching regulators to supply the various voltages required by the Handheld, as well as charging and battery management circuitry for the NiMH

battery pack. Figure 1-4 shows the positions of various jumpers and connectors on the Power Board.

Several jumpers on the power board allow you to configure its operation as described in Table 1-4.

Table 1-4 Power Board Jumpers

Label	Default Setting	Description
J1 DB25-1 / GND	GND	If installed closest to the edge of the board, connects pin 1 of the DB-25 to signal DB25_1 on the Personality Board; if it is installed closest to the DB-25 connector, pin 1 of the DB-25 is instead connected to ground.
J2 DB25-2 / EXT_VIN	EXT_VIN	If installed closest to the edge of the board, connects pin 2 of the DB-25 to signal DB25_2 on the Personality Board; if installed closest to the DB-25 connector, pin 2 of the DB-25 is instead connected to signal EXT_VIN so that the Handheld may be powered through the DB-25 connector.
J3 Bright Backlight	OFF	Sets the LED backlight current to a software-chosen 510 mA or 285 mA; when not installed sets it to software-chosen 255 mA or 140 mA.
J4 Force-On	OFF	Holds the Handheld in the ON state and can not be turned OFF.
J5 Auto-On	ON	Enables the Handheld to turn on automatically when power is applied to the power jack or to the EXT_VIN line.

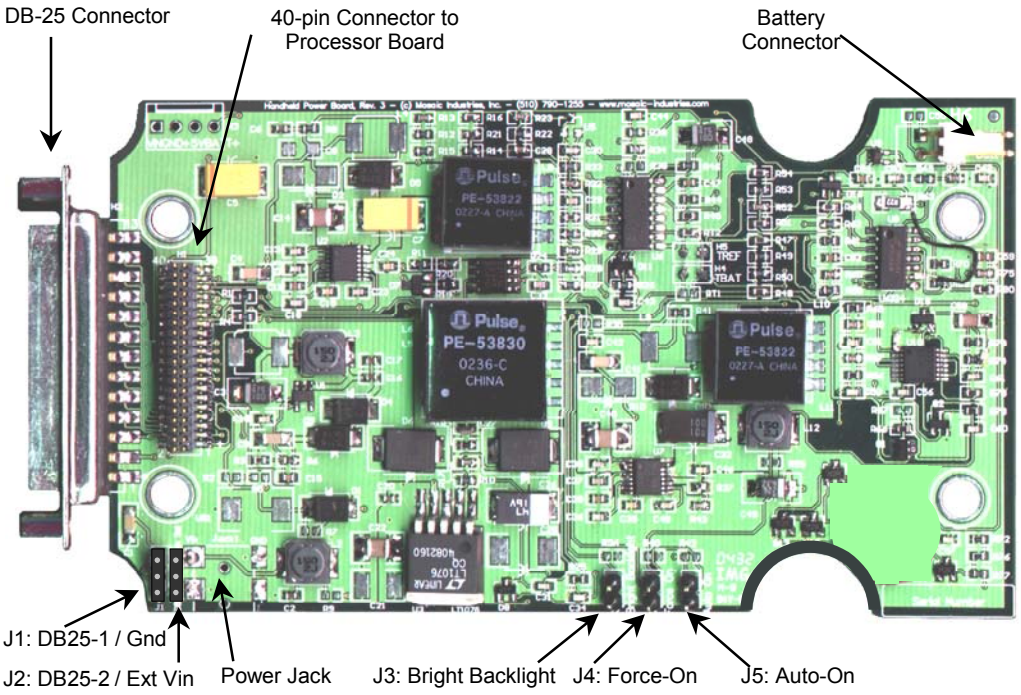


Figure 1-4 Headers and Jumpers on the Handheld's Power Board.

Chapter 2

Powering Your Handheld

The Mosaic Handheld provides an internal battery pack for portability. You can use a plug-in power adapter to recharge the batteries, or to power your Handheld directly. While recharging you can continue to use the instrument. In tethered operation, you can also bring power in through the Handheld's DB-25 connector.

Controlling Power

Automatic Turn-On

When the Auto-On jumper, J5, on the Power Board is connected, the Handheld automatically turns on when power is applied to either the external power jack or to the EXT_VIN signal. The EXT_VIN signal appears on pin 1 of the Power Header, H3, of the Power Board. Power ground and EXT_VIN may also be connected to pins 1 and 2 respectively of the Handheld's DB-25 by setting the jumpers on J1 and J2 to the position closest to the DB-25.

Manual Turn On and Off

The lower right button of the keypad is dedicated to turning the instrument on and off. Manually pressing the button when the Handheld is OFF turns it ON, and holding it down in excess of three seconds while it is ON turns it OFF.

Programmable Shut-Down

For orderly shutdown your software can intercept the press of the ON/OFF button, perform an appropriate exit routine, and then shut down the instrument. Whether or not you shut down the instrument programmatically, a keypress of the ON/OFF button of greater than three seconds will force a hardware shutdown. Further, whether or not a user presses the ON/OFF button, you can always programmatically shut down the instrument at any time.

To detect a keypress on the ON/OFF button, you must connect jumper J13 on the processor board. Doing this connects processor Port A pin PA0 to the ON/OFF key. Configure PA0 as an input capture (IC1) to evoke an interrupt on a rising edge. To enable programmatic shutdown, reconfigure PA0 as an output, set it high for at least 10 microseconds, then set it low. This action will cause an immediate shutdown.

Continuously ON

The Handheld is held continuously ON whenever external or battery power is present by connecting jumper J4 on the Power Board. With this jumper installed the Handheld can not be turned OFF short of removing all sources of power.

Using Battery Power

The Handheld may be powered by six rechargeable size AA NiMH 2100mAH batteries for the times shown in Table 2-1. The battery pack provides 15 watt-hours of energy between charges. The batteries are specified with a nominal cell voltage of 1.2 V (for a pack voltage of 7.2 v) and actually range from 1.375 for a fully charged but unloaded cell to 1.0 V at full discharge.

Table 2-1 Typical Battery Operation Times Using Six 2100 mAH Cells.

Service Time (Hours)	Conditions	
10	Backlight OFF	0 WildCards installed.
6	Backlight ON	0 WildCards installed.
5	Backlight OFF	4 WildCards Installed
4	Backlight ON	4 WildCards installed.

Replacing the Batteries

The batteries are replaced by first removing the battery cover which is held in place with six Phillips machine screws. A battery holder is permanently affixed to the Handheld. The batteries are a tight fit in the holder, but they can be carefully removed and replaced by lifting them up at the center, where each series pair of AA cells meet. When replacing them you must be sure to insert them with the correct polarity; + and – signs are inscribed in the aluminum holder.

Charging the Batteries

The batteries charge automatically when the power adapter is plugged in, whether the instrument is ON or OFF at the time. There are two charging states, fast-charge (at a 1.3 A rate) and trickle charge (at a 72 mA rate). Whenever the power adapter is plugged in the batteries charge at either of these two rates, with the charger automatically switching between rates.

When power is first applied, the batteries fast-charge to approximately 80% of full charge within 90 minutes, then trickle charge for the remainder of their capacity over five hours. At high ambient temperatures the fast charge is terminated earlier to prevent battery overheating, so more time is required for trickle charging.

The Handheld starts fast-charging the batteries when they are first installed and the power adapter is plugged in, or when the power adapter is first plugged in. It switches to trickle-charge when any of the conditions of Table 2-2 are met. For safety, fast-charge is inhibited as long as the battery tem-

perature is less than 1°C; fast-charge commences when the battery temperature rises above the 1°C threshold, and always ends if the batteries reach 52.5°C.

Table 2-2 Conditions for Fast-Charge Termination.

Parameter	Value at Termination
Fast-Charge Time Limit	> 132 minutes
High and Low Temperature Limits	<1°C or > 52.5°C
Max Rate of change of Temperature (dT/dt)	> 1°C per minute
High and Low Battery Voltage limits	<1.05 V/cell or > 1.84 V/cell
Change in Battery Voltage (-dV/dT)	< –5 mV/cell per minute

Be Careful !

Use Only Size AA NiMH Batteries:

When using battery power you must use only nickel metal hydride (NiMH) size AA batteries. Different sized batteries, or alkaline, nickel cadmium (NiCad), or lithium ion (Li-Ion) batteries, can not be charged safely in the Handheld and may overheat, burst, catch on fire or otherwise damage the Handheld.

Don't Trickle Charge Too Long:

To get optimum life from your batteries they should not be left on trickle charge for longer than needed. So your instrument should not be left plugged in to a power source when it is not used. Instead, it should be trickle charged for a maximum of 12 hours (typically, 5-7 hours are needed), then unplugged.

Battery Lifetime

NiHM batteries are a great battery technology to use in handheld instruments. They are more reliable and have much better cycle life than NiCAD batteries. Although they aren't as light as Lithium batteries, they are an order of magnitude less costly and have much greater lifetime. The shelf life of NiMH batteries is reliably several years if they are maintained between 10°C and 30°C and if they are charged at least once each year. They are conservatively specified by their manufacturer at a one-year shelf life at temperatures from -20°C to +35°C. Do not store them where the temperature fluctuates or is ever less than -20°C or greater than +45°C.

The cycle life of NiMH batteries depends on the conditions under which they are charged and discharged. Ideally, they should be charged at temperatures between 10°C and 30°C, where they charge most efficiently. Charging should not be initiated at temperatures less than 0°C or greater than 40°C, nor continue at temperatures greater than 55°C. The Mosaic Handheld limits fast-charge to battery temperatures between 1°C and 52.5°C.

Under ideal conditions state-of-the-art NiMH batteries lose only 10-20% of capacity over 500 charge cycles. Significantly reduced service time despite proper charging means that the life of the battery has been attained, and the batteries should be replaced. Normally the batteries should last at least 2 years if not overcharged or undercharged.

Tethered Operation

In tethered operation, the Handheld may be powered by applying any unregulated DC input voltage between 15 and 30 volts. If you power the board with an unregulated DC supply, the onboard circuitry is protected by a built-in surge protector and electromagnetic interference (EMI) suppression circuitry. This improves the reliability of the electronics in harsh industrial environments. The Handheld requires 20 watts. Of this, 17 watts is used for fast charging and all other instrument operations need only 3 watts. Consequently, if batteries are not used only 3 watts is required to fully power the instrument.

You can supply power to the Handheld by connecting jumpers J1 and J2 on the Power Board. If these jumpers are installed closest to the edge of the board, pins 1 and 2 of the DB-25 are connected to the Personality Board; if they are installed closest to the DB-25 connector, pins 1 and 2 of the DB-25 can be used to supply ground and power (EXT_VIN) so that the Handheld may be powered through the DB-25 connector as well as through its power jack.

Using the Power Adapter

The AC adapter provided with your Handheld supplies 24-28V at 20 W. Plug the adapter into the jack located on the right side of the enclosure near the bottom.

Power Metering

The Handheld can measure its own battery voltage and current.

Monitoring Battery Current

The current is converted to a voltage and presented as a signal named MON_CHARGER. This signal is routed to the Processor Board where it can be connected to PE6 (one of the processor's 8-bit A/D inputs) by installing Processor Board jumper J3. The relationship between this voltage and the battery current is given in Table 2-3.

Table 2-3 MON_CHARGER Voltage as a function of battery current.

MON_CHARGER Voltage	Battery Current	Condition
0.8	-1.3 A	Battery is fast charging at 1.3 A
0.58	-.071 A	Battery is trickle charging at 71 mA
0.564 to 0.291	0 to 1.5 A	Battery is discharging during operation
V	I = (0.564 – V) / 0.182	Battery is discharging during operation

The battery current, I_{bat}, is related to the MON_CHARGER voltage as,

Eqn. 2-1 $$I_{bat} = (0.564 - MON_CHARGER) / 0.182$$

This current is negative during charge, and positive during discharge.

Monitoring Battery Voltage

The battery voltage is monitored through an attenuating (by a factor of 4) resistive voltage divider and presented as a signal named MON_BATT/4. This signal is routed to the Processor Board where you can connect it to PE7 by installing Processor Board jumper J4. The measured voltage is roughly one quarter of the battery voltage. However the battery is in series with a 0.22 ohm resistor, so that the measured voltage is actually the sum of the battery voltage and the small voltage drop across the resistor. Consequently, the actual battery voltage is given by,

Eqn. 2-2 $$V_{bat} = 4 \times MON_BATT/4 + I_{bat} \times 0.22 \text{ , or,}$$
$$V_{bat} = 4 \times MON_BATT/4 + 0.22 \times (0.564 - MON_CHARGER) / 0.182$$

in which I_{bat} is the signed battery current of Eqn. 2-1. For example, while the battery is fast-charging the actual battery voltage is four times the measured MON_CHARGER/4 minus 0.286V.

For an example of how to measure battery voltage take a look at the source code of the demo application that shipped with your Handheld.

Chapter 3

Your First Program

This Chapter will get you started using the Control-C language to program your Handheld. It will guide you through the installation of the Mosaic IDE, an integrated editor, compiler, 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 Handheld's extended memory space, and prints the results to the terminal;*
- ⇨ *Selectively execute any program function using the Handheld's on-board debugging environment (called QED-Forth);*
- ⇨ *Use the Handheld'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 and Control-C Compiler

The Mosaic IDE, which includes the Control-C Compiler, 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* (the editor used within the Mosaic IDE), the *Mosaic Terminal* (previously called *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 then turn it back on; lesser restarts don't always restart fully.

You are now ready to talk with your Handheld!

Turning on Your Handheld

Familiarize yourself with the locations of the power and serial connectors as shown in Chapters 1 and 2. After finding them follow these steps to turn on your system and establish communications with it:

1. Connect your PC serial cable to the Handheld. You can use the 10-pin-socket-to-DB9 cable to connect to the Handheld's internal serial header (on the Processor Board with the instrumetn open and the two halves connected with the high-density interconect cable) or you can use the DB25-to-DB9 cable to connect to the DB25 connector on the bottom of the Handheld. Connect the female end of the 9-Pin serial communications cable to your computer terminal's RS-232 serial communications port. You can use any of your PC's COM ports. COM2 is usually available, but some PCs only have COM1 available. If your computer does not have an RS-232 serial port, low cost USB-to-RS-232 serial cables are available; contact Mosaic Industries for details.
2. Power up your PC computer or terminal.
3. 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 (COM2)". (If COM2 is tied up with another service, such as a fax/modem, you may want to use COM1 instead.)
    3. You'll now have a dialog box called "Communications Port (COM2) Properties". Click the general tab and make sure you have these settings:

Property	Value
Baud Rate	19200
Data Bits	8
Parity	None
Stop Bits	1
Flow Control	Xon/Xoff

4. We recommend that you use *Mosaic Terminal*, the terminal program that comes with the Mosaic IDE. You can start the terminal by double clicking the Mosaic IDE executable (the primary application of the Mosaic IDE) and choosing the terminal toolbar button which looks like this:

(The appearance of this and other toolbar icons may change in subsequent versions of the Mosaic IDE.) You can also start the terminal by double clicking the application "MosaicTerminal.exe".

The terminal starts using COM2 by default at a speed of 19200 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 linefeed (`^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. If you use another terminal program, you must specify that it use the linefeed character as its *prompt character*. It might be denoted as LF, ^J, ascii decimal 10, or ascii hex A.

If your PC is set up to use a COM port other than COM2 Mosaic Terminal will respond with a warning box saying "Invalid port number". If so, just go to the Mosaic Terminal menu item "Settings→Comm→Port" and choose the COM port you chose when configuring Windows in step 3. above.

5. Verify the baud rate of the Mosaic Terminal by going to "Settings→Comm→Baud Rate" and making sure that 19200 is selected. This is the baud rate used to communicate with the QScreen Controller. If the baud rate is incorrect, garbled characters may appear in the terminal when you try to communicate with the controller.

6. Plug the Handheld's power adapter 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 side of the Handheld and turn it ON by pressing the ON/OFF keypad button in the lower right corner of the keypad.

    1. If no application program is running you should now see the following response:

        ```
        Coldstart QED-Forth V4.4x
        ```

    2. If there is a demonstration program running and if you hit the enter key while the cursor is in your terminal window you should see the Handheld respond with,

        ```
        ok
        ```

The serial communications responses indicate that your Handheld is now working!

If Something Doesn't Work

If 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 Mosaic Terminal (using the menu item "Settings → Comm"), and recheck the communications properties of the Windows communications port.

4. Perform a "special cleanup".

> ***Doing a "Special Cleanup"***
>
> If you ever need to return your Handheld to its factory-new condition, just do a *Special Cleanup*: With the power on, install jumper J11 on the processor board, install and remove the reset jumper, J12, then remove J11. This procedure will remove any application programs and reinitialize all operating system parameters to their factory-new condition. Note that this procedure removes any demo programs too.

If you still are having trouble, email or give us a call.

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 Mosaic Terminal 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.
- *Mosaic Terminal* 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 compiled C programs into the memory of the Handheld. *Mosaic Terminal* 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.

  You can type characters directly into the terminal window, and they will be accepted by the Handheld's line editor and interpreter. This mode of interaction is convenient when debugging or typing short code fragments. If you are sending source code to the Handheld, it is best to create a file first. The file can be saved to disk to provide a record of your work, and the terminal program can be used to download the file to the Handheld. You can also use the terminal to record your debugging sessions and save them as a file on disk.

The TextPad Tool Bar

Along with the standard tools you expect in a text editor you'll find some 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 *Debug* icon calls the C compiler and assembler only, the *Single-Page Compile* icon performs a standard build of your program, and the *Multi-Page Compile* icon 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 *Mosaic Terminal* program. Each of these tools is described in more detail below.

The 'Debug Tool' Finds Syntax Errors in C Programs

The Debug Tool, designated by the "Magnifying Glass" icon, invokes the compiler and assembler only – it does not produce downloadable code. Use it to quickly check the syntax of a program and find compilation errors without performing the full build which would be needed to download the program into the microcontroller.

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

The Single-Page Compile Tool, designated by the "Single-Page" icon, performs a standard, single-page memory model build of your program. If you are compiling a program whose compiled size does not exceed 32 Kbytes of memory you should use this mode for fastest execution. Although this program size is sufficient for many applications, you may need to use the multi-page build if your application grows beyond 32 K. If you see the following warning printed in the compiler output, then you must switch to the multi-page memory model build (using the Multi-Page Make icon):

```
WARNING: Input section ".doubleword" from 'progname.o11' is not used !
```

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

The Multi-Page Compile Tool, designated by the "Multi-Page" icon, invokes the C compiler's multi-page build mode. Programs compiled in this mode may be many pages in length limited only by the amount of FLASH installed in the Handheld. It is always a good programming practice to break large projects into multiple smaller source code files for organization, and the multi-page build also uses this organization for distributing the compiled program across multiple 32 Kbyte pages. Thus, no source code file may contain more than 32 Kbytes worth of compiled source code or the above warning will be issued and the program will not run. A more detailed description of this behavior is available in Chapter 4.

You may wonder why there are both "Single-Page" and "Multi-Page" compile modes. The reason is that C function calls between pages take just a little longer to execute (calls to functions on a different page take 49 microseconds while those on the same page or to common memory take only 11.5 or 13.75 microseconds, respectively). Because most function calls are to functions on the same page or to common memory, page changes are rare; the average execution speed of multi-page C applications is still quite fast.

'Mosaic Terminal' Communicates with Your Product

The Terminal icon launches the communications program, Mosaic Terminal. When you launch Mosaic Terminal for the first time, check the communications settings (Settings→Comm) to verify that the serial port is set correctly for your computer.

Your First C Program

Now that we've learned about the Handheld's hardware, established serial communications, and installed the Mosaic IDE on the PC, it's time to compile, download and execute a C program. We'll also explore the Handheld's on-board operating system and use it to interactively debug a program.

Compiling a Program

In this section we'll be running a simple program that does some computation and communicates its results on the serial port. The program is one of several examples for use with the Control-C IDE in the "\Mosaic\Demos_and_Drivers\Misc\C Examples" directory. Let's compile, download, and run it.

Start TextPad, and open the source-code file "getstart.c" from the C_Examples directory. You should see the source code in an open window – browse through it to get a feel for it. You'll see that the final routine in the file is called **`main()`**; it's the top-level executable program. The first routine within **`main()`** that is called when the program starts is **`InitVars()`**. Note that in the run-from-place applications of embedded systems it's important to initialize all variables with a run-time procedure when the program starts. Variables that are initialized when the program is compiled are not automatically initialized when the program runs; you should have a runtime routine in your code that does that.

Clicking on the Single-Page Compile icon will compile and produce a downloadable form of the program, named "getstart.dlf". A new window named 'Command Results' will appear so that you can watch the compilation process. When compilation has finished, you can scroll the Command Results window up and look for warnings, which don't prevent creating a valid download file, and errors, which do. You should see two warnings near the beginning:

```
GETSTART.C(126): Warning: Expression is always TRUE !
GETSTART.C(197): Warning: Symbol 'unused_variable' is never used in function 'main' !
```

We deliberately inserted into 'main' a variable named **`unused_variable`** that is never used in the function. If you double click on an error or warning line in the command results, TextPad will jump to the corresponding line in the affected source file. Despite the warnings, the program should have compiled successfully; the command results will end with:

```
Qcc-> Creating QED Download File: getstart.dlf
Tool Completed Successfully
```

You can quickly switch between the Command Results window and your source code file either by hitting Ctrl-Tab, or by clicking on the file tabs at the bottom of the TextPad window.

The file named "getstart.dlf" is ready to be downloaded to the microcontroller using the Mosaic Terminal program.

Downloading and Running the Program

If it is not already open, launch Mosaic Terminal either from the 'Start' menu or using the TextPad toolbar or dropdown menu. It's most convenient to use the Terminal icon on the TextPad toolbar.

You should be able to hit enter at the Mosaic Terminal prompt and see the 'ok' response with the microcontroller plugged in and turned on. If this is not the case, check your communications settings and cabling.

Now, select 'File → Send File' from the Mosaic Terminal menu and enter the "\Mosaic\Demos_and_Drivers\Misc\C Examples" directory, or wherever you compiled the program. Set the file type to "Download Files (*.dlf)" and select "**getstart.dlf**". You will see various commands and hex data scrolling on the screen as the file is downloaded to the microcontroller. When the download is complete, the text will turn from gray to green to indicate that it is finished. Now, it's time to run your program.

To execute the top level function of your code, simply type 'main' and press enter,

```
main←
```

The 'Enter' key is represented by the ← symbol in the line above.

The getstart program will respond with:

```
Starting condition:
The radius is       0; the circular area is      0.
 ok
```

While on its face that doesn't seem a very impressive response, you're running your first program! This particular example program uses multitasking. The program runs a background task called **CalculationTask** continuously, incrementing a radius variable and using it to compute a new area. The program is running in its own task, leaving the communications task free so you can continue to interact with the controller.

You will notice that you can hit enter, and use the interactive debugging capabilities even though the program is still running. For example, try executing the following function interactively from the terminal:

```
Announce( )←
```

Note that you must type the space after the **(** character. Each time you execute this function you'll notice that the output is different, as the radius is being continuously incremented by the background task. Now try executing,

```
Nap( )←
```

which puts the background CalculationTask **ASLEEP**. If you again execute

```
Announce( )←
```

several times, you will notice that the radius and area are no longer being updated by the **CalculationTask**. To wake up the **CalculationTask** again, type

```
Wakeup( )←
```

and notice that the calculation is again being performed by the task.

You may want to stop the program; in particular you'll need to stop it before attempting any new downloads. This can be done most easily by simply entering '**warm**' at the microcontroller's

prompt. The warm restart causes a soft reset to occur, terminating any background tasks that may be running.

After having run this program, you may want to play with the other example programs in the "\Mosaic\Demos_and_Drivers\Misc\C Examples" directory. We strongly recommend that you compile these programs and work through the examples as suggested in the text of this manual. This will provide you with a thorough "hands-on" introduction to the Control-C programming environment.

Interactively Debugging Your Program

We have seen how to interactively call the **main()** function from the terminal to execute our entire program; most C development environments let you do this. But the Handheld's operating system makes it easy to interactively execute any designated function in your program. By simply preceding a function definition or prototype with the **_Q** keyword (we chose "**_Q**" as a unique keyword that suggests *QED*), you can ensure that the function will be interactively callable from your terminal.

An example: Announce( ) Displays an Area and Radius

For example, to display a summary of the current values of the radius and calculated circular area variables, we would like to call the function **Announce( )**.

Using the editor, look near the top of the **GETSTART.C** file and you'll see that its definition is:

```
_Q void Announce(void)
{ printf("\nThe radius is %6u; the circular area is %5.4g.\n",radius,area);
}
```

The **void** keywords indicate that the **Announce( )** function does not return a value, and does not expect any input parameters to be passed to it.

The **_Q** declarator instructs the compiler that we want to be able to interactively call this function using the on-board QED-Forth interpreter. The names and execution addresses of all functions defined with the **_Q** designator are placed in the **.DLF** download file so that QED-Forth will recognize them and will be able to interactively execute them.

The **printf()** function invoked in **Announce( )** prints the specified string to the serial1 port. The parameters of the **printf()** function are well defined by the ANSI standard, and are described in many excellent texts. Briefly, the **\n** is an escape sequence that instructs **printf** to insert a newline at the designated places in the string. The **%** characters are formatting symbols that tell the compiler to substitute the listed arguments (in this case, the radius and area) for the **%** sequences at runtime. The **%6u** sequence tells the compiler to display the radius as an unsigned decimal number with a minimum field width of 6. The **%5.4g** sequence tells the compiler to display the area using either decimal or exponential notation with a precision of 4 decimal places to the right of the decimal point, and a minimum field width of 5.

The **printf()** function in Control-C differs from the ANSI standard in one respect: the maximum length of a printed string is limited to 80 characters instead of the standard 255 characters. This limitation also applies to the related functions named **sprintf()** (which writes a string to a buffer)

and **scanf()** (which inputs a string). Of course, you can handle strings longer than 80 characters by using multiple calls to these functions.

Interactively Calling Announce( )

To interactively call this function, simply type at your terminal

```
Announce( )←
```

followed by a carriage return (indicated by the arrow above). Spaces are important to the QED-Forth interpreter which processes this command; make sure that there is no space between the function name Announce and the opening parenthesis (, and there must be at least one space after the opening parenthesis. If QED-Forth does not recognize your command, it will repeat what you typed followed by a "**?**" character and wait for another command, so you can try again. The case of the letters does not matter: you can use all uppercase, all lowercase, or any combination when typing commands for the QED-Forth interpreter.

After calling **Announce( )**, you should now see the message

```
The radius is      0; the circular area is      0.
```

on your screen, except that the printed values of the radius and area will correspond to the values they had when you executed the "**WARM**" command to stop the calculations. Then you will see an additional line of text starting with "**Rtn:**" that summarizes the return value of the function in several formats, followed by the "**ok**" prompt. Because the **Announce()** function has no return value, the return value summary is not relevant. The "**ok**" prompt indicates that QED-Forth has successfully called the function and is now ready to execute another command.

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.

Area Calculation

The next function defined in **GETSTART.C** is called **IncrementRadius()**. This simple function increments the radius variable, and resets it to 0 when it exceeds the **MAX_RADIUS** constant. As described below, **IncrementRadius()** is called from the infinite loop in **CalcForever()**; this results in the radius taking on all integer values between 0 and 1000.

The next function defined in the **GETSTART.C** file calculates the area of a circle; its definition is:

```
_Q float CalcArea(uint radius)
{  return PI * radius * radius;
}
```

As described above, the **_Q** designator flags this function as one that can be called interactively. The "float" keyword declares that the function returns a floating point value, and the parameter list tells us that the function expects a single unsigned integer (**uint**) as its input. (Note: **uint** and other

useful type abbreviations and declarations are defined in the **TYPES.H** header file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory.)

To interactively test this function with an input radius of 5, type at your terminal

```
CalcArea( int 5)←
```

followed by a carriage return. QED-Forth uses spaces as delimiters; consequently, you must type at least one space after the (character and after the "**int**" keyword. You should see something like the following response at your terminal:

```
Rtn: 17053 5242 =0x429D147A=fp: 78.54
```

This line summarizes the returned value in several formats, including decimal or hexadecimal 16-bit values, 32-bit hexadecimal, and floating point. Because the **CalcArea()** function returns a floating point (**fp**) value, the final number on the line, labeled

```
=fp: 78.54
```

is the relevant return value. Indeed, 78.54 is the area of a circle that has the specified radius of 5. You can execute the function with any integer input as the radius, and verify that it returns the correct circular area. This capability enables interactive testing of the function over its allowed range of input values. Such thorough function-by-function testing of a program facilitates rapid development of reliable programs.

In the next chapter the interactive debugging process will be explored in more detail. You will learn how to examine the values of static variables and Forth arrays, pass parameters by value or by reference, generate hexadecimal and ascii dumps of memory contents, and modify the contents stored in variables and Forth arrays.

Restrictions on the Use of _Q

Nearly every function in the many sample programs in the **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory is declared with the **_Q** keyword to facilitate easy debugging. There are, however, two restrictions associated with the use of the **_Q** declarator.

First, a function defined using the **_Q** keyword cannot use

```
...
```

(ellipsis) in its parameter list; rather, the number of input parameters must be specified when the function is defined. (If you try to define the **_Q** function with an ellipsis as an input parameter, the compiler will issue a warning and remove the **_Q** specifier, so you will not be able to interactively call the function during debugging.)

The second restriction is that the **_Q** function cannot be called via a function pointer if the function accepts input parameters. In other words, do not use the **_Q** declarator if:

a. You need to call the function using a function pointer; and,

b. The function accepts input parameters.

This restriction does not affect many functions. Any function declared using `_Q` can always be called in the standard way (that is, by invoking the function name followed by parentheses that contain any input parameters). Moreover, any `_Q` function can be called indirectly via a function pointer (by passing its name without any parentheses) if the function's input parameter list is "`void`".

An Introduction to Extended Memory

The Handheld's onboard operating system, called QED-Forth, provides numerous run-time services, including providing a heap memory manager. Using this memory manager we can access the controller's extended memory.

1 Megabyte Addressable Memory Space

The standard 68HC11 processor can address 64 kilobytes of memory using 16-bit addressing. The Handheld expands the address space to 1 Megabyte, addressing the lower 32 Kbytes of the processor's memory space by means of a 5-bit "Page Latch" that selects one of 32 pages. The 32 pages times 32 Kbytes per page yields 1 Megabyte of addressable memory. The upper 32 Kbytes of the 68HC11's address space is called the "common memory". This address space is always accessible, regardless of the contents of the Page Latch.

Available Common RAM

The ANSI C compiler supports the standard 16-bit addressing scheme via its *small memory model*. It also supports a *medium memory model* that allows functions to be called on any specified page using a 24-bit address (16-bit standard address plus an 8-bit page). All C variables and C arrays, however, must be accessible using a simple 16-bit address. For practical purposes, this means that variables and C arrays must reside in the Handheld's available 8 kilobytes of available common RAM located at addresses 0x8E00 to 0xADFF. In multitasking applications, this RAM is also used for task areas; each task requires 1 Kbyte of common RAM area.

You are of course free to use ANSI-standard C arrays located in the variable area in common RAM. These arrays allow you to use standard C pointer arithmetic, and their use is explained in all of the C textbooks. However, if you need to store *a lot* of data, the available 8K of common RAM may not be sufficient. But don't worry – you can still use all the memory.

Built-in Array Library Routines Manage Access to Paged Memory

The `FORTH_ARRAY` routines that reside in ROM on the Handheld provide an efficient means of accessing the large paged address space for storage of data. The pre-defined `DIM()` macro makes it easy to dimension a 2-dimensional array to hold signed or unsigned characters, integers, longs, or floating point values. Other pre-defined library functions handle storing, fetching, and copying data to and from the arrays. These QED-Forth functions are callable from C, and provide access to a large contiguous memory space that is very useful for real-time data storage and analysis.

Each array is referred to using a named 16-bit pointer to a "parameter field" structure in common RAM. Once the array has been "dimensioned", this structure holds the number of rows and columns, data size, and a pointer to the QED-Forth heap where the array is allocated. The ROM-

resident heap manager allocates and deletes the arrays in real time under the control of the C program, thereby maximizing the effective use of available paged RAM.

This section introduces the use of the arrays, and as we'll see in a later chapter, they are very useful for storing data from the Handheld's A/D convertors. The header file named **ARRAY.H** in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory contains all of the function and macro definitions that are used to access Forth arrays, including the **DIM()**, **FARRAYFETCH()** and **FARRAYSTORE()** macros that are mentioned in this section.

Declaring and Dimensioning a FORTH ARRAY

Let's look at the example code in the **GETSTART.C** file. Approximately 1/3 of the way into the file, you'll find a section called "Array Dimensioning, Storing and Fetching". The first command in this section is:

```
FORTH_ARRAY   circle_parameters;
```

which declares a new **FORTH_ARRAY** named circle_parameters and allocates storage for the structure in the variable area in common RAM. **FORTH_ARRAY** is a **struct typedef** (see the **ARRAY.H** file) that specifies how the dimensioning information for the array is to be stored. Whenever we want to call a function to operate on this array, we will pass the pointer

```
&circle_parameters
```

as an argument to the function.

After using **#define** directives to define some dimensioning constants, we encounter the following function definition:

```
_Q void DimAndInitFPArray(float value,int rows,int cols,FORTH_ARRAY* array_ptr)
{  int r,c;
   DIM(float, rows, cols, array_ptr);        // dimension; allocate in heap
   for(c = 0; c < cols; c++)                 // for each column
      for(r=0; r< rows; r++)                 // for each row
         FARRAYSTORE(value,r,c,array_ptr);   // store in array
}
```

The function dimensions a **FORTH_ARRAY** and initializes all elements of the array to have a specified floating point value. The inputs are the floating point value, the number of rows and columns, and a pointer to the **FORTH_ARRAY** structure in common memory. After declaring the automatic variables **r** and **c**, the **DIM()** macro is invoked to emplace the dimensioning information in the **FORTH_ARRAY** structure, and allocate memory for the array in the heap.

The first parameter expected by **DIM()** is a type specifier; type definitions and abbreviations are defined in the **TYPES.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory. Valid type arguments for **DIM()** include the following:

```
char  unsigned char  uchar
int   unsigned int   uint
long  unsigned long  ulong
float xaddr
```

The next two input parameters expected by **DIM()** are the number of rows and columns, and the final input parameter is a pointer to the **FORTH_ARRAY** structure. The nested **for()** statements

cycle through each row and column element in the array, calling the macro **FARRAYSTORE()** to store the specified value into the array element. **FARRAYSTORE()** expects a floating point value, row and column indices, and a pointer to the **FORTH_ARRAY** as its inputs.

The starting "**F**" in the name **FARRAYSTORE()** means "floating point"; a parallel macro named **ARRAYSTORE()** is used for arrays that contain signed or unsigned **char**, **int**, or long data.

The **SaveCircleParameters()** function in the **GETSTART.C** file calls the macro **FARRAYSTORE()** to store the radius and area as floating point values in their respective columns of the **circle_parameters** array. Then it increments the **row_index** variable, handling overflow by resetting the **row_index** to zero to implement a circular storage buffer.

The next function in **GETSTART.C** is called **PrintFPArray()** which prints an array of floating point values to the terminal. Its definition is as follows:

```
_Q void PrintFPArray(FORTH_ARRAY* array_ptr)
{  int r, c;
   putchar('\n');
   for (r = 0; r < NUMROWS(array_ptr); r++)        // for each row
   {   for  (c = 0;c < NUMCOLUMNS(array_ptr);c++)  // for each col
         printf("%9.4g  ",FARRAYFETCH(float,r,c,array_ptr));
         // min field width=9;precision=4;g=exp or decimal notation
      putchar('\n');    // newline after each row is printed
      PauseOnKey();  // implement xon/xoff output flow control
   }
}
```

As usual, the **_Q declarator** allows this function to be called interactively from the terminal. **PrintFPArray()** expects a pointer to a **FORTH_ARRAY** as its input parameter, and uses 2 nested **for()** statements to print the contents of the array one row at a time.

The **printf()** statement invokes the Forth library macro **FARRAYFETCH()** to fetch the contents of the array at the specified row and column. **FARRAYFETCH()** returns the value stored in the array; it expects a type specifier (used to cast the return value to the required type), row and column indices, and a pointer to the **FORTH_ARRAY** as its inputs.

The **%9.4g** argument to **printf()** specifies that the number should be printed using either decimal or exponential formatting (whichever displays better precision), with 4 digits to the right of the decimal point and a minimum field width of 9 characters. The putchar('**\n**') statement inserts a newline character after each row is printed. The **PauseOnKey()** function is a handy library routine that serves 2 purposes:

- It implements XON/XOFF output flow control to avoid "inundating" the terminal with characters faster than the terminal can process them, and
- It allows the user to abort the printout by typing a carriage return from the terminal.

For further details, please consult the definition of **PauseOnKey()** in the Control-C Glossary.

To see how the **DimAndInitFPArray()** function is called, scroll down to the function named **CalcForever()** in the **GETSTART.C** file. The first statement in the function is:

```
DimAndInitFPArray(0.0,CIRCLE_ROWS,CIRCLE_COLUMNS,&circle_parameters);
```

where 0.0 is the floating point value to be stored in each element, the constants **CIRCLE_ROWS** and **CIRCLE_COLUMNS** specify the number of rows and columns in the array, and **&circle_parameters** is a pointer to the **FORTH_ARRAY**.

Interactively Dimension, Initialize and Print the Array

It is easy to interactively call the functions that we've examined. The syntax that we'll type at the terminal looks similar to an ANSI C function prototype, with one of the following type declarators being used before input parameters that are passed by value:

```
char     int      long     float
char*    int*     long*    float*
```

When passing the address of a variable or a structure, use only the name of the variable or structure, without any additional declarators or **&** operators. All of this is explained in detail in a later Chapter; for now, the goal is see how easy it is to use the interactive function calling tools.

For example, to interactively dimension and initialize the **circle_parameters** array to have 10 rows, 2 columns, with each element initialized to a value of 34.56, type the following line at your terminal:

```
DimAndInitFPArray( float 34.56,int 10,int 2,circle_parameters)←
```

Remember to type at least one space after the **(** character, and after the **float** and **int** keywords. QED-Forth will respond to your command with a line of text that summarizes the return value of the function, followed by the "**ok**" prompt. We can ignore the return value summary, because this function does not return a value.

Now to verify that the initialization was performed correctly, we can type at the terminal:

```
PrintFPArray( circle_parameters)←
```

and, as always, we make sure that there is a space after the (character. Note that we do not use the **&** (address-of) operator before the **circle_parameters** argument; it turns out that **circle_parameters** has already been defined in QED-Forth as the base address of the **FORTH_ARRAY** structure.

QED-Forth calls the function which prints the contents of the **circle_parameters** array, and then summarizes the return information (which we can ignore in this case). You can verify that the value of each array element is the same one that you specified when you called the **DimAndInitFPArray()** function. (Slight differences in the values are due to rounding errors in the floating point conversion and printing routines.) Using this interactive method, you can test each function with a variety of dimensioning and initialization information.

An Introduction to Multitasking

Many instrumentation and automation applications can be logically conceived of in terms of a set of distinct "tasks" that cooperate to solve the problem at hand. For example, a program that manages a hand-held sensing instrument might have one task that acquires sensory data, another that performs calculations to process the data, and a third task that displays the results on a liquid crystal display.

Using the Handheld's built-in multitasking executive confers significant advantages when designing real-time systems. Breaking up a complex program into easily understood modular tasks speeds debugging, improves maintainability, and prevents source code modifications of one task from adversely affecting the required real-time performance of another task.

The Task Activation Routine

In a multitasking environment, a "task" is an environment capable of running a program. After declaring (naming) a new task (which also allocates a 1 Kbyte task area), its environment is "built" by initializing its required stacks, buffers and pointers in the 1 Kbyte task area. Then the task is "activated" by associating it with an "activation routine" that performs a specified set of actions.

A typical task activation routine is the **`CalcForever()`** function in the **`GETSTART.C`** file. Its definition is straightforward:

```
_Q void CalcForever(void)
// this infinite loop function can be used as a task activation routine
{  DimAndInitFPArray(0.0,CIRCLE_ROWS,CIRCLE_COLUMNS,&circle_parameters);
   while(1)                     // infinite loop
   {  IncrementRadius();        // updates radius variable
      area = CalcArea(radius);  // updates area variable
      if(radius%10 == 0)        // on even multiples of 10...
      SaveCircleParameters();   // save data in FORTH_ARRAY
      Pause();                  // give other tasks a chance to run
   }
}
```

The first thing that this function does is to dimension and initialize the **`circle_parameters`** array. Then it enters an infinite loop that increments the radius variable, calculates the corresponding circular area and stores it in the area variable, and saves the radius and area in the **`circle_parameters`** array if the radius is an even multiple of 10. The function calls **`Pause()`** on every pass through the loop. **`Pause()`** is a multitasking function that instructs the multitasking executive to change to the next task (if any) in the round-robin task list. This enables "cooperative multitasking", in which a task willingly lets other tasks run by executing **`Pause()`**. The other type of multitasking, also supported by the Handheld, is "pre-emptive multitasking", in which an interrupt-driven timeslice clock forces a task switch on a periodic basis.

In summary, the **`CalcForever()`** function is an infinite loop that manages the calculation and storage of the radius and circular area. This function can be the "activation routine" for a standalone task running in a multitasking environment.

Declare, Build and Activate a Task

The short section titled "Multitasking" in the **`GETSTART.C`** file demonstrates how easy it is to set up a task using the pre-defined macros. First we declare the new task as:

```
TASK CalculationTask;
```

The **`TASK`** typedef allocates a 1 Kbyte task structure named **`CalculationTask`** in the common RAM.

The function **`SetupTask()`** builds and activates the new task; its definition is:

```
void SetupTask()
{  NEXT_TASK = TASKBASE;                          // empty task loop before building
   BUILD_C_TASK(HEAP_START,HEAP_END,&CalculationTask);  // private heap
   ACTIVATE(CalcForever, &CalculationTask);       // define task's activity
}
```

The first statement empties the round-robin task loop by setting the **NEXT_TASK** pointer in the task's user area to point to the task's own **TASKBASE**. The next statement invokes the **BUILD_C_TASK()** macro which expects starting and ending addresses for the task's heap, and the address at which the task is located. We have defined the constants **HEAP_START** and **HEAP_END** to specify a task-private heap occupying 1 Kbyte on page 0. The task base address is simply **&CalculationTask.** **BUILD_C_TASK()** sets up all of the stacks, buffers and pointers required by the task.

The final statement in **SetupTask()** invokes the **ACTIVATE()** macro which expects a pointer to the activation function (which is **CalcForever**) and the **TASKBASE** address (which is **&CalculationTask**).

Multiple tasks can be declared, built and activated in the same way.

Putting a Task Asleep

A "sleeping" task remains in the round-robin task loop, but is not entered by the multitasking executive. The status of a task can be changed from **AWAKE** to **ASLEEP** and back again by simply storing the appropriate constant in the **user_status** variable in the task's **USER_AREA**. The **USER_AREA** is a task-private structure initialized by **BUILD_C_TASK()** that contains the pointers that a task needs to operate; it is defined in the **USER.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory. The **USER_AREA** structure is the first element in the **TASK** structure.

The **Nap()** function in **GETSTART.C** is a simple function that puts the **CalculationTask** asleep:

```
_Q void Nap(void)    // put calculation task asleep
{  CalculationTask.USER_AREA.user_status = ASLEEP;
}
```

This function simply stores the **ASLEEP** constant into the **user_status** variable in the **CalculationTask**'s **USER_AREA** structure. A similar function named **Wakeup()** stores the **AWAKE** constant into **user_status** to wake up the task. We'll see how to use these functions in the next section.

The main Function Gets Us Going

The **main()** function is the highest level routine in the program. Its definition is:

```
void main(void)
// Print starting area and radius, build and activate CalculationTask.
{  int unused_variable; // an example of how warnings are handled!
   InitVars();
   printf("\nStarting condition:");
   Announce();         // print starting values of radius and area
   SetupTask();        // build and activate the CalculationTask
}
```

As you recall, the declaration of the **unused_variable** was inserted to demonstrate how the Control-C IDE highlights the source code line associated with compiler errors and warnings. **InitVars()** performs a runtime initialization of the variables used by the program; this is very important, because compile-time initializations won't ensure that variables are properly initialized after the program has run once, or after the processor is restarted.

After initializing the variables, **main()** announces the starting values of radius and area and then calls **SetupTask()** to build and activate the **CalculationTask**. To execute the program, simply type at your terminal:

```
main←
```

You'll see the following message:

```
Starting condition:
The radius is       0; the circular area is       0.
ok
```

The "**ok**" prompt lets you know that QED-Forth is ready to accept more commands. We have set up a two-task application: the default startup task (named the **FORTH_TASK**) is still running the QED-Forth interpreter, and the **CalculationTask** that we built is running the **CalcForever()** activation routine. At any time we can monitor the current values of radius and area by interactively calling the function:

```
Announce(  )←
```

Remember to type a space after the (character, and you may have to type slowly so that the multitasking program does not miss any of the incoming characters from the terminal. To view the contents of the circular buffer array named **circle_parameters**, type the command:

```
PrintFPArray( circle_parameters)←
```

To suspend the operation of the **CalculationTask**, type:

```
Nap( )←
```

Now notice that successive invocations of:

```
Announce(  )←
```

all show the same values of the radius and area; this is because the **CalculationTask** is no longer updating them. To re-awaken the **CalculationTask**, simply type:

```
Wakeup(  )←
```

To abort the multitasking program altogether and return to a single task running the QED-Forth monitor, you can perform a "**warm**" restart by typing:

```
WARM←
```

The QED-Forth startup message will be displayed.

Of course, if you want to run the program again, you can type main or any of the interactively callable function names at any time. Remember to type

```
WARM ←
```

or

```
COLD←
```

before trying to download another program file; the Handheld can't run multiple tasks and accept a download file at the same time. (Both **WARM** and **COLD** re-initialize the system, but **COLD** performs a more thorough initialization and causes QED-Forth to immediately "forget" the definitions of the C functions that were sent over in the **.DLF** download file).

Autostarting Your Application

You can configure QED-Forth to automatically execute a specified application program after every reset, restart, and ABORT. This makes it easy to design a production instrument based on the Handheld; the instrument will automatically perform its required function when it is turned on or reset.

QED-Forth provides two functions named AUTOSTART and PRIORITY.AUTOSTART that allow you to specify a startup routine. Both write a pattern in memory that instructs QED-Forth to execute a user-specified program. AUTOSTART stores the pattern in EEPROM which is inside the 68HC11 processor chip, and PRIORITY.AUTOSTART stores the pattern near the top of page 4 which is typically in PROM in a final turnkeyed system. The EEPROM-based AUTOSTART function is convenient during program development and debugging, or in the development of one-of-a-kind systems. But because the startup pattern is stored in EEPROM inside the 68HC11, it is cannot be automatically transferred with the application program to a different board.

The PRIORITY.AUTOSTART routine should be used for PROM-based production systems. It installs the startup pattern in PROM, so simply reproducing the PROM and plugging it into any Handheld turns that board into a turnkeyed instrument controller. In other words, the startup instructions are stored in the same PROM as the application program itself.

Let's assume that you want to want to run the main routine every time you turn on, reset, or restart the Handheld. The following command:

```
CFA.FOR main AUTOSTART
```

leaves the extended code field address (cfa) of main on the stack. AUTOSTART then writes a pattern into EEPROM comprising a 16-bit flag (equal to 1357H) followed by the 32-bit extended cfa of the specified startup program. All subsequent resets and restarts will call the specified application program after QED-Forth initializes the system.

To specify the startup vector so that it can eventually reside in PROM, we would execute a different command:

```
CFA.FOR main PRIORITY.AUTOSTART
```

PRIORITY.AUTOSTART writes a pattern starting at 7FFAH on page 4 comprising a 16-bit flag (equal to 1357H) followed by the 32-bit extended cfa of the specified startup program. All subsequent resets and restarts will call the specified application program after QED-Forth initializes the system.

The priority autostart and autostart locations are checked each time QED-Forth executes ABORT, which is called after every reset, COLD or WARM restart, or error. ABORT first checks the prior-

ity autostart location at 7FFAH\4, and if 1357 is stored there it executes the program whose xcfa is stored in the four bytes starting at 7FFCH\4. If the priority autostart pattern is not present, or if the specified priority startup program finishes executing and "returns", ABORT then checks the autostart pattern at AE00H in common memory. If 1357 is stored there it executes the program whose 32-bit xcfa is stored in the four bytes starting at AE02H.

To remove the autostart pattern or patterns, execute:

```
NO.AUTOSTART
```

This command clears the priority startup pattern at 7FFAH\4 and the startup pattern at AE00H.

Summary

Now you've worked through the **GETSTART.C** program in detail. You know how to compile, download and execute programs, perform simple floating point calculations, print formatted strings and numbers to the terminal, dimension and access **FORTH_ARRAY**s in paged memory, define a multitasking application with an interactive terminal interface, and autostart an application. That's pretty good considering that this is your first C program on the Handheld!

Chapter 4

The IDE: Writing, Compiling, Downloading and Debugging Programs

In this Chapter we'll explore the Handheld's tools for writing, editing, downloading and debugging your application program. You'll learn:

- ⇨ *How to efficiently use the editor and compiler to write and compile both short and long programs;*
- ⇨ *Coding and file-naming conventions;*
- ⇨ *How to access the Handheld's onboard funcitons; and,*
- ⇨ *How to interactively debug your programs.*

Writing Programs

Using the Editor/Compiler

In the prior Chapter we introduced the Mosaic IDE and the tools you can use to edit, compile, and download your program. Briefly, the text pad toolbar provides the following buttons:

The *Debug Tool* invokes the compiler and assembler only – it does not produce downloadable code. Use it to quickly check the syntax of a program and find compile errors.

The *Single-Page Compile Tool* performs a standard, single-page memory model build of your program for programs that do not exceed 32 Kbytes of memory. If you see the following warning printed in the compiler output, then you must switch to the multi-page memory model build (Multi-Page Make icon):

```
WARNING: Input section ".doubleword" from 'progname.o11' is not used !
```

The *Multi-Page Compile Tool* invokes the C compiler's multi-page build mode for long programs. Still, each source code file should be less than 32 Kbytes worth of compiled source code or the above warning will be issued and the program will not run.

The Terminal icon launches the communications program, Mosaic Terminal, which is used to download your compiled program to the Handheld.

Sylistic Conventions

Code Comments

At the top of the **GETSTART.C** source code file are some comments that tell what the program does. Single- or multi-line comments can be enclosed in the standard

```
/*    */
```

delimiters. The double-slash

```
//
```

token means that the remainder of the line is a comment. Note that the editor colors all comments differently to make it easy to distinguish comments from source code. C keywords are also colored differently than user-defined routines. You can change the default colors if you like.

Style Conventions

The example programs on your CD-ROM follow several stylistic conventions. Here is a brief summary:

- Macros and constants are spelled with **CAPITAL_LETTERS**.
- Variable names are spelled with **small_letters**.
- Function names use both capital and small letters, with capital letters indicating the start of a new *subword* within the function name. For example:

```
void SaveCircleParameters(void)
```

To minimize the need to skip from one file to another, we have decided not to group all **#define** statements in a header file that is separate from the program being compiled. Rather, the **#define** statements are defined close to where they are used in the program file that is being compiled.

File Naming Conventions

For backward compatibility with DOS and Windows 3.1, all filenames have 8 or fewer characters. C source code files have the .C extension, and header files have the .H extension. When you use the Make Tool utility to compile a source code file with the filename,

```
NAME.C
```

several files with the extensions shown in Table 4-1 are created:

Table 4-1 Files Created by the C Compiler.

FILENAME.ext	Description
NAME.C	Source code text file created by you, the programmer
NAME.A11	Assembled output text file created by C11 compiler
NAME.O11	Object code binary file created by ASM11 assembler

FILENAME.ext	Description
NAME.LCF	Linker command text file created by CC or CCM batch file
NAME.S	S-record (raw download ascii file) created by linker
NAME.DLF	Final download file created by CC or CCM batch file; includes S records and definitions for QED-Forth
NAME.MAP	Map file listing created by linker
NAME.MEM	Symbols map file
NAME.USE	Memory usage summary
NAME.BAK	Backup file sometimes created by the editor

While this list may seem overwhelming, you won't have to worry about most of these files. You'll create your `NAME.C` and `.H` source code and header files in a directory of your choice, run the automated Make Tool by clicking on the Make icon, and send the resulting `NAME.DLF` download file to the Handheld using the Terminal program. In fact, unless you tell it otherwise, the editor's "File" menu will typically show you only files with the `.C` and `.H` extensions ("Source Files"); you won't have to wade through the files with the other extensions. Similarly, the Mosaic Terminal typically lists only files with the `.DLF` extension, so it will be easy to select the download file to send to the Handheld.

Using Function Prototypes

This stylistic convention deserves its own section. We strongly urge that you define or prototype each function before it is called. If the compiler generates a warning that a function has been called without a prototype, we recommend that you check your source code and insert the required function prototype, or move the definition of the function so that it is defined before it is called.

A prototype is a declaration that specifies the function name and the types of its return value and input parameters. For example, the following is a valid function prototype:

```
_Q float CalcArea( unsigned int radius);
```

This declaration specifies that `CalcArea()` is a function that expects one unsigned integer input and returns a floating point value. As discussed below, the `_Q` tags the function as one that is interactively callable during debugging. The `CalcArea()` function can then be defined later in the source code file. If a function is prototyped in one file and defined in another, add the `extern` specifer before the prototype.

You can preface any function prototype with the `_Q` tag if you want to interactively call the function from the terminal. The Handheld's onboard operating system maintains a list, called the *dictionary*, of the names of functions tagged with the `_Q` so that it can recognize them when you send a command line from your terminal.

Prototype and Declare the Parameter Types of Every Function

Defining or prototyping a function before it is called allows the compiler to help find parameter passing errors, and it also prevents unnecessary *promotion* of parameters that can render the code slower and defeat the Handheld's interactive function-calling capability. To avoid unwanted promotion and runtime errors, each and every parameter in the function prototype or function definition must be preceded with a type specifier. For example, leaving out the **`unsigned int`** keywords in the prototype for **`CalcArea()`** above would lead to promotion of the input parameter, possibly resulting in a *runtime error* message from the compiler or linker.

Using function prototypes and definitions that explicitly specify the type of each and every input and output parameter results in more readable and reliable code.

Accessing the Standard (Kernel) Library Functions

The command

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

near the top of the **`GETSTART.C`** file is a preprocessor directive that includes all of the relevant header files for the Handheld, and all of the standard C header files (such as **`stdio.h`**, **`math.h`**, **`float.h`**, **`string.h`**, etc.) We strongly recommend that this statement be placed at the top of each C program file that you write. It gives you access to all of the pre-coded library routines that reside on the Handheld. These routines let you control the A/D converters, digital I/O, serial ports, real-time clock, and many other useful functions. The **`ALLQED.H`** file also gives you access to the Handheld's multitasking and paged memory capabilities, as well as the standard ANSI C library functions including printing and string conversion routines such as **`printf()`** and **`sprintf()`**. Including these files is very efficient; it generates almost no additional runtime code while giving you access to very powerful capabilities.

You can call any of these functions from within your C code. There is one limitation however:

Do not nest functions of the type `_forth`.

Many functions that are callable from C are actually of the **`_forth`** type. This includes functions that are in the kernel on the Handheld, or are part of software distributions such as the Graphical User Interface (GUI) Toolkit. A call to one of these **`_forth`** functions may not be made from within the parameter list of a call to another **`_forth`** function.

There is always a straightforward way of avoiding such nesting of function calls: simply use a variable to hold the required intermediate return value/parameter. For example, if you need to use the **`_forth`** function **`FetchChar()`** to fetch the first character from the extended address returned by the **`_forth`** function **`DisplayBuffer()`** in paged memory, you could execute the following statements:

```
static xaddr buffer_xaddress = DisplayBuffer( );
FetchChar( buffer_xaddress);
```

This code is correct, while nesting the call to `DisplayBuffer()` inside the parameter list of `FetchChar()` would be incorrect.

Initializing Variables

Caution:
RAM-Resident Variables & Arrays Must Be Initialized Within Functions

A common mistake made when creating application programs for embedded systems is the use of *compile-time initialization* for RAM-based quantities such as variables and arrays. While this approach of initializing quantities outside of function definitions may work during program development, it fails when the device goes into production because the variables and arrays are not properly initialized when power is cycled.

Only *run-time initialization*, i.e., initializations that are performed within functions (which are in turn called by the autostart program), will occur reliably in an embedded application.

Even users with battery-backed RAM in their systems should always perform initializations within functions. This approach will avoid hard-to-diagnose field failures that result from corrupted data in a battery-backed RAM that is never re-initialized to valid values.

Feel free to call Mosaic Industries for help with this or other programming issues.

Compiling Programs

Compiling Multiple Source Code Files

When writing large programs it is often useful to break up the program into multiple source code files. The Make Tool allows you to accomplish this in one of two ways.

1. First, you can designate one of your source code files as the primary file, and insert into this file statements of the form,

   ```
   #include "\path\filename"
   ```

   to include the other source code functions, where \path is the standard DOS path specification that allows the file to be located. The primary file **must have a .C extension.**

2. Second, you can split your source code into several files that start with the same sequence of characters, such as:

   ```
   CODE.C    // this is the primary file that you compile
   CODE1.C   // CODE1.C and CODE2.C are subsidiary files that are automatically
   CODE2.C   // compiled and linked when CODE.C is compiled
   ```

All of these files must have the `.C` extension. Then, when you compile `CODE.C` using the Make Tool (by clicking on the Hammer icon), all of the files with a `.C` extension whose name starts with `CODE` (namely, `CODE.C`, `CODE1.C` and `CODE2.C`) will be compiled and linked together. Note that

the Make Tool cannot handle filenames that start with a numeral or filenames that contain the dash character; thus a source code file named `1CODE.C` or `MY-CODE.C` cannot be compiled.

Finally, note that one and only one of the program files must include a function named `main()`.

Using the Interactive Debugger

In the prior chapter, you gained experience using the debugging environment that lets you interactively execute any designated function with input arguments of your own choosing. Now we'll look more closely at the operation of the debugging environment, and explain how to use it to examine and manipulate the values of static variables, Forth Array elements, and memory locations.

The interactive debugging environment conveys several advantages. First, you can test each function of a program individually without changing the `main()` function and recompiling. This saves compilation and download time. Second, the environment makes it easy to test each function with a wide range of input parameters, allowing you to isolate bugs that might otherwise be missed until later in the program development cycle. Such thorough function-by-function testing of a program facilitates more rapid development of reliable programs.

We'll start by learning how to use the interactive environment to examine the values of static variables. The explanation of how this works involves taking a brief high-level look at the interactive QED-Forth language that is built into the Handheld. Understanding how QED-Forth operates will empower you to take full advantage of the debugging capabilities of the Handheld.

Overview of the Forth Language and Programming Environment

The QED-Forth interactive environment makes it easy to examine the contents of static variables. A brief overview of how the Forth language works will help clarify the procedure.

The Forth Data Stack

Forth is a stack-oriented high level language that combines the interactive benefits of an interpreter with the speed of a compiler. Unlike C, FORTH is implemented as a two-stack language. In addition to the return stack that most languages use to keep track of function calls and returns, FORTH has a *data stack* that is used to pass parameters. All arithmetic, logical, I/O, and decision operations remove any required arguments from the data stack and leave the results on the data stack. This leads to *postfix* notation: the operation is stated after the data or operands are placed on the stack. This is the same notation used by Hewlitt Packard's RPN (*reverse polish notation*) calculators.

Unlike C, Forth uses spaces as delimiters to distinguish different keywords and tokens. For example, a C compiler can easily parse the addition expression:

```
5+4
```

as three distinct tokens: 5, +, and 4. But because the above expression was typed without any spaces, Forth would interpret the expression as a single token, assume it's the name of a function, and would try to find it in its dictionary. In Forth the expression must be entered as:

```
5  4  +
```

which includes the required spaces and uses postfix notation to add the numbers and leave the result on the data stack.

To see how this works, we'll talk to the interactive QED-Forth interpreter on the Handheld. To start, enter the terminal now: if the terminal program is already active, click on its window or hold down the "Alt" key and press "Tab" until the terminal announcement appears on your screen. If you haven't started the terminal program yet during this session, double-click on the *Mosaic Terminal* icon to start it up. Connect and power up your Handheld; pressing the Return key should cause QED-Forth's `ok` prompt to appear in the terminal window.

To start, we'll ensure that the current number base is decimal by typing the command,

```
DECIMAL←
```

from the terminal. With each character you type QED-Forth echoes the character in you terminal window. The back arrow in the line above indicates that you pressed the Enter key which sends a carriage return character; but you won't see it as an echoed character on your screen. QED-Forth executes this command when the terminating carriage return is received. Also recall that QED-Forth case-insensitive, so you can freely mix upper and lower case letters. Now we can put some numbers on the QED-Forth data stack by typing :

```
5  7←
```

followed by a carriage return. QED-Forth responds:

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

We have underlined QED-Forth's response for clarity. 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 data stack is a feature of the debugging environment. To disable the stack print, you could execute (that is, type at your terminal) the `DEBUG OFF` command. 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 QED-Forth `*` 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 QED-Forth **-** (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:

```
. ←
```

(that's right, the command is simply a *dot*, the period on your keyboard) which prints the response:

```
30  ok
```

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 the Forth language 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. This interactive function execution has been harnessed to speed development of C programs for the Handheld.

QED-Forth Numeric Printing Functions

There are a variety of QED-Forth printing functions, and some related functions that set the current number base and clean up the data stack. Here is a short list of useful functions that can be executed interactively:

Function	Description
.	Prints a 16 bit signed integer in the current number base
U.	Prints a 16 bit unsigned integer in the current number base
D.	Prints a 32-bit signed long in the current number base
PrintFP	Prints an ANSI-C floating point number
HEX	Sets the number base to hexadecimal
DECIMAL	Sets the number base to decimal
SP!	Clears all items off the stack without printing anything

Each of the printing routines removes a number from the data stack and prints it to the terminal. Because characters are promoted to unsigned **int** in Forth, the **.** (dot) function is also used to print 8-bit character data. The **PrintFP** function was specifically written to display floating point numeric output from C programs, as internally QED-Forth uses a non-ANSI floating point representation for its own floating point numbers.

The default QED-Forth number base after a **COLD** restart is **DECIMAL**. The number base can be changed to hexadecimal by executing **HEX**. All non-floating-point numbers typed at the terminal or printed by QED-Forth are converted using the current number base (corresponding to the most recent execution of **DECIMAL** or **HEX**). Floating point numbers are always converted using the decimal number base.

Displaying the Values of Static Variables

Now that we understand how the Forth data stack works, the procedure for examining variables will make sense. The examples presented here use code from the **GETSTART.C** program that we've already discussed in detail. If you have already downloaded the program, you are ready to go. If your board is presently running a multitasking application and you want to download a new file, type

```
WARM←
```

to stop the program so that a new download file can be accepted.

If you have not yet compiled the **GETSTART.C** program and you want to do the exercises here, first compile it by opening **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES\GETSTART.C** in the TextPad editor, click on the Make Tool, and after the compilation, enter the Mosaic Terminal and use the "Send Text File" menu item to send **GETSTART.DLF** to the Handheld. To run the program, type

```
main←
```

at your terminal – this initializes all the pointers and variables. After typing main, let's type

```
Nap( )←
```

to put the calculation task asleep; remember to type at least one space after the **(**. This stops the variables from being updated in the background.

Let's start by initializing the contents of the radius variable to 5 by interactively executing (typing) from the terminal:

```
SetRadiusAndArea( int 5)←
```

Remember to type at least one space after the (character and after int. This function is defined in **GETSTART.C** as:

```
_Q SetRadiusAndArea( uint r)
{  radius = r;
area = CalcArea( r);
}
```

It assigns the specified input parameter to the unsigned integer **radius** variable, and assigns the corresponding circular area to the floating point area variable.

Now we can check the value of **radius**. The following interactive command places the contents of the integer variable named **radius** on the Forth data stack:

```
int  radius←
```

QED-Forth responds with:

```
ok  ( 1 ) \ 5
```

Because radius is defined as an unsigned integer, we use the unsigned integer printing routine named **U.** (U-dot) to remove the value from the Forth data stack and print it. Type

```
U.←
```

to print the radius an unsigned integer. QED-Forth responds with:

```
5 ok
```

To speed things up, we can type the entire command sequence on one line so that QED-Forth immediately prints the result. Type:

```
int  radius  U.←
```

and QED-Forth responds with:

```
5 ok
```

To interactively examine the contents of the floating point area variable, type the command sequence:

```
float  area  PrintFP←
```

and QED-Forth responds:

```
78.54 ok
```

which is indeed the area of a circle whose radius equals 5.

Extracting the Value Referenced by a Pointer

Sometimes C programs add an additional layer of indirection, referencing a value by means of a pointer. An example of this technique appears in the **GETSTART.C** program in the form of the static variables **radius_ptr** and **area_ptr**; they are defined as:

```
static uint*   radius_ptr;
static float*  area_ptr;
```

In the **InitVars()** function near the end of the program, these pointers are initialized as follows:

```
radius_ptr = &radius;
area_ptr = &area;
```

In other words, **radius_ptr** holds the address of a variable that represents the radius, and **area_ptr** holds the address of a variable that represents the area. Given the **radius_ptr** and **area_ptr**, we want to be able to extract the value of radius and area. The following keywords can be executed interactively to accomplish this:

```
char*      int*      long*      float*
```

Note that there cannot be any spaces before the ***** in each keyword, and there must be at least one space after the ***** and before any subsequent number or variable name.

For example, to print the radius you can type:

```
int*  radius_ptr  U.←
```

The **int*** keyword fetches the 16-bit address from **radius_ptr** and from that location fetches the integer contents. **U.** then prints the answer to the terminal. Similarly, to print the area you can type:

```
float*  area_ptr  PrintFP←
```

The **float*** keyword fetches the 16-bit address from **area_ptr** and from the resulting location fetches the floating point contents. **PrintFP** then prints the result.

Signed versus Unsigned Numbers

Note that the type specifier used above does not specify signed versus unsigned numbers; rather, the printing function determines whether the number is interpreted as signed or unsigned. For example, type the following two command lines from the terminal and see how QED-Forth responds:

```
65535   U.←
65535   . ←
```

In the first instance, QED-Forth prints 65535, while in the second instance, QED-Forth prints -1 (we're assuming that you have not changed the number base to `HEX`). The same binary pattern (in this case, all 16 bits of the number are set) can represent either 65535 or -1 depending on how the number is interpreted and printed. Thus by choosing the printing function, you can control whether a number is displayed as a signed or unsigned quantity.

Summary

In summary, to display the contents of a simple static variable, type a command of the form:

```
type  variable_name  print_function_name
```

where type is one of the following keywords:

```
char      int      long      float
```

To display the contents of a static variable that is pointed to by a pointer, type a command of the form:

```
type      pointer_name      print_function_name
```

where type is one of the following keywords:

```
char*    int*     long*     float*
```

Use Type Keywords To Interactively Call C Functions

The same family of familiar C type-declaration keywords that we used to fetch the contents of variables is also used to facilitate interactive calling of C functions. Recall that these keywords are:

```
char    int     long    float
char*   int*    long*   float*
```

We have seen that these keywords are used in two different contexts while debugging. In the prior chapter we used them to declare the type of an input parameter while interactively calling a function. For example, we can interactively type from the terminal:

```
CalcArea( int 5)←
```

where the `int` keyword is used with the same syntax as an ANSI-C function prototype to declare the input arguments to the called function.

Second, we used the type keywords in this chapter to extract the value from a variable, as in the interactive QED-Forth command

```
int  radius  U.←
```

which prints the contents of the radius variable as an unsigned integer.

These two contexts for the use of the **int** keyword are related. For example, to calculate the area corresponding to the current value of the radius variable, we can interactively execute:

```
CalcArea( int radius)←
```

and QED-Forth prints the resulting floating point area in its summary of the return value. The **int** keyword serves two complementary purposes here: it tells QED-Forth that the input parameter is a 16-bit integer, and it extracts the value of the radius variable so the variable is *passed by value*.

When interactively calling a function, all parameters that are passed by value should be preceded by the appropriate type keyword. However, when passing the address of a variable or a structure, simply state the variable or structure name without any type specifiers or **&** (address-of) operators.

For example, the function prototype for the **DimAndInitFPArray()** function in **GETSTART.C** is:

```
Q void DimAndInitFPArray(float val,int rows,int cols,FORTH_ARRAY* array_ptr)
```

and the program includes the array declaration:

```
FORTH_ARRAY circle_parameters;
```

which declares **circle_parameters** as a **FORTH_ARRAY** structure in memory. As we shall see, executing (typing) the name **circle_parameters** in QED-Forth leaves the address of the array structure on the stack, so there is no need for additional type declarators or **&** operators. Thus to interactively dimension the array to have 10 rows, 2 columns and a initialization value of 12.34, we type from the terminal:

```
DimAndInitFPArray( float 12.34,int 10,int 2,circle_parameters)←
```

To verify that this worked, you can execute:

```
PrintFPArray( circle_parameters)←
```

which displays the contents of the newly initialized **circle_parameters** matrix.

Displaying the Values of FORTH_ARRAY Elements

The same type specifier keywords that let you examine static variables can also be used to examine any specified element in a two-dimensional **FORTH_ARRAY**. The syntax is parallel to what we have already used; the difference is that we now append the row and column indices in square brackets after the array name to specify which element should be fetched.

For example, recall that **circle_parameters** is a **FORTH_ARRAY** that is dimensioned to hold 10 rows and 2 columns of floating point data. To print the contents of the first element in the array at [row=0, col=0], we type:

```
float  circle_parameters[ 0, 0]  PrintFP←
```

and QED-Forth prints the result. While this array notation is not exactly like the standard C syntax, it is straightforward. To print the element whose row index is 5 and whose column index is 1, type:

```
float  circle_parameters[ 5, 1]  PrintFP←
```

As you might expect, there must not be a space before the `[` character, and there must be at least one space after the `[` character. This is because

```
circle_parameters[
```

is defined as a space-delimited QED-Forth function in the **GETSTART.DLF** file, as explained later in this chapter.

All of the keywords that we learned about above can be used to fetch the contents of appropriately dimensioned arrays. Arrays that are dimensioned to hold character, integer, long, or float data are accessed using the **char**, **int**, **long** and **float** keywords, respectively, in front of the array name. If for some reason you use a **FORTH_ARRAY** to hold 16-bit pointers , the **char***, **int***, **long*** and **float*** keywords can be used in a manner exactly analogous to the description in the earlier section of this chapter.

Assigning Values to Static Variables and FORTH_ARRAY Elements

You can interactively change the contents of any static variable or FORTH_ARRAY element using the following assignment keywords:

```
=char    =int    =long    =float
```

Each of these keywords expects to be preceded by the address of a variable or **FORTH_ARRAY** element, and expects to be followed by a valid number, variable name, or **FORTH_ARRAY** element specifier. As expected, the value of the right hand side is assigned to the variable or array element on the left hand side of the assignment expression.

For example, to change the current value of radius to 22, simply type:

```
radius  =int  22↵
```

This syntax was designed to be similar to a C statement that assigns the value 22 to the radius variable. As you might guess, **=int** is a single keyword defined in QED-Forth, so there cannot be any spaces between **=** and **int**. Similarly, the other tokens in the expression must be separated by spaces; thus there is at least one space after **radius** and at least one space before **22**.

To set the current value of the floating point area variable to 1520. type:

```
area =float 1520.↵
```

To assign the current value of the area variable to element [0, 1] in the **FORTH_ARRAY** **circle_parameters**, you can execute:

```
circle_parameters[ 0,1] =float  area↵
```

To check that these operations actually worked, we can execute the following commands to examine the contents of the affected variables and array elements:

```
int radius  U.↵
float  area  PrintFP↵
float  circle_parameters[ 0,1]  PrintFP↵
```

Under the Hood of the QED-Forth Interactive Debugger

This section is for the curious among you; you need not read or understand this section to use the QED-Forth interactive debugger. However, it will give you additional insight into the debugging environment.

Variable Declarations

In the example above, `radius` is defined in the `GETSTART.DLF` download file as a QED-Forth constant whose value is the address of the radius variable. To see for yourself, use you editor to open the `GETSTART.DLF` file. Select "Open" from the editor's "file" menu, set the "List Files of Type" option to either "Text Files" or "All Files", and double click on `GETSTART.DLF` in the `\MOSAIC\DEMOS_AND_DRIVERS\MISC\C_EXAMPLES` directory. The top portion of the file is the hexadecimal dump of the compiled C code in the Motorola S2 record format. Near the bottom of the file you'll see some `CONSTANT` declarations. Among them is the declaration:

```
008E03 CONSTANT radius
```

which defines radius as a QED-Forth constant that places the hexadecimal value `8E03` on the stack. You can verify this by clicking on the Terminal window and typing:

```
HEX radius  U.←
DECIMAL←
```

from the terminal. This command sequence instructs QED-Forth to print the hexadecimal address of the radius variable, and then return to decimal base. Note that if you want to pass the address of the radius variable as a parameter to a function (also known as *passing a pointer* or *passing by reference*), you leave out the `int` keyword before `radius` in the parameter list.

The keyword `int` is actually a QED-Forth function that examines the next token in the input stream; if it is already a number such as 5 or 3.2, `int` simply converts it to the nearest integer. If the next token is a variable address (such as `radius`), `int` extracts the 16-bit contents stored at the address. To see this behavior for yourself, try the following commands at your terminal:

```
int  5  U.←
int  5.45  U.←
int  radius  U.←
```

These three statements all yield identical results if the value of `radius` is still 5.

Function Declarations

Returning to the `GETSTART.DLF` file that you opened in the editor, scroll to the area just above the list of `CONSTANT` definitions and you will see a set of lines starting with the `:` (colon) character. In Forth, the `:` character marks the start of a new definition (function or subroutine), and the `;` (semicolon) marks the end of the definition. These are the *function definitions* that tell QED-Forth the names and execution addresses of each function in `GETSTART.C` that was preceded by the `_Q` declarator. Among these functions you will find some familiar ones including:

```
SetRadiusAndArea(
CalcArea(
DimAndInitFPArray(
```

```
PrintFPArray(
```

The body of each of these Forth definitions defines the compilation address and invokes the routine `CALL.CFN` (meaning *call-C-function*). `CALL.CFN` accepts an optional list of comma-delimited parameters terminated by a closing `)` and then sets up the stack frame and calls the function.

So when you type the interactive command

```
CalcArea( int radius)←
```

with a terminal enter key, here's what happens:

1. When QED-Forth accepts the carriage return, it starts interpreting the command line that has been entered. It looks for the first space-delimited token, and it finds the token:

   ```
   CalcArea(
   ```

2. It looks in its dictionary, and sure enough, it finds that this token has been defined; the definition was compiled when the `GETSTART.DLF` download file was sent to the Handheld.

3. When QED-Forth executes the `CalcArea(` token, it executes the `CALL.CFN` routine which starts looking for a terminating `)` character, and processes any tokens that are present.

4. The next space-delimited token found is `int`, which looks for the next token (in this case, `radius`). Because `radius` is not a number, `int` assumes that it is a variable and extracts the 16-bit contents from the address that is left on the Forth data stack by `radius`. The contents are left on the Forth data stack.

5. The terminating `)` is found, so the `CALL.CFN` routine pushes the items on the Forth data stack onto the C stack in the proper order to make a legal C stack frame, and then executes the `CalcArea()` function as defined in the C program at the specified execution address.

6. When the `CalcArea()` function returns, QED-Forth traps its return value from the 68HC11's registers and prints the value using integer and floating point formats.

FORTH_ARRAY Declarations

Near the bottom of the `GETSTART.DLF` file you can find the definition of `circle_parameters[` that facilitates examining and modifying any element of this array. The QED-Forth definition is:

```
: circle_parameters[
  circle_parameters  DO[]
;
```

As described above, the `:` character marks the start of a new definition, and the `;` marks the end of the definition. The body of the definition is simple: the constant `circle_parameters` leaves the base address of the `FORTH_ARRAY` structure on the stack, and `DO[]` does the rest of the work. `DO[]` is defined in the QED-Forth kernel; it searches for a row index followed by a comma, and a column index followed by a terminating `]` character. Then it passes the specified row, column, and array parameter field address to the Forth function named `[]` (*brackets*) which places the 32-bit extended address of the array element on the stack. This extended address can be used as the argument to the familiar keywords that we have discussed such as `char`, `int`, `long`, `float`, `=char`, `=int`, `=long`, `=float`, etc. Thus all of the following are legal debugging commands:

```
float circle_parameters[ 3,0]  PrintFP←
circle_parameters[ 2,1] =float  area←
circle_parameters[ 5,0] =float 345.←
```

Some of you may have noticed that `CalculationTask[` is also declared to QED-Forth as a potential `FORTH_ARRAY` in the `GETSTART.DLF` download file; yet we know that `CalculationTask` is a task identifier, not a `FORTH_ARRAY`. The reason for this is that Make Tool always declares the last variable allocated in the common RAM as a potential `FORTH_ARRAY`; it does this because there it can't determine the allocated size of the last variable. The extra definition of `CalculationTask[` does no harm (as long as we don't try to use it improperly).

Summary

The Make Tool calls the `QCC.EXE` executable program to create the QED-Forth debugging declarations that appear at the bottom of the `.DLF` download file. This program has to decide whether each compiler symbol in the `.OUT` file is a callable function, a variable, or a `FORTH_ARRAY`. The Make Tool identifies callable functions by detecting the `_pascal?` tag that the compiler places there in response to the `_Q` specifier, and in response prints the `functionname(` definition into the `.DLF` file. The Make Tool identifies variables by detecting whether the corresponding address lies in the common RAM area, and in response prints a QED-Forth `CONSTANT` declaration into the `.DLF` file. Finally, it tentatively identifies `FORTH_ARRAY`s by checking the size of each variable; if there are exactly 18 bytes allocated to one item in the common RAM, it decides that the associated name should also be declared as a `FORTH_ARRAY` by printing the `name[` definition in the `.DLF` download file. To be safe, the Make Tool always declares the last variable as a `FORTH_ARRAY` because it cannot be sure of its allocated size.

Other Useful QED-Forth Functions

QED-Forth is a complete language that includes over a thousand pre-defined functions, all of which reside in ROM on the Handheld. Many of these functions are declared in the header files in the `\MOSAIC\FABIUS\INCLUDE\MOSAIC` directory, and so are callable from C. The names and descriptions of these functions are detailed in the *Control C Glossary* in the documentation package. But there are also additional routines described there that are useful while debugging; these allow you to:

- Modify the contents of EEPROM on the 68HC11 processor.
- Dump the contents of a specified region of memory in hex and ascii format using the `DUMP` command.
- Specify a new baud rate for the serial port to speed downloads using the `BAUD1.AT.STARTUP` command.
- Configure the Handheld to execute a specified program each time a reset or restart occurs using the `AUTOSTART` or `PRIORITY.AUTOSTART` command.
- Dump out a replica of the board's program memory space in Intel Hex or Motorola S2 record format to archive your production code.

In sum, the versatile QED-Forth language enhances the power of Control C by providing many operating system functions as well as an interactive debugging environment that speeds program development and testing.

The Handheld Kernel vs. Prior Kernels

The Handheld, QCard and QScreen products use a QED-Forth operating system kernel denoted as V4.4x, where the 'x' may take on any numeric value. There are several minor differences between V4.4x and the V4.0x kernel used on the QED Board, Panel-Touch Controller, and QVGA Controller products. Briefly, five functions have been added to the V4.4x kernel, and 19 device functions have been removed. The removed functions are device drivers associated with hardware that is not implemented on the Handheld/QCard/QScreen products. In addition, the V4.4x kernel boots up at a default serial baud rate of 19,200 baud, compared to 9600 baud on prior kernels.

Table 4-2 lists the new functions, and Table 4-3 lists the removed functions. Descriptions of the new functions are provided in the glossary below and in the function reference document that accompanies the Handheld, QCard and QScreen products.

Table 4-2 Functions added to V4.4x kernel

C Name:	Forth Name:
Buffer_To_SPI( )	BUFFER>SPI
Bytes_To_Display( )	BYTES>DISPLAY
Calc_Checksum( )	CALC.CHECKSUM
Clear_Boot_Vector( )	CLEAR.BOOT.VECTOR
Set_Boot_Vector( )	SET.BOOT.VECTOR

C programmers must include the files named v4_4update.c and V4_4update.h to gain access to the five new functions. These files are located in the

\Mosaic\Fabius\Include\Mosaic\v4_4Update

directory in the software distribution CD. Simply #include both the v4_4update.h and v4_4update.c files in one of your source files, and also #include v4_4update.h in any other source files that use these new kernel routines.

Table 4-3 Summary of functions deleted from V4.4x.

C Name:	Forth Name:
InitPIA( )	INIT.PIA
PIAStore( )	PIA.C!
PIAFetch( )	PIA.C@
PIAChangeBits( )	PIA.CHANGE.BITS
PIAClearBits( )	PIA.CLEAR.BITS
PIASetBits( )	PIA.SET.BITS
PIAToggleBits( )	PIA.TOGGLE.BITS

C Name:	Forth Name:
ClearHighCurrent( )	CLEAR.HIGH.CURRENT
SetHighCurrent( )	SET.HIGH.CURRENT
PPA_ADDRESS	PPA
PPB_ADDRESS	PPB
PPC_ADDRESS	PPC
FastSetDAC( )	(>DAC)
FastAD12Multiple( )	(A/D12.MULTIPLE)
FastAD12Sample( )	(A/D12.SAMPLE)
SetDAC( )	>DAC
AD12Multiple( )	A/D12.MULTIPLE
AD12Sample( )	A/D12.SAMPLE
InitAD12andDAC( )	INIT.A/D12&DAC

Summary of Modified Memory Map Functions

The Kernel's internal memory map functions have been modified to be aware of the Handheld, QCard and QScreen's memory. In the "standard map", the Handheld has flash at pages 4-7 that swaps with RAM on parallel pages 1-3, plus flash at hex pages 10-17 that swaps with RAM on parallel hex pages 18-1F. In the "download map", flash and RAM are swapped: flash is present on pages 1-3 and 18-1F, and RAM is present on pages 4-6 and 10-17. The C development environment transparently handles the loading of program code into flash, so C programmers typically do not have to be concerned with these issues.

Chapter 5

Making Effective Use of Memory

The Handheld contains plenty of memory for your application. The standard Handheld provides 512K Flash for your program, and 128K of battery-backed RAM for variable storage. An enhanced memory option is available to provide up to 1 MB Flash and 512K RAM. EEPROM is also available for long-term nonvolatile data storage. This chapter:

- ⇨ *Provides the memory map – that is, the addressable locations for each type of memory;*
- ⇨ *Shows you how to address memory using the C programming language; and,*
- ⇨ *Shows you how to efficiently use memory by using arrays.*

The Handheld's Memory Map

In its standard configuration the Mosaic Handheld hosts 512K Flash and 128K RAM.

Expanded Memory Option

The total RAM on the standard configuration, or "128K", Handheld is actually 129K, comprising a 128K SRAM memory chip and 1K of onboard RAM on the 68HC11 processor. An expanded memory option is available that provides a 512K SRAM in place of the 128K SRAM. The default memory option's 128K memory chip may be battery backed using a Li battery if the real time clock option is also chosen. If the expanded memory option is chosen, the memory can be battery backed by the NiMH battery pack.

Pages Expand the Processor's Addressable Memory

The Handheld uses a paged memory system to expand the processor's 64Kbyte address space to 2 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 a 6-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 heap memory manager and array routines allow you to think of the paged memory as contiguous memory for data storage. The operating system automatically handles function calls and returns among the pages. There is very little speed penalty associated with changing pages.

Figure 5-1 below illustrates the memory map of the Handheld. 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 64 pages of memory selected by an 6 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 64 pages (32K per page) occupy the paged memory at addresses 0x0000 to 0x7FFF. The first 10 pages of the Handheld's 20 pages of installed memory are shown in the figure. The expanded memory option provides a total of 48 pages of memory.

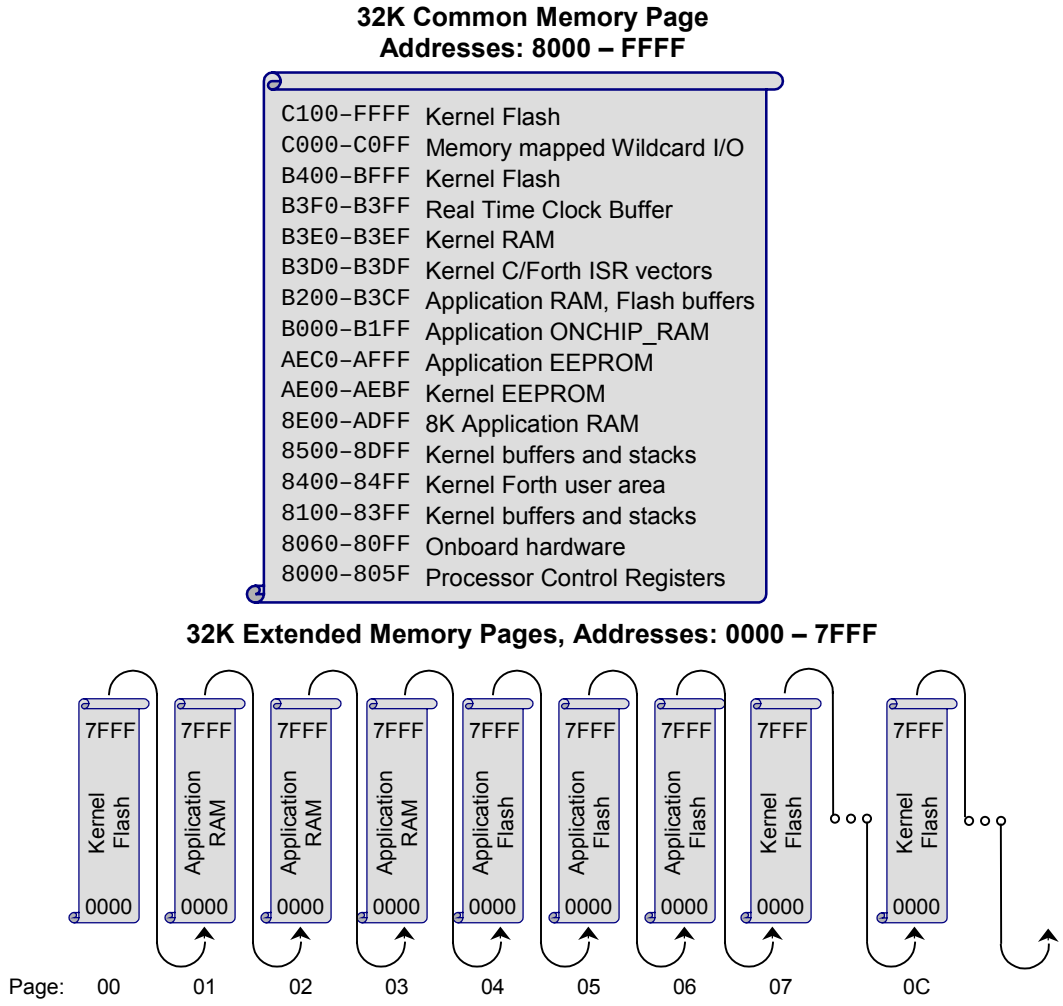


Figure 5-1 The paged memory space of the Handheld.

Table 5-1 Partition of Flash and RAM among Kernel and Application Functions

Function	Size	Memory Page	Memory Address	Physical Location
Kernel Flash (64K)				
Kernel	32K	00	0000 – 7FFF	Handheld Flash Socket 1
Kernel	1K	0E	Various	Handheld Flash Socket 1
Kernel	19K	common	B400 – FFFF	Handheld Flash Socket 1
Kernel	12K	0F	0000 – 2FFF	Handheld Flash Socket 1
GUI Toolkit	2K	7	0000 – 1F4A	Handheld Flash Socket 1
Kernel RAM (4K)				
Kernel	3.5K	common	8000 – 8DFF	Handheld RAM
Kernel	0.5K	0E	Various	Handheld RAM
RTC	48 bytes		B3D0 – B3FF	68HC11
GUI Toolkit	75 bytes		8E00 – 8E4B	Handheld RAM
GUI Toolkit Heap Programmer specifies location in RAM dedicated to the GUI Heap	variable	variable	variable	Handheld RAM
Flash write-buffer, available for an application at run-time	464 bytes	common	B200 – B3CF	68HC11
Kernel EEPROM (192 bytes)				
Kernel	192 bytes	common	AE00 – AEBF	68HC11
Application Flash (448K or 960K)				
Application code	96K	04 – 06 (01– 03)	0000 – 7FFF	Handheld Flash Socket 1
Kernel Extensions	96K	07,0C,0D	0000 – 7FFF	Handheld Flash Socket 1
Application code and data	256K	10 – 17 (18 – 1F)	0000 – 7FFF	Handheld Flash Socket 1
Application code and data available with the extended memory option	512K	20 – 2F	0000 – 7FFF	Handheld Flash Socket 2
Application RAM (125K to 509K)				
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, some used by GUI Toolkit, optionally battery-backed	8K	common	8E00 – ADFF	Handheld RAM
Arrays and heap memory, programmer specifies location of heap used by GUI Toolkit, optionally battery-backed	20K	0F	3000 – 7FFF	Handheld RAM
Arrays and heap memory, optionally battery-backed	96K	01 – 03 (04 – 06)	0000 – 7FFF	Handheld RAM
Arrays and heap memory, available with extended memory option	128K	08-0B	0000 – 7FFF	Handheld RAM
Arrays and heap memory, available with extended memory option	256K	18 – 1F (10 – 17)	0000 – 7FFF	Handheld RAM
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 memory during program download, i.e., the *download* memory map.
2. Only the 128K RAM on the Handheld may be battery-backed. The Handheld that comes with the Starter Kit has 512K of non-battery backed RAM.
3. Application code is free to reside on pages 04-06 and 10-17.
4. Addresses from 8000 through FFFF comprise common memory that is visible to code on all pages.

Table 5-1 on page 63 shows the distribution of Flash and RAM among kernel services and your application program, providing addresses and page numbers for each memory block. The Handheld's onboard operating system uses some of the Flash, RAM and EEPROM, but most remains available for your application program.

The Flash, RAM and EEPROM are distributed among the common memory, which is limited to 32K, and the paged memory, which is limited only by available hardware. The following sections discuss the important addresses and pages in these two memory areas.

Kernel vs Application Memory Space

Of the Handheld's 512K of Flash memory, 448K 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 129K of RAM, 125K is available for application program use. In the "standard map", the pages are distributed among Kernel and Application Flash and RAM as: Kernel Flash (page 00 and parts of the common page, 0E and 0F); Kernel RAM (parts of the common page, 0E and 0F); Application Flash (pages 04 to 07, 0C to 0D, and 10 to 17); and Application RAM (pages 01 to 03 (and pages 08 to 0B and 18 to 1F with the expanded memory option).

Most of the Flash memory is available as two blocks of contiguously addressable memory on pages 04-06 and 10-17. RAM for your application program is also available in the paged memory in one contiguously addressable chunk, filling pages 01-03. 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.

The Standard and the Download Memory Maps

The Handheld has an alternate memory map called the "download map" that is used to facilitate the loading of code. To download code into the board the RAM and Flash addresses are swapped. Code is then downloaded into RAM that is addressed where Flash usually resides. Then the code is copied into the Flash, and the RAM and Flash addresses swapped back to the standard map.

In the download map, hex pages 4-7 and 10-17 are RAM, while hex pages 1-3 and 18-1F are Flash. In other words, the blocks of RAM at hex pages 1-3 and 18-1F in the standard map become Flash in the download map, and the blocks of Flash at hex pages 4-6 and 10-17 in the standard map become RAM in the download map.

C programmers never have to worry about the download map; the C development environment transparently handles the downloading process and automatically loads the code into flash memory for you.

The functions ALL.TO.FLASH and ENABLE.DOWNLOAD make it easy for Forth programmers to manage the download process for compiled applications up to 96 Kbytes long.

Common Memory

The *common memory* is also partitioned between the operating system and application program. Table 5-2 on page 66 shows the common memory addresses used by the operating system, and in boldfaced type those addresses available to the application program.

Referring to Table 5-2 on page 66, there are four unencumbered common memory areas your application can use:

0x8E00 – 0xADFF: These 8 Kbytes are available for your application. The Control-C compiler uses this area for static variables, arrays, task areas, etc.

0xAEC0 – 0xAFFF: 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.

0xB000 – 0xB1FF: The 68HC11's 1 Kbyte of on-chip RAM is located at 0xB000 – 0xB3FF.

- Locations 0xB000 – 0xB1FF are totally unencumbered and always available for your use (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 0xB200 – 0xB3CF are reserved for the flash programming routines.
- Locations 0xB3D0-0xB3DF are reserved for support of Forth interrupt service routines called from C-compiled programs; and,
- Locations 0xB3F0 – 0xB3FF are reserved for the real-time clock buffers.

0xB200 – 0xB3CF: These 464 bytes are also available for run-time variable storage, but are used by the kernel during download as a Flash buffer, or anytime that TO.FLASH is used, even at run time.

The remainder of the common memory area is used by either the processor or the kernel. 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. Locations 0xB3D0-0xB3EF are reserved for support of Forth interrupt service routines called from C-compiled programs, and locations 0xB3F0 – 0xB3FF are reserved for the real-time clock buffers. Locations 0xB400 – 0xBFFF and 0xC100 – 0xFFFF contain kernel code. A *notch* at 0xC000 – 0xC0FF is not decoded by any onboard devices, but is used for addressing WildCards. The Handheld does not bring out the processor's address/data bus for do-it-yourself

memory mapped peripherals; instead, a direct interface to WildCard modules at addresses 0xC000-C0FF on pages 0-7 provides for versatile off-the-shelf I/O expansion.

Table 5-2 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
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 and bold entries indicate memory available for application programs.

Locations 0xB400 – 0xBFFF and 0xC100 – 0xFFFF contain kernel code. A *notch* at 0xC000 – 0xC0FF is not decoded by any onboard devices; this location is used to communicate with the Wildcards.

Paged Memory

Occupying this memory space are 512K or 1MB Flash and 129K or 513K RAM. Of the Handheld's maximum of 1M of Flash memory, all but 64K is available for your application program and data storage. The 64K of flash 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 513K of RAM, 509K is available for application program use. A smaller memory option of 128K RAM is available that allows the RAM to be battery backed.

Table 5-1 illustrates the partitioning of the onboard memory between the operating system (Kernel) and your application functions. Most of the Flash memory is available as three blocks of contiguously addressable memory on pages 4-6, 10-17, and 20-2F. RAM for your application program is also available in three chunks, filling pages 1-3, 8-B, and 18-1F. 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.

Addressing Memory in C

Although 6-bits are sufficient to address the 64 possible pages, the page is padded out to a more standard 16-bit date type so that the full address, lower 16 bits plus 16-bit page, occupies 32 bits. We'll refer to this full address as an *xaddress* (32-bit extended addresses). Three macros are available in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC\types.h** file to simplify the manipulation of xaddresses and their constituent 16-bit addresses and pages. These C macros are:

```
TO_XADDR
XADDR_TO_ADDR
XADDR_TO_PAGE
```

Multi-page C programs rely on a "page change" routine in the common kernel memory to call functions on other pages. Unlike the Forth compiler, the C compiler is not "page smart", and does not know at compile time whether a page change is needed. In fact, page changes are rarely needed, because most functions call other functions that are located on the same page or in common memory. Calls to functions on the same page or to common memory take only 11.5 or 13.75 microseconds, respectively, while function calls to other pages require just under 49 microseconds. Because page changes are rare, the average execution speed of multi-page C applications is not significantly impacted by the need for page changes.

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 Handheld'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 pre-coded flash programming routine. Programming time is approximately 60 milliseconds per kilobyte.

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

```
DownloadMap()  PageToFlash()  PageToRam()
StandardMap()  ToFlash()      WhichMap()
```

The FLASH programming functions use a buffer in the 68HC11's on-chip RAM starting at hex addresses B200-B3CF. The remaining on-chip RAM at B000 to B1FF is available to you. Also, because FLASH programming is generally not done at run-time, you can still use the Flash buffer for run-time variables.

Software Development Using Flash Memory

Because code cannot be downloaded or compiled directly into flash memory, the flash memory map implements *page swapping* to provide a mechanism for getting the compiled code into the flash memory. There are two page-swap modes: one is called the Standard Map and the other is called the Download Map. As the names suggest, the Standard Map is used during run-time, and the

Download Map is used during downloading C-compiler S-records from the PC to the Handheld. The two maps are very similar; the effect of changing from the Standard to the Download map is to swap the locations of pages between the Flash and the RAM.

Table 5-4 Addressing the Flash and RAM in *Standard* and *Download* Memory Maps

	Flash Pages		RAM Pages	
Standard Address Map	04 – 06	10 – 17	01 – 03	18 – 1F
Download Address Map	01 – 03	18 – 1F	04 – 06	10 – 17

In normal operation the Flash memory is addressed on pages 04-06 and 10-17, and the RAM is addressed on pages 01-03 and 18-1F. During download their addresses are swapped, so that the Flash is addressed at pages 01-03 and 18-1F and the RAM at pages 04-06 and 10-17.

To see how it works let's consider a hypothetical download. Suppose you have compiled code intended to load into the Flash and run from it at addresses on page 4. Automated commands contained in the download file establish the download map, load the code into RAM, transfer the code to flash, and re-establish the standard map. In this case, the download file would:

1. Swap the addresses of the RAM and Flash (by executing the command `DOWNLOAD.MAP`) so that the RAM is now addressed on page 04;
2. Download the code to its proper addresses on page 04;
3. Copy the code (using the command `PAGE.TO.FLASH`) from page 04 into the Flash addressed on page 01; then,
4. Swap the RAM and Flash addresses back (by executing the command `STANDAD.MAP`) so that the Flash is now addressed on page 04, and the RAM on page 01 is available for run-time use by your program.

You can now execute your programs on page 04.

The Control-C download file does all this for you so you don't need to worry about the details. But if you're interested, just peruse the download file in the editor where you'll see the commands it uses to manage memory during the download process.

You can now run the program by typing

```
MAIN←
```

or any function name that was preceded with the _Q designator. (By the way, the _Q does not compromise performance in any way; it simply makes it possible for the PC-resident batch routines to send out the execution addresses of the designated functions to the Handheld to simplify debugging).

Locating Nonvolatile Data in EEPROM

The Handheld's built-in EEPROM provides an ideal place to store calibration constants or other data that must be changed from time to time, but that must be retained even when power is removed. The EEPROM (Electrically Erasable Programmable Read-Only Memory) can be modified up to 10,000 times before it loses its ability to retain data. The **ANALOGIO.C** file presents an example of how to locate a static "variable" in EEPROM.

You can read from EEPROM locations just as you would from any other memory location. But writing to them is done using special kernel functions. These functions take approximately 20 milli-seconds per byte to reprogram the EEPROM cell. The EEPROM variable should be declared as an un-initialized static variable; these are located by the linker in the "data" section, which normally points to system RAM where normal variables are stored. By following the syntax presented here, you can relocate the data section to point to EEPROM while defining the EEPROM variables, and then restore the data section to its standard RAM location. To define an EEPROM variable, use the following code:

```
#pragma  option data=.eeprom  // put the following variables in eeprom
static uchar numsamples;
static int  nonvolatile_int;
static float calibration_value;
#pragma  option data=.data  // restore the data area to RAM
```

The **#pragma** statements are pre-processor directives that are interpreted by the linker. In the code fragment above, we located three nonvolatile variables in EEPROM; note that we did NOT include initializers in the declaration statements. Initializers don't make any sense for EEPROM variables, because special functions must be called to store values into EEPROM, so initialization can't be accomplished by placing initialization data in the download file. These EEPROM variables must be initialized programmatically at run-time.

To store data into the EEPROM variables, use the following functions which are declared in the **XMEM.H** file in the **\FABIUS\INCLUDE\MOSAIC** directory:

```
void StoreEEChar(char value, char* addr)
void StoreEEInt(int value, int* addr)
void StoreEELong(long value, long* addr)
void StoreEEFloat(float value, float* addr)
```

To learn how to interactively modify the contents of EEPROM variables, read the glossary entries for these functions in the Control-C Glossary.

These EEPROM storage functions are easy to use. For example, to store the value 123 into the character variable **numsamples**, you would place the following statement in your program:

```
StoreEEChar( 123, &numsamples);
```

To avoid wearing out the EEPROM by executing unneeded write cycles, these functions check whether each EEPROM byte already holds its specified contents. If so, the write is not performed. Thus there is no penalty for redundant execution of commands that initialize particular locations in EEPROM.

While EEPROM variables *must* initialized programmatically at run-time the first time they are used, they don't need to be re-initialized each time the processor starts up because the nonvolatile EEPROM retains the data. Even so, initializations can be performed every time the processor starts up, with no adverse effects on the life span of the EEPROM. For example, initialization code in an autostart routine could execute ATTACH functions to ensure that all needed interrupt vectors are properly initialized each time the processor restarts. If the EEPROM cells have been corrupted for some reason, the ATTACH command installs the correct contents, but if the specified interrupt vector information is already in the EEPROM, the memory cells are not needlessly rewritten.

> ***Interrupts are disabled during writes to EEPROM***
>
> All of the EEPROM storage routines globally disable interrupts while each EEPROM byte is being programmed, and it takes 20 milliseconds to program each byte. Thus you should avoid storing values in EEPROM while time-critical events are being serviced by interrupts.
>
> For experts and the curious: Interrupts are disabled during stores to EEPROM because QED-Forth vectors all interrupts via the EEPROM, and the 68HC11 hardware does not allow any EEPROM cells to be read while a single EEPROM cell is being written to. Thus if an interrupt occurs while one of the EEPROM storage functions is writing to EEPROM, the interrupt will not be able to read the instruction code in the interrupt vector. Disabling interrupts prevents this error, but the interrupt service is delayed until the EEPROM write is finished.

Write-Protecting EEPROM

It is possible to write-protect locations within the EEPROM to ensure the integrity of calibration constants or other vital information. This is done using the EEPROM block protect register named **`BPROT`** (MC68HC11F1 Technical Data Manual mc68hc11f1.rev3.pdf, p.4-13). Four blocks of size 32, 64, 128, and 288 bytes may be individually protected by storing an appropriate configuration value to **`BPROT`**. The contents of the **`BPROT`** register may be changed using the C function **`InstallRegisterInits`**; please consult its glossary entry for details.

To make a turnkeyed application maximally "bullet-proof" and fail-safe, consider using the **`BPROT`** register to protect the first three blocks in the EEPROM totaling 224 bytes. This protects the on-board kernel's configuration region (the first 32 bytes in EEPROM) plus the interrupt vectors (the next 160 bytes in EEPROM) plus an additional 32 bytes available the programmer. The remaining 288 bytes of EEPROM then remain available for modification by the application program.

Using C Arrays and Forth (Kernel) Arrays

Storing Data Acquisition Results in C Arrays and Forth Arrays

Programs written in Control-C use space in common memory to store variables. You may store simple variables or arrays of variables there using standard C syntax. However, common memory is a limited to approximately 9K. It can get used quickly in multitasking systems because each task requires a task area of about 1K. Consequently, the programmer may require access to additional

RAM. Access is provided through the use of Kernel Arrays, also called Forth Arrays. Using Forth Arrays you may dynamically dimension arrays of virtually any size in the extended address space – and their memory allocation is automatically handled by the kernel's heap memory manager.

The code presented in the sample program **ANALOGIO.C** uses a C array and a **FORTH_ARRAY** to store the results of multiple A/D conversions. This section uses that code as an example to discuss some interesting features of both C Arrays and **FORTH_ARRAY**s.

Declaring a C Array

The use of C arrays is discussed in detail in all standard C texts. In this program, the one-dimensional 16-element character array named **results_8** is declared and allocated in RAM using the statement:

```
uchar results_8[DEFAULT_NUMSAMPLES];
```

where **DEFAULT_NUMSAMPLES** is a constant equal to 16. The arrays are easy to use. For example, the following C statement assigns the last element in the array to a static variable named **my_variable**:

```
my_variable = results_8[15];
```

To see another simple example that demonstrates how C arrays are accessed, look at the **InitAnalog()** function in the **ANALOGIO.C** file. The **results_8** array is zeroed by executing the following statement:

```
for(i=0; i< DEFAULT_NUMSAMPLES; i++)
results_8[i] = 0;          // zero the array
```

Note that this array is dimensioned and allocated by the compiler and linker. In contrast, **FORTH_ARRAYS** are dimensioned and allocated dynamically by the run-time program itself.

Converting a 16 Bit Address to a 32 Bit xaddress

The **AD8ToCArray()** function that we just used provides an interesting example of type conversion. The definition of the function is:

```
_Q void AD8ToCArray( int channel)
{  EXTENDED_ADDR buffer;
buffer.sixteen_bit.addr16 = results_8;
buffer.sixteen_bit.page16 = 0;
AD8Multiple(buffer.addr32,0,DEFAULT_NUMSAMPLES,channel);
}
```

The purpose of **AD8ToCArray()** is to properly call **AD8Multiple()** which is defined in the **ANALOG.H** file. **AD8Multiple()** is optimized to use a **FORTH_ARRAY** buffer, and so expects a 32 bit buffer xaddress instead of a simple 16 bit buffer address. To convert the simple 16 bit address returned by **results_8** into a 32 bit extended address, we take advantage of the **EXTENDED_ADDR** union defined in the **TYPES.H** file in the **\FABIUS\INCLUDE\MOSAIC** directory. The union is defined as:

```
typedef union{    xaddr addr32;
struct{ uint  page16;
char* addr16;
```

```
} sixteen_bit;
}  EXTENDED_ADDR;
```

To convert a 16 bit address into a 32 bit xaddress, we use the **EXTENDED_ADDR** typedef to declare an instance of the union (named "**buffer**" in this example), store the 16 bit address into the **buffer.sixteen_bit.addr16** element, and store 0 (the default page) into the **buffer.sixteen_bit.page16** element. Then we reference the corresponding 32 bit xaddress via the **buffer.addr32** element of the union. While it is rare that you will have to convert from 16 bit to 32 bit address types, this example provides a template for how to do it.

A Review of FORTH ARRAYs

FORTH_ARRAYs have two key advantages. First, they are allocated in paged memory, so they allow your program to access the large 1 Megabyte memory space of the Handheld. In contrast, C arrays must reside in the available common RAM which is limited to approximately 9 kilobytes on the Handheld. Second, they can be dynamically dimensioned, re-dimensioned and de-allocated (deleted) while your program is running; this boosts efficiency by maximizing the use of the available memory.

To define a new Forth Array, simply use the **FORTH_ARRAY** typedef followed by a name of your choice. For example, in the **ANALOGIO.C** file the following declaration appears:

```
FORTH_ARRAY  results_12;
```

Before the **FORTH_ARRAY** can be accessed at runtime, it must be dimensioned. This is typically accomplished by calling the **DIM()** macro defined in the **ARRAY.H** header file. For example, to dimension the **results_12** array to have 10 rows and 1 column of integer data, we would execute:

```
DIM(int, 10, 1, results_12);
```

In the **ANALOGIO.C** file, the pre-defined macro named **DIM_AD12_BUFFER()** invokes the **DIM()** routine for us (its definition is in the **ANALOG.H** file in the **\FABIUS\INCLUDE\MOSAIC** directory).

After the **FORTH_ARRAY** is dimensioned, it can be accessed by a family of macros and functions that are defined in the **ARRAY.H** header file and are described in the Control-C Glossary. These include functions that fetch from, store to, and calculate the address of individual elements, swap and copy entire arrays, fill an array with a specified character, and delete the array so that it no longer requires memory in the heap. The **PrintForthArray()** and **InitAnalog()** functions in **ANALOGIO.C** provide examples of how to call a few of these functions.

Printing the Contents of a FORTH ARRAY

The **PrintForthArray()** function presented in **ANALOGIO.C** is a more general version of the **PrintFPArray()** function in **GETSTART.C** as discussed in an earlier chapter. The function is defined as follows:

```
_Q void PrintForthArray(int float_flag, FORTH_ARRAY* array_ptr)
// works for FORTH_ARRAYS dimensioned using the standard DIM() macro.
// float_flag is true if array holds float numbers, false otherwise.
{ int r, c;
putchar('\n');
for (r = 0; r < NUMROWS(array_ptr); r++)         // for each row
```

```
{   for  (c = 0; c < NUMCOLUMNS(array_ptr); c++) // for each column
if(float_flag)
printf("%9.4g  ",FARRAYFETCH(float,r,c,array_ptr));
else
printf("%9ld  ",ARRAYFETCH(long,r,c,array_ptr));
putchar('\n');       // newline after each row is printed
PauseOnKey();        // implement xon/xoff output flow control
}
}
```

After calling **putchar()** to output a newline character, we enter nested **for()** statements that print the contents of each element. Because the compiler treats floating point numbers differently than numbers stored in other formats, we use **FARRAYFETCH()** to access floating point arrays, and **ARRAYFETCH()** to access **char**, **int** or **long** arrays. The **PauseOnKey()** function is called once per row to suspend the Handheld's printed output if the terminal program has sent the **XOFF** handshake character; the printout resumes when the terminal sends the **XON** character. **PauseOnKey()** also gives the user the ability to terminate the printout by typing a carriage return character from the terminal.

This function can be tailored to meet the detailed needs of your application. You can change the **printf()** formatting, or insert extra carriage returns to confine the printout to one screen width.

Chapter 6

Real Time Programming

This chapter provides an introduction to real time programming. You'll learn:

- ⇨ *About the timeslice clock and how to use it;*
- ⇨ *All about interrupts, and how to use them to respond to events.*

The Timeslicer and Task Switching

The Built-In Elapsed Time Clock

The Handheld's multitasking executive maintains an elapsed time clock whenever the timeslicer is active. Please consult the TIMEKEEP.C program in the **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory for examples of using the elapsed time clock.

Your program can start the timeslice clock by calling the function:

```
StartTimeslicer()
```

The timeslicer increments the long variable named **TIMESLICE_COUNT** each timeslice period. The default timeslice period is 5 milliseconds (ms), and this can be modified by calling

```
ChangeTaskerPeriod()
```

as described in the Control-C Glossary. The function

```
InitElapsedTime()
```

sets **TIMESLICE_COUNT** equal to zero. The function

```
ReadElapsedSeconds()
```

returns a long result representing the number of elapsed seconds since **InitElapsedTime()** was called.

To attain the full 5 millisecond resolution of the elapsed time counter, we can write a simple function that converts the **TIMESLICE_COUNT** into elapsed seconds as well as the number of milliseconds since the last integral second. For example, let's examine some code from the **TIMEKEEP.C** file in the **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory:

```
#define DEFAULT_TIMESLICE_PERIOD    5   // {ms}; system default
```

```
#define MS_PER_SECOND  1000
static long start_time;                 // saves starting count of TIMESLICE_COUNT
_Q void MarkTime(void)
{   start_time = TIMESLICE_COUNT;
}
_Q void PrintElapsedTime(void)
{   long elapsed_ms =
DEFAULT_TIMESLICE_PERIOD*(TIMESLICE_COUNT - start_time);
long seconds = elapsed_ms / MS_PER_SECOND;
int ms_after_second = elapsed_ms % MS_PER_SECOND;
printf("\nTime since mark is: %ld seconds and %d ms.\n",
seconds, ms_after_second);
}
```

The **MarkTime()** function simply stores the **TIMESLICE_COUNT** in the **start_time** variable. The timeslicer is continually incrementing its counter, and when you later call **PrintElapsedTime()**, **start_time** is subtracted from the latest **TIMESLICE_COUNT** and multiplied by the timeslice period to calculate the elapsed number of milliseconds. This is converted into the elapsed seconds by dividing by 1000, and the remainder is the number of milliseconds since the last integral elapsed second.

To try it out, use the Mosaic IDE's editor to open the **TIMEKEEP.C** file in the **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory, click on the Make Tool to compile the program, and use the terminal to send **TIMEKEEP.DLF** to the Handheld. At your terminal, type:

```
main←
```

to start the timeslicer and initialize the program. Now at any time you can mark a starting time by typing at the terminal:

```
MarkTime( )←
```

and you can print the elapsed seconds and ms since the last mark by typing:

```
PrintElapsedTime( )←
```

which will produce a response of the form:

```
Time since mark is: 3 seconds and 45 ms.
```

Using Interrupt Service Routines (ISRs)

The on-chip resources of the 68HC11 include an A/D converter, timer system, pulse accumulator, watchdog timer, serial communications port, high speed serial peripheral interface, and general purpose digital I/O. The 68HC11's 21 interrupts can enhance the performance of these facilities. Interrupts allow rapid response to time-critical events that often occur in measurement and control applications. For example, you can use an interrupt to create a pulse-width modulated (PWM) output signal.

Interrupt Recognition and Servicing

68HC11 interrupts fall into two main categories: nonmaskable and maskable.

Maskable Interrupts

Maskable interrupts may be freely enabled and disabled by software. They may be generated as a result of a variety of events, including signal level changes on a pin, completion of predetermined time intervals, overflows of special counting registers, and communications events.

Recognition and servicing of maskable interrupts are controlled by a global interrupt enable bit (the I bit in the condition code register) and a set of local interrupt mask bits in the hardware control registers. If a local interrupt mask bit is not enabled, then the interrupt is "masked" and will not be recognized. If the relevant local mask bit is enabled and the interrupt event occurs, the interrupt is recognized and its interrupt flag bit is set to indicate that the interrupt is pending. It is serviced when and if the global interrupt bit (the I bit) is enabled. An interrupt that is not in one of these states is inactive; it may be disabled, or enabled and waiting for a triggering event.

When an interrupt is both recognized and serviced, the processor pushes the programming registers onto the stack to save the machine state, and automatically globally disables all maskable interrupts by setting the I bit in the condition code register until the service routine is over. Other maskable interrupts can become pending during this time, but will not be serviced until interrupts are again globally enabled when the service routine ends. (The programmer can also explicitly re-enable global interrupts inside an interrupt service routine to allow nesting of interrupts, but this is not recommended in multitasking applications). Non-maskable interrupts (reset, clock monitor failure, COP failure, illegal opcode, software interrupt, and XIRQ) are serviced regardless of the state of the I bit.

When an interrupt is serviced, execution of the main program is halted. The programming registers (CCR, ACCD, IX, IY, PC) and the current page are pushed onto the return stack. This saves the state of execution of the main program at the moment the interrupt became serviceable. Next, the processor automatically sets the I bit in the condition code register. This disables interrupts to prevent the servicing of other maskable interrupts. The processor then fetches an address from the "interrupt vector" associated with the recognized interrupt, and starts executing the code at the specified address. It is the programmer's responsibility to ensure that a valid interrupt service routine, or "interrupt handler" is stored at the address pointed to by the interrupt vector.

The CPU then executes the appropriate interrupt handler routine. It is the programmer's responsibility to ensure that a valid *interrupt service routine* is stored at the address pointed to by the interrupt vector. The interrupt vectors are near the top of memory in the onboard ROM. The ROM revectors the interrupts (using jump instructions) to point to specified locations in the EEPROM. The **`ATTACH()`** routine installs a call to the interrupt service routine at the appropriate location in the EEPROM so that the programmer's specified service function is automatically executed when the interrupt is serviced. **`ATTACH()`** also supplies the required **`RTI`** (return from interrupt) instruction that unstacks the programming registers and resumes execution of the previously executing program.

An interrupt handler must reset the interrupt flag bit (not the mask bit) by writing a 1 to it, and perform any tasks necessary to service the interrupt. When the interrupt handler has finished, it executes the RTI (return from interrupt) assembly code instruction. RTI restores the CPU registers to their prior values based on the contents saved on the return stack. As explained below, this also re-

enables interrupts by clearing the I bit in the CCR (condition code register). Thus other interrupts can be serviced after the current interrupt service routine has completed.

Nonmaskable Interrupts

Six of the 68HC11F1's 21 interrupts are nonmaskable, meaning that they are serviced regardless of the state of the global interrupt mask (the I bit in the CCR). Events that cause nonmaskable interrupts include resets, clock monitor failure (triggered when the E-clock frequency drops below 10 kHz), Computer-Operating-Properly (COP) failure (triggered when a programmer-specified timeout condition has occurred), execution of illegal opcodes, execution of the SWI (software interrupt) instruction, and an active low signal on the nonmaskable interrupt request pin named /XIRQ.

Three types of interrupts initiate a hardware reset of the 68HC11:

- Power-on or activation of the reset button
- Computer-Operating-Properly (COP) timeout
- Clock monitor failure

These are the highest priority interrupts, and are nonmaskable. Serviced immediately, they initialize the hardware registers and then execute a specified interrupt service routine. QED-Forth sets the interrupt vectors of these interrupts so that they execute the standard startup sequence. The service routines for all but the main reset interrupt may be changed by the programmer with the `Attach()` utility.

If a nonmaskable interrupt is enabled, it is serviced immediately upon being recognized. The importance of these interrupts is reflected by the fact that most cause a hardware reset when serviced. The following table gives the name of each nonmaskable interrupt and a description of its operation. They are listed in order of priority from highest to lowest:

Table 6-1 Nonmaskable Interrupts.

Interrupt Name	Description
Reset	Recognized when the /RESET (active-low reset) pin is pulled low, this highest priority nonmaskable interrupt resets the machine immediately upon recognition and executes the standard QED-Forth restart sequence.
Clock Monitor Failure	Enabled or disabled via the CME (clock monitor enable) bit in the OPTION register, this interrupt is recognized if the E-clock frequency drops below 10 kHz. It resets the processor hardware and executes a user-defined service routine. QED-Forth installs a default service routine for this interrupt that performs the standard restart sequence.
COP Failure	After enabling the computer operating properly (COP) subsystem, failure to update COP registers within a predetermined timeout period triggers this interrupt which resets the processor and executes a user-defined service routine. QED-Forth installs a default service routine for this interrupt that performs the standard restart sequence.
Illegal Opcode Trap	This interrupt occurs when the processor encounters an unknown opcode. QED-Forth installs a default service routine for this interrupt that performs the standard restart sequence.

Interrupt Name	Description
SWI	Software interrupts are triggered by execution of the SWI opcode. After being recognized, an SWI interrupt is always the next interrupt serviced provided that no reset, clock monitor, COP, or illegal opcode interrupt occurs. SWI requires a user-installed interrupt handler.
/XIRQ	Enabled by clearing the X bit in the condition code register, an /XIRQ interrupt is recognized when the /XIRQ (active-low nonmaskable interrupt) pin is pulled low. This interrupt is serviced immediately upon recognition. It requires an appropriate user-installed interrupt handler.

The service routine for the reset interrupt cannot be modified by the programmer. The service routines for the clock monitor, COP failure, and illegal opcode trap interrupts are initialized to perform the restart sequence, but this action may be changed by the programmer (see the Glossary entry for **`InitVitalIRQsOnCold()`** for more details). No default actions are installed for the SWI and /XIRQ interrupts, so before invoking these interrupts the user should install an appropriate interrupt service routine using the **`ATTACH()`** command.

Servicing Maskable Interrupts

Maskable interrupts are controlled by the I bit in the condition code register (M68HC11 Reference Manual, mc68hc11rm.rev4.1.pdf, Sections.5.7 and 5.8). When the I bit is set, interrupts are disabled, and maskable interrupts cannot be serviced. When clear, interrupts can be serviced, with the highest priority pending interrupt being serviced first. In sum, a locally enabled maskable interrupt is serviced if:

it has been recognized, and

it has the highest priority, and

the I bit in the condition code register is clear.

If a maskable interrupt meets these criteria, the following steps are taken to service it. First, the programming registers and page are automatically saved on the return stack. Note that the condition code register, CCR, is one of the registers saved, and that the saved value of the I bit in the CCR is 0. Next, the CPU automatically sets the I bit to 1 to temporarily prevent maskable interrupts from being serviced. Control is then passed to the interrupt handler code, which you must provide and post using **`ATTACH()`**. The interrupt handler clears the interrupt flag bit set by the trigger event and performs any tasks necessary to service the interrupt. It terminates with an RTI instruction which restores the saved values to the programming registers. Execution then resumes where it left off.

Recall that when the interrupt service began, the processor's first action was to store the programming registers on the return stack. At that time, the I bit in the CCR equaled 0 indicating that interrupts were enabled, and the bit was stored as 0 on the return stack. After stacking the machine state, the processor set the I bit to disable interrupts during the service routine. When the programming registers are restored to their prior values by RTI, note that the I bit is restored to its prior cleared state, indicating that interrupts are again enabled. In this manner the processor automatically disables interrupts when entering a service routine, and re-enables interrupts when exiting a service routine so that other pending interrupts can be serviced.

Nested Interrupts

While the programmer can explicitly clear the I bit inside an interrupt service routine to allow nesting of interrupts, this is not recommended as it can cause crashes in multitasking applications.

Interrupt Priority

Multiple pending interrupts are serviced in the order determined by their priority. Interrupts have a fixed priority, except that the programmer may elevate one interrupt to have the highest priority using the HIPRIO register. Nonmaskable interrupts always have the highest priority when they are recognized, and are immediately serviced. The following table lists the fifteen available maskable interrupts in order of highest to lowest priority:

Table 6-2 Maskable Interrupts, from Highest to Lowest Priority.

Interrupt Name	Description
/IRQ	/IRQ is an active-low external hardware interrupt which is recognized when the signal on the /IRQ pin of the 68HC11 is pulled low (MC68HC11F1 Technical Data Manual, p.2-5).
Real Time Interrupt	The RTI provides a programmable periodic interrupt (MC68HC11F1 Technical Data Manual, p.5-5).
Input Capture 1	An IC1 interrupt is recognized when a specified signal transition is sensed on port A, pin 2 (MC68HC11F1 Technical Data Manual, p.9-1 ff.).
Input Capture 2	An IC2 interrupt is recognized when a specified signal transition is sensed on port A, pin 1 (MC68HC11F1 Technical Data Manual, p.9-1 ff.).
Input Capture 3	An IC3 interrupt is recognized when a specified signal transition is sensed on port A, pin 0 (MC68HC11F1 Technical Data Manual, p.9-1 ff.).
Output Compare 1	An OC1 interrupt is recognized when the main timer's count becomes equal to OC1's timer compare register (MC68HC11F1 Technical Data Manual,p.9-6 ff.).
Output Compare 2	An OC2 interrupt is recognized when the main timer's count becomes equal to OC2's timer compare register (MC68HC11F1 Technical Data Manual,p.9-6 ff.).
Output Compare 3	An OC3 interrupt is recognized when the main timer's count becomes equal to OC3's timer compare register (MC68HC11F1 Technical Data Manual, p.9-6 ff.).
Output Compare 4	An OC4 interrupt is recognized when the main timer's count becomes equal to OC4's timer compare register (MC68HC11F1 Technical Data Manual, p.9-6 ff.).
I4O5	Depending on its configuration, an I4O5 (input capture 4/output compare 5) interrupt is recognized when a specified signal transition is sensed on port A, pin 3, or when I4O5's timer compare register is equal to the main timer's count (MC68HC11F1 Technical Data Manual, p.9-1 ff.).
Timer Overflow	A TOF interrupt occurs when the free-running count in the TCNT (timer count) register overflows from FFFFH to 0000H (MC68HC11F1 Technical Data Manual, p.9-1).
Pulse Accum Overflow	A PAOVF interrupt occurs when the PACNT (pulse accumulator count) register overflows from FFH to 00H (MC68HC11F1 Technical Data Manual, p.9-15 ff.).
Pulse Accum Edge	A PEDGE interrupt occurs after a signal edge is detected on port A, pin 7 (MC68HC11F1 Technical Data Manual, p.9-15 ff.).

Interrupt Name	Description
SPI Event Interrupt	An SPI (serial peripheral interface) interrupt occurs after a byte transfer is completed, or a write collision or a mode fault is detected (MC68HC11F1 Technical Data Manual, p.8-1 ff.).
SCI Event Interrupt	An SCI (serial communications interface) interrupt occurs when the transmit data register is empty, or the transmission is complete, or the receive data register is full, or an idle line is detected. The handler must determine which of these four events caused the interrupt (M68HC11 Reference Manual, Section 9.5.2).

Elevated Priority

After being recognized, a locally enabled maskable interrupt will be serviced when the I bit is clear, and when it has the highest priority among the pending interrupts. Note that interrupts are not necessarily serviced in the order in which they are recognized, but in order of priority among those pending.

You can elevate one maskable interrupt at a time to receive highest priority servicing. This is accomplished by configuring four priority-selection bits named PSEL0, PSEL1, PSEL2, and PSEL3 located in the HPRIO (high priority) register (MC68HC11F1 Technical Data Manual, p.5-7). The default highest priority maskable interrupt is /IRQ. A table in MC68HC11F1 Technical Data Manual, p.5-8 lists the states of the priority selection bits needed to elevate an interrupt's status to the highest priority.

Interrupt Flag and Mask Bits

Each maskable interrupt is enabled and disabled by a local mask bit. An interrupt is enabled when its local mask bit is set. When an interrupt's trigger event occurs, the processor sets the interrupt's flag bit.

The local mask bit should be used to enable and disable individual interrupts. In general, you should avoid setting the global I bit in the condition code register (**CCR**) using **DISABLE_INTERRUPTS()** unless you are sure that you want to disable all interrupts. Time-critical interrupt service routines such as the timesliced multitasker cannot perform their functions when interrupts are globally disabled.

Some of the Handheld's library functions globally disable interrupts for short periods to facilitate multitasking and access to shared resources. A list of these functions is presented in the Control-C Glossary document.

Interrupt trigger events can occur whether or not the interrupt is enabled. For this reason, it is common for flag bits to be set before an interrupt is ready to be used. Unless an interrupt's flag bit is cleared before it is enabled, setting the local mask bit will force the system to recognize an interrupt immediately. Unfortunately, the event which set the interrupt's flag bit occurred at an unknown time before the interrupt was enabled. Depending on the interrupt handler's task, this can cause erratic initial behavior, collection of an incorrect initial data point, or begin an improper sequence of events (for example, cause a phase shift in an output waveform). To avoid these problems, it is recommended that you enable an interrupt by first clearing its flag bit and then immediately setting its mask bit.

! An Interrupt Flag Bit Is Cleared By Writing a 1 to it !

Although mask bits can be set and cleared by storing the desired value in them, flag bits are unusual. Since flag bits are set by trigger events, it is not possible to set them via software. In order to clear an interrupt flag bit, a logical one must be stored into the flag bit's location – that clears it to zero!

To clear a specified flag bit, write a pattern to the flag register with a 1 in the bit position of the flag that must be cleared. All of the other flag bits in the flag register then remain unchanged. See M68HC11 Reference Manual Section 10.4.4 for examples.

External Hardware Interrupts /IRQ and /XIRQ

Two external interrupts, /IRQ (active-low interrupt request) and /XIRQ (active-low nonmaskable interrupt request) allow external hardware to interrupt the 68HC11F1 (M68HC11 Reference Manual, Section 2.4.6). The / prefix to each of these names indicates that the signals are active-low. Pull-up resistors on the Handheld hold these signals high during normal operation, and an interrupt is recognized when either signal is pulled low by an external source. The /IRQ input is maskable and is not serviced unless the I bit in the condition code register is clear. If the CPU is servicing an interrupt when the /IRQ line goes low, the external interrupt will not be recognized until the interrupt being serviced has been handled. Unlike all the other maskable interrupts, /IRQ does not have a local interrupt mask. The /XIRQ external interrupt is not available on the Handheld.

The /IRQ pin is accessed and controlled via the Wildcard Port Header (for pin locations see Appendix A). It operates as an active-low input to the processor. An external device can drive the line LOW to signal an interrupt. Alternatively, several open-collector devices can be wired together on the same line, so that any one of them can interrupt the processor by pulling the request line low. This is called "wired-or" operation. In either case, the external device must pull the line low long enough to be detected by the CPU.

Note that the PORTA input capture lines can also be configured to interrupt the processor when an external event occurs.

Configuring /IRQ Interrupts

In its default state, after each reset or restart, the /IRQ pin is configured as an edge-triggered input. In this mode, the 68HC11 latches the falling edge, causing an interrupt to be recognized. This frees peripheral devices from having to hold the /IRQ line low until the CPU senses the interrupt, and prevents multiple servicing of a single external event.

The disadvantage of this configuration is that multiple edge-triggered interrupts cannot be reliably detected when used with wired-OR interrupt sources. If you are using multiple wire-or /IRQ inputs, you can specify level-sensitive interrupt recognition by clearing a bit named IRQE (IRQ edge-sensitive) in the OPTION register (M68HC11 Reference Manual, Section 5.8.1). IRQE is a "protected bit" in OPTION that must be written within the first 64 E cycles after a reset. The QED-Forth word `InstallRegisterInits()` (described in the glossary) may be used to specify a value that is automatically stored into OPTION upon each reset.

Using /IRQ

To use the /IRQ external interrupt, define an interrupt handler and install it using the pre-defined identifier IRQ.ID and the interrupt **Attach** utility, as described in the Glossary entry for **Attach**.

If interrupts have not yet been enabled globally, then execute:

```
ENABLE_INTERRUPTS↵
```

Whenever /IRQ is pulled low, your interrupt handler will be executed. Note that there is no local interrupt mask for the /IRQ interrupt, so your interrupt handler routine need not clear an interrupt request flag.

Routines that Temporarily Disable Interrupts

Certain kernel routines temporarily disable interrupts by setting the I bit in the condition code register. These routines are summarized in the "Library Functions that Disable Interrupts" chapter of the Control-C Glossary. A review of that list will assist you in planning the time-critical aspects of your application.

Interrupt Latency

The time required between the processor's initiation of interrupt servicing and the execution of the first byte of the specified service routine is called the interrupt latency. Most of the 68HC11's interrupts have an inherent latency of 12 machine cycles during which the registers are saved on the return stack and the interrupt vector is fetched. This corresponds to 3 microseconds (µs). QED-Forth's interrupt latency is longer because the interrupts are re-vectored via the EEPROM to allow the programmer to modify the vectors, and because the page must be changed. The latency of service routines installed with **ATTACH()** is 34 machine cycles, or 8.5 µs. That is, the first opcode of the user's service routine is executed 8.5 µs after interrupt service begins. After the service routine's concluding **RTS** executes, an additional 20 cycles (5 µs) lapses before the originally interrupted program resumes execution. 12 of these cycles are accounted for by the **RTI** instruction, and the other 8 cycles are required to restore the original page.

Interrupt Latency	
Time to Enter an Interrupt Service Routine:	8.5 µsec
Time to Leave an Interrupt Service Routine:	5.0 µsec

Writing Interrupt Service Routines

Maskable interrupts have a local mask bit which enables and disables the interrupt, and a flag bit which is set when a trigger event occurs. For maskable interrupts, an interrupt triggering event is recognized when the flag and mask bits are both set. In order to avoid premature recognition of a maskable interrupt, it should be enabled by first clearing its flag bit and then setting its mask bit. Once an interrupt has been recognized, it will be serviced if it is not masked by the I bit in the CCR. Multiple pending interrupts are serviced in the order determined by their priority. Interrupts have a fixed priority, except that the programmer may elevate one interrupt to have the highest priority.

Nonmaskable interrupts always have the highest priority when they are recognized, and are immediately serviced.

When an interrupt is serviced, the machine state (specified by the programming registers) is saved and the I bit in the CCR register is set. This prevents other pending interrupts from being serviced. The CPU then executes the appropriate interrupt handler routine. The interrupt handler is responsible for clearing the interrupt flag bit. For most interrupts this is accomplished by writing a one to the flag bit. After completing its tasks, the interrupt handler executes an RTI instruction to restore the machine state, subsequently clearing the I bit in the CCR. The CPU is now ready to service the next, highest priority, pending interrupt. If there is none, processing of the main program continues.

To use interrupts you need to create and post an interrupt service routine using the **ATTACH()** macro. We'll look at this process in detail, then discuss how interrupts are implemented on the 68HC11.

To use an interrupt to respond to events, follow these four steps:

1. Use **#define** to name all required bit masks related to servicing the interrupt, and look in the Motorola 68HC11F1 documentation and the **QEDREGS.H** file (in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory) to find the names of all registers that relate to the interrupt. These bit mask and register names will simplify the creation of a readable service routine.

2. Use C or assembly code to define an interrupt service routine which will be executed every time the interrupt occurs. The function must have a **void** stack picture; it cannot return a value or expect input parameters. This function must reset the interrupt request flag (by writing a 1 to it!) and perform any necessary actions to service the interrupt event. Note that the service routine is a standard function; it is not defined using the **_interrupt** keyword.

3. Write a function that installs the interrupt service routine using the **ATTACH()** command. **ATTACH()** initializes the interrupt vector in EEPROM to call the specified service routine, and **ATTACH()** also supplies the **RTI** (return from interrupt) instruction that correctly terminates the service routine.

4. Write functions to enable and disable the interrupt. Enabling the interrupt is accomplished by clearing the interrupt's flag bit by writing a 1 to it, and then setting its mask bit. It may also be necessary to clear the **I** bit in the **CCR** to globally enable interrupts. This can be accomplished by executing **ENABLE_INTERRUPTS()**.

ATTACH() Makes It Simple

It is easy to define an interrupt service routine and **ATTACH** it to a specified interrupt. You define your service routine in either assembly code or in high level C. Thus the service routine can be debugged just like any other C function. You then call **ATTACH()** to bind the service routine to the interrupt.

The following constants have been defined as identifiers for the 68HC11 interrupts in the **INTERUPT.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory:

Table 6-3 68HC11 Interrupts.

Identifier	Interrupt description
`SCI_ID`	Serial communications interface
`SPI_ID`	Serial peripheral interface
`PULSE_EDGE_ID`	Pulse accumulator edge detection
`PULSE_OVERFLOW_ID`	Pulse accumulator overflow
`TIMER_OVERFLOW_ID`	Timer overflow
`IC4_OC5_ID`	Timer input capture 4/output compare 5
`OC4_ID`	Timer output compare 4
`OC3_ID`	Timer output compare 3
`OC2_ID`	Timer output compare 2
`OC1_ID`	Timer output compare 1
`IC3_ID`	Timer input capture 3
`IC2_ID`	Timer input capture 2
`IC1_ID`	Timer input capture 1
`RTI_ID`	Real-time interrupt
`IRQ_ID`	IRQ external pin
`XIRQ_ID`	IRQ external pin (pseudo-nonmaskable)
`SWI_ID`	Software interrupt
`ILLEGAL_OPCODE_ID`	Illegal opcode trap
`COP_ID`	COP failure (reset)
`CLOCK_MONITOR_ID`	Clock monitor failure (reset)

The `ATTACH()` macro expects as inputs a function pointer to your service routine, and an interrupt identifier. It sets up the interrupt vector in EEPROM so that subsequent interrupts will execute the specified service routine. The code installed by `ATTACH` includes the `RTI` instruction that terminates the interrupt service sequence. The `StartFunctionTimer()` routine presented earlier shows how `ATTACH()` is called.

Implementation Details

The interrupt vectors near the top of memory are in ROM; locations that cannot be modified by the programmer. The contents of these locations point to a series of locations in the EEPROM (at AE20-AEBFH) which can be modified and, if desired, write-protected using the BPROT register. `ATTACH()` writes some code at the EEPROM locations corresponding to the specified interrupt. This code loads the code field address of the user's service function into registers and jumps to a routine that saves the current page, changes the page to that of the user's service function, and calls the service function as a subroutine. When the user-defined service function returns, the code installed by `ATTACH()` restores the original page and executes `RTI` (return from interrupt) to complete the interrupt service process. This calling scheme ensures that the interrupt service will be properly called no matter which page the processor is operating in when the interrupt occurs. And because

the interrupt calling routine which is installed by **ATTACH()** ends with an **RTI**, your service routine can end with a standard **RTS**, return or **}** which makes debugging much easier.

The following sections explain how to define and install interrupt service routines that enhance the usefulness of many of the 68HC11's hardware features.

The following example illustrates how to write and use an interrupt service routine.

An Example: Periodically Calling a Specified Function

Many times a program needs to execute a specified action every X milliseconds, where X is a specified time increment. We can use an output compare interrupt to accomplish this. We'll set up an interrupt service routine that executes once per millisecond (ms), and maintains a **ms_counter** variable that is incremented every millisecond. The variable **time_period** specifies the time increment, in ms, between calls to the specified function which is named **TheFunction()**. For this simple example, **TheFunction()** inverts the contents of the static variable named **action_variable**.

Listing 6-1 TIMEKEEP.C, An Example of an Interrupt Service Routine

```
#define OC3_MASK  0x20          // used to set/clear OC3 interrupt flag and mask
#define ONE_MS  500             // 500 counts of 2us TCNT = 1 ms
#define DEFAULT_TIME_PERIOD  1000   // Execute once per second

static int ms_counter;          // runs from 0 to 65535 before rolling over
static uint time_period;        // specifies time in ms between function calls
static int next_execution_time; // next scheduled value of ms_counter
static int action_variable;     // state is toggled by TheFunction()

_Q void TheFunction(void)       // the function simply complements
{   action_variable = !action_variable;  // a variable
}

_Q void FunctionTimer(void)
// This interrupt service routine is called by an OC3-based clock interrupt and
// simply calls TheFunction periodically.
{  ms_counter++;
   if(ms_counter == next_execution_time)
   {  TheFunction();
      next_execution_time = next_execution_time + time_period;
   }
   TOC3 += ONE_MS;        // set OC3 count for next interrupt in 1 ms
   TFLG1 = OC3_MASK;      // reset the oc3 interrupt flag by writing a 1
}

_Q void StopFunctionTimer(void)
{   TMSK1 &= ~OC3_MASK;   // clear OC3I to locally disable OC3
}

_Q void StartFunctionTimer(void)
// inits variables and locally enables OC3 interrupt;
// does not globally enable interrupts!
{  StopFunctionTimer();        // locally disable OC3 while we set it up
   ATTACH(FunctionTimer, OC3_ID);  // post the interrupt service routine
   ms_counter = 0;
   time_period = next_execution_time = DEFAULT_TIME_PERIOD;   // 1/second
   action_variable = 0;        // state is toggled by TheFunction()
   TOC3 = TCNT + ONE_MS;       // start after a 1 ms delay
   TFLG1 = OC3_MASK;           // clear interrupt flag OC3F
   TMSK1 |= OC3_MASK;          // set OC3I to locally enable OC3
}
```

In this program, we define a bit mask named **OC3_MASK** which has bit 5 set and all other bits clear. From inspection of the register summary in the Motorola 68HC11F1 booklet, we see that this mask isolates the Output Compare 3 (**OC3**) mask bit in the **TMSK1** register, and isolates the **OC3** interrupt flag bit in the **TFLG1** register. The other relevant registers are the 16 bit free-running counter register named **TCNT** which increments every 2 microseconds, and the Timer Output Compare 3 register named **TOC3**. If the **OC3** interrupt is enabled by setting its mask bit = 1 in **TMSK1** and by globally enabling interrupts using **ENABLE_INTERRUPTS()** or the assembly instruction **CLI**, then an interrupt occurs when the count in **TCNT** matches the count in **TOC3**. Thus we can control when the next interrupt occurs by writing a specified count to **TOC3**.

TheFunction() is our prototypical function that simply toggles the **action_variable** between the values 0 and 1. The goal of our interrupt service routine is to call **TheFunction()** exactly once per second.

FunctionTimer() is the **OC3** interrupt service routine. It increments the **ms_counter** variable, and checks if **ms_counter** equals **next_execution_time**. If so, it calls **TheFunction()** and updates **next_execution_time** by adding **time_period** to it. Then **FunctionTimer()** increments the contents of the **TOC3** register to set up the next interrupt in 1 ms, and clears the interrupt request flag by writing a 1 to the **OC3** flag bit in the **TFLG1** register. Note that **FunctionTimer()** does not have any input parameters or a return value. Moreover, it is not defined using the **_interrupt** keyword which would insert an **RTI** (return from interrupt) instruction at the end of the function. Rather, **ATTACH()** will supply the **RTI** instruction for us. Because **FunctionTimer()** does not end with an **RTI**, we can easily test the **FunctionTimer()** service routine using our standard interactive debugging techniques.

StopFunctionTimer() simply clears the local **OC3** interrupt mask bit in the **TMSK1** register to disable the **OC3** interrupt.

StartFunctionTimer() first locally disables **OC3** to prevent an interrupt while the service routine is being posted. Then it calls:

```
ATTACH(FunctionTimer, OC3_ID);
```

to ensure that **FunctionTimer()** is called every time the **OC3** interrupt occurs; **ATTACH()** also installs a return sequence that supplies the required **RTI** (return from interrupt) opcode. **ATTACH()** is described in detail later in this chapter. Note that its input parameters are a pointer to the interrupt service routine **FunctionTimer**, and a pre-defined constant named **OC3_ID** that identifies the interrupt. All of the interrupt identifier constants are summarized in Table 6-3.

After calling **ATTACH()**, **StartFunctionTimer()** initializes the timing variables, and initializes **TOC3** so that the first interrupt will occur in 1 ms. It then clears the interrupt flag by writing a 1 to the **OC3F** flag bit in the **TFLG1** register using the statement:

```
TFLG1 = OC3_MASK;
```

Clearing the interrupt flag bit before enabling the interrupt is a highly recommended procedure that ensures that all prior pending **OC3** interrupts are cleared before the interrupt occurs. Finally, **StartFunctionTimer()** locally enables the **OC3** interrupt by setting the mask bit in **TMSK1** with the statement:

```
TMSK1 |= OC3_MASK;
```

To start the interrupt, **main()** simply calls **StartFunctionTimer()** followed by **ENABLE_INTERRUPTS()**. After you compile and download the **TIMEKEEP.C** program and type:

```
main←
```

from your terminal, the **OC3** interrupt is running *in the background*. To monitor the state of the **action_variable**, interactively type at your terminal:

```
See( )←
```

and you will see the variable's value change from 0 to 1 exactly once per second. Type any key to terminate the **See( )** function.

This short program provides a template that shows how a function can be periodically called with a period specified by the variable **time_period**. Of course, in your application the called function would perform a more useful action than does **TheFunction()** in this simple example. You could make other enhancements; for example, a foreground task could manipulate the contents of **time_period** to change the frequency at which **TheFunction()** is called, and you could use **ms_counter** to measure elapsed time with 1 ms resolution.

Note that, to maintain timing accuracy, the interrupt service routine should have a worst-case execution time of under 2 ms; otherwise the **FunctionTimer()** will miss the interrupt when **TCNT** matches **TOC3**, and an extra delay of 131 ms will occur while the **TCNT** timer rolls over. In general, interrupt service routines should be short and simple. Complex calculations should be done by foreground tasks, and the interrupt routines should perform the minimum actions necessary to service the time-critical events. An example of this approach is presented in the *Turnkeyed Application Program*.

Cautions and Restrictions

Note that the OC2 interrupt is used as the multitasker's timeslice clock. Before using this interrupt for another purpose, make sure that you don't need the services provided by the timeslicer which supports the multitasking executive and the elapsed time clock.

The main restriction on interrupt service routines is that they must not call **_forth** (kernel) library functions unless the instructions in the next section are followed. The Glossary document and the header files in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory specify which functions are of the **_forth** type.

Summary

Using interrupts requires:

- ☑ coding an interrupt service routine;
- ☑ using **ATTACH()** to bind it to the appropriate interrupt; and,
- ☑ enabling its local interrupt mask.

Calling Kernel Functions From Within ISRs

There is a special consideration when calling **_forth** library functions from interrupt service routines. Fortunately, this restriction can be overcome by simply including the **FORTHIRQ.C** file with your source code, and following the simple example presented in the file. **FORTHIRQ.C** is present in the **MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory.

The method is very simple: just place a call to the function

```
BeginForthInterrupt();
```

at the top of your interrupt service routine (or, at the minimum, before any **_forth** functions are called). Before the final exit point of the interrupt service routine, place a call to the function

```
EndForthInterrupt();
```

That's all there is to it. The ability to call **_forth** library functions from interrupt service routines makes it easier to manage page-mapped I/O devices on an event-driven basis.

Chapter 7

Failure and Run-Time Error Recovery

This chapter describes a variety of useful hardware features of the 68HC11F1:

- ⇨ *The processor's external hardware interrupt /IRQ, may be used by external devices to request immediate service.*
- ⇨ *Three nonmaskable interrupts cause a hardware reset: the external reset, the COP, and the clock monitor. The main reset is activated on power-up or when the /RESET pin is pulled low for more than 4 machine cycle. Enabling the computer operating properly circuit, COP, sets up a watchdog timer that resets the processor unless a special register is periodically updated. This provides a means of recovering from crashes in an embedded application. Use of the COP feature requires installation of an autostart routine which services the COP. The clock monitor backs up the COP by resetting the machine if the system clock fails.*
- ⇨ *STOP and WAI instructions are available to put the CPU in low power modes with different degrees of power savings*
- ⇨ *Finally, an on-board jumper allows selection of the standard operating mode or the special cleanup mode.*

Getting Started and Getting Stopped – Restarts and Resets

External Hardware Resets

The main reset interrupt of the 68HC11 processor is activated upon power-up or when the active-low /RESET signal is pulled low. The processor does not distinguish between a power-on reset and a reset caused by a low level on the /RESET input pin; both result in the same hardware initialization and software restart sequence.

The /RESET line is normally held high by a pull-up resistor. You can pull the /RESET line low by pushing on the reset switch. Moreover, any peripheral device can reset the processor by driving the /RESET signal low for at least 2 microseconds using an open-collector output.

The active-low /RESET signal is controlled by the power monitor circuitry. On power-up, the monitor asserts the reset signal until the positive supply has stabilized above 4.5 Volts.

Internal Resets

The 68HC11 resets itself when a failure condition is detected by either the computer-operating-properly (COP) or the clock monitor circuit. When either of these failure conditions occur, the processor drives the /RESET line low for less than 4 machine cycles to reset itself and any peripherals that are connected to the /RESET line. The processor then determines which failure (COP or clock monitor) caused the reset, and branches to the associated service routine. QED-Forth initializes the interrupt vectors for the COP and clock monitor to perform the standard restart sequence, and the programmer may change the vectors if desired. The operation of the COP and clock monitor are described in the following sections.

Crashes

A computer "crashes" when it executes a set of instructions that it is not supposed to. This can cause the processor to write over memory locations that are not write-protected. The processor may get into an infinite loop of legal instructions (in which case it will not respond to your commands), or it may eventually execute an "illegal opcode". Illegal instructions are detected by the processor's illegal opcode trap and result in a restart , in which case you will see the QED-Forth startup message on your terminal, or execution of the autostart program, if present.

The best response to a crash during program development is to push the reset button. This initializes all of the registers and performs a restart. In most cases a "warm restart" will be performed, which should allow you to continue programming with access to all of the words that you have defined. In other cases, the state of the user area or the dictionary may be corrupted. If QED-Forth detects the corruption, it will automatically execute a "cold restart"; otherwise you may execute COLD which performs the restart. The cold restart re-initializes all of the user variables that control QED-Forth's operation.

Resets versus Restarts

To clarify the discussion of crashes, some terms must be defined. A "reset" is an initialization process invoked by the hardware of the 68HC11, while a "restart" is an initialization process controlled by software.

A reset can be caused by any of four events:

- power is applied to the processor
- the reset button is pushed
- the clock monitor detects a clock failure
- the computer operating properly (COP) circuit detects a failure

The hardware of the 68HC11 is configured by a set of registers that reside at locations 8000H through 805FH. (These hardware registers should not be confused with the programming registers D, X, Y, etc.) The reset initializes essentially all of the registers, and then initiates an interrupt response sequence. The interrupt calls a specified response program whose address is stored in an interrupt vector near the top of memory. The power-on and reset-button resets share the same inter-

rupt vector at FFFE. The clock monitor and COP resets are re-vectored to addresses in EEPROM where the programmer can install customized service routines, if desired. All of these service routines are initialized to perform the default restart routine.

A "restart" is an initialization process performed by software. After a (hardware-invoked) reset, the 68HC11 calls a restart routine which re-initializes some of the registers to accommodate QED-Forth, and initializes other memory locations including all or part of the user area. A restart can also be invoked solely via software, by executing the kernel words COLD or WARM. When the illegal opcode trap detects an illegal instruction, it calls a restart routine, but does not perform a hardware reset. Note that a reset always results in a restart, but that a restart can be performed without a reset.

COLD is the most comprehensive software-invoked initialization command. Executing COLD after a crash usually puts the machine into a well-known state by completely initializing the user area which controls QED-Forth's operation. But COLD does not initialize all of the registers. Therefore, in crashes where the contents of key hardware registers are corrupted, it may be necessary to perform a hardware reset by pushing the reset button or powering the machine off and on again.

Cold versus Warm Restarts

There are two types of restart: cold and warm. A cold restart initializes all of the parameters used by the QED-Forth system. These parameters are stored in the "user area", which is a 256-byte block of memory in the common RAM. All of the memory management pointers, format variables to control numeric conversion, quantities that enable the compilation of local variables, and many other system values are stored in the user area. COLD initializes these to default values. COLD also initializes several vital interrupt vectors so that they will perform the startup sequence if they are invoked. These vital interrupts --clock monitor, computer operating properly, and illegal opcode trap-- were discussed in the last chapter.

A warm restart, on the other hand, assumes that most of the user variables have already been properly initialized. A warm restart initializes only a few of these parameters, including stack pointers (it clears the stacks) and some multitasking variables (it makes sure that a single task is running and that it has control of the serial port).

A warm restart preserves the prior number base (whatever you had set it to before the restart occurred) while a cold restart always sets the base to decimal. A warm restart preserves the user's memory map and QED-Forth's ability to find user defined words, while a cold restart sets a default memory map and forgets all words except those in the original kernel.

The default restart program decides whether to perform a cold or a warm restart by checking a location in the user area to see if a specified pattern (1357H) is stored there. If the correct pattern is present, the restart program assumes that the user area is already properly initialized, so it performs a warm restart. If the location does not contain the proper value, the restart program assumes that some event (perhaps a crash) has corrupted the user area, so a cold restart is executed to force the system to a known state.

Because the Handheld's common RAM is battery backed (except for the 1K of RAM at B000H-B3FFH on the 68HC11 itself), the user area (including the location where the startup pattern is stored) maintains its contents even when it is powered down. Thus a warm restart will be performed

most of the time when you turn on the Handheld. This is convenient: it means that access to the words you defined, your memory map, and the contents of the user area are not altered by removal of power. It also means that pushing the restart button and powering the machine off and on again have similar effects, except that powering the machine off loses the contents of the 1K of RAM on the 68HC11 at addresses B000H-B3FFH.

If a crash over-writes the user area, the next restart will be a cold restart. QED-Forth signals a cold startup by printing a COLDSTART statement before the QED-Forth V4.4x startup message is printed. If the crash did not corrupt the startup pattern in the user area, a warm restart would be performed, and you could continue debugging. In most cases, all of the words that you defined would still be accessible. If the machine is behaving in an unpredictable manner, however, it may be necessary to reset the machine and perform a cold restart to establish a known initialized state.

Recovery Tricks

Some crashes may be difficult (but not impossible!) to recover from. For example, if the name area of the dictionary is corrupted, QED-Forth may not be able to find even the most basic commands in the dictionary. If every command you give is met with the ? error message, try executing COLD. The FIND word in the interpreter is programmed to always recognize the word COLD, even if the dictionary is corrupted.

If All Else Fails, Use the Special Cleanup Mode

These recovery techniques may not work if you have a buggy autostart word or a major crash. If typing COLD or pressing the reset button does not greet you with the standard "QED-Forth V4.4x" prompt, you may need to use the special cleanup mode to restore your system to a proper state. This involves installing Jumper J1 and then pressing the reset button. The special cleanup procedure places the Handheld in the same state it was in when it was shipped from the factory.

The COP Watchdog Timer and Clock Monitor

In many embedded control applications, it is important that processor crashes be detected quickly so that the system can rapidly be returned to a proper operating condition. The Computer Operating Properly subsystem, also known as a "watchdog timer" or "COP", provides this capability. It gives the programmer a way to force a processor reset if an application program crashes or gets lost. When enabled, the COP resets the processor if the application program fails to periodically update a specified register within a predetermined time-out period. The COP time-out period is programmable to any of four values between 8 msec to 0.5 seconds.

To use the COP, design and debug an application program that, in addition to performing all of its normal tasks, periodically writes a 2-byte pattern to the COP reset (COPRST) register as described below. The specified pattern must be written before the COP "times out". Then install the application as an autostart routine using the QED-Forth word AUTOSTART or PRIORITY.AUTOSTART, and enable the COP.

If the application program ever allows the time-out period to be exceeded without writing the specified pattern, the COP resets the processor. Presumably the pattern will not be properly written if the

processor crashes for any reason, so the COP provides a way of automatically resetting the processor to recover from crashes. Then, because the application program has been installed as an autostart routine, the application is automatically restarted when the COP forces a reset.

Be Careful with the COP

Before enabling the COP, make sure that a debugged application program that properly updates the COPRST register has been installed as an Autostart() or PriorityAutostart() routine. If the startup program is improperly designed so that it is unable to service the COP on time, the COP will reset the machine, thereby invoking the startup program again, and leading to an infinite series of COP resets.

If you find yourself in this situation you can return the Handheld to its "pristine" state by entering the special clean-up mode: install Jumper J1 and then press the reset button to resume normal operation with the COP disabled and any autostart routine removed.

The COP feature should prove trouble-free as long as the application program is:

- ☑ fully debugged;
- ☑ capable of updating the COPRST in a timely fashion; and,
- ☑ installed as an autostart routine.

Configuring the COP

Three bits are used to configure and enable/disable the COP. They are named CR0, CR1, and NOCOP. CR0 and CR1 are located in the OPTION register. These bits determine the amount of time which can elapse between updates of the COPRST register by the application program. If the time-out period is exceeded, the COP forces a reset. The four available time-out periods are:

Table 7-1 COP Time-out Period

CR1	CR0	Time-out Period
0	0	8.192 ms
0	1	32.768 ms
1	0	131.07 ms
1	1	524.5 ms

The CR1 and CR0 bits in the OPTION register may be modified only during the first 64 cycles after a reset. The function InstallRegisterInits() makes it easy to specify a value that will be automatically stored into the OPTION register after every reset; consult its glossary entry for details.

The third control bit is called NOCOP and is located in the CONFIG register. The Handheld is shipped with this bit set so that the COP is disabled. To enable the COP, clear this bit. The CONFIG register's contents are non-volatile, and so are maintained even after the processor has been powered down.

Servicing the COP

Servicing the COP is accomplished by writing 55H and AAH to the COPRST register. Although the order of the writes is important, the number of intermediate instructions between them is inconsequential. The two writes must be performed before the time-out period has elapsed. Once AAH has been stored, the COP will need to be serviced again before the next time-out period has elapsed.

The Clock Monitor

The clock monitor provides a second level of security by monitoring the main system clock and resetting the processor if the clock signal disappears or oscillates too slowly. The clock monitor does not initiate a reset as long as the E-clock frequency is greater than 200 kHz (the E-clock frequency is one quarter the frequency of the on-board crystal). A reset is always triggered at E-clock frequencies below 10 kHz, and may be triggered at frequencies as high as 200 kHz.

The clock monitor is primarily used as a backup for the COP. The COP relies on the clock's presence for reliable operation, and the clock monitor can ensure that the processor is safely reset if the clock fails.

Enabling the clock monitor is accomplished by setting the CME (clock monitor enable) bit in the OPTION register. This bit may be set or reset at any time. A second bit named FCME (force clock monitor enable) is also involved. When the FCME bit is in its default state of 0, the bit has no effect, and when FCME is set, the clock monitor feature cannot be disabled until a reset occurs. We will assume that FCME is 0, and that the CME bit controls the clock monitor. See MC68HC11F1 Technical Data Manual, p.5-3 for further details. Note also that if the clock monitor is enabled, a STOP assembly instruction will trigger a reset because it stops the clock, as discussed in the "Low Power Modes" section below.

Processor Operating Modes

Low Power Modes

The 68HC11F1 has two low power modes. These modes are enabled by assembly instructions STOP and WAI (wait). The STOP command puts the CPU into its lowest power-consumption mode by stopping all clocks, thereby stopping all processing (MC68HC11F1 Technical Data Manual, p.5-17). If the clock monitor is enabled, a reset will be triggered when the clocks stop due to a STOP instruction. To use a STOP instruction when the clock monitor reset is enabled, disable the monitor before the STOP instruction, and re-enable it after returning from the STOP.

Pulling either /RESET or /IRQ low wakes the processor up after a STOP instruction. Pulling the reset line low awakens the CPU and performs the standard reset startup sequence. For the CPU to be awakened by the /IRQ line going low, the I bit in the CCR register must be clear so that interrupts are globally enabled. When /IRQ goes low and the I bit is clear, execution begins with the /IRQ handler and then executes the code following the STOP instruction.

The STOP instruction is executed as a NOP unless the S bit in the CCR is cleared. After clearing the S bit, any occurrence of a STOP instruction puts the CPU into its lowest power mode. After each reset or restart, QED-Forth leaves the S bit in the CCR in its default set position, meaning that the STOP mode is disabled.

WAI Low Power Mode

The WAI instruction also puts the 68HC11F1 in a low power mode. However, clocks are not disabled in the wait mode, so power consumption is greater than the STOP mode. After a WAI instruction, the machine state is stacked and processing stops. Power savings can be increased by setting the I bit in the CCR and disabling the COP. Further savings can be achieved by disabling the on-chip subsystems, including executing A/D8.OFF to turn off the A/D (MC68HC11F1 Technical Data Manual, pp.5-17...5-18).

The WAI low power state can only be exited by an unmasked interrupt or by pulling the /RESET pin low. When an unmasked interrupt occurs, (for example /IRQ goes low, the COP is not serviced, clock monitor failure or reset occurs), the appropriate interrupt handler is executed and then processing continues with the instructions following the WAI. Implementing the WAI lower power mode is accomplished by simply executing WAI.

Summary of Low Power Modes

In sum, power can be saved by putting the CPU in a low power mode while processing is not required. The 68HC11F1 has two low power modes with different degrees of savings. Both modes are terminated by unmasked interrupts. While the WAI instruction can be called without any preparation, the STOP instruction must be enabled by clearing the S bit of the CCR register.

Operating Modes of the 68HC11F1 CPU

The 68HC11F1 microcontroller has four operating modes: expanded nonmultiplexed, special test, single chip, and special bootstrap modes (M68HC11 Reference Manual, Section 3 and MC68HC11F1 Technical Data Manual, pp.4-1...2). The standard operating mode is expanded nonmultiplexed, meaning that the processor has access to expanded memory beyond its on-chip memory, and that the address and data lines are not multiplexed together (as they are on other members of the 68HC11 family). The Handheld also makes use of the special test mode, renaming it the “special cleanup” mode. This mode makes it possible to rapidly recover from any programming error that causes repeated machine crashes. The single chip mode takes away the ability of the processor to address external memory, and special bootstrap allows startup code to be inserted into the processor; these two modes are not used on the Handheld.

The processor’s operating mode is determined by the states of two pins named MODA and MODB (refer to the schematic in Appendix C). On the Handheld, MODA is always high and MODB may be pulled LOW by installing Jumper J1; this invokes the special cleanup mode. When Jumper J1 is not installed, the board is in the standard operating mode.

Special Cleanup Mode

The Special Cleanup Mode is useful if a buggy startup routine has been installed (using the AUTOSTART or PRIORITY.AUTOSTART words) or if invalid register initializations have been specified (for example, using the InstallRegisterInits() word). To recover from these problems, simply enter the special cleanup mode by installing Jumper J1 and pressing the reset button. This completely re-initializes the system software to its "pristine" state, and displays the QED Forth startup message at your terminal.

Chapter 8

Chapter 8: 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 keypad.

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* 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 key.

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_SCREEN3	An object that contains a collection of other functionally related objects such as graphics, textboxes, and plots.
GUI_KEYPAD0 to GUI_KEYPAD3	An object tied to the physical keypad for each GUI screen.
GUI_BATTERY	The object tied to the rechargeable batteries.
GUI_TIMER	An object that repeatedly calls event procedures.
GUI_ERROR	An object that contains the most recent GUI Toolkit error.
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	A custom proportional font, 10 pixels tall, that is used to render strings in textboxes.
GRAPHIC	An object that contains a single image as a property.
FONT	An object that contains a single image composed of the 255 images of each character of a font that is used to render strings in textboxes.
TEXTBOX	An object that uses a font object to render strings onto a screen.

Object	Description
DATA_ENTRY_KEY	An object associated with a key on the keypad that adds a character to a textbox.
ACTION_KEY	An object associated with a key on the keypad that responds to a user's touch by calling an associated event procedure.
SHIFT_KEY	An object associated with a key on the keypad that changes the shift state.
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 Handheld 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 four simple screens. The first screen is an alphanumeric keypad example that allows you to enter text and show it on the display. The second screen allows you to adjust the contrast and brightness of the display. The third screen allows you to change the length of each beep that is sounded when you press a key. The fourth screen displays a textbox using a custom font of 8 point Comic Sans. As we describe each part of the GUI Toolkit, we'll use the alphanumeric 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 8-1 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 8-1 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 128 pixels and a height of 128 pixels, a resolution of 72 pixels per inch, and using the bitmap image mode. 128x128 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 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 8-1 is shown in Figure 8-2 .

Figure 8-2 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. 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 placement of an image in the vertical direction. The size of an image is limited by its placement and the size of the display. For example, an image placed in the upper left corner (at coordinates 0,0) can be up to 128 pixels wide by 128 pixels tall. However, if the image is moved 8 pixels to the right and 12 pixels down (to coordinates 8,12), the maximum size the image can be is 120 pixels wide by 116 pixels tall.

Transferring Your Images to the Handheld 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 Handheld Controller, it stores your image data in RAM and then copies it into Flash. You must transfer the Image Data File using Mosaic's Terminal Program, to the Handheld 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 display_screen.bmp, sound_screen_bmp, and welcome.bmp shown in the following listing. The constants contain the addresses of the image data in Flash memory on the Handheld Controller. For more information on the memory map of the Handheld Controller see *Chapter 4: Making Effective Use of Memory*.

Listing 8-1 The first few lines of the Image Header File.

```
#define DISPLAY_SCREEN_BMP 0x7101F36
#define SOUND_SCREEN_BMP 0x102756
.
.
```

```
.
#define WELCOME_BMP 0x103497
```

Coding Your Application

Once you have designed and built the user interface for your application and you have transferred the images to the Handheld 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;
- ☑ Routines that load objects onto screens; and,
- ☑ An endless loop that executes event procedures

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 Handheld 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.

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 Installation CD. Look in the Demos and 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 controller.
- ◙ 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 C

Move the library.c and library.h files into the same directory as your other C source code files and use the following directives in your highest level source code file:

```
#include "library.h"
#include "library.c"
```

The library.h contains the prototypes and constants of the GUI Toolkit while library.c contains the code to call the various GUI Toolkit methods. You should include library.c only once in your highest level source code file but include library.h in every additional source file that uses GUI Toolkit constants or functions.

Note that all of the functions in the kernel extension are of the _forth type. While they are fully callable from C, there are two important restrictions. First, _forth functions may not be called as part of a parameter list of another _forth function. Second, _forth functions may not be called from within an interrupt service routine unless the instructions found in the file named

\fabius\qedcode\forthirq.c

are followed.

NOTE: If your compiler was purchased before June 2002, you must update the files, qlink.bat and qmlink.bat in your /fabius/bin directory on your installation before using the kernel extension. You can download a zip file of new versions from

http://www.mosaic-industries.com/Download/new_qlink.zip

The two new files should be placed in c:\Fabius\bin. This upgrade only has to be done once for a given installation of the C compiler.

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 Handheld Controller. Then, be sure to include the Image Header File in your application as shown below:

Listing 8-2 The first few lines of the demo program.

```
#include <\mosaic\allqed.h>  // Include all of the Mosaic utilities.
#include "library.h"         // Define prototypes and constants of the GUI Toolkit.
#include "library.c"         // Code used to call the GUI Toolkit functions.
#include "image_header.h"    // Include constants that define 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 8-3 Initializing the GUI Toolkit

```
#define GUI_HEAP_START 0x0F3000L
#define GUI_HEAP_END  0x0F6FFFL
.
.
void main( void )
{
  Initialize_GUI ( GUI_HEAP_START, GUI_HEAP_END );
  .
  .
}
```

`Initialize_GUI` performs several actions: it sets up a heap memory structure for the GUI Toolkit; it sets up and activates a task for the GUI Toolkit; it initializes and starts the timeslicer; 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 8k 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:

- checks to see if new objects need to be created or existing objects need to be modified;
- polls the keypad looking for events;
- checks to see if a user defined event procedure needs to be called (as part of the GUI Timer); and,
- calls **`Pause`**.

When an action key is pressed, the GUI Task sends a message indicating that the event procedure 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 8-4 Servicing GUI Events

```
TASK taskbase;
.
.
void GUI_Monitor ( void )
{
  Initialize_GUI ( GUI_HEAP_START, GUI_HEAP_END);
  .
  while(1)
  {
    .
    .
    Service_GUI_Events();
    Pause();
  }
}
void main( void )
{
  .
  BUILD_C_TASK(0, 0, &taskbase);    // build task
  ACTIVATE( GUI_Monitor,&taskbase); // activate task
  StartTimeslicer();                // starts elapse time clock, enables interrupts
  Pause();                          // start next task immediately
}
```

The GUI Toolkit is a resource whose access must be controlled. Only the GUI Toolkit (running in the GUI Task) polls the keypad, 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 create and use a resource variable to assure that only one task accesses the GUI Toolkit at a time. See the chapter on Real Time Programming for more information on using resource variables.

The GUI Task should have an opportunity to run approximately once every 50 milliseconds. Calling the GUI Task more frequently will not improve performance because the GUI Task polls the timeslicer to assure it is not called more than once every 50 milliseconds. Calling the GUI Task less frequently will not cause any errors but may result in sluggish performance. Because the GUI Toolkit polls the timeslicer, the timeslicer must be running or the GUI Toolkit will freeze. 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, display, four predefined screens, four predefined keypads, battery, timer, pen, buzzer, and 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 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 adjusting the contrast or controlling the backlight. The **GUI_DISPLAY** contains four pre-instantiated screens called **GUI_SCREEN0** to **GUI_SCREEN3**. A property of GUI Screens is their **VISIBLE** 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**.

Each GUI Screen has an associated keypad named **GUI_KEYPAD0** to **GUI_KEYPAD3**. A keypad is a collection of keys; the possible types of keys are action keys, data entry keys, and shift keys. Action keys generate events when a user presses the key (press event), holds down the key (hold event), and releases the key (release event). Data entry keys are used to enter numbers, letters, or symbols into textboxes. You can overload data entry keys so that the same key can be used to enter multiple numbers, letters, or symbols. For example, in the demo, the number 9 data entry key is used to enter the number "9" as well as the letters "W", "X", "Y", "Z", "w", "x", "y", or "z". Shift keys are used to change the shift state to enable the overloading of data entry keys.

The Battery Object is called the **GUI_BATTERY**, and it is tied to the physical rechargeable batteries. Properties of the **GUI_BATTERY** include the battery voltage, battery current usage, battery state, and whether the charger is on or off.

The Timer Object is called the **GUI_TIMER**, and it is used to repetitively call user defined event procedures. The **GUI_TIMER** is useful for calling routines that need to be called every so often to perform tasks like reading the battery.

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 a custom proportional font that is10-pixels tall. The default font is used to render string data from a textbox onto a screen where it can be shown.

Creating New Objects

The GUI Toolkit also has other objects, like graphics and textboxes, 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**, **ACTION_KEY**, **DATA_ENTRY_KEY**, **SHIFT_KEY**,

SCREEN, 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 **graphicWelcome**. Later, we'll set the image of the graphic and then load it into a screen.

Listing 8-5 Creating a New Graphic.

```
int graphicWelcome;
.
.
.
void Initialize_Screen0 ( void )
{
  graphicWelcome = New_Object ( GRAPHIC );
  .
  .
  .
}
```

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.

In the demo, **Initialize_Screen0** creates a graphic and then sets it's image property using a constant defined in the Image Header File.

Listing 8-6 Setting the Property of a Graphic.

```
int graphicWelcome;
.
.
.
void Initialize_Screen0 ( void )
{
  graphicWelcome = New_Object ( GRAPHIC );
  Set_Property ( graphicWelcome, IMAGE, WELCOME_BMP );
  .
  .
  .
}
```

Creating and Initializing Controls

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

1. Use the **New_Object** method to create an action key and store the object reference to the action key into a variable. By default, the action key will beep when it receives a press event. You can change this behavior by setting the **KEYPAD_BEEP** property of the keypad containing the action key (**GUI_KEYPAD0** in this case) to **GUI_FALSE**.
2. Initialize any required action key properties such as **BLOCK_ON_HOLD**. The next section discusses when to use blocking in your application.
3. Define the **PRESS_EVENT_PROCEDURE** for the action key. Writing event procedures is covered after the next section on blocking.
4. Associate the event procedure with the action key using the **Set_Property** method.
5. Insert the action key into a keypad using the **Insert_Key** method. Inserting a key into a keypad associates the key with a screen. When the associated screen becomes visible, the key becomes active (i.e. the key will responded to presses). The key positions are shown in the following table:

Table 8-1 The key positions for a keypad.

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15
20	24	28	32
21	25	29	33
22	26	30	34
23	27	31	On / Off

The on / off key is tied to power circuitry so it can not be moved or assigned to different function.

The following code shows the **Initialize_Screen0** function from the demo that illustrates how to create an action key that, when pressed, will beep, and execute an event procedure.

Listing 8-7 Creating and Initializing a Control.

```
int actionkeyFunction1;
.
.
.
void Initialize_Screen0 ( void )
{
  .
  .
  .
  actionkeyFunction1 = New_Object( ACTION_KEY );
  Set_Property(actionkeyFunction1, BLOCK_ON_PRESS, GUI_TRUE);
  Set_Property(actionkeyFunction1, PRESS_EVENT_PROCEDURE, (long)
    Function1_event_procedure_ptr);
  // Position the function 1 key to be in the upper left corner of the keypad.
  Insert_Key ( GUI_KEYPAD0, actionkeyFunction1, 0 );
  .
  .
  .
}
```

Defining and initializing other controls like data entry keys and shift keys are handled in the same way. Examples of how to define and initialize data entry keys and shift keys are shown in the demo.

When To Use Blocking

Turning off the service events property of the GUI Toolkit (also called blocking) disables the keypad and keys from changing GUI Toolkit properties. Blocking is required when the state of the GUI Toolkit must remain unchanged for a short time period. Blocking is usually associated with action keys with event procedures that need to read, write, or change properties or calls GUI Toolkit methods. As shown in the previous listing, the block on press property of the action key is set because its event procedure (as shown in the next listing) changes the visible property of a screen. Thus once the action key is pressed, no other keys will be able to change GUI Toolkit properties (so another key that is simultaneously pressed can't make another screen visible causing a conflict). If you configure an action key to block, it is your responsibility to re-enable the servicing of events in the event procedure. Typically, the re-enabling of events is the last thing you do in an event procedure before exiting.

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 Function 1 Action Key changes the visible screen. The following listing shows the code for the action key's event procedure.

Listing 8-8 Writing the Event Procedure.

```
void Function1_Event_Procedure ( void )
{
  // Set GUI Screen 3 to be the visible screen.
  Set_Property ( GUI_SCREEN3, VISIBLE, GUI_TRUE ) ;
  // The action key associated with this event procedure must be configured to
  // block on press since the event procedure must change the visible property
  // of a screen.  Re-enable the servicing of events here at the end of the
  // event procedure.
  Set_Property ( GUI_TOOLKIT, SERVICE_EVENTS, GUI_TRUE );
}
// We need to store the address of the event procedure into a function pointer which
// will be used to associate the event procedure with the action key.
#include "begin_event_procedure_pointers.h"
.
.
.
xaddr (*function1_event_procedure_ptr)(void)    = Function1_Event_Procedure;
.
.
.
#include "end_event_procedure_pointers.h"
```

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 a graphic for **`GUI_SCREEN0`** and then loads it into the screen.

Listing 8-9 Loading a Control into a Screen.

```
void Initialize_Screen0 ( void )
{
  graphicWelcome = New_Object ( GRAPHIC );
  Set_Property ( graphicWelcome, IMAGE, WELCOME_BMP );
  Load ( GUI_SCREEN0, graphicWelcome, 0, 0 );
  .
  .
  .
}
```

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 (112,0), 8 pixels of the image will hang off the edge of the screen (since the width of a screen is 128 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 (104,0). 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 8-10 Handling Errors.

```
void Print_Error ( void )
{
  switch (Read_Error())
  {
    case HEAP_FULL: printf("Can't create any more new objects.\n"); break;
    case X_OUT_OF_RANGE: printf("X coordinate is out of range.\n"); break;
   // You would put other error codes and error messages here.
  }
  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 prepackaged 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_BATTERY** and **GUI_DISPLAY**), some collect other objects together (GUI Screens), some respond to events (controls), 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_BATTERY**, **GUI_BUZZER**, **GUI_FONT**, GUI Screens, GUI Keypads, and **GUI_PEN**. You can not create more of these object types. For other object such as graphics, textboxes, keys, screens, 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. More detailed descriptions of each method, property, and value is available in the GUI Toolkit Glossary.

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 8-2 Properties of the GUI_TOOLKIT.

Property	Values	Meaning
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	Unsigned Long	Read only. Contains the number of bytes remaining in the GUI Heap.
CURRENT_SCREEN	**GUI_SCREEN0** to **GUI_SCREEN3**	Contains the current visible screen.
SERVICE_EVENTS	**GUI_TRUE**, **GUI_FALSE**	If false, events are not serviced. All events generated by pressed keys are ignored.

Property	Values	Meaning
STANDARD_FONT	GUI_FONT or custom font object	Contains the reference to the font object used to render string data in textboxes. At initialization, STANDARD_FONT is set to GUI_FONT which is a proportional font with a height of 10 pixels. You can set this property to your own custom font, if desired.
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.

Methods

Table 8-3 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_BATTERY, GUI_PEN, GUI_FONT, GUI_BUZZER, GUI Screens, and GUI Keypads. Sets up and activates a task in common memory. Initializes and starts the timeslicer. Calling any other GUI Toolkit method before calling Initialize_GUI will crash the controller.
Service_GUI_Events	None	None	Executes the event procedures of controls. This method must be included in a task as part of your 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 8-4 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_FONT	**Set_Property**	The object reference is not a font when setting the **STANDARD_FONT** property of the GUI Toolkit or setting the **TEXTBOX_FONT** property of a textbox.
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
WIDTH_OUT_OF_RANGE	**Set_Property**	The width will create a plot or textbox that is wider than the width of the screen.
HEIGHT_OUT_OF_RANGE	**Set_Property**	The height will create a plot or textbox 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 8-5 Properties of the GUI_DISPLAY.

Property	Values	Meaning
WIDTH_IN_PIXELS	128xels	Read only. Contains the width of the display in pixels.

Property	Values	Meaning
HEIGHT_IN_PIXELS	128xels	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. Initialized to GUI_TRUE.
BRIGHT_BACKLIGHT	GUI_TRUE, GUI_FALSE	If true and the backlight is on, supplies more current to the backlight to increase the brightness. Initialized to GUI_FALSE.

GUI Screens

A screen is a collection of functionally related objects, including textboxes, plots, and graphics that are presented together on the display. Screens are fixed in size to 128 pixels wide by 128 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.

You can also create additional screens that reside in the heap. Heap screens, unlike GUI Screens, can not be made visible. Heap screens allow you to build many different screens when you are initializing your application and then quickly switch them into GUI Screens at run time using the Copy_Screen method. Heap screens are described in detail in a later section.

Properties

Table 8-6 Properties of GUI Screens.

Property	Values	Meaning
WIDTH_IN_PIXELS	128 pixels	Read only. Contains the width of the screen in pixels.
HEIGHT_IN_PIXELS	128 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	Unsigned Long	Read only. Contains the number of objects contained in the screen.

Property	Values	Meaning
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 rendered image of the screen and its objects in the display.

Methods

Table 8-7 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.
Copy_Screen	Source Screen Reference, Destination Screen Reference	None	Copies the contents of the source screen into the destination screen. The contents of the destination screen are over written.
Clear	Screen Reference	None	Unloads all objects from the screen and erases the screen.
Refresh	Screen Reference	None	Re-renders graphics, 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 8-8 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, Copy_Screen	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 Keypads

Each GUI Screen has a corresponding keypad that holds the keys that animate the screen. **GUI_SCREEN0** is associated with **GUI_KEYPAD0**, **GUI_SCREEN1** is associated with **GUI_KEYPAD1**, **GUI_SCREEN2** is associated with **GUI_KEYPAD2**, and **GUI_SCREEN3** is associated with **GUI_KEYPAD3**. GUI Keypads can contain three different types of keys - action keys, data entry keys, and shift keys. Each type of key is covered in a later section. Only the keypad of the visible screen is active (i.e. responds to key presses).

Properties

Table 8-9 Properties of the `GUI Keypads`.

Property	Values	Meaning
DATA_ENTRY_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when a non-overloaded data entry key is pressed or an overloaded data entry key is pressed and a timeout has been reached. If zero, no event procedure is executed.
INT_DATA_ENTRY_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when an overloaded data entry key is pressed and a timeout has not been reached. If zero, no event procedure is executed.
SHIFT_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when any shift key is pressed. If zero, no event procedure is executed.
DATA_ENTRY_TEXTBOX	Textbox object	The textbox that contains the keycodes of pressed data entry keys.
SHIFT_STATE_TEXTBOX	Textbox Object	The textbox that contains a string corresponding to the shift state.
KEYPAD_BEEP	**GUI_TRUE, GUI_FALSE**	If true, sounds the buzzer when a key is pressed.
KEYPAD_BEEP_DURATION	Unsigned Integer	Controls the length of time (in 5 ms intervals) the piezo electric buzzer is on when a key is pressed. The default value is 1, corresponding to 5 milliseconds.
KEY_CODE	1 – 255	Read only. Contains the key code of the last pressed data entry key. See the following section on Data Entry Keys for more information on Key Codes.
PRIOR_KEY_CODE	1 – 255	Read only. Contains the key code of the data entry key that was pressed before the last data entry key.
KEY_EVENT	**PRESS_EVENT, HOLD_EVENT, RELEASE_EVENT, NO_EVENT**	Read only. Contains the last event.

Property	Values	Meaning
PRIOR_KEY_EVENT	PRESS_EVENT, HOLD_EVENT, RELEASE_EVENT, NO_EVENT	Read only. Contains the event that occurred before the last event.
KEY_POSITION	0 – 15 or 20 – 34	Read only. Contains the key position of the last key. See Table 8-1 for the key positions on the keypad.
PRIOR_KEY_POS	0 – 15 or 20 – 34	Read only. Contains the key position of the key pressed before the last key.
KEY_TYPE	ACTION_KEY, DATA_ENTRY_KEY, SHIFT_KEY, NO_KEY	Read only. Contains the type of key that was last pressed.
PRIOR_KEY_TYPE	ACTION_KEY, DATA_ENTRY_KEY, SHIFT_KEY, NO_KEY	Read only. Contains the type of key that was pressed before the last key.
PRESS_NUMBER	Unsigned Integer	Read only. Contains the number of times the last data entry key was pressed before a timeout expired.
SHIFT_STATE	SHIFT_UP, SHIFT_NEUTRAL, SHIFT_DOWN	Contains the current shift state.

Methods

Table 8-10 Methods of the `GUI Keypads`.

Method	Inputs	Outputs	Meaning
Insert_Key	Keypad Reference, Key Reference, and Key Position	None	Inserts a key object into the specified keypad object at the specified position. Valid positions are shown in Table 8-1.

Errors

Table 8-11 Errors of the `GUI Keypads`.

Error	Associated Method	Meaning
INVALID_KEY	Insert_Key	The specified object reference is not a valid key.
INVALID_KEYPAD	Insert_Key	The specified object reference is not a valid keypad.
KEY_POS_OUT_OF_RANGE	Insert_Key	The specified key position is not between 0 and 15 or 20 and 34.

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 8-12 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	Unsigned 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 8-13 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 8-14 Errors of the GUI_PEN.

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

GUI_BUZZER

The **GUI_BUZZER** corresponds to the piezo electric beeper and controls the length of the beep when the **Buzz** method is called. The **GUI_BUZZER** is useful for alarms and system notifications.

Properties

Table 8-15 Properties of the GUI_BUZZER.

Property	Values	Meaning
BEEP_TIME	Unsigned Long	Controls the length of time the piezo electric buzzer is on during a call to the **Buzz** method. 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.
BEEPER_ON	**GUI_TRUE**, **GUI_FALSE**	If true, turns the piezo electric buzzer on.

Methods

Table 8-16 Methods of the GUI_BUZZER.

Method	Inputs	Outputs	Meaning
Buzz	None	None	Turns the piezo electric buzzer on for **BEEP_TIME**.

GUI_BATTERY

The **GUI_BATTERY** contains properties corresponding to the 6 AA rechargeable Nickle Metal Hydride batteries. The **GUI_BATTERY** is used to monitor the state of the batteries. For more information about the lifetime, replacement, and charging of the batteries, see the chapter on Powering Your Handheld.

If reading the battery voltage, current draw, battery state, and charger state, be sure to install jumpers J3 and J4 on the Motherboard to connect the A/D inputs to the battery. Before you read any properties of the battery, be sure the call **Read_Battery** to update the battery properties. If all 8 Port E lines are required by your application, don't install the jumpers on the Motherboard and don't call **Read_Battery**. To monitor the battery and use the remaining Port E lines as digital or analog inputs, you must get the 8 bit A/D resource before reading the other Port E lines and then release the resource when finished.

Properties

Table 8-17 Properties of the GUI_BATTERY.

Property	Values	Meaning
BATTERY_STATE	**FULL_BATTERY, MEDIUM_BATTERY, LOW_BATTERY, NO_BATTERY**	Read only. Contains the current state of the battery.
BATTERY_VOLTAGE	Unsigned Long	Read only. In the 16 least significant bits, contains the current battery voltage in millivolts.

Property	Values	Meaning
BATTERY_CURRENT	Unsigned Long	Read only. In the 16 least significant bits, contains the current draw of the Handheld in milliamps.
CHARGER_STATE	GUI_TRUE, GUI_FALSE	Read only. If true, the battery charger is on.

Methods

Table 8-18 Methods of the GUI_BATTERY.

Method	Inputs	Outputs	Meaning
Initialize_Battery	None	None	Turns on the 8-bit A/D circuitry of the processor which is used to read the battery voltage, current usage, battery state, and charger state.
Read_Battery	None	None	Reads the battery voltage, current usage, battery state, and charger state and stores it into the appropriate properties.

GUI_TIMER

The GUI_TIMER allows you to repetitively call non-critical event procedures during a specified interval. The GUI_TIMER is useful for calling routines every so often – like reading the battery once a minute and showing the voltage on the display. If the repetitively called event procedure reads, writes, or changes any GUI Toolkit property or calls a GUI Toolkit method, the first line of the event procedure must disable the servicing of events and the last line of the event procedure must re-enable the servicing of events. See the demo program for an example. The GUI_TIMER should only be used for non-critical event scheduling because events may be delayed by interrupt service routines and other tasks. For critical event timing, use an interrupt service routine.

Methods

Table 8-19 Methods of the GUI_TIMER.

Method	Inputs	Outputs	Meaning
Schedule_Event	Extended Address, Interval, Start Time, End Time	None	Schedules an event procedure to be repetitively called. The interval, start, and stop time is specified in 5 ms increments. If 0 is specified for the start time, the event procedure is called immediately. If 0 is specified for the end time, the event procedure is repetitively call forever or until the event procedure is removed by Remove_Event. No error checking is performed.
Remove_Event	Extended Address	None	Removes an event procedure from being called by the GUI_TIMER. No error checking is performed.

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 Handheld 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 Handheld 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 Mosaic's Terminal Program. 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 8-20 Properties of Graphic Objects.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 128 pixels	Read only. Contains the width of the image in pixels.
HEIGHT_IN_PIXELS	1 to 128 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 Handheld 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 Mosaic's Terminal Program, the terminal program provided by Mosaic Industries. The Image Data File, by default, puts the converted image data into RAM on page 0x01 and then transfers the data into Flash at page 0x10. For more information about the memory map of the Handheld Controller, see *Chapter 4: Making Effective Use of Memory*.

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 bitmap (*.bmp) and PCX (*.pcx) files. 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 Conversion Program has a main screen, an advanced options screen, a font selection 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 type of controller you are using. A button labeled "Convert Files Now" starts the conversion process. In the advanced options screen there are controls that select your programming language, desired target memory location for the image data, and filenames for the Image Data and Image Header files. All of the advanced options are set to default values that will work for most applications. In the font selection screen there are controls that allow you to preview and select fonts to use with the GUI Toolkit. Once a font is selected, a bitmap image is made of all of the characters of the font and a data file is created that contains the width of each character. The bitmap file and the data file are then stored into the currently selected directory. The Image Conversion program will then add the font bitmap file and the character width data file along with your images into the Image Data File and the Image Header File. Be sure to select your working directory before selecting a font and be sure to select the BMP image format to convert your selected fonts. 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 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 a single bitmap image of all 255 characters joined together along with a data file that contains the widths of each character. Fonts are created on the PC using the Image Conversion Program (see the previous section for more information). The GUI

Toolkit provides a custom proportional font that is 10 pixels tall called the **GUI_FONT**. The demo program uses a Comic Sans font to demonstrate how to create and use font objects. While you can't create another instance of the **GUI_FONT**, you can create your own custom fonts as described in the previous section.

Properties

Table 8-21 Properties of the Font Object.

Property	Values	Meaning
HEIGHT_IN_PIXELS	10 – 128 pixels.	Read only. Contains the height of a single character in pixels.
IMAGE	Extended Address	Contains a pointer to the image of the characters contained in the font.
IMAGE_WIDTH_DATA	Extended Address	Contains a pointer to the array that contains the width in pixels of each character contained in the font.

Textbox Object

Textboxes are used in conjunction with fonts to render 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 a 10 pixel tall, proportional font; each character has a different width depending on the glyph or symbol of the character. If you want to change the font of a textbox, you must change it before you set the size of the textbox. Once you have set the size of a textbox, the font can not be changed.

The size of a textbox is uninitialized when it is instantiated. The size of the textbox must be set (using the **Set_Property** method along with the **HEIGHT_IN_PIXELS** and **WIDTH_IN_PIXELS** properties) before a character or string is written into the textbox. Once the size of a textbox is set, it cannot be changed. When the size of a textbox is initialized, a block of memory in the heap is allocated for the textbox to render its string into. The string in a textbox must be rendered before it can be shown on the display. The **RENDER** property controls whether or not the string in a textbox is rendered. The **RENDER** property is set to **GUI_TRUE** when a textbox is instantiated. Turning this property off is useful when a string in a textbox needs to be changed or modified frequently but does not need to be shown on the display.

Properties

Table 8-22 Properties of the Textbox Object.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 128 pixels	Write once. Contains the width of the textbox in pixels.

Property	Values	Meaning
HEIGHT_IN_PIXELS	10 to 128 pixels	Write once. Contains the height of the textbox in pixels.
TEXTBOX_ROWS	Unsigned Long	Read only. The 16 least significant bits contain the number of rows of the textbox. Equals to the textbox height divided by the font height.
STRING_LINES	Unsigned Long	Read only. The 16 least significant bits contain the number of lines the string in the textbox occupies when rendered. Based on the number of characters in the string and the textbox width.
FIRST_VISIBLE_LINE	Unsigned Long	The 16 least significant bits contain the line number of the string that is written to the first row of the textbox. Typically used for scrolling long strings in textboxes. See the demo for an example.
STRING_LENGTH	Unsigned Long	Read only. The 16 least significant bits contain the length of the string in the textbox.
TEXTBOX_FONT	Font reference	Contains a reference to the font object used to render the string in the textbox. This property can only be set before the size of the textbox is set.
RENDER	GUI_TRUE, GUI_FALSE	If true, immediately renders changes to the textbox string.
INVERT	GUI_TRUE, GUI_FALSE	If true, inverts the background and rendered string. If using this property, be sure to set this property to the desired value before writing to the textbox.
BORDER	GUI_TRUE, GUI_FALSE	If true, draws a one-pixel border around the textbox.

Methods

Table 8-23 Methods of the Textbox Object.

Method	Inputs	Outputs	Meaning
Add_Character	Textbox Reference, Character	None	Adds a character to the end of the string in a textbox.
Delete_Character	Textbox Reference	None	Deletes a character from the end of the sting in a textbox.
STRING_TO_TEXTBOX	Textbox Reference, String	None	Writes a string up to 255 characters into a textbox. Overwrites any existing string.
Textbox_To_String	Textbox Reference	String	Reads the first 255 characters of the string in a textbox.
Clear	Textbox Reference	None	Sets all of the characters in the textbox to ASCII space.

Errors

Table 8-24 Errors of the Textbox Object.

Error	Associated Method	Meaning
INVALID_OBJECT	Clear	The object reference is not a textbox.
INVALID_TEXTBOX	Add_Character, Delete_Character, Textbox_To_String, STRING_TO_TEXTBOX	The object reference is not a textbox.
HEIGHT_OUT_OF_RANGE	Set_Property	The height is greater than the height of the display or less than the height of the textbox font.
WIDTH_OUT_OF_RANGE	Set_Property	The width is greater than the width of the display or less than the width of the widest character of the textbox font.

Controls

Controls are objects that respond to keypad events such as key presses and releases. The GUI Toolkit has three types of controls for data entry and instrument control – Action Keys, Data Entry Keys, and Shift Keys; each are described in the following sections.

Action Key Object

Action keys are used to invoke custom program actions such as turning on a motor, changing the contrast level of the display, or starting a process. The custom program actions are performed using event procedures or functions that you write that can be assigned to run when an action key is pressed, held, or released. Event procedures cannot receive or return values but they can modify global variables. If you need to write an event procedure that needs to read, write, or modify GUI Toolkit properties, the servicing of events must be disabled using **BLOCK_ON_PRESS**, **BLOCK_ON_HOLD**, and **BLOCK_ON_RELEASE** to prevent data corruption. If using **BLOCK_ON_PRESS**, **BLOCK_ON_HOLD**, and **BLOCK_ON_RELEASE**, remember to re-enable the servicing of events (by setting the **GUI_TOOLKIT** property, **SERVICE_EVENTS** to **GUI_TRUE**) prior to leaving the event procedure. When an Action Key object is instantiated, all of its properties are initialized to 0.

Properties

Table 8-25 Properties of the Action Key Object.

Property	Values	Meaning
PRESS_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the key is pressed. If zero, no event procedure is executed.
HOLD_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the key is held. If zero, no event procedure is executed.

Property	Values	Meaning
RELEASE_EVENT_PROCEDURE	Extended Address	Execute this user defined event procedure when the key is released. If zero, no event procedure is executed.
BLOCK_ON_PRESS	GUI_TRUE, GUI_FALSE	If true, disables the servicing of events when the key is pressed.
BLOCK_ON_HOLD	GUI_TRUE, GUI_FALSE	If true, disables the servicing of events when the key is held.
BLOCK_ON_RELEASE	GUI_TRUE, GUI_FALSE	If true, disables the servicing of events when the key is released.

Data Entry Key Object

Data entry keys are used to enter alphanumeric characters and symbols into the DATA_ENTRY_TEXTBOX of a keypad. Data entry keys may be overloaded; that is the same key can be used to enter many different characters or symbols. The alphanumeric characters and symbols are stored as key codes in the Data Entry Key object. Key codes are numbers from 1 to 255 that refer to the images of the characters of a font that are copied to a textbox when a string is rendered. Key codes are stored into the Date Entry Key object by using the Set_Property method. When a Data Entry Key object is instantiated, all of its properties are initialized to 0; be sure to initialize the Data Entry Key object (by storing key codes into the properties) before you use it.

Properties

Table 8-26 Properties of the Data Entry Key Object.

Property	Values	Meaning
SHIFT_NEUTRAL_VALUE	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed and the shift state is shift neutral.
NUM_SHIFT_UP_VALUES	0 to 5	Contains the number of shift up values in the Data Entry Key.
SHIFT_UP_VALUE0	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is first pressed and the shift state equals to shift up.
SHIFT_UP_VALUE1	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed twice and the shift state equals to shift up.
SHIFT_UP_VALUE2	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed three times and the shift state equals to shift up.
SHIFT_UP_VALUE3	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed four times and the shift state equals to shift up.
SHIFT_UP_VALUE4	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed five times and the shift state equals to shift up.

Property	Values	Meaning
NUM_SHIFT_DOWN_VALUES	0 to 5	Contains the number of shift down values in the Data Entry Key.
SHIFT_DOWN_VALUE0	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is first pressed and the shift state equals to shift down.
SHIFT_DOWN_VALUE1	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed twice and the shift state equals to shift down.
SHIFT_DOWN_VALUE2	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed three times and the shift state equals to shift down.
SHIFT_DOWN_VALUE3	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed four times and the shift state equals to shift down.
SHIFT_DOWN_VALUE4	1 to 255	The key code that is added to the Data Entry Textbox when the data entry key is pressed five times and the shift state equals to shift down.

Shift Key Object

Shift keys are used to set the Shift State of the keypad. Changing the Shift State allows Data Entry Keys to be overloaded. When a Shift Key object is instantiated, the Shift Type is set to SHIFT_NEUTRAL, and SHIFT_TOGGLE is set to GUI_FALSE.

Properties

Table 8-27 Properties of the Shift Key Object.

Property	Values	Meaning
SHIFT_TYPE	SHIFT_UP, SHIFT_NEUTRAL, SHIFT_DOWN	Contains the shift type of shift key.
SHIFT_TOGGLE	GUI_TRUE, GUI_FALSE	If true, causes the shift key when pressed, to toggle between the SHIFT_NEUTRAL state and value in the SHIFT_TYPE property.

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 120 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 8-11 Using the Plot Object.

```
int plotRandom;
void Example_Plot( void )
{
  int random_num;
  // Create a new plot object.
  plotRandom = New_Object( PLOT );
  // Center plot object on the screen.  Plot size is 240 x 120 pixels.
  Load(GUI_SCREEN0, plotRandom, 0, 0);
  random_num = Random();
  Add_Data(plotRandom, (random_num % 119));
  while(1)
  {
    random_num = Random ();
    Add_Data(plotRandom, (random_num % 119));
    Refresh(GUI_SCREEN0);
    Pause();
  }
}
```

Properties

Table 8-28 Properties of the Plot Object.

Property	Values	Meaning
WIDTH_IN_PIXELS	8 to 128 pixels	Contains the width of the plot in pixels.
HEIGHT_IN_PIXELS	1 to 128 pixels	Contains the height of the plot in pixels.
BUFFER_SIZE	Unsigned 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 8-29 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 8-30 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.

Screen Object

Screens that are created using the `New_Object` method reside in the heap. Screens that reside in the heap can not be made visible - the only difference between heap screens and GUI Screens. Heap screens are useful for applications with many different screens. All screens can be instantiated and loaded when the application first starts up. Then, during run time, the screens that are needed can be quickly copied to the GUI Screens for viewing.

Properties

Table 8-31 Properties of Screen Objects.

Property	Values	Meaning
`WIDTH_IN_PIXELS`	128 pixels	Read only. Contains the width of the screen in pixels.
`HEIGHT_IN_PIXELS`	128 pixels	Read only. Contains the height of the screen in pixels.
`VISIBLE`	`GUI_FALSE`	Read only. All non GUI screens have this property set to `GUI_FALSE` since they can not be made visible.
`NUM_OBJECTS`	Unsigned Long	Read only. Contains the number of objects contained in the screen.
`IN_DISPLAY`	`GUI_FALSE`	Read only. All non GUI screens have this property set to `GUI_FALSE` since they reside in the heap
`SCREEN_ADDRESS`	Extended Address	Read only. Contains the heap handle to the address of the rendered image of the screen.

Methods

Table 8-32 Methods of Screen Objects.

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, X Coordinate	None	Unload an object from a screen at the specified x and y coordinates.
`Copy_Screen`	Source Screen Reference, Destination Screen Reference	None	Copies the contents of the source screen into the destination screen. The contents of the destination screen are over written.
`Clear`	Screen Reference	None	Unloads all objects from the screen and erases the screen.
`Refresh`	Screen Reference	None	Re-renders graphics, 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 8-33 Errors of Screen Objects.

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

Customizing the Keypad Overlay

You can customize the "look and feel" of your instrument by creating your own graphic to be used over your keypad. Figure 8-3 shows an example graphic.

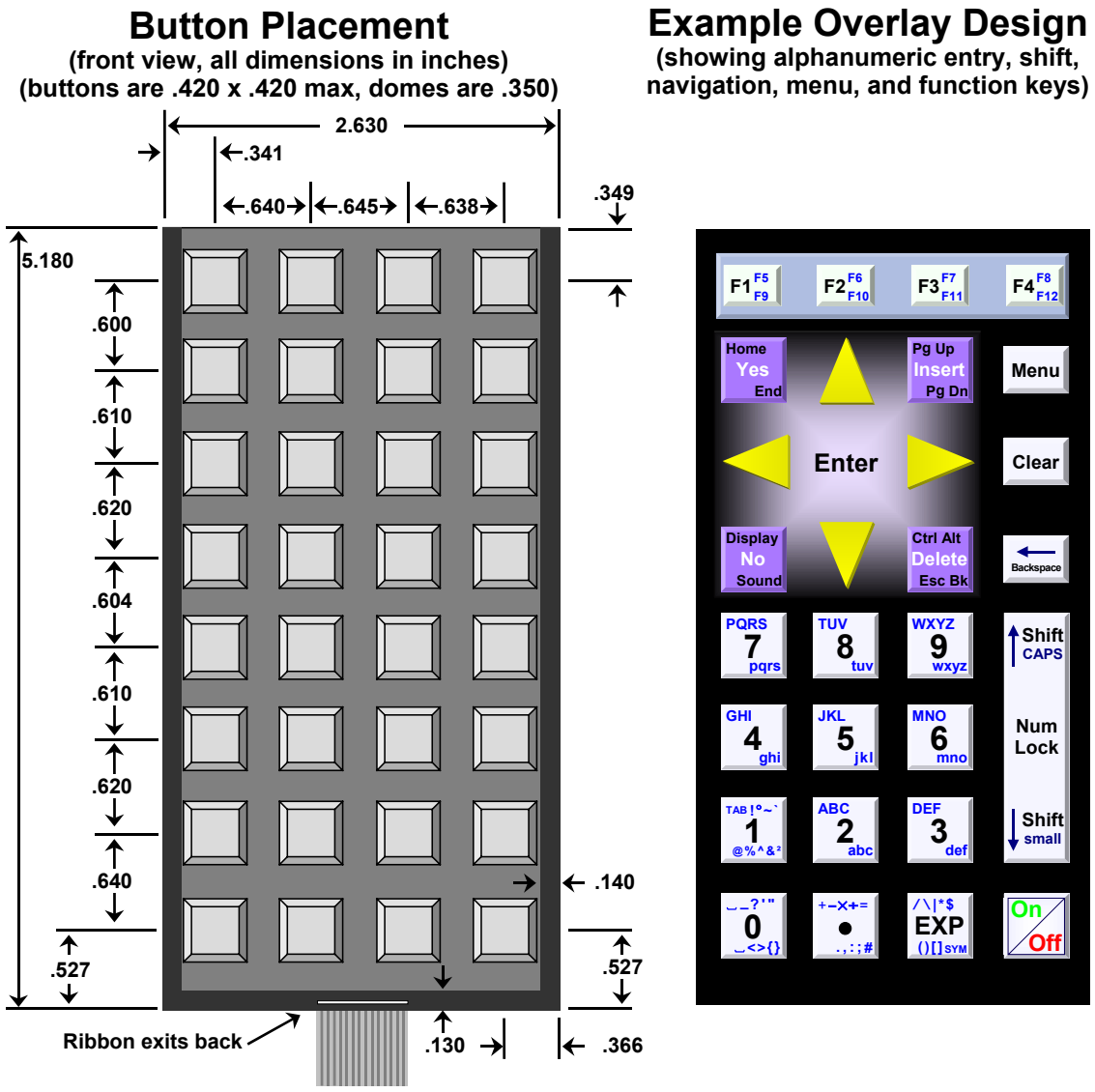


Figure 8-3 Keypad layout showing precise button positions, and a sample design for key layout.

The keypad includes a clear plastic overlay which forms a pocket into which you can place a paper graphic. When shipped from the factory, the keypad is loosely attached with tape to the enclosure; the adhesive on the back of the keypad is not used to securely fasten the keypad to the enclosure. This allows you to pull up the keypad and install your own custom graphic. Once you have finalized your keypad design, you can remove the backing on the keypad (exposing the adhesive) and

permanently attach the keypad to the enclosure. You can also remove the clear plastic overlay and instead use an adhesived plastic sheet with your own graphic.

Figure 8-3 provides the dimensions of the keypad, showing the precise placement of the buttons. Note that they are not arrayed perfectly evenly; your graphic design should take into account their precise placement. also shows an example overlay design including alphanumeric entry keys, shift and menu keys, navigation keys, and function keys.

One key is special: The bottom right corner key is used to turn the handheld on and off. It is directly wired to the handheld's power circuitry and it is not scanned with the other keypad keys by the keypad driver routines. Your graphic should show this dedicated key as the on/off button.

Using the Keypad Pocket

You can print a graphic image on ordinary photo paper and insert it into the keypad pocket. We've had the best success printing on "Epsom Photo Quality Ink Jet Paper (Matte Finish)" Part Number S041062, or equivalent: 27 lb, ISO 90 brightness (or greater), 4.9 mil, coated paper. Using a thin paper will not interfere with the buttons' tactile feedback, while much thicker paper stock may. After printing, the image can be cut to size to fit within the pocket, i.e., to 2.350 x 5.050 in., and carefully slid into the pocket. You can also achieve a better finished look if the image is inserted into the keypad pocket slightly underlapping the black border of the keypad's top clear plastic layer. If so, cut the image slightly larger. To insert it you will need to peel back the pocket's top clear plastic layer. Peel it back starting at the top – it is pressure-sensitive-adhesived at the side and bottom edges only. Then place the overlay onto the keypad, carefully centering it so that its edges slightly underlap the black borders of the top plastic layer, and carefully aligning its top edge with the top edge of the keypad. Then place the top layer back on, and reseal the pocket by running a finger around the bottom and side edges to reattach the pressure sensitive adhesive.

Printing on a Plastic Overlay

Alternatively you can print directly on plastic sheets and use them in place of the top layer on the existing keypad. For use as a top layer, the cropped size of your graphic should be 2.630 by 5.180 in.

Chapter 9

Digital and Timer-Controlled I/O

Overview of Available Digital I/O

The Handheld provides up to 8 digital I/O lines, 8 analog input lines (which may be used as digital intputs), and up to three communications channels. The digital I/O lines originate from two 8 bit ports on the CPU (68HC11) named PORTA and PORTE. Table 9-1 summarizes the uncommitted digital I/O available.

Table 9-1 The Handheld's Uncommitted Digital I/O

I/O Lines	Type	Port Address	Comments / Alternate Uses
6	Timer-controlled inputs or outputs including 3 input-capture, 3 output-compare, and pulse accumulator	PA 0-2, 5-7	Bit-by-bit configured by the application as inputs or outputs, including: Timed inputs: PA 0-2 Timed outputs: PA 5-7 Pulse accumulator: PA 7

There are a total of 6 uncommitted digital I/O lines for your use. After initialization or reset, all digital I/O lines are configured as inputs, but they may be reconfigured as outputs.

In addition to these I/O lines there are several committed to other services on the controller; these are summarized in Table 9-2.

Table 9-2 Committed I/O pins

Service	Port	Pins
8-bit A/D	CPU PORTE	PE 0-7
Serial 2	CPU PORTA	PA 3-4

For applications requiring even more digital I/O, I/O lines usually committed to the 8-bit A/D and the secondary serial port may be redirected as general purpose digital I/O if these other services are not needed. Table 9-3 summarizes the digital I/O lines gained if other services are not used.

Table 9-3 Additional digital I/O lines made available if other services are forfeited and their committed I/O pins freed.

Services Used		Digital I/O Available		
8-bit A/D	Serial 2	Inputs	Outputs	Total
Yes	Yes	0 to 6	0 to 6	6
No	Yes	6 to 14	0 to 6	14
No	No	8 to 16	0 to 8	16

The maximum number of digital inputs and outputs is 16 (up to 16 can be configured as inputs, up to 8 as outputs) if none are used for the 8-bit A/D or the secondary serial port. Table 9-3 summarizes the *digital* I/O and alternate use 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 Handheld with a wide variety of digital accessories.

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: 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 Handheld originate from a Motorolla 68HC11 processor chip. The 68HC11 provides two 8 bit ports named PORTA and PORTE.

Table 9-4 summarizes the digital input/output available on the Handheld including the names, addresses, and number of signals associated with the digital ports on the 68HC11. 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.

Table 9-4 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)
PORTE	800A	8	Bytewise digital or analog input	8 bit A/D:	PE0-7 (8)

Table 9-5 specifies the named data direction register which controls the input/output direction, or specifies the functions that configure each digital I/O port.

Table 9-5 Digital I/O port data direction registers and configuration functions.

Port Name	Configured By
68HC11:	
PORTA	`DDRA`
PORTE	`AD8On()` `AD8Off()`

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 Handheld.

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 Handheld 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.

PORTA

PORTA is configurable as input or output on a bit-by-bit basis. To configure **PORTA**, use an assignment statement to write to the **DDRA** (Data Direction Register A) register. **DDRA** and all 68HC11 register names are defined in the **QEDREGS.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory. Writing a 1 to a bit position in **DDRA** configures the corresponding port bit as an output, and writing a 0 to a bit position configures the corresponding bit as an input. For example, the following C statement configures **PORTA** as all outputs:

```
DDRA = 0xFF;
```

To configure PORTA as all inputs, use the statement:

```
DDRA = 0x00;
```

If we want to configure bits 0-6 as inputs, and bit 7 as output, we can execute:

```
DDRA = 0x80;
```

To change the state of an output bit on **PORTA** of the 68HC11 chip, use an assignment statement with **PORTA** on the left hand side to write to the port's data register named **PORTA**. For example, if **PORTA** is configured as all outputs, the following C statement sets all **PORTA** bits high:

```
PORTA = 0xFF;
```

To read the state of **PORTA**, use an assignment statement with **PORTA** on the right hand side to read the port's data register. For example, the following code fragment reads **PORTA** and places the results in the variable named **latest_porta_state**:

```
static unsigned char latest_porta_state;
latest_porta_state = PORTA;
```

PORTE

PORTE (named in the **QEDREGS.H** file) is an 8 bit analog or digital input port. **PORTE** is configured as a digital input after a reset or restart, and is read in the same way that **PORTA** is read: simply use it as the right hand side of an assignment statement. For example, to read the digital state of PORTE, your program could execute the statements:

```
static unsigned char latest_porte_state;
latest_porte_state= PORTE;
```

To configure PORTE for analog input, use the function:

```
AD8On()
```

To turn off the 8 bit A/D and revert to a digital input port, use:

```
AD8Off()
```

(For experts and the curious: **AD8Off()** turns the 8 bit analog converter off by clearing the A/D power up bit named **ADPU** in the **OPTION** register; **AD8On()** sets the **ADPU** bit.)

Using Uninterruptable Operators

The Importance of Uninterruptable Operators

Care must be taken when performing "read/modify/write" operations in applications that use interrupts or multitasking. Operations such as setting or clearing individual bits in a byte while leaving other bits unchanged are called "read/modify/write" operations because they involve reading the port, modifying the read contents, and writing the result back to the port. Unpredictable results can occur if more than one interrupt service routine or task tries to access a single port or memory location at the same time using a read/modify/write sequence. The simplest solution to this problem is to access the memory location or port using an "uninterruptable" read/modify/write operator.

The following scenario illustrates the importance of 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 PORTA. Assume that TASK1 is controlling the state of bit 4, and TASK2 controls bit 7. Let's assume that bit 4 is low when TASK2 tries to execute the following code:

```
static unsigned char mask = 0x80;
PORTA |= mask;
```

TASK2 is merely trying to set the top bit in PORTA to 1, but this statement may have unintended consequences. The compiler generates code that reads the contents of PORTA, performs a bitwise OR with the contents of mask, and stores the result back into PORTA. 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 final store to PORTA is executed. Unfortunately, this store command erroneously sets bit 4 back to the low state! TASK2 was interrupted after it read the state of PORTA but before it had a chance to write the new contents, so it undoes the change that TASK1 made in the state of PORTA bit 4! This can indeed cause problems in an application program.

Pre-coded Read/Modify/Write Functions

The pre-coded read/modify/write functions avoid this problem by disabling interrupts for ten to sixteen cycles (2.5 to 4 microseconds at a 16 MHz crystal speed). This prevents the corruption of the contents when different tasks or interrupts share access to a single location. The following functions are uninterruptable:

```
void  ChangeBits( uchar data, uchar mask, xaddr address )
void  ClearBits( uchar mask, xaddr address )
void  SetBits( uchar mask, xaddr address )
void  ToggleBits( uchar mask, xaddr address )
```

Additional uninterruptable operators are declared in the **XMEM.H** header file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory; these routines are described in detail in the Control-C Glossary.

Create Your Own Uninterruptable Functions

It is easy to create your own uninterruptable functions using the **_protect** keyword. For example, the following uninterruptable function sets specified bits in a port or memory byte:

```
void _protect  SetBitsUninterrupted( uchar mask, char* address )
{  *address |= mask;
}
```

In response to the **_protect** keyword, the compiler ensures that interrupts are temporarily disabled while **SetBitsUninterrupted()** is executing, and that the global interrupt mask (the I bit in the condition code register) is restored to its prior state after the function terminates.

Simple stores to and fetches from 1-byte or 2-byte memory locations are intrinsically uninterruptable because they are implemented with single assembly-language opcodes. However, the 68HC11 processor does not have a single opcode that can access a 32 bit memory location. Thus, problems can arise when one task writes to a floating point or long 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. In these cases uninterruptable operators should be used. An example is presented by the functions named:

```
PeekFloatUninterrupted()
PokeFloatUninterrupted()
```

which are defined using the _protect keyword in the **TURNKEY.C** example program found in the last chapter of this book.

Connecting Hardware to the Digital Outputs

On the Handheld the processor's port A is 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 on page A-3 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 port A pins also have inherent diode clamps to the chip's +5V supply voltage, V_{DD}, but it is 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 are similar in electrical characteristics to the SN54/74HC digital logic family. They can source or sink up to 25 mA and are guaranteed to source 0.8 mA while providing a valid logic high and to sink 1.6 mA at a valid logic low, although they generally do much better. 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 is loaded, the V_{OL} and V_{OH} levels

rise or fall. 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, it can be modeled as a voltage source of either zero or five volts and an equivalent series resistance of 40 ohms. At greater output currents the resistance increases until at the greatest specified current for any one pin, 25 ma., the equivalent resistance is 60 ohms. At this current the voltage degradation of the V_{OL} or V_{OH} is 1.5 volts. Figure 9-1 and Figure 9-2 illustrate this variation.

These figures can be used to choose component values for particular circuits. For example if we wish to use a pin of Port A to drive a light-emitting diode we would place the LED in series with a resistor and connect them between an output pin and ground. The resistor limits the current, to the LED. From the LED 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 calculate it as,

Eqn. 9-1 $\mathbf{R = (V_{OH} - 2.2\ V\ ) / 10\ mA}$

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.

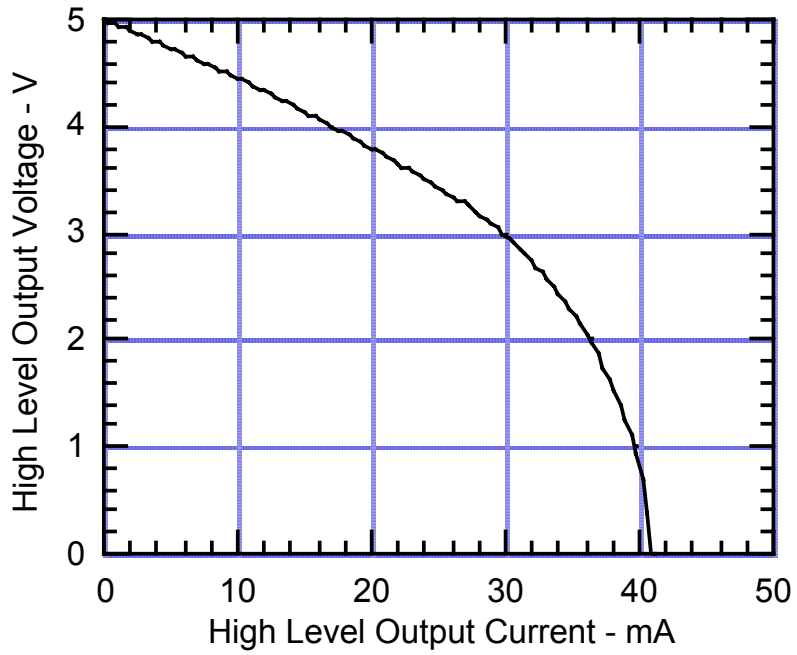


Figure 9-1 Degradation of the Port A or Port D output high voltage with current. The maximum current allowed for continuous operation is 25 ma.

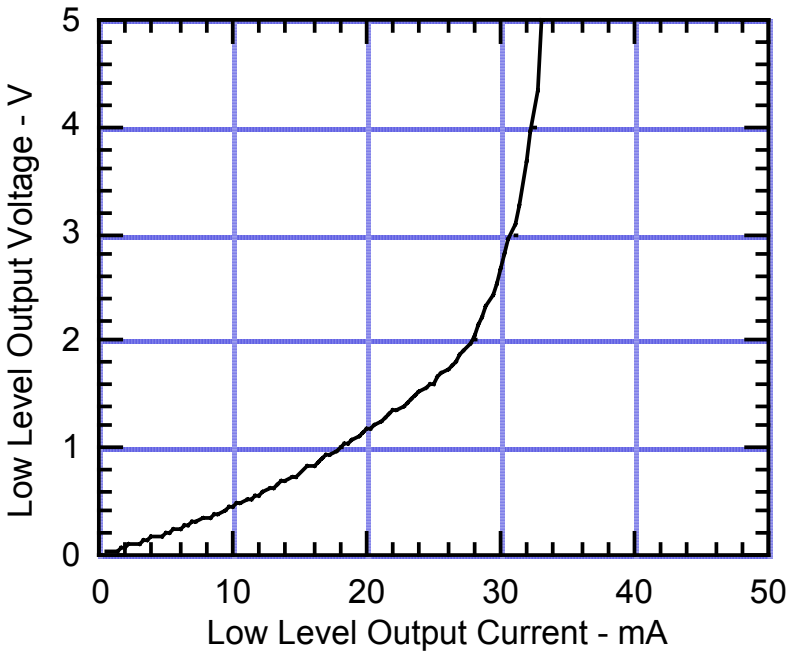


Figure 9-2 Degradation of the Port A or Port D output low voltage with current. The maximum current allowed for continuous operation is 25 ma.

The Processor's Output Compare Functions

The processor's programmable timer subsystem contains 5 output compare (OC) functions (named OCx for x from 1 to 5) associated with PORTA output pins PA7 through PA3 (in descending order). These output compare functions allow you to specify actions that take place at particular, well determined times. Using output compares, it is easy to set up real-time clocks, cause periodic execution of code, and generate precisely timed synchronous or asynchronous waveforms. They can be used to implement a stepper motor controller, pulse generator, pulse width modulation (PWM) signals, precisely timed output pulses, timesliced multiplexing, or serial communications.

Output-compare functions work by automatically changing PORTA output pins and/or invoking an interrupt service routine (ISR) whenever the contents of a free-running 16-bit counter (TCNT) matches the contents of user-set output-compare registers, TOCx. When these contents match, we say that a "successful output compare" has occurred. Thus the programmer can precisely specify a future time at which an action will occur by simply storing the time as a 16-bit value in the appropriate output compare register, TOCx. The free running counter counts at a programmable rate, from 0 to 65536, then rolls over to zero and continues counting. Its rate is one count each 2 microseconds, for a rollover period of 131.072 milliseconds. Consequently, you can set up output compares to trigger events with a resolution of 2 microseconds, and up to 131.072 milliseconds into the future (or arbitrarily longer if you keep track of rollovers on TCNT).

Because TCNT is clocked by a prescaler driven from the system E clock, you can change the count rate by modifying the prescaler's division ratio, from its current value of 8 to 1, 4, or 16, changing its rollover period to 16.384, 65.536, or 262.144 milliseconds. If you do though, the system

timeslicer will be affected. We find that a 2 microsecond tick rate provides sufficient resolution for most applications.

Each of the five output compare subsystems has a 16-bit TOCx register, a successful compare OCxF (interrupt) flag, and an interrupt mask OCxI, where "x" is the output compare number. OC1 can control any of pins PA3 through PA7 and it has its own register to specify which of PORTA pins are affected. Output compares OC2, OC3, OC4 and OC5 each control a single pin. They each have a pair of output mode/level bits, OMn and OLn, which determine the effect that each successful compare has on PORTA bits PA6, PA5, PA4, and PA3 respectively. The processor automatically sets an output compare's OCxF flag bit in the TFLG1 register when the contents of the TCNT register and the OC's TOCx register are equal. At the same instant, the state of the associated PORTA pin is set, cleared, or toggled as specified by the output mode bits. In addition, if the output compare's OCxI mask bit in the TMSK1 register is set, an interrupt is recognized when a successful compare occurs.

An active output compare function can cause a signal change on a PORTA pin at a specified time T, and/or trigger an interrupt at time T:

- Signal Change – To cause a signal change on a PORTA pin when time T equals the contents of TCNT, the PORTA pin must be configured as an output, and the output compare function must be enabled. OC1 is enabled by storing the data to be output in OC1D and specifying the pins to be changed in OC1M, and OC2-OC5 are enabled simply by storing a 2-bit code into TCTL1 specifying the desired signal change. Using the CFORC register, it is also possible for software to immediately force a state change on a timer-controlled signal without causing an interrupt.
- Interrupt – To trigger an interrupt when time TCNT = T, an interrupt handler must be installed, and the output compare's local interrupt must be enabled by setting bits in TMSK1.

Table 9-6 Output Compares and their properties

Output Compare	Controlled PORTA Pin	Comparison Register	Comments and Alternate Use
OC1	PA3, 4, 5, 6, and/or 7	TOC1	May control multiple pins simultaneously, or be paired with another OC to jointly control pin PA3, 4, 5, or 6.
OC2	PA6	TOC2	The OC2 timer is used by the kernel's timeslicer and elapsed time clock functions. If you do not use these functions, pin PA6 may be controlled by OC1 or used for general purpose I/O.
OC3	PA5	TOC3	Not used by the kernel.
OC4	PA4	TOC4	Used as an output by the secondary serial port. Available if you do not need Serial 2.
OC5	PA3	TI4O5	Shared with Input Capture 4, which is used as an input by the secondary serial port. Available if you do not need Serial 2.

As summarized in the table, the output compares are not all identical in function, and some are used by the operating system. OC1 and OC5 differ from the others slightly in function, OC2 is used by the operating system's timeslicer, and OC4 and OC5 are used by the secondary serial port:

- OC1 is special in that it can synchronously control any of PORTA pins PA7, PA6, PA5, PA4, and PA3. Even when OC1 is used to control several PORTA pins, the timer and interrupt functions of those pin's output compares may still be used independently of the pin (to implement clocks, etc.). While the other output compares can be used to set, clear, or toggle an output pin, OC1 can be used only to set or clear a pin, but not to toggle it.
- OC2 is used by the timeslicer. Consequently, if you need the services of the timeslicer (timesliced task switching or elapsed time clock function) make sure that you do not use OC2 for other functions. Even if you do use the timeslicer, pin PA6 is still available for use as general purpose I/O, or as an output controlled by OC1.
- OC4 is used as an output by the secondary serial port, so you can't use it or its associated pin PA4 if you need the second RS232 serial link.
- OC5 shares its timer register and output pin with input capture 4 (IC4). OC5 operates like the other output compares, but it must be initialized by clearing the I4/O5 bit of the PACTL (pulse accumulator control) register before it may be used. Also, IC4/OC5 and associated pin PA3 are used by the secondary serial port, so be sure not to use these resources if you need the second RS232 serial link.

Pulse and PWM Generation Techniques

The processor's output compare functions provide lots of flexibility for creating single pulses or pulse-width-modulated waveforms. Most methods are variations on this algorithm:

1. The desired start time of the pulse is programmed by storing an appropriate count in the output-compare register (TOCx) of OCx, and the OCx interrupt is enabled by setting a flag bit in TMSK1.
2. OCx's mode/level bits (OMx and OLx) are configured to automatically set the output compare's corresponding output either high or low, depending on the polarity of the desired pulse (this action enables the output compare).
3. When the compare occurs, the pin state is automatically changed and an interrupt service routine called.
4. The interrupt service routine (ISR) reprograms the output compare to automatically change its pin back to its inactive level on the next compare;
5. The ISR also increments the output-compare register by a value corresponding to the desired duration of the pulse.

Since the pin-state is changed by hardware automatically at specific values of the free-running counter, the pulse width is controlled accurately to the resolution of the free-running counter irrespective of software latencies. By repeating the actions for generating a pulse, you can generate an output signal of a specific frequency and duty cycle. While software latency and execution times do

not affect the timing of the waveform, they do impose limits on the frequency and duty cycles attainable. The different methods of generating PWM signals differ primarily on where the software execution times are constrained to fit, either within the on time, the off time, or the waveform's period as a whole.

The following is a quick summary of some of some specific ways you can use output compares to generate pulses or PWM waveforms. You can find complete descriptions of the registers mentioned in Motorola's MC68HC11F1 Technical Data Manual. Example 1 shows you how to generate a tirggered pulse, Examples 2 and 3 represent exceptional and instructive methods of generating PWM signals, and examples 4-6 are commonly used PWM methods. Example 6 provides code for generating "failsafe" PWM signals.

Example 1 – Creating a Single Precise Pulse from an External Trigger

Suppose you'd like to output a single pulse on PA5 with very precise duration in response to a triggering event, for example the leading edge of an input pulse on PA0. You'd like the output pulse to start a precise, fixed time after the initiating trigger and to last some precise duration, between 2 microseconds and 131 milliseconds. Further, the output pulse should occur after the first occurrence of the trigger (after enabling the system) and then stay off rather than being retriggered by subsequent input pulses.

To do this you would use one input capture (IC3) and two output compares (OC1 and OC3). Because the trigger may occur anytime an input capture is used to determine the precise time of its leading edge. And because the output pulse width must be precisely controlled, and its duration can potentially be shorter than any interrupt service routine, the pulse should be turned on and off by output compares. The input capture invokes an interrupt service routine that enables the output compares. For our example we'll assume the desired pulse start time is 10 milliseconds (or 5000 TCNT counts) after the trigger is detected, and its duration is precisely 10 microseconds (5 TCNT counts). Here's one way to do it using three routines:

- ⇨ An initialization routine (called ENABLE) that configures and enables IC3 to capture a rising edge on PA0, forces PA5 off to initialize it, and configures and enables OC3 to turn off pin PA5;
- ⇨ An interrupt service routine for IC3 (called IC3_ISR) that programs the comparator registers of OC1 and OC3 and enables OC1; and,
- ⇨ An interrupt service routine for OC1 (called OC1_ISR) that disables OC1 so that after the output pulse is first turned on it is not turned on again.

Here's how to write the routines:

ENABLE – Inhibit IC3, OC1, and OC3 interrupts by storing 0x0xxxx0 to TMSK1 (using CLEAR.BITS with a mask of 0xA1). Configure OC1 to set PA5 high on an output compare by setting bit 5 of OC1D, but don't enable it using OC1M yet – the IC3_ISR will be responsible for doing that. Configure OC3 to clear its associated pin (PA5) on an output compare by storing appropriate mode/level bits (OM3 and OL3), that is xx10xxxx, into the timer control register (TCTL1). Initialize PA5 to the OFF state by forcing an early output compare on OC3. This is done by setting bit 5 of CFORC. Configure and enable IC3 to capture a rising edge

PA0. This is done by clearing bit 0 of DDRA (to set the data direction of PA0 to input), and setting the lower two bits of TCTL2 (the two corresponding to IC3) to xxxxxx01 to capture a rising edge. Clear the interrupt flag bit (IC3F) left over from any prior edge detection, if any, by writing a one to IC3F in TFLG1. Finally, enable an IC3 interrupt by setting the IC3I bit in TMSK1.

IC3_ISR – On entering the ISR routine first disable further interrupts by clearing the IC3I bit in TMSK1, and clear the interrupt flag bit (IC3F) by writing a one to IC3F in TFLG1. Then read TIC3 to determine the trigger time, we'll call it TT. Set up OC1 to turn on output pin PA5 at time TT+5000 by setting TOC1=TT+5000 and configure OC3 to turn it back off just 5 counts later, by setting TOC3=TT+5005. Enable an OC1 interrupt by setting the OC1I bit in TMSK1. Finally enable OC1 and tie it to PA5 by setting bit 5 in OC1M.

OC1_ISR – This interrupt service routine will be invoked when the output pulse is turned on. It only needs to disable OC1 so that further pulses are not produced until the system is re-enabled by executing ENABLE again. Disable OC1 by clearing bit 5 in OC1M. Disable interrupts on OC1 by first clearing the OC1I bit in TMSK1, then clear the interrupt flag bit (OC1F) by writing a one to OC1F in TFLG1.

Example 2 – Two OCs Generate a PWM Waveform Without Interrupts

Using two output compares you can generate a PWM waveform of any duty cycle (from 1/65536 to 65535/65536) without using interrupt service routines. As no ISR is used, no software resources are needed to maintain the waveform, and there is no impact on overall system performance. The PWM waveform is free – from a software perspective. So what's the catch? The catch is that the waveform must have a fixed period, equal to the rollover period of the free-running counter, or 131.072 milliseconds. If you can live with that, here's how it's done: Two output compares are used, one sets the output pin at a particular value of TCNT, and the other simply resets the pin at another TCNT value. One of the output compares must be OC1 – because it can be used in conjunction with another to control the same pin. Let's use OC1 to turn ON an output pin (PA5) whenever TCNT hits 0x0000, and OC3 to turn it OFF whenever TCNT hits 0x1000. More specifically,

1. Disable interrupts caused by OCs by storing 0x00 to the timer interrupt mask register 1 (TMSK1). Now, output compares will not cause interrupts. Even so, they can still control output pins.

2. Use OC3 to turn OFF the pulse whenever TCNT=0x1000 (for example, for a duty cycle of 1/16) by storing 0x1000 into TOC3. Configure OC3 to clear its associated pin (PA5) on a successful compare by storing appropriate mode/level bits (OM3 and OL3), that is xx10xxxx, into the timer control register (TCTL1). Now, whenever TCNT hits 0x1000 pin PA5 will be cleared. We configure the OFF transition first because we don't want the pin to get stuck ON before we're done configuring our output compares, just in case.

3. Use OC1 to turn ON the pulse whenever TCNT=0x0000 by storing 0x0000 into TOC1. Associate OC1 with pin PA5 by setting bit 5 in the output compare mask register OC1M. Configure OC1 to set the pin high by storing 0x20 into the OC1 data register, OC1D. Now, whenever TCNT hits 0x0000 pin PA5 will be set high.

Because an ISR isn't used, there is no possibility of software delays influencing the PWM output.

Advantages: Precise transition time control, all duty cycles possible with 16-bit resolution, no software needed to keep things running, failsafe in that a software crash is not likely to affect operation.

Disadvantages: Only a single channel, only a single PWM period.

Example 3 – A Single OC-Driven ISR Generates Many PWM Channels

What if you want to generate many channels of PWM waveforms, more than there are output compares? In the prior example output compares were used to generate a precisely-timed high-resolution waveform on a single channel without the assistance of an ISR. This example provides the other extreme, an unlimited number of channels of low-resolution PWM signals are generated by a single output-compare-driven interrupt service routine.

The scheme uses a single output compare to periodically invoke an interrupt service routine. The OC is not tied to a PORTA pin, rather it is used only as a dedicated clock that calls the ISR at a fixed interval corresponding to the smallest ON or OFF time that can be produced. Each time the ISR is called it updates all the PWM outputs using any general purpose output pins available. It may do this by reading values from a look-up table, counting, or more computationally.

Because the ISR latency can vary depending on what other interrupt-driven services are running on the controller there is some jitter (or variance) on the transition times, producing a slightly varying PWM duty cycle. This variation can be compensated on the average if the ISR reads TCNT to determine the actual ON and OFF times and modulates the next ON or OFF time to attain the desired PWM duty cycle averaged over a number of cycles – but of course that requires greater execution time.

The sum of the ISR latency and execution times must be less than the difference between adjacent OC times, placing a limit on the smallest ON or OFF time attainable. If the ISR is delayed so long that the next OC time is missed, the next ISR doesn't occur until TCNT rolls over and another match occurs. Consequently rollover delays of a TCNT period may be inserted into the desired ON or OFF times.

For an example of software generated PWM signals with 8-bit resolution and best averaging properties see "MI-AN-056 Optimal PWM Algorithm" and "MI-AN-058 Using Port PPA for PWM".

Advantages: Any number of channels can be accommodated.

Disadvantages: ISR latency causes jitter in the transition times; ON or OFF times smaller than the ISR latency and execution time are not possible; rollover delays possible if timing criteria not met; not failsafe.

Example 4 – One OC and ISR Generates a "No Jitter" PWM Signal

The simplest way of generating a precise PWM waveform with arbitrary duty cycle and period is to use a single output compare to automatically turn on and off an output pin and an interrupt service routine that reprograms the output compare after each transition. The "off" transition invokes the ISR which sets up the turn-on time and programs the next output state to be "on", and the "on" transition invokes the same ISR to set up the turn-off time and the next "off" state. The on and off times must each be great enough to contain the latency and execution time of the ISR. So duty cycles that

would require very small on or off times are not attainable. If the ISR is delayed so that it does not program the next transition in time, than the output compare doesn't find a match until TCNT rolls all the way around. In this case rollover delays of approximately 131 msec may be inserted into either the on or off time.

An example of this approach is presented by the function named `OC3Service()` in the `TURNKEY.C` example program found in the last chapter of this book.

Advantages: Transitions are precise, with no jitter; duty cycle and period are programmable over a wide range.

Disadvantages: ON or OFF times smaller than the ISR execution time and latency are not possible; rollover delays are possible; not failsafe – a software crash can leave the output stuck ON.

Example 5 – Two OCs and ISRs Generate a "No Jitter" PWM Signal of any Duty Cycle

To obtain ON and OFF times that may be each as small as a single clock tick, two OCs and ISRs are required. The "off" transition invokes an ISR which sets up the next turn-off time, and the "on" transition invokes a different ISR to set up the next turn-on time. Each ISR simply increments the next ON or OFF comparison register by the period. There is no particular restriction on the shortness of the ON or OFF times, either can extend down to just a single count of TCNT, but their sum, the period, must be great enough to contain the latency and execution times of both ISRs. Because OCs are used to drive the output pin, the transition times are exact, with no jitter. If service of either ISR is delayed for more than a period then a rollover delay may be inserted into either the on or off time.

Advantages: Transitions are precise, with no jitter; duty cycle and period are programmable over a wide range, duty cycle extends fully from 1/65536 to 65535/65536.

Disadvantages: Rollover delays are possible; not failsafe – a software crash can leave the output stuck ON.

Example 6 – One OC and ISR Generates a "Failsafe" PWM Signal

A single OC and associated ISR is used. The OC invokes an ISR for each transition. For the ON transition, the ISR is responsible for setting the output pin. It also computes the next turn-off time based on the time at which the pulse is actually turned on, and programs the OC to automatically turn it back OFF at the turn-off time. At the next OC the pin is turned off by hardware and the ISR is again called. This time the ISR just sets the turn-on time for the next OC time, disconnecting the OC from the pin so that the pin is not automatically set. This sequence of events produces pulses whose duration is invariant with respect to ISR delay, but that may jitter back and forth within their fixed period. Despite any jitter, the duty cycle and period are both precisely controlled. If the ISR is delayed by more than the off time, rollovers are inserted into only the off time, never the on time. So the pulse on times are failsafe so long as the processor's clock is running.

Advantages: Failsafe operation assures an ON pulse is never longer than desired and the pulses turn off on a software crash. ON time may be as small as a single TCNT count.

Disadvantages: Pulse position jitter; rollover delays are possible in the OFF time; minimum OFF time must be greater than the IST latency and execution time.

Table 9-7 PWM Methods.

Method	ISR Used	Output Compares	Turn ON/OFF	Jitter	D.C. Accuracy	D.C. Range	Rollovers inserted?	Period
2	no	OC1 and one other	OC/OC	none	Perfect	any	no – failsafe	Fixed at 131.072 ms
3	yes	any OC	ISR/ISR	yes	limited by ISR delays	Ton, Toff >ISR	yes	on and off times must each be greater than ISR latency and execution times
4	yes	any OC	OC/OC	no	Perfect	Ton, Toff >ISR	yes	on and off times must each be greater than ISR latency and execution times
5	yes	OC1 and one other	OC/OC	no	Perfect	Ton, Toff unlimited	yes	P>ISR1+ISR2
6	yes	any OC	ISR/OC	yes	Perfect	Ton unlimited, Toff>ISR	failsafe – into only the off time	

Chapter 10

Data Acquisition Using Analog to Digital Conversion

This chapter describes the analog inputs available on the Handheld, explains how to connect the converters to external signals, and details the built-in driver routines that make the analog inputs easy to use. Simple code is presented to calculate measured voltages based on A/D readings.

Data Acquisition Using the Handheld

Many instrument applications require monitoring of analog signals. Analog to digital (A/D) converters can perform this function. An A/D converter samples analog signals and converts them to digital values that can be stored, processed, or displayed.

The resolution of an A/D is specified in bits. For example, an *8-bit* A/D converts an analog signal into one of 256 discrete digital numbers.

Table 10-1 Analog I/O

I/O Lines	Type	Port Address	Comments / Alternate Uses
8	8-bit 0-5 V analog inputs	PE 0-7	Alternately may be used as digital inputs.

The Handheld hosts an analog to digital converter to address a wide variety of instrumentation and control applications. It includes an 8-channel 8-bit analog to digital (A/D) converter that is built into the 68HC11 processor chip. It converts unipolar signals with a nominal 0 to +5 volt range;

The analog inputs are brought out to the 24 pin Field Header on the Handheld; the connector diagrams in Appendix B 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.

Examining the Demonstration Program

The code discussed in this section is located in the file named **AD8.C** in the **\MOSAIC\DEMOS_AND_DRIVERS\MISC\C EXAMPLES** directory. Most of the functions in this file are interactive versions of functions declared in the **ANALOG.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory; they are described in detail in the Control-C Glossary.

We recommend that you compile and download the **AD8.C** program now so that you can interactively work through the exercises in this chapter. Simply use your TextPad editor to open **AD8.C**, and click on the Make Tool to create the download file named **AD8.DLF**. Then enter the Mosaic Terminal program, and type:

```
COLD←
```

to terminate any prior multitasking program that might be running, and select the "Send File" menu item to send **AD8.DLF** to the Handheld.

Fundamentals of Analog to Digital Conversion

An analog to digital converter samples an analog signal and outputs a digital number that is proportional to the analog signal. The A/D converters on the Handheld sample input voltages and communicate the digital result to the 68HC11 processor.

The analog input signal must be within the input range of the A/D converter. On the Handheld, the lower bound of the range is equal to the voltage on VRL (voltage reference/low) and the upper bound of the allowable input range is equal to the voltage on VRH (voltage reference/high). By default, their values are 0 Volts (analog ground) and 5.0 Volts (analog +5V), respectively.

The converter measures the input voltage with a certain specified resolution. The resolution is the granularity with which the measurement is performed. It can be specified as a number of bits or as a voltage increment. For example, an A/D converter with only 1 bit of resolution and a 5 Volt input range would classify all voltages from 0 to just under 2.5 Volts as the digital value 0, and voltages from 2.5 Volts to 5 Volts as the digital value 1. This converter has a resolution equal to 1 bit, which corresponds to 2.5 Volts per count.

The converter measures the input voltage with a specified accuracy. The accuracy tells how close the measured value is to the actual voltage. A typical 8 bit A/D converter is accurate to within plus or minus one least significant bit.

Determining the Resolution of an A/D Converter

The V_{RL} and V_{RH} analog input reference pins define the lower and upper voltages that can be converted by the 8 bit A/D. The resolution depends on the input voltage range. The measurement resolution of a B-bit A/D, expressed in Volts per count, is

Eqn. 10-1 $\text{Resolution} = (V_{RH} - V_{RL}) / 2^B$ **[Volts per count]**

where 2^B is the number of counts that can be represented by a B-bit number. From this equation we see that resolution becomes finer (better) as B grows larger or as the reference voltage range (VRH - VRL) gets smaller. For the 8 bit A/D the resolution is

Eqn. 10-2 8 bit Resolution = $(V_{RH} - V_{RL})/256$ [Volts per count]

With the default V_{RL} and V_{RH} of 0 V and 5 V, respectively, the resolution of the 8 bit converter is 19.5 mV per count.

Converting an A/D Count into Its Equivalent Voltage Reading

To convert the 8-bit count returned by the A/D converter into an equivalent voltage, use the formula

Eqn. 10-3 Input Voltage = V_{RL} + (Count * Resolution)

Combining this with Eqn. 10-1 yields

Eqn. 10-4 Input Voltage = V_{RL} + Count * $(V_{RH} - V_{RL}) / 2^B$

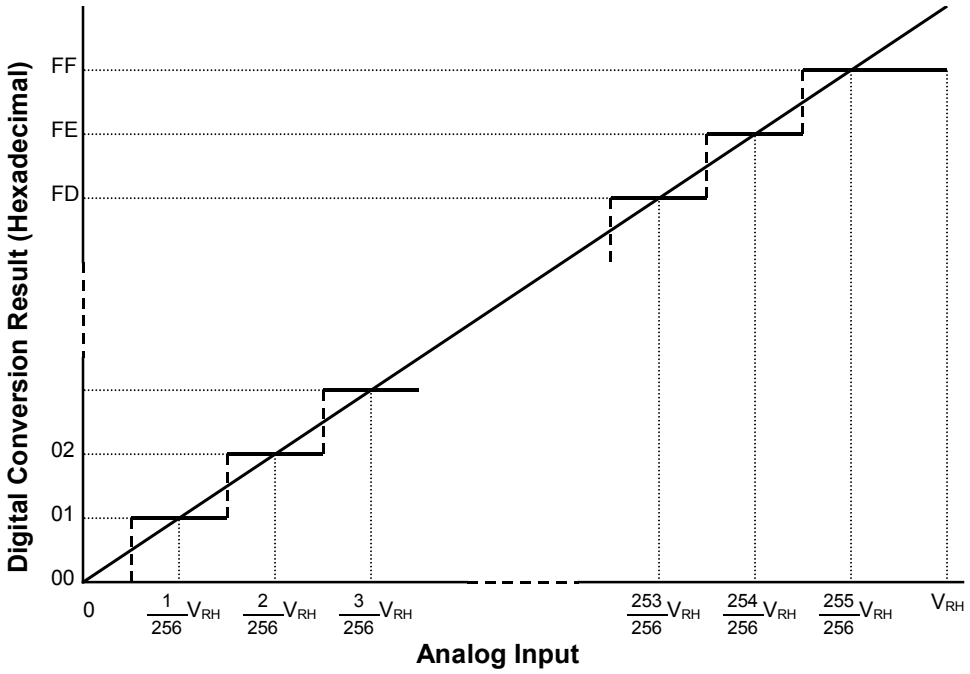


Figure 10-1 Conversion function for the 8-bit A/D.

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 Handheld is first

powered up, or after a reset or restart, the 8 bit A/D converter is disabled and **PORTE** is configured as a digital input port. To turn on the A/D converter, a program must call the function:

```
AD8On()
```

Another function named **AD8Off()** is available to turn off the 8 bit A/D so that PORTE once again acts as a digital input port.

The function named

```
InitAnalog()
```

defined in the **AD8.C** file calls **AD8On()**. If you interactively execute **InitAnalog()**, as,

```
InitAnalog( )←
```

the 8 bit A/D converter will be turned on and ready for use.

In an autostarting application you should include **InitAnalog()** 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 **PE0**. The 8 bit A/D inputs are the PORTE signals named PE0 through PE7 available at pins 17 through 24 on the Field Header (see Appendix A). Each analog signal is converted to a number between 0 and 255 indicating its value relative to VRL (the low voltage reference) and VRH (the high voltage reference). The default values are VRL = 0 Volts (analog ground) and VRH = 5 Volts. The exact voltage difference between VRH and VRL varies slightly from board to board; it is a good idea to measure the value on your board to obtain the most exact voltage equivalents of the measured A/D results.

Interactively Perform the Conversion

The **Convert8()** function is defined near the middle of the **AD8.C** file; you can see from its definition that it is simply an interactively callable version of the **AD8Sample()** function. Now that you have connected an input voltage to channel **PE0**, you can type from your terminal:

```
Convert8( int 0)←
```

The printed return value summary displays the conversion count; you'll see a printout that looks something like this for a 1.5 volt signal:

```
Rtn:  -30722    77  =0x87FE004D =fp:  -3.822E-34
```

We know that this function returns an integer, so we identify "77" (hex 0x004D) as the return value; the other numbers in the summary are irrelevant. If the input is 1.5 volts, the result should be approximately 77 counts (256 * 1.5 / 5.0). 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 255. If the input is exactly half of (VRH - VRL), the result will equal decimal 128.

Multiple 8 Bit A/D Conversions with Results Stored in a C Array

The function **AD8Multiple()** which is defined in the **ANALOG.H** file expects as inputs a buffer xaddress (32 bit extended address), a sampling interval parameter, the number of samples, and a channel number. When called, it performs the specified number of conversions and saves the results as single bytes in the specified buffer. As explained in the Glossary, the sampling interval parameter specifies the timing of the samples, with 0 representing the fastest sampling, and 65,535 representing the slowest sampling. You can sample at up to 100 kHz (100,000 samples per second) using the 8-bit A/D provided the processor is devoted to this single task.

The **AD8ToCArray()** function defined in **AD8.C** uses a standard C one-dimensional array named **results_8** as the data buffer. The function accepts a channel number as input, performs conversions at the fastest sampling speed (100 kHz), and places the results in the array. The number of samples is specified by the **DEFAULT_NUMSAMPLES** constant which equals 16. Because we have connected the channel 0 input (AN0), let's perform the multiple conversion on this channel. Type at your terminal:

```
AD8ToCArray( int 0)←
```

remembering to type at least one space after the **(** character. This function does not return a value, so the printed return value summary is not relevant. To see the results of the conversion, you could write a simple C function that prints the contents of **results_8**; this is left as an exercise for you. You can also use the debugging routine named **DUMP** to view a hexadecimal dump of the buffer. To do this, type at your terminal:

```
results_8  0  16  DUMP←
```

DUMP is a QED-Forth function that expects as inputs an address (in this case, **results_8** puts the address on the data stack), a page (0, representing the common page), and the number of bytes to be dumped (16). You should see a printout similar to this if you connected a 1.5 volt signal:

```
pg  addr 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
00  8E2D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D 4D  MMMMMMMMMMMMMMMM
```

This tells us that the contents of the 16 bytes starting at address 0x8E2D (which is the address of **results_8)** all equal 0x4D. To the right of the hex dump the ascii equivalent of the memory contents is displayed; this is not relevant for the numerical data here. We conclude that the A/D conversion result was 0x4D, equivalent to decimal 77; this is the same value returned by the **Convert8()** function above. Your results may vary slightly depending on the exact value of supply voltage on the Handheld, as the +5V supply is used as the default reference voltage.

For Experts: How To Use Additional Features of the 8 Bit A/D

The built-in driver routines for the 8 bit A/D are easy to use and address the requirements of most applications. If you wish to gain a detailed understanding of the operation of the 8 bit A/D, or need to use one of its special modes, the information in the Analog-To-Digital Section of the MC68HC11F1 Technical Data Manual may prove useful. For example, one of the operating modes allows the 8 bit A/D converter to continuously sample four different analog inputs in rapid succession.

Chapter 11

Serial Communications

RS-232 and RS-485 Communications

The Handheld has two serial communications ports: a primary serial port called Serial 1 that supports both RS232 and RS485 protocols, and a secondary serial port called Serial 2 that supports RS232. The Serial 1 port is implemented with the 68HC11's on-chip hardware UART (*Universal Asynchronous Receiver/Transmitter*). Serial 2 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.

Table 11-1 Serial Communications Channels

Channels	Type	Port Address	Comments / Alternate Uses
1	Serial 1: RS232/485 hardware UART at up to 19.2 KBaud	PD 0-1, PD 5 used for RS-485.	
1	Serial 2: RS232 software UART at up to 4800 Baud	PA 3-4	PA 3-4 may be used as timer-controlled I/O (one input-capture and one output-compare) if Serial 2 is not used. PA 3-4 may also be used for hardware handshaking for Serial 1.
1	Synchronous Serial Peripheral Interface at 2 MBaud	PD 2-5	
3	Serial communications channels		

The primary channel's 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. It controls the serial-to-parallel and parallel-to-serial conversion and performs all of the timing functions necessary for *asynchronous* serial communications. The communications is asynchronous because no synchronizing clock signal is transmitted along with the data. Rather, the UART deduces the correct time to sample the incoming signal based on the start and stop bits in the signal itself. (The Handheld also supports fast synchronous serial communications via the Serial Peripheral Interface described later in this Chapter.)

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 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. The Serial 1 port can be configured for either RS232 or RS485 communications at up to 19200 baud. The Serial 2 port is dedicated to RS232 communications at up to 4800 baud.

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 Handheld's serial ports. Its simplest implementation requires only three wires: one to transmit serial data, a second to receive serial data, and a third to provide a common ground reference. RS232 allows both communicating parties to transmit and receive data at the same time; this is referred to as *full duplex* communications. RS232's greatest benefit is its universality; practically all personal computers can use this protocol to send and receive serial data. If your computer does not have an RS-232 serial port, low cost USB-to-RS-232 serial cables are available; contact Mosaic Industries for details.

RS232 uses inverse logic; that is, a positive bit at the 68HC11 UART is inverted by the onboard RS232 driver chip and appears as a negative signal on the serial cable. The terminal's serial receiver chip re-inverts the signal to its positive sense. The data bits are also transmitted in reverse order, with the least significant bit transmitted first, after a start bit. The specified signal levels of approximately +/- 9 Volts are derived from the Handheld's +5 Volt supply by a dual RS232 driver chip that has a built-in charge pump voltage multiplier.

RS485

RS485 is another protocol supported by the primary serial port on the Handheld. 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.

Like RS232, the data bits are transmitted in reverse order, with the least significant bit transmitted first. The RS485 protocol uses differential data signals for improved noise immunity; thus RS485 can communicate over greater distances than RS232. An RS485 transceiver is present on the Handheld, and its data direction is controlled by pin 5 of port D of the 68HC11.

Serial Connectors and Configuration Options

The primary and secondary serial communications ports are accessible through the Handheld's 10 pin, dual row Communications Header (H4) on the Processor Board. Several serial communications cables are also supplied with Handheld Starter Kits.

The pinout of the Handheld's Communications Header (H4) is shown in the following table.

Although the RS232 protocol specifies functions for as many as 25 pins, each communications channel requires only three for simple serial interfaces: TxD1 (transmit data), RxD1 (receive data), and DGND (digital ground). The RS232 protocol specifies the use of two separate grounds, a signal ground and a protective (or "chassis") ground. The Handheld does not differentiate between these. To provide a convenient means of attaching two grounds to the serial cable, there are several pins (labeled DGND) on the communications connector that are connected to the controller's ground plane.

Table 11-2 Serial Communications Header, H4

Signal	Pins	Signal
/TxD1	1 – 2	/RxD1
DGND	3 – 4	/MOD1.CS
RS485 XCVR-	5 – 6	RS485 XCVR+
/TxD2	7 – 8	/RxD2
DGND	9 – 10	+VBAT

Given the availability of ready-made communications cables, it is not necessary to study or understand the following descriptions of cable connections. These detailed signal descriptions and cable diagrams are presented to provide complete information for those who have special communications requirements and for those who wish to make their own application-specific communications cables. Most computers conform to IBM PC AT-compatible RS232 interfaces which use 9-pin D-Type connectors, consequently the Handheld brings out its serial ports to two female 9-pin D-Type connectors on the PowerDock. Table 11-3 shows the connection diagram for a standard 9-pin serial cable.

Table 11-3 Serial Cable Connections.

Cable Pin	PC AT or Terminal	Handheld
1	NC	DTR1/DSR1
2	RxD	TxD1
3	TxD	RxD1
4	DTR	DSR1/DCD1
5	GND	GND
6	DSR	DTR1/DCD1
7	RTS	CTS1
8	CTS	RTS1
9	NC	NC

We can gain insight into the operation of the RS232 protocol by examining the signal connections used for the primary serial port in Table 11-3. The transmit and receive data signals carry the messages being communicated between the Handheld and the PC or terminal. The Handheld's transmit data signal TxD1 (pin 2 on the 9-pin serial connector) is connected to the terminal's receive data signal RxD (pin 2 on its 9-pin connector). Likewise, the terminal's transmit signal TxD is con-

nected to the Handheld's receive signal RxD1. Chassis and signal grounds are connected together to the digital ground (DGND) signal.

From the Handheld's point of view, these three signals (TxD, RxD, and ground) are the only connections required to perform serial communications. While these signals provide a data path, they do not provide hardware *handshaking* that allows the two communicating parties to let each other know when they are ready to send or receive data.

The RS232 protocol provides for four handshaking signals called *ready to send* (RTS), *clear to send* (CTS), *data set ready* (DSR), and *data terminal ready* (DTR) to coordinate the transfer of information. The Handheld, however, does not implement hardware handshaking. Rather, it relies on software handshaking via transmission of XON/XOFF characters to coordinate data transfer and ensure that information is not lost when one of the communicating parties is busy.

Many terminals and PCs, however, do rely on hardware handshaking to determine when the other party (in this case the Handheld) is ready to accept data. By connecting pairs of these handshaking signals together, the terminal or PC can be made to think that the Handheld is always ready to send and receive data. Thus in Table 11-3 , RTS1 is connected to CTS1, and DSR1 is connected to DTR1 and DCD1 onboard the Handheld using zero ohm shorting resistors. These signals may alternatively be redirected to the digital inputs and outputs used by the second serial port if hardware handshaking is required.

The secondary serial port is connected similarly except that the onboard connection of RTS to CTS, and DSR to DTR are permanent.

Enabling RS485 Communications

If your application requires RS485, use the primary serial port (serial1) for RS485 communications, and use the secondary serial port (Serial 2) to program and debug your application code using the RS232 protocol. The default serial routines used by the onboard kernel assume that full duplex communications are available, so you cannot use the RS485 protocol to program the controller. You can use it to communicate with other devices.

A jumper, J9 on the Processor Board, configures the primary serial port for either RS232 or RS485 operation.

> For RS232 operation: Remove the jumper shunt from J9. In this case, cable connections may be made to Serial 1 on either the 10-pin Serial Communications Header or the DB25 Connector if jumpers J1-J3 are configured appropriately on the Personality Board.
>
> For RS485 operation: Install the jumper shunt onto J9. In this case, cable connections must be made to Serial 1 at pins 5 and 6 of the 10-pin Serial Header.

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. The standard C serial I/O routines such as **`printf()`**, **`scanf()`**, **`putchar()`**, and **`getchar()`** give you high level access to the serial ports. All high

level routines call the following low level revectorable serial primitives to access the currently active serial port:

```
int   AskKey(void)   // returns a flag that is true if an input char is waiting
char  Key(void)      // waits for and returns the next input char
void  Emit(char)     // outputs the specified char to the serial port
```

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

Let's do a quick experiment to see how easy it is. We'll use code from the **GETSTART.C** program. If you have already downloaded the program, you are ready to go. If your board is presently running a multitasking application, type

```
WARM←
```

to stop the program now.

If you have not yet compiled the **GETSTART** program and you want to do the exercises here, open **GETSTART.C** in your TextPad editor, click on the Make Tool, and after the compilation is done, enter Mosaic Terminal by clicking on the terminal icon and use the "Send File" menu item to send **GETSTART.DLF** to the Handheld.

Switching the Default Serial Port

Before running the program, let's switch to the secondary 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 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 "Comm" 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 on the PowerDock. Typing a carriage return at the terminal should now produce the familiar "**ok**" response via the Serial2 port.

Now type:

```
main←
```

and you'll see the familiar starting message of the **GETSTART.C** program:

```
Starting condition:
The radius is       0; the circular area is       0.
ok
```

In fact, the program works the same as it did before, but now it is using the secondary serial port instead of the primary port -- and you didn't even have to recompile the code!

For those of you interested in the details, here's how it works: The low-level serial driver routines named **Key()**, **AskKey()** 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.

You can invoke the C version of this routine by calling

```
UseSerial2()
```

anywhere within your C program's source code file. Function prototypes for this function and other versatile serial I/O routines are defined in the **COMM.H** header file, and are described in detail in the *Control-C Glossary*.

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 19200 baud). A hardware reset (pressing down on the reset switch) has the same effect. If you do this now, remember to move the Handheld's serial connector back to Serial Port 1, and to change the terminal's baud rate back to 19200 baud using the "Comm" item under the terminal's "Settings" menu.

If you always want the Handheld 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.

All of these functions that we are calling interactively via the operating system can also be called from C in your program source code; their C function prototypes are as follows:

```
void UseSerial1( void );
void UseSerial2( void );
void Baud2( int baud );
void Serial1AtStartup( void );
void Serial2AtStartup( int baud );
```

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

```
SERIAL_ACCESS = RELEASE_ALWAYS;
SERIAL_ACCESS = RELEASE_NEVER;
```

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 Handheld's kernel software contains a complete set of high level driver routines for the Serial2 port, and these functions are summarized in the Control-C 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 Handheld 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 Control-C 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 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 Handheld 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

```
void Baud1AtStartup( int baud );
void Serial2AtStartup( int baud );
```

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.

Multi-Drop Communications Using RS-485

Connecting computers together in multi-drop networks is common in factories and laboratories. In these distributed processing networks, a variety of machines and instruments work locally, but communicate and share data or resources with one another globally using a single serial link. You can use the Handheld's RS485 link to create such a multi-drop serial network.

In the most common multi-drop RS-485 protocol, one computer is designated as a "master" and the rest of the computers or devices on the serial bus are designated as "slaves". At any given time, only the master and a single "active" slave communicate. The remaining "inactive" slaves may actively receive, or listen to, data on the communications line, but only one slave at a time can transmit a message. If more than one slave tried to drive the transmit line simultaneously, their serial drivers would fight with each other for control of the bus. To ensure that no two devices drive the network at the same time, it is necessary that each slave device be able to disable it's own RS-485 data transmitter.

Software Implementation of an RS485 Network

Because the requirements of every multi-drop application are so unique, it is difficult to specify or design a software protocol that meets everyone's needs. This section describes the QED-Forth routines that control the RS485 transceiver, and presents some ideas that may prove useful in designing a multi-drop data exchange protocol.

The Handheld controls the RS485 transceiver with bit 5 of Port D of the processor. When this bit is high, the transceiver is in transmit mode. When it is low, the transceiver is in receive mode. QED-Forth includes three built-in routines to facilitate control of the RS485 transceiver. They are:

```
void InitRS485( void );
void RS485Receive( void );
void RS485Transmit( void );
```

`InitRS485()` configures Port D to ensure that bit 5 is an output. **`RS485Receive()`** clears bit PD5 to place the transceiver in receive mode, and **`RS485Transmit()`** sets bit PD5 to place the transceiver in transmit mode.

To use a Handheld as a slave in a multi-drop network, simply define a word, (named `Silence(void)`, for example) that when executed calls `RS485Receive()` to wait for any pending character transmission to complete, then disable the transmitter, and then execute a routine such as `Key()` to listen to the communications on the serial bus. The `Silence()` routine searches the incoming serial characters for a pre-determined keyword (for example, the ascii "name" of this particular slave). When the network master wants to talk to this particular slave, it outputs the slave's ascii name onto the serial bus. When the keyword name is received by the `Silence()` routine running in the slave, the slave Handheld executes `RS485Transmit()` to send an acknowledgment to the master (which should now be listening to the serial bus to accept the acknowledgment). The master and slave can then exchange data.

The data exchange format may be a line of ascii text. The master and slave could even exchange ascii QED-Forth commands. When the exchange is complete, the slave can again execute the `Silence()` routine to disable its transmitter and begin listening for its name.

Synchronous Serial Peripheral Interface (SPI)

The Serial Peripheral Interface, SPI, is a fast synchronous serial interface. It provides a convenient means of connecting the Handheld to a variety of peripheral devices, including analog to digital and digital to analog converters, real time clocks, and other computers which use high speed communication.

The SPI can transfer data much more rapidly than an asynchronous serial link – its maximum rate is 2 Megabits/second.

After configuring the SPI system to communicate on a properly connected network of devices, sending and receiving data is as simple as writing and reading a register. The QED-Forth kernel includes pre-coded drivers that configure and control the SPI for maximum speed data transfers. This chapter describes those drivers, and presents code that makes it easy to configure the SPI for different data transfer rates and formats.

SPI Bus Pins

Hardware is interfaced to the SPI via three PORTD pins named SCK, MOSI, and MISO brought out to pins 7, 8, and 10 on the Wildcard Port Header (see Appendix B). The SCK (serial clock) pin is a configurable synchronous data clock output. This signal synchronizes the exchange of bytes between the Handheld and its peripherals. The byte-sized messages are transmitted and received via the MOSI (master out/slave in) and MISO (master in/slave out) pins. The /SS (active-low slave select) is typically used to enable data transfers by slave devices when it is active low. For the Handheld, /SS is not used for SPI communication because it is used to control the direction of the RS485 transceiver; you can use any digital I/O line as a /SS signal. A ground connection is also necessary to ensure that the communicating devices have a common voltage reference.

When the Handheld controls the network, it is referred to as a "master"; otherwise, it is a "slave". The distinction between master and slave is an important one. The device that initiates a data transfer is the master, and all other devices on the network are slaves. Only one active master may con-

trol the network at a time; however, the device that assumes the role of master may change according to an appropriate protocol.

If you are using the Handheld as a slave device and require the /SS signal for your external SPI hardware, configure one of the Port A pins on the Field Header as an input pin. By polling the Port A pin or by setting up an interrupt service routine, you can configure the Handheld to ignore the SCK input when /SS is high and keep MISO in a high-impedance state so that it does not interfere with the SPI bus. When the /SS input goes low, the slave (or Handheld in this case) transfers data in response to the SCK clock input that is initiated by the master.

If you are using the Handheld as a master device, each external SPI device will require a separate select line (/SS). You can implement the slave select lines by configuring Port A pins as outputs. Remember that the /SS is active low so to select a device you need to set the pin low; otherwise the pin should idle high.

There are many possible configurations of master/slave networks. Regardless of the network, however, there are only four signals used: SCK provides a synchronized clock, MOSI and MISO signals are used for data transmission and reception, and /SS configures the Handheld as a master or slave device. In this section we will consider the most general and simple configurations.

SPI Network Connections

Configured as a master device, the Handheld transmits bytes via the "master out/slave in" pin, MOSI. It receives bytes sent by a slave device via the "master in/slave out" pin, MISO. Transmissions are always initiated by the master device, and consist of an exchange of bytes. As the master transmits a byte to an active slave (that is, a slave with its /SS input active low), the master receives a byte from the slave. It may be that only the byte sent from the master to the slave is meaningful; nevertheless, each device simultaneously transmits and receives one byte. The only difference between the master and slave devices is that the master initiates the transmission.

Slave devices use the master in/slave out pin, MISO, for transmitting, and the master out/slave in pin, MOSI, for receiving data. The following wiring diagram illustrates how the MOSI, and MISO pins of a master and a slave would be connected to exchange data:

Master		Slave
MOSI	→	MOSI
MISO	←	MISO
SCK	→	SCK
/SS	→	/SS
GROUND	↔	GROUND

The status of a device as master or slave determines how the various pins must be configured. The arrows in the diagram point to pins configured as inputs, and originate from output pins. Thus, the master has only one input, MISO, which is the slave's only output. Note that the master device outputs the clock synchronization signal SCK to the slave's SCK which is configured as an input. Also, in the diagram, the master's /SS (slave select) is configured as an output. By setting this out-

put LOW, the slave's input /SS is pulled LOW. The GROUND line serves as a common voltage reference for the master and slave.

There are a variety of ways the MOSI, MISO, SCK and /SS pins on your Handheld can be connected. The one you choose depends on the specific device, or devices you will be connecting to. In some circumstances a one-way data flow may suffice. For example, a Handheld connected to a serial A/D converter might have these connections:

Master Handheld		Slave Serial A/D Device
MOSI	→	not connected
MISO	←	Conversion Output
SCK	→	CLK
/SS (Port A pin)	→	/CS
GROUND	↔	GROUND

In this example, the Handheld selects the serial A/D by outputting a LOW signal on /SS. Even though the MOSI pin is not connected to anything, the master initiates a transmission using a "dummy" byte. The SCK pin clocks the serial A/D's CLK input which causes the A/D's conversion result to be transferred to the master via the MISO line.

The Handheld allows the details of the synchronous communications protocol to be customized for compatibility with a variety of peripherals. The next section describes the registers that configure and control the Handheld's SPI.

Configuring the SPI

The SPI is configured and accessed via four registers:

Name	Description	Reference
SPCR	SPI control register	MC68HC11F1 Technical Data Manual, p.8-5
SPSR	SPI status register	MC68HC11F1 Technical Data Manual, p.8-7
SPDR	SPI data register	MC68HC11F1 Technical Data Manual, p.8-7
DDRD	Port D data direction	MC68HC11F1 Technical Data Manual,p. 6-4

Given a properly wired network and a properly configured SPCR control register, a master device may transmit a message by simply storing the byte to the SPDR data register. This automatically activates the SCK clock which synchronously transmits the data. As the master transmits its data, 8 bits of data are simultaneously received. The received data byte is accessed by reading SPDR data register. This ability to exchange messages means that the SPI is capable of full duplex communication. Once the data has been exchanged, a flag bit in the SPSR status register is set to indicate that the transfer is complete. If the programmer has enabled the local interrupt mask for the SPI, an interrupt is recognized at this point. Any required SPI output signals must be configured as outputs,

either by calling **InitSPI()** or by setting the appropriate bits in the Port D data direction register DDRD.

The **InitSPI()** function provides a convenient way to initialize the SPI as the master at a 2MHz baud rate. This function properly configures the directions of the SPI I/O pins, and configures the data transfer such that data is valid on the falling trailing edge of the clock, with the clock idling in the low state. This configuration works for many SPI devices, including the optional battery-backed real-time clock. Consult the data sheets for any peripheral devices that you are interfacing to the SPI and, if a different configuration is needed, follow the instructions below to set up the appropriate SPI data transfer protocol.

The **BufferToSPI()** function implements fast data transfer from a specified buffer in the controller's memory to an SPI device. This function cannot accept incoming data; consult its glossary entry for details.

Multitasking applications with more than one SPI device should control access to the SPI bus using the resource variable **SPI_RESOURCE** and the access control functions **GET()** and **RELEASE()**. Consult their glossary entries for details.

Initializing the SPI Control Register

The SPI control register, SPCR, contains 8 bits which must be initialized for proper control of the Handheld's SPI (M68HC11 Reference Manual, Section 8.6.2). These bits are:

Bit Name	Description
SPIE	SPI interrupt enable
SPE	SPI system enable
DWOM	Port D wired-or mode
MSTR	Master
CPOL	Clock polarity
CPHA	Clock phase
SPR1	SPI clock rate select bit1
SPR0	SPI clock rate select bit0

The DWOM bit (port D wired-or mode) should always be set to 0. Setting DWOM to 1 takes away the processor's ability to pull the Port D signals high unless there is a pull-up resistor on each bit of the port. Setting this bit to 1 without installing pull-up resisters corrupts the operation of the serial communications interface which uses bits 0 and 1 of Port D.

Setting SPE (SPI enable) to 1 turns on the SPI system. This bit should be set only after all other SPI configuration is complete.

SPIE is a local interrupt mask that allows an interrupt to be recognized when an SPI data transfer has completed, or if a write collision or mode fault is detected.

Setting the MSTR bit initializes the Handheld as a master, and clearing the MSTR bit initializes it as a slave. If the /SS pin of the master is an input and if a low input level is detected, the processor sets

the MODF bit in the SPI status register a "mode fault" condition. This detects the presence of more than one master on the SPI bus.

The CPOL, CPHA, SR1 and SPR0 configure the SCK pin's clock polarity, clock phase, and clock rate. These signals are described in detail below.

SPI Clock Signal Configuration

The SCK pin's synchronous clock signal has configurable phase, polarity and baud rate so that it can interface to a variety of synchronous serial devices. In general, all devices on a network should use the same phase, polarity, and baud rate clock signal. In some cases, however, a sophisticated network may have device groups on a network that use different clock configurations. Although the devices would share the same network, communications would only be understandable by members of the same group.

The clock's polarity is controlled by a bit named CPOL (clock polarity) and its phase is controlled by CPHA (clock phase). CPOL determines whether the clock idles in the low state (CPOL = 0) or the high state (CPOL = 1). If the clock idles in the low state, the leading edge of the clock is a rising edge. If the clock idles in the high state, the leading edge of the clock is a falling edge. The CPHA bit determines whether data is valid on the leading or trailing edge of the clock. Note that the data is changed by the transmitting device one half clock cycle before it is valid.

The following table summarizes the combinations of CPHA and CPOL settings:

CPOL	CPHA	Data is valid on the:
0	0	Rising, leading edge
0	1	Falling, trailing edge
		= default set by `InitSPI()`
1	0	Falling, leading edge
1	1	Rising, trailing edge

Many serial devices require a clock that idles in the low state (CPOL = 0), and expect valid data to be present on rising clock edges. Thus in this common configuration the transmitting device outputs the data when the clock goes low, and the receiving device samples the valid data when the clock goes high (CPHA = 0). In other words, data is valid on the "rising leading edge" of the clock.

To interface devices that support synchronized serial interfaces, but are not configurable like the Handheld, determine the device's requirements for clock phase and polarity and configure the Handheld's CPHA and CPOL accordingly. It is important to note that when the CPHA bit is 0, the /SS line must be de-asserted and re-asserted between each successive data byte exchange (68HC11 Reference Manual, Section 8.3.2). If the CPHA bit is 1, the /SS line may be tied low between successive transfers.

SPI Baud Rate

The two lowest order bits in the SPCR control register, named SPR1 and SPR0, determine the data exchange frequency expressed in bits per second; this frequency is also known as the baud rate.

This setting is only relevant for the master device, as it is the master's clock which drives the transfer. SPR1 and SPR0 determine the baud rate according to the following table:

SPR1	SPR0	SPI Frequency (bits/s)
0	0	2.0 MHz = default set by `InitSPI()`
0	1	1.0 MHz
1	0	250 kHz
1	1	125 kHz

Summary of the SPCR Control Register

The SPIE bit in the SPCR (SPI control register) enables SPI interrupt handling. The SPE bit turns on the SPI system. The DWOM bit determines whether Port D needs pull-up resistors; it should be set to 0. The MSTR bit determines whether the device is a master or slave. The CPOL and CPHA bits configure the synchronous clock polarity and phase and specify when valid data is present on the MISO and MOSI data lines. Finally, for master devices, the SPR1 and SPR0 bits determine the baud rate at which data is exchanged.

SPI Status Register Flags

There are three flag bits implemented in the SPSR (SPI status register). They are:

Flag	Condition
SPIF	SPI transfer complete
WCOL	Write collision
MODF	Mode fault

Any of these conditions may generate an interrupt if the SPIE (SPI interrupt enable) bit in the SPCR control register is set.

The SPIF is set when a data transfer is complete, and is cleared by a read of the SPSR status register, followed by a read or write to the SPDR data register. Thus, resetting the SPIF flag is very simple. After a data transfer is initiated by writing to the SPDR data register, the processor may poll the SPSR status register until the SPIF flag is set. Then reading the data that was received (by reading the SPDR) or initiating a new data transfer (by writing to the SPDR) automatically clears the SPIF flag. Alternatively, the if the SPI interrupts are enabled, the SPI interrupt handler determines what caused the interrupt by reading the SPSR register to see which of the three status bits is set. If SPIF is set, reading the received data or initiating a new data transfer automatically clears the SPIF bit.

A write collision occurs when a byte is written to the SPI data register, SPDR, while data is being exchanged. The WCOL flag is set when a write collision occurs. The data transfer that is in process when the write collision occurs is completed. WCOL is cleared by a read to the SPSR followed by a read or write to the SPDR.

A mode fault occurs when the SPI senses that a multimaster conflict (MC68HC11F1 Technical Data Manual, p.8-7) exists on the network as explained above in connection with the /SS input. When a mode fault is detected, the processor:

1. disables the SPI outputs by clearing the bits in the Port D data direction register (DDRD),
2. clears the MSTR bit in the SPCR to configure the SPI as a slave,
3. clears the SPE bit to disable the SPI, and
4. generates an interrupt if the SPIE bit in the SPCR is enabled

These steps greatly reduce the chance that the communicating devices might be damaged by contention on the SPI bus. The MODF bit is cleared by a read of the SPSR followed by a write to the SPCR.

SPI Data Transfers

A data transfer is initiated by a master device when it stores a message byte into its SPDR register. If a slave device has already stored a byte into its SPDR register, that byte will be exchanged with the master's byte. Once the bytes have been exchanged, the master may write a new byte to initiate another byte exchange. Although data byte transfers are easily executed once the network has been wired and configured properly, a carefully executed software protocol may be required to ensure data integrity.

Summary

The flexibility and power of the 68HC11's serial peripheral interface supports high speed communication between the 68HC11 and other synchronous serial devices. The interface can be used to support analog to digital and digital to analog converters, networks of many computers controlled by a single master, or networks of devices controlled by several coordinated masters. Pre-coded device drivers configure the SPI for a standard data format, and it is easy to customize a data format and baud rate for your application. With careful design, many peripherals can communicate via the SPI, and powerful multi-processor systems can be linked using this high speed bus.

Chapter 12

The Battery-Backed Real-Time Clock

A battery-backed real time clock (RTC) may be optionally included. If so, the RTC's battery also backs-up the 128K RAM. Note however that if the expanded memory option is chosen, providing 512K or RAM, then none of the RAM can be battery-backed. The accuracy of the clock is better than +/- 2 minutes per month.

A 7 ma-hr Lithium rechargable battery is used. This backup battery charges automatically while +5V is applied to the board. When +5V is not applied the battery keeps alive the RTC and the RAM so that information is not lost. The length of time the battery lasts varies from board to board primarily because RAM chips vary device-to-device in their leakage currents. Their leakage currents also depend strongly on ambient temperature so that while a battery may last a long time at normal or low ambient temperatures, at greater temperature it may discharge quickly.

We have measured battery discharge times for typical devices and have found the following:

- Actual measurements at normal temperatures (<40°C) show the battery should backup the memory more than two years between recharging. Our actual measurements of the RAM current draw show it to draw much less than its "typical" spec from its datasheet.
- Assuming the "typical" current specifications of the RAM data sheets the battery should last 139 days; and,
- At the worst case current specification for the RAM, battery life would be 44 days at normal temperatures, and as little as 6 days if operated continuously at an ambient temperature of 70°C.
- If the RAM is not battery backed, but the RTC installed, typical RTC retention time at 25°C should be 300 days.

Figure 12-1 illustrates the dependence of battery lifetime between charges while backing up the 128K RAM and powering the RTC, and ambient temperature.

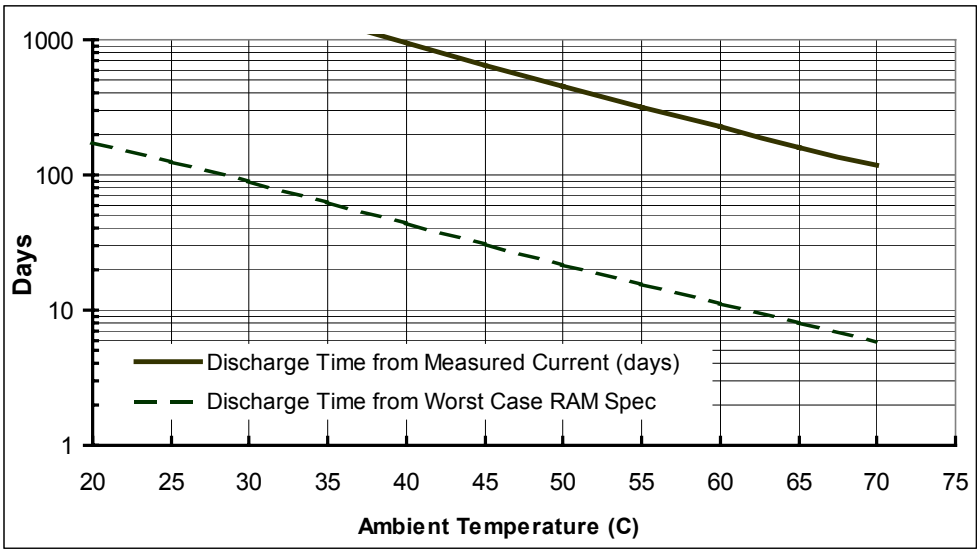


Figure 12-1 Battery discharge time as a function of ambient temperature.

After each cold restart the kernel re-initializes the RTC chip to charge the battery and to use battery backup on removal of power. The maximum battery recharge time is 22 hours.

Setting and Reading the Real Time Clock

The built-in library functions **`SetWatch()`** and **`ReadWatch()`** make it easy to set and read the real time clock. The **`SetWatch()`** and **`ReadWatch()`** functions use the top 16 bytes of the 68HC11F1's on-chip RAM as a buffer to hold intermediate results during the data transfers, and the top 8 bytes at addresses 0xB3F8-0xB3FF serve as the **`watch_results`** structure that contains the results returned by the most recent call to **`ReadWatch()`** as described below.

Because the structure that is written to by **`ReadWatch()`** is at a fixed location, this code is not re-entrant. This is not a problem in single-task applications or applications where only one task uses the watch. But it can cause problems if multiple tasks are executing **`ReadWatch()`**. For example, assume that task #1 calls **`ReadWatch()`**, but before it can access the contents of the structure using the assignment statement, the timeslice interrupt occurs and task#2 proceeds to call **`ReadWatch()`**, then the contents of the structure could be changed before task#1 is able to execute its assignment statement. To avoid this situation, the multitasking application can be configured so that only one task calls **`ReadWatch()`** and **`SetWatch()`** and shares the data with other tasks.

Another option is to define a resource variable to mediate access to the watch. The routines needed to accomplish this are declared in the **`MTASKER.H`** file and described in detail in the Control-C Glossary. The following brief example illustrates how to design a re-entrant function that returns the current **`WATCH_MINUTE`**:

```
RESOURCE watch_resource;      // declare resource variable: controls
                              // watch access
  _Q int CurrentMinute(void) // reads watch, returns current minute
   {  int minute;
      GET(watch_resource);    // get access to watch; Pause() if
                              // another task has it
      ReadWatch();            // updates contents of watch structure
```

```
    minute = WATCH_MINUTE;
                          // you can also transfer other contents
                          // from the watch structure to
                          // "task-private" variables here
    RELEASE(watch_resource); // release access to watch so other
                          // tasks can use it
    return minute;
}
```

The **CurrentMinute()** function can be simultaneously called from multiple tasks without causing any conflicts. The **GET()** and **RELEASE()** macros automatically mediate access to the watch, ensuring that only one task has access at a time.

The battery-backed clock is pre-set at the factory to Pacific Time in the United States. To re-set the smart watch to your time zone, your program can call the function:

```
void SetWatch(hundredth_seconds,seconds,minute,hour,day,date,month,year)
```

The interactively callable routine named **SetTheWatch()** is defined in the **TIMEKEEP.C** file so you can set the watch from your terminal. As explained in the Control-C Glossary, the hour parameter ranges from 0 to 23, the day from 1 to 7 (Monday=1 in this example), and the year parameter ranges from 00 to 99. For example, if it is now 10 seconds past 5:24 PM on Wednesday, May 26, 2004, you could interactively set the watch by typing:

```
SetTheWatch( 0, 10, 24, 17, 3, 26, 5, 4)←
```

The watch is set and read using 24-hour time, where midnight is hour 0, noon is hour 12, and 11 PM is hour 23. Note that you may assign any day of the week as "day number 1".

The function **ReadWatch()** writes the current time and date information into a structure named **watch_results** that occupies the top 8 bytes of on-chip RAM at addresses 0xB3F8-0xB3FF. Pre-coded macros name the structure elements so it is easy to access the time and date information. For example, the following simple function in the **TIMEKEEP.C** file reads the smart watch and prints the time and date:

```
_Q void SayDate(void)
{   ReadWatch();     // results are placed in watch_results structure
printf("\nIt is now %d:%d:%d on %d/%d/%d.\n",
WATCH_HOUR, WATCH_MINUTE, WATCH_SECONDS,
WATCH_MONTH, WATCH_DATE, WATCH_YEAR);
}
```

It simply calls **ReadWatch()**, and then executes a **printf()** statement using the pre-coded structure macros to reference the time parameters in the **watch_results** structure. If you called this function immediately after setting the watch as described in the prior section, the response at your terminal might be:

```
It is now 17:24:31 on 5/26/04.
```

After compiling **TIMEKEEP.C**, you can interactively type at your terminal:

```
SayDate( )←
```

at any time to see a display of the current time and date.

For backwards compatibility with the QED product line, the hundredths of a second field is present but it is not used. The hundredths of a second value is required for SetWatch() but it is ignored and it is always 0 from ReadWatch().

Chapter 13

Customizing Your Handheld Instrument

Your Handheld comes with 17 I/O signals including timer controlled digital I/O, 8 lines of 8-bit A/D, and two serial ports. You can further customize its I/O by:

- ⇨ *Adding WildCards for virtually any kind of I/O;*
- ⇨ *Routing any I/O signals to the Handheld's DB-25 connector;*
- ⇨ *Filtering and/or protecting any of the I/O; and,*
- ⇨ *Adding custom circuitry.*

Using the Personality Board to Customize I/O

The Handheld's Personality Board allows you to place your own custom circuitry into the Handheld, to mount WildCards for custom-off-the-shelf I/O, to route any of the Handheld's signals to its DB-25 connector, or to filter, protect, or condition any of its I/O signals.

Figure 13-1 shows the Personality Board. On the left is prototyping area you can use to mount custom sensors, actuators or to include your custom circuitry. On the right are field connections for up to four WildCards. Each of the field connectors for the WildCards is mirrored with a sea of holes to provide access to any of the field signals. Corresponding hole patterns for the DB-25 connector and the Processor Board's native I/O are provided so that you may point-to-point wire from any I/O signal to the DB-25 connector.

The Personality Board includes protection circuitry for you to use to protect up to eight signals. This protection circuitry comprises SIDACtor overvoltage protection devices mated with current-limiting resistors. The schematic for the protection devices is found on the "Protection" page of the *Handheld Schematic*.

Figure 13-2 and Figure 13-3 show the internal and external dimensions of the Handheld. There is considerable volume between the Personality Board and the Handheld's back shell. This volume is roughly in the shape of two adjacent cubes, one of approximately 17.7 in^3 (4.0" x 3.4" x 1.3") available for mounting Wildcards or your custom hardware, and another of 5 in^3 (appx. 1.5" x 2.8" x 1.2") over the prototyping area. Even if you fill the Handheld with four Wildcards, there is still the sizeable prototyping area available for your custom devices.

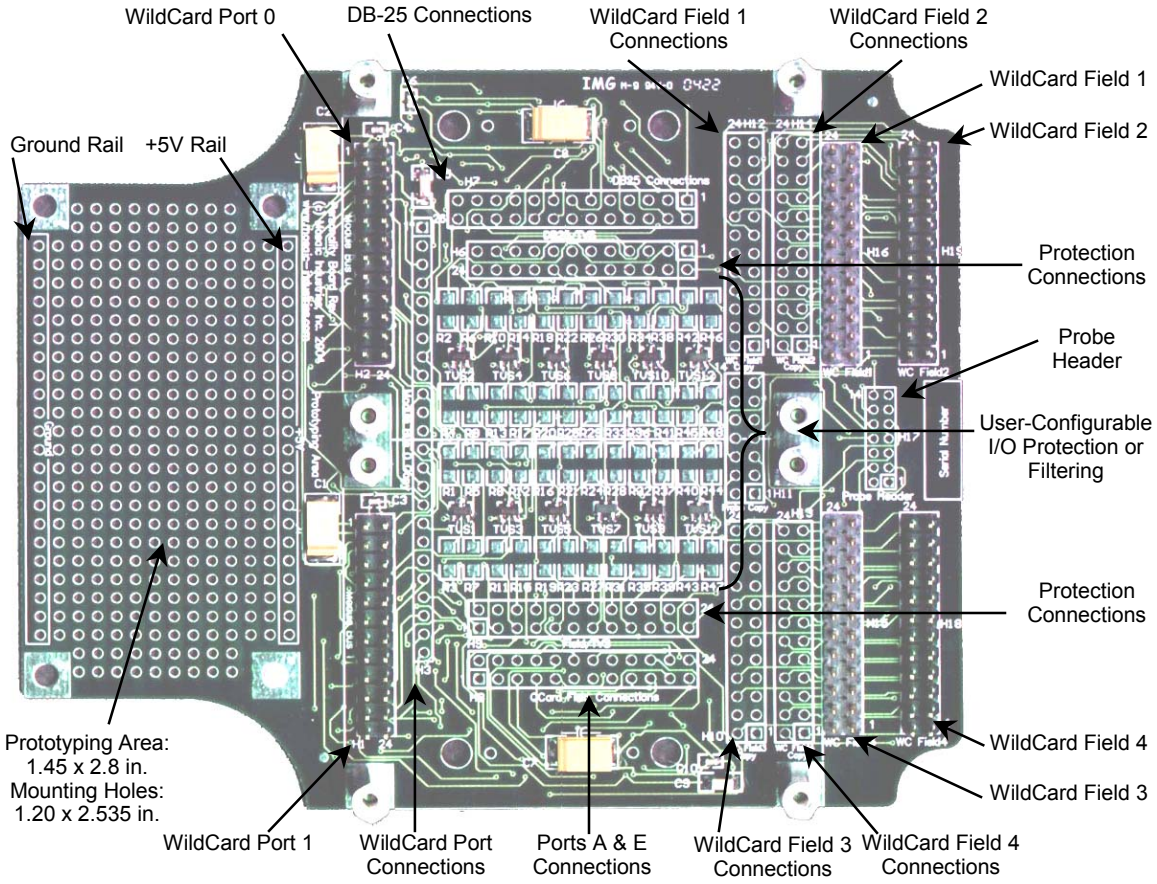


Figure 13-1 The Personality Board showing room for custom circuitry on the left, and protection circuitry in the center.

Routing I/O Into the Instrument

Figure 13-1 shows that the Personality Board includes a hole pattern corresponding to the DB-25 signals, labled “DB-25 Connections”. All the DB-25 lines are routed for you from the connector to these holes. You may point-to-point wire on the Personality Board from any of the WildCard field headers, the Port A and E Connections, or your custom circuitry to these holes.

Mounting Additional Components

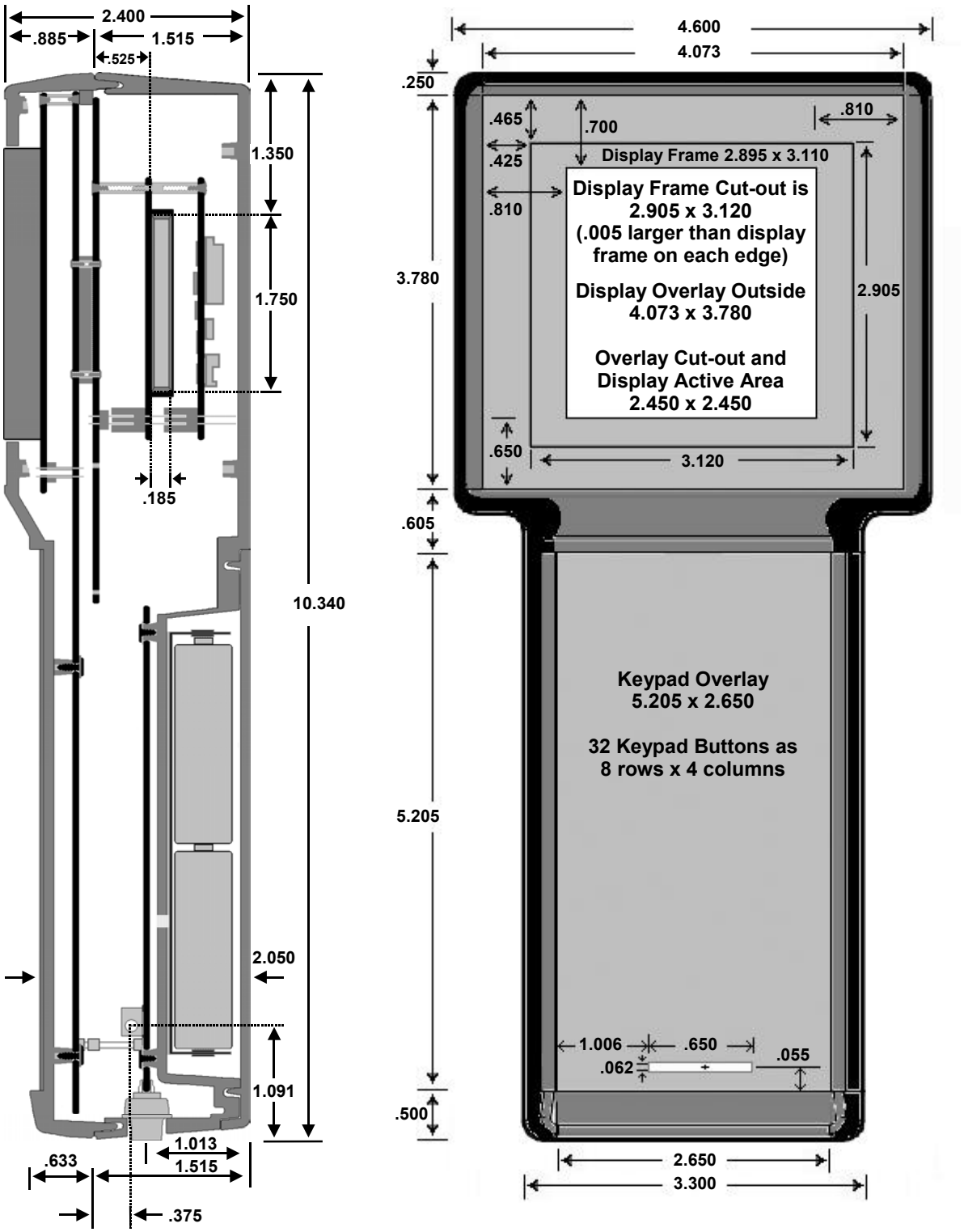
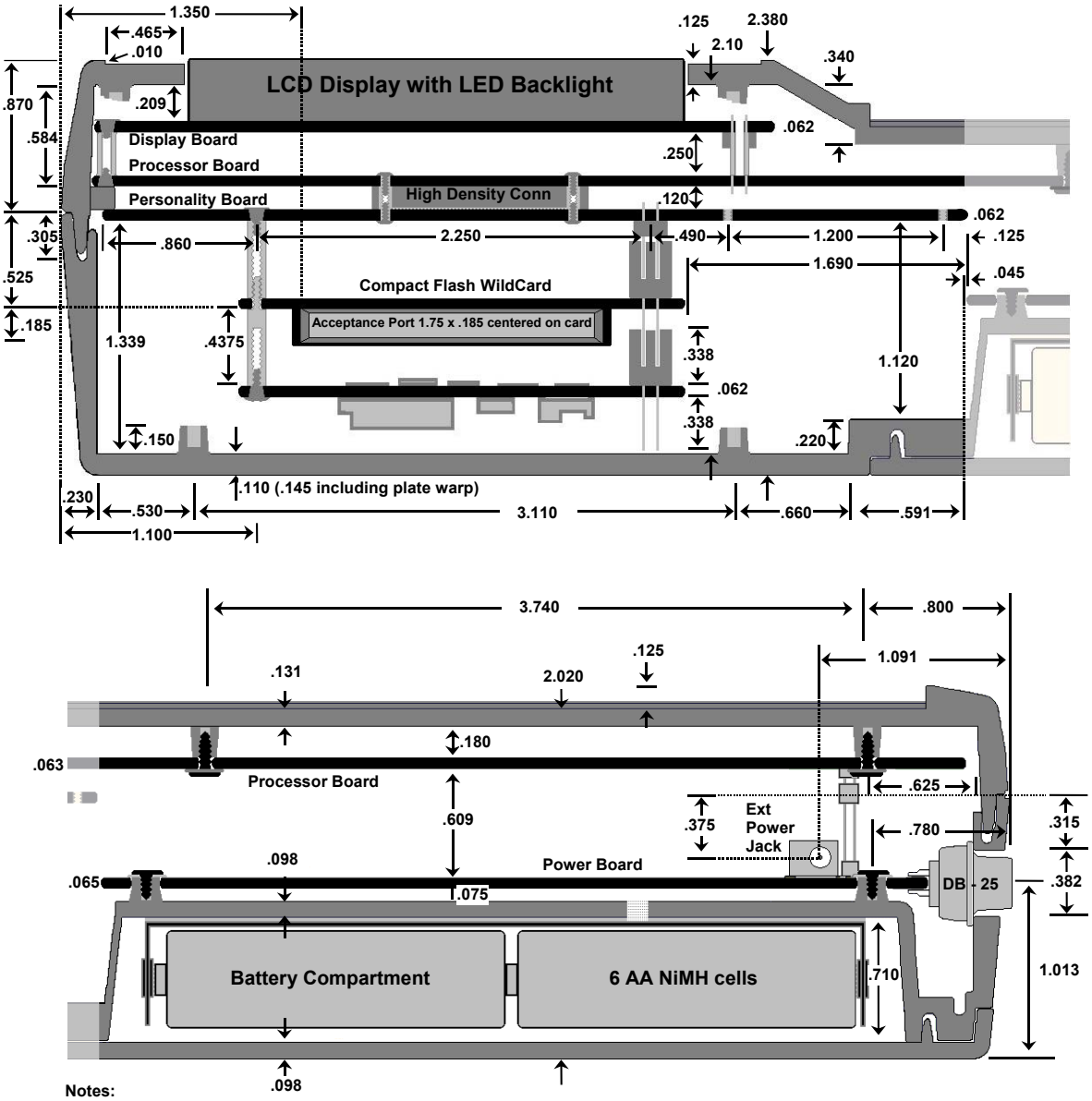


Figure 13-2 The Handheld's external dimensions (in inches).



Notes:

All dimensions ± .005, distances that span the top and bottom shell are ± .010.

Display protrudes appx. .030 from front shell surface.

Processor/Personality board spacing is .125 at the spacer, .115 at the high density connector, and appx. .120 at the WildCard connectors.

Figure 13-3 The Handheld's internal dimensions (in inches).