The QED-Flash Board

Hardware and Software Update Documentation

Version 1.1

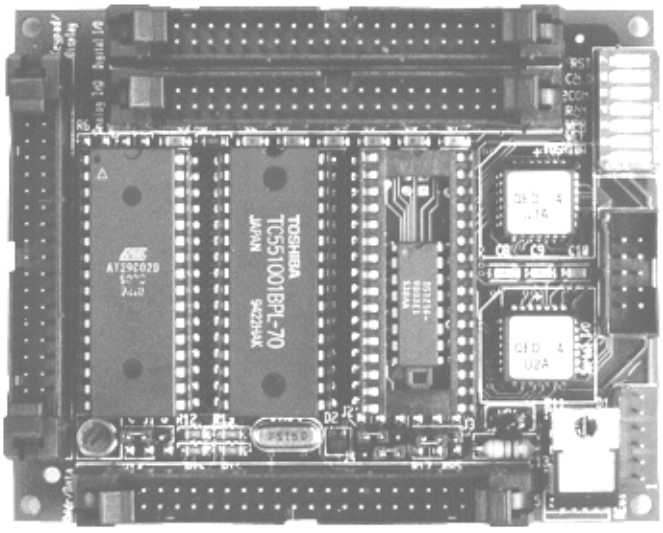


Table of Contents

Introduction

The QED-Flash Board is an enhanced version of the QED Board, a compact I/O-rich controller. This document describes the new hardware and software features of the QED-Flash Board, and updates the documentation provided in the QED Hardware and Software Manuals. Control C users should note that the C software documentation is found in “Getting Started with the QED Board Using the Control C Programming Language” book.

After describing the hardware features of the QED-Flash Board, this document details the QED-Flash memory map, illustrates how to compile a program into flash, summarizes the new software features, and itemizes the changes to the QED Kernel. A glossary of new software routines is included at the end of this document, followed by Appendices that present the QED-Flash Board connector pinouts and schematics.

Hardware Enhancements

The hardware enhancements and features found on the QED-Flash Board include:

- Flash memory for code and data storage
- Keypad and display changes
- New location for the real time calendar clock
- New DIP switch and jumper configuration
- Onboard high current MOSFET drivers
- Support for unipolar (0 to 5 V) and bipolar (-5 to +5 V) 12 bit A/D conversions
- Support for up to 256 Kilobytes static RAM in the center socket
- Power fail monitor

Each of these is described in detail below. Note that the latter four items are present on the QED-3 Board as well as the QED-Flash Board; the discussion of these feaures is presented here as a cogent update to the QED Hardware Manual.

Flash Memory

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 QED-Flash Board'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.

The "developer version" of the QED-Flash Board (shipped with the Industrial Control System, Product Design Kit, and Developer Package) is configured with 256 Kbytes of nonvolatile re-programmable flash memory in socket S1, 128 Kbytes of battery-backed RAM in socket S2, and a battery-backed real-time clock in socket S3. The SmarTouch Controller Starter Kit comes with 256K flash in S1 and 128K of battery-backed RAM in S2. At least 128K in S2 is required for compiling or field-upgrading flash-resident code. Standard production versions of the QED-Flash Board are configured with 256K flash in S1 and 32K RAM in S2. Production versions can be enhanced with additional memory, battery-backup, and real-time clock options.

The onboard 256K flash memory device in socket S1 reserves 128K for user programs and data. It also contains a 128 Kbyte QED Kernel comprising a complete multitasking operating system, debugger, interactive Forth compiler, assembler, and hundreds of pre-coded device driver functions. Easy-to-use functions are available to move code into flash, and to transfer data into flash from anywhere in the QED Board's memory space. Programming is easily accomplished from any PC using ANSI C, Forth, or assembly language.

Keypad and Display Changes

The keypad and display interface has changed significantly as the product evolved from the QED-2 to the QED-3 and finally to the QED-Flash Board. The QED-3 Board expanded upon the capabilities of the QED-2 Board by adding a faster 8-bit memory mapped display interface, a direct read/write interface to Hitachi 128x240 pixel graphics displays that can show up to 16 lines of text with 40 characters per line, and support for a 4x5 resistive touchscreen as well as the standard 4x5 keypad. The QED-Flash Board adds the ability to directly connect a Toshiba 128x240 pixel graphics display .

Chapter 10 of the QED-Hardware Manual discusses the keypad and display interface of the QED-2 Board which did not have a full 8-bit display data bus or MOSFET high-current drivers; consequently, the connector diagram in Figure 10.1 of the Hardware manual is outdated. The connector pinouts and schematics in Appendix A and B of the QED-Hardware Manual are correct for the QED-3 Board. Note that odd-numbered pins 19 through 33 on the keypad/display header implement four high-current MOSFET drivers and their associated grounds and snub signals. The display bus is a fast 8-bit memory-mapped read/write interface that supports character or graphics displays. The keypad and display connector of the QED-Flash Board, shown on page A-2 of this document, has further evolved to support Toshiba graphics displays by changing the signals associated with pins 30, 32, and 34 to DGND, +5VDD, and VEE, respectively.

Direct Connection to Toshiba and Hitachi Graphics Displays

All models of the QED Board provide a direct connection to character displays up to 4 lines by 20 characters in size. The QED-Flash Board also provides a direct interface for graphics displays up to 128x240 pixels that use the popular Hitachi HD61830 and Toshiba T6963 driver chips. Simply flipping a switch (DIP switch #1) on the QED-Flash Board selects the Hitachi or Toshiba display type, and the correct display control signals are automatically generated. A jumper (J1, next to the potentiometer) can be set so that the onboard potentiometer provides proper contrast adjustment for graphics displays (the contrast adjustment range spans -15V to +5V for graphics displays, and 0 to +5V for character displays). Thus the QED-Flash Board does not need an adapter board to interface to non-backlit character or graphics displays. However, a Graphics Interface Board (GIB) is still required for touchscreen users, or for those who want to take advantage of the power supply on the GIB to drive a backlight.

More Versatile Keypad/Touchscreen Interface

The keypad interface now supports a resistive 4 row by 5 column touchscreen in addition to the standard 4x5 keypad. The touchscreen can overlay a graphics display, enabling you to implement a modern touch-sensitive menu-driven user interface. While a standard contact-type keypad exhibits close to 0 Ohms resistance when a key is pressed, resistive touchscreens exhibit a "connection" of up to 30 kOhms when the screen is touched. The QED-Flash keypad scanning circuitry accommodates these larger resistances, and capacitors have been added to increase the noise immunity of the circuitry. The keypad pin assignments are identical to those of the QED-2 and QED-3 Boards.

New Location for the Real-Time Calendar Clock

The standard location for the optional battery-backed Real-Time Clock (RTC, also known as the "smartwatch") is in socket S2. The RTC is itself a socket made by Dallas Semiconductor that contains a clock chip, crystal and battery. The smartwatch RTC socket also battery-backs a 32K RAM chip that is plugged into it. QED-3 and QED-Flash Boards with the "non-flash" memory map accommodate the RTC in the center S2 socket (for a detailed description of the QED-Flash memory map, see "The QED-Flash Memory Map" section below). The only drawback to this scheme is that Dallas does not make the components necessary for a 128 Kbyte memory integrated with a battery-backed RTC at the required 100 nanosecond or faster access time.

This limitation is removed on flash-carrying QED-Flash Boards. A board that has flash memory in socket S1 and has DIP switches 3 and 4 ON (indicating the flash memory map) accommodates a smartwatch RTC in the S3 socket. No memory is installed in S3; up to 256K of flash and RAM can be installed in S1 and S2, respectively. The built-in functions that set and read the watch automatically access the watch at position S3 on flash-carrying QED-Flash Boards.

DIP Switch Settings

The QED-Flash Board has a 7-position DIP switch (compared to a 6-position switch on prior boards). The switches are located on the memory side of the board next to the communications connector. The following table summarizes the purpose of each switch:

#	Legend	Effect of Switch in the ON State
1	TOSHIBA	Configures the display interface for direct connection to a Toshiba graphics display. (The OFF state is used for character displays and Hitachi graphics displays, or with any graphics display via the Graphics Interface Board).
2	WEF	Write Enables the entire Flash memory device in the S1 socket.
3	WPR	Write Protects the entire RAM (pages 4, 5, 6, 7) in the S3 socket.
4	ROM	Configures the S3 (RAM/ROM) socket to accommodate a PROM. (The OFF state configures S3 to hold a RAM). NOTE: If both switches 3 and 4 are ON, flash is assumed to be in S1 and the flash memory map is used.
5	2COM	Enables serial port 2 communications hardware for use as a second RS232 port. (The OFF state configures bit 3 of PORTA for use as general-purpose I/O.)
6	COLD	Puts the QED Board into "special cleanup mode" upon the next power-up or reset.
7	RST	Resets the QED Board; return switch to OFF position to resume normal operation.

Discussion of DIP Switch Settings

Caution is in order regarding Switch 1: connecting a character display, non-Toshiba graphics display, or Graphics Interface Board (this includes the SmarTouch Controller) while DIP switch 1 is ON may prevent proper operation of the QED Board. If this situation occurs, power down the QED Board and remove the display and/or correct the switch position.

Switch 2 is useful for customers who are using the flash memory for code storage, and who do not need to modify the contents of the flash memory during normal operation. The write-protect switch "bullet-proofs" the code against modification. It also helps avoid "flash lockout". The flash device is

set up to "lock out" all memory accesses for 10 milliseconds (msec) after an "errant" write to the flash (that is, a write cycle that was not preceded by the required "unlocking sequence" that grants access to write to the flash). Thus, if an application program inadvertently executes a write cycle while the flash is being addressed, the QED Board will "crash" for at least 10 msec because the program code cannot be read by the processor. This frequently results in "silent crashes" that do not return the "QED-Forth V4.x" prompt. Setting Switch 2 OFF avoids this situation. Of course, all application programs should be thoroughly debugged so that there are no "inadvertent writes" to the flash memory.

In general, switch 4 selects RAM versus PROM in socket S3, and switch 3 indicates whether the device in S3 should be write-protected. The following table itemizes the effects of the 4 possible settings of these 2 switches:

SW#4	SW#3	Result
OFF	OFF	RAM is in S3, RAM not write protected; uses non-flash memory map.
OFF	ON	RAM in S3, pages 4 - 7 of RAM write protected; uses non-flash memory map.
ON	OFF	PROM in S3; uses non-flash memory map.
ON	ON	Flash in S1; uses flash memory map. No memory allowed in S3. (Real-time clock may be placed in socket S3.)

If both switches 3 and 4 are ON, the QED Board assumes that a 128K or 256K flash memory is in socket S1, and the "flash" memory map is used. You may notice that we have taken advantage of the fact that a PROM does not need to be write-protected; we use this unneeded switch combination to select the flash memory map.

The 2COM, COLD and RST switches perform the same functions as the like-named switches on the QED-3 Board as explained in the "Getting Started with the QED Board" booklet.

Jumper Settings

The QED-Flash Board has four 3-post jumpers, labeled J1 through J4. As shown in the schematics at the end of this document, each jumper configuration can be optionally implemented using surface-mounted zero-ohm resistors to support volume OEMs who do not wish to allow moveable jumpers on the board. The purpose of each jumper is explained in turn.

J1 is located beneath the flash socket S1 as shown in figure A.1. If a jumper shunt is placed across the pin closest to the Keypad/Display Connector and the center pin (i.e., shorting pins 1 and 2), the contrast potentiometer (pot) is configured for a character display. If a jumper shunt is placed across the center pin and the pin closest to the crystal (i.e., shorting pins 2 and 3), the contrast pot is configured for a graphics display.

The contrast adjustment on the SmarTouch Controller is controlled with a potentiometer on the Graphics Interface Board. Thus, SmarTouch Controller or users need not worry about the J1 jumper; the J1 jumper shunt may be in either position, or omitted entirely.

Jumpers J2-J4 are located beneath socket S3 as shown in figure A.1. The schematic on page B-3 depicts jumpers J2 and J4 with pin 1 on the right. Note that the physical location of pin 1 on the QED-Flash Board is on the left when looking at the board in the orientation shown in Figure A.1.

Jumper J2 is located above jumper J4 and to the left of J3. If the jumper shunt is installed across the pin closest to the crystal and the center pin (i.e., shorting pins 1 and 2), 32K or 128K RAM can be installed in S2. If the jumper shunt is installed across pins 2 and 3, socket S2 can accommodate 128K or 256K RAM.

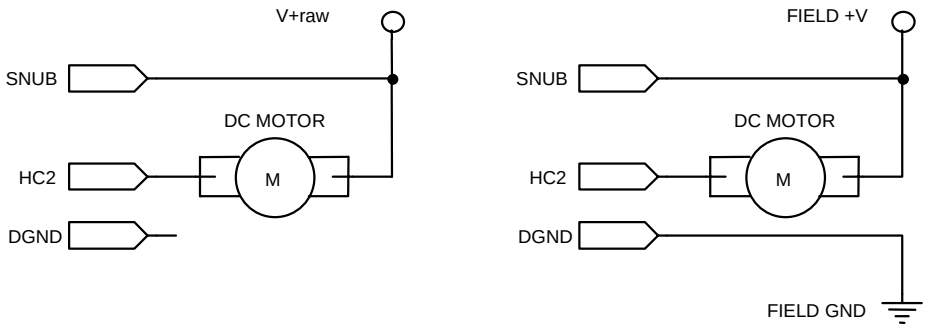
Jumper J4 is located between jumper J2 and the Address/Data Connector. If the jumper shunt is installed across the pin closest to the crystal and the center pin (i.e., shorting pins 1 and 2), 64K or 128K ROM can be installed in S1. If the jumper shunt is installed across pins 2 and 3, socket S1 can accommodate 128K or 256K flash.

Jumper J3 is located to the right of J2. It determines whether the RS-232 or RS-485 protocol is used for the primary serial port. If the jumper shunt is installed across the pin closest to J2 and the center pin (i.e., shorting pins 1 and 2), the primary serial port is configured for full-duplex RS-232 communications. If the jumper is installed across pins 2 and 3, the primary serial port is configured for half-duplex RS-485 multi-drop communications.

Onboard High Current Drivers

Four MOSFET high current drivers are available at the keypad/display connector on the QED-Flash Board (and the QED-3 Board), replacing the "No Connect" pins that were present on the QED-2 board. Each driver can sink up to 150 mA continuously, or up to 1 amp on an intermittent basis at voltages as high as 60 V. Onboard snubber diodes suppress inductive kickback, so the drivers can be used to control solenoids, motors, and other inductive loads.

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



Connecting a load using onboard power. Connecting a load using external power.

Figure 1. How to Connect a DC Load to the High Current Drivers.

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

Users of the SmarTouch Controller should note that the Graphics Interface Board (GIB) uses high current driver channels 0 and 1. HC0 controls the piezoelectric buzzer, and HC1 controls a relay that turns the backlight on.

Unipolar and Bipolar 12 Bit A/D Conversions

Each QED-Flash Board comes standard with an 8 channel 12-bit A/D (analog-to-digital) converter and an 8 channel 8-bit D/A (digital-to-analog) converter. Of course, the 68HC11's 8 channel 8-bit A/D is also supported. On the QED-3 and QED-Flash Boards, the 12 bit A/D converter supports bipolar conversions and faster conversion times (because the chip select is generated using a PAL instead of port bit PPB7 on the PIA).

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

The software drivers for the 12 bit A/D support the unipolar and bipolar conversion options. For Forth users, the software routines are discussed in “Changes in A/D Conversion” in the Software Enhancements section of this document. For Control C users, A/D Conversion is covered in Chapter 8 of “Getting Started with the QED Board Using Control C Programming Language”

Up to 256K Static RAM in S2

The center memory socket (S2) on the QED-Flash Board can accommodate a 32K, 128K, or 256Kbyte static RAM. A three-post jumper located at J2 (at the bottom of the S3 socket, nearest to the crystal and above J4 as shown in Figure A.1) configures the QED-Flash Board for either 32K/128K RAM or 256K RAM in S2. To configure the board for a 32K SRAM or 128K SRAM, connect the center jumper pin to the leftmost jumper pin (the pin nearest to the crystal). To configure the board for a 128K or 256K SRAM, connect the center jumper pin to the right jumper pin.

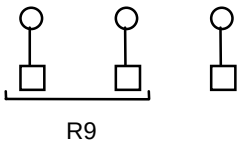


Figure 2. S2 memory configuration jumper.

Power Fail Monitor

The QED-Flash Board has a power fail monitoring circuit shown in Figure 3 that can be used to warn the application program of an impending power failure. By installing an appropriate resistor, you can select the supply voltage at which the power fail signal is asserted. The power fail warning can be configured to generate an nonmaskable interrupt, /XIRQ. This allows the application program to perform any required "cleanup" tasks (such as saving crucial status information in battery-backed RAM) before power is lost.

The following 3 steps configure the power fail feature of the QED Board:

1. Connect PFI Input (pin 21 on the Digital I/O Connector) through an external resistor to the external power source that you wish to monitor. Resistor value selection is discussed below. Volume OEM users can install the resistor as a surface-mount device at position R30 on the QED-Flash Board.
2. Install a zero-ohm resister at location R7, located just above the flash socket (S1) on the top of the QED-Flash Board, to connect the power fail output to the processor's XIRQ input.
3. Enable the nonmaskable interrupt after every reset as explained in Chapter 8 of the QED Hardware Manual.

When the PFI Input (pin 21 on the Digital I/O Connector) falls to less than 1.25 V, the Power Fail Output (/PFO) of the MAX708 chip in Figure 3 goes active low, triggering the /XIRQ interrupt if it is enabled and if R7 is installed. The value of the user-supplied resistor between the PFI input and the monitored voltage supply named V.supply is calculated using a simple voltage divider equation as,

$$R = \left(V.supply \; / \; 1.25 \right) \; - \; 1 \qquad (k\Omega)$$

where R is expressed in units of kΩ. For example, to configure a power fail warning when V.supply goes below 6V, choose R30 = 3.8kΩ. Typically, V.supply is the V+raw input to the QED Board, but any power supply voltage can be monitored using this circuit.

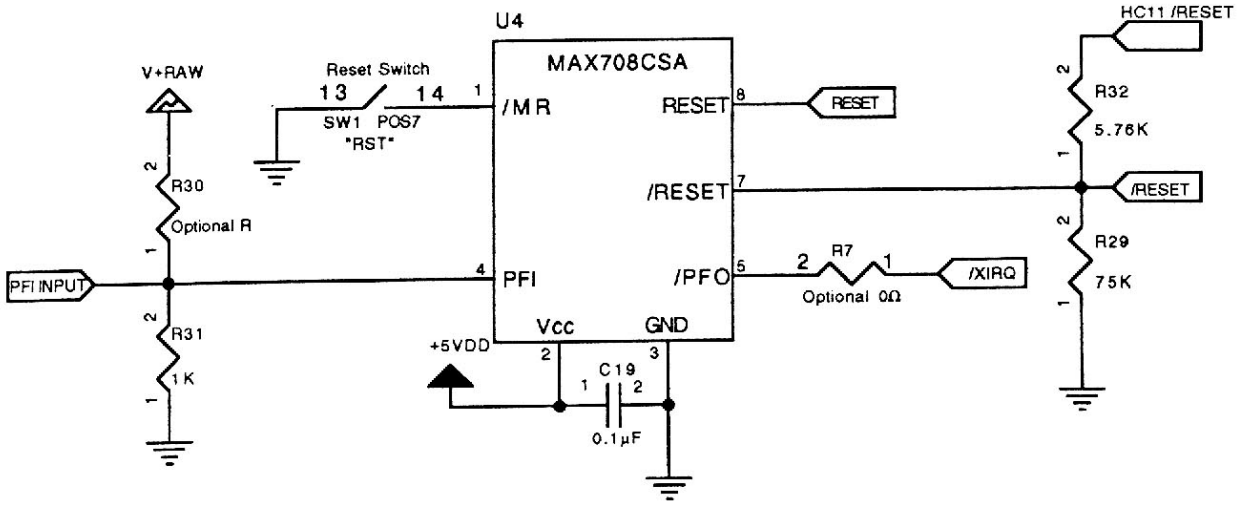


Figure 3. The MAX708CSA Power Fail Input and Power Fail Output circuitry. Other circuitry not related to the power-fail detection is not shown here; see the Reset Circuitry page of the QED-Flash Schematics in Appendix B to view the complete reset control and power monitor circuit.

1.5 Volt Reference

The QED Hardware Manual contains warnings in Chapter 6 regarding the stability of the 1.5 volt reference. These warnings are outdated. The 1.5 volt reference may be used to provide a stable reference voltage for all of the digital-to-analog converters on the QED Board, and can also provide a reference voltage for external circuitry.

The QED-Flash Memory Map

Common Memory

The "common memory" at locations 8000 - FFFF hex is the same for flash-carrying and non-flash versions of the Board. A short summary is presented here; all addresses are hexadecimal.

The processor's registers are located at 8000 - 805F, onboard hardware occupies addresses through 80FF, and the operating system reserves memory through location 8DFF for user areas, buffers, and stacks. For example, the default user area that runs the interactive Forth interpreter occupies 8400 - 84FF.

The 8 Kbytes at locations 8E00 - ADFF are available RAM for the user. The C compiler uses this area for static variables, arrays, task areas, etc. Similarly, the Forth memory map routine `USE.PAGE` locates the variable area starting at address 8E00 in common memory. (This is different from the QED-2 Board which located the variable area on page fifteen; Forth programmers should consult the glossary entry for `USE.PAGE`.)

The HC11's on-chip EEPROM (Electrically Erasable Programmable Read Only Memory) is located at AE00 - AFFF. Locations AE00 - AEBF are reserved by the operating system for use by the `SAVE` and `RESTORE` utilities, and for interrupt vectors. EEPROM at AEC0 - AFFF is available to the user.

The HC11's 1 Kbyte of on-chip RAM is located at B000 - B3FF. Locations B3F0 - B3FF are reserved for the real-time clock buffers. Locations B3D0-B3DF are reserved for support of Forth interrupt service routines called from C-compiled programs. On flash-carrying boards, locations B200 - B3CF are reserved for the flash programming routines. Locations B000 - B1FF are always available to the programmer (this area is named `ONCHIP_RAM` in the C linker command file; C programmers can locate data in this area using a `#pragma` directive).

Locations B400 - BFFF and C100 - FFFF contain kernel code. The "notch" at C000 - C0FF is not decoded by any onboard devices, and provides a convenient place for the user to memory map I/O that must be accessed quickly (that is, without requiring a page change). Of course, an almost limitless amount of I/O can be mapped onto pages in the QED Board's 8 Megabyte address space.

Paged Memory

As described above, DIP switches 3 (WPR) and 4 (ROM) select the "flash" or "non-flash" memory map. This selection affects the allocation of paged memory among the 3 sockets on the QED Board. Pages 0 - F are decoded on the QED Board, and pages 10 - FF are available to the user.

Note that the optional Memory Interface Board (which supports a PCMCIA card plus flash and RAM) is mapped on pages 10 - 3F (on-board flash, RAM, and bank control logic) and pages 80 - BF (PCMCIA linear flash or RAM cards up to 32 Megabytes in size, or ATA flash cards of unlimited size). Up to 4 Digital I/O Boards can be interfaced to the QED Board; the are mapped at pages DC through DF. The PAL on the prototyping board and ICS backplane decodes pages E0 through FF for customer use during prototyping.

The flash memory map allows up to 256 Kbytes of flash memory in socket S1, and up to 256 Kbytes RAM in socket S2, with no memory allowed in socket S3. As explained below, a "page-swapping" approach is used to allow code to be downloaded (compiled) into RAM, transferred to flash, and then executed from flash.

The non-flash memory map accomodates up to 128 Kbytes PROM in S1, up to 256 Kbytes RAM in S2, and up to 128 Kbytes RAM or PROM in socket S3.

Non-Flash Memory Map

Figure 4 illustrates the non-flash memory map. As the caption explains, the top 32 Kbytes at hex addresses 8000 - FFFF of the processor's native 64 Kbyte memory map is "common memory" which is always accessible without a page change. Data stacks and variables are typically kept in "common RAM". Frequently accessed kernel library routines are located in the "common ROM". The lower 32 Kbytes are addressed as one of 256 pages, yielding an 8 Megabyte address space. Page 0 and the lowest 12 Kbytes on page fifteen contain kernel library routine and function names, respectively. Pages 12 and 13 are reserved for kernel use. All other pages are available to accommodate memory (for programs or data) or memory-mapped I/O. Up to 512 Kbytes of memory can be installed on the QED Board itself.

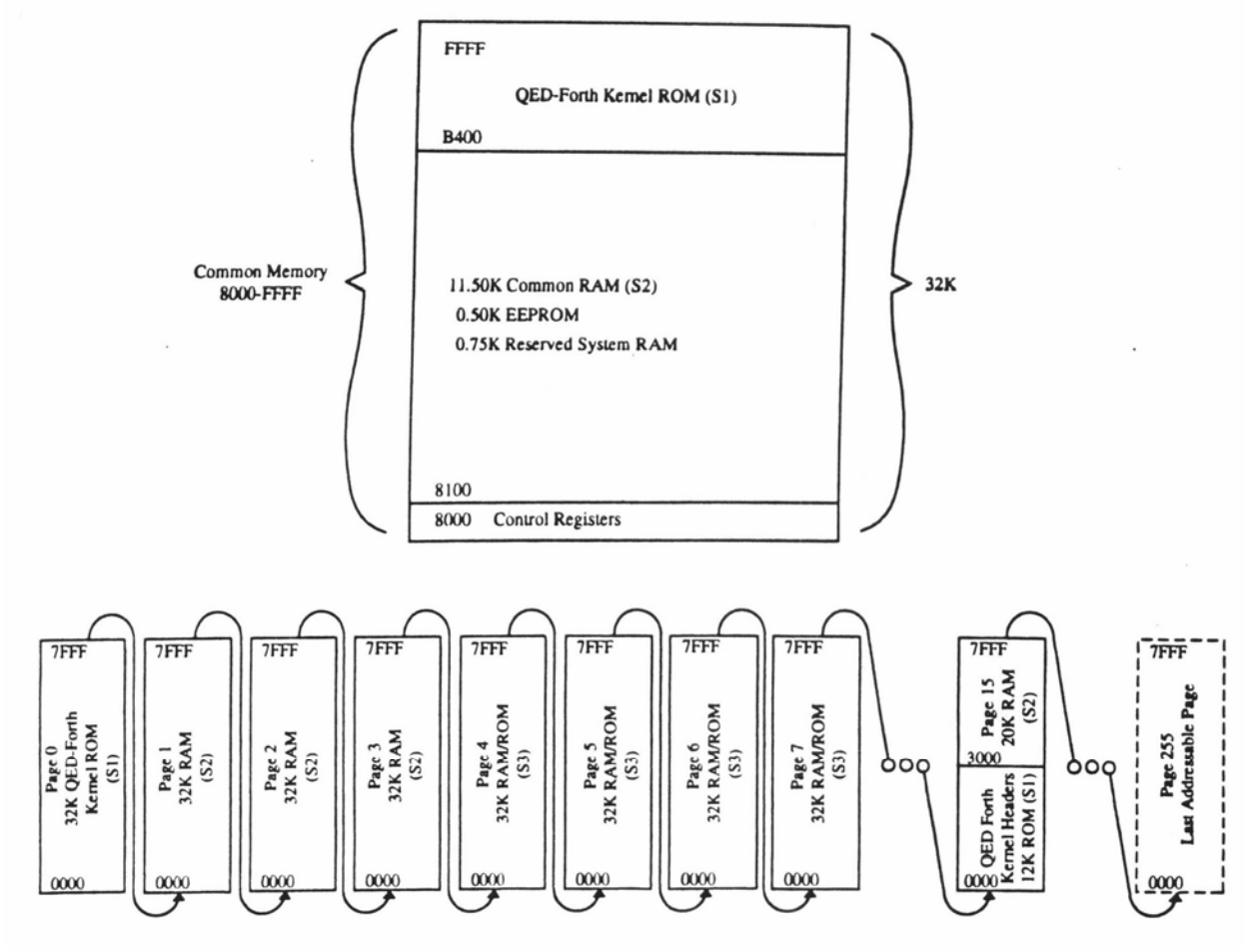


Figure 4. Diagram of the QED Board non-flash memory map**.** The 32K of "common memory" at addresses 0x8000 to 0xFFFF (the upper half of the processor's memory space) is rapidly accessible without a page change**.** Up to 256 pages (32K per page) occupy the "paged memory" at addresses 0000 to 0x7FFF**.** If 128K memories are installed in sockets S2 and S3, the common memory as well as pages 0-7 and 15 are present on the QED Board**.**

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.

Using this non-flash memory map, users who do not need flash memory can use the QED-FLASH board with RAM and/or PROM memory, and all programs and data will be handled properly with no code changes. Existing source code must be recompiled for the QED-FLASH Board. The table in Figure 5 presents another way to look at this non-flash memory map; for simplicity, only the user-modifiable non-common memory is listed.

S1 PROM	**S2 RAM**	**S3 RAM/PROM**
Kernel (non-modifiable)	Pages 1 – 3	Pages 4 – 7
	Pages 8 – 11	
	Page 15 at addresses 3000 - 7FFF hex	

Figure 5. User-modifiable pages in the non-flash memory map assuming a 256K RAM in S2 and a 128K RAM or PROM in S3. Pages are listed in decimal. If a 32K RAM is installed in S2, only page 15 (3000-7FFF) and common RAM are available in the S2 device. If a 128K RAM is installed in S2, pages 1, 2, 3, 15 and common RAM are available in the S2 device.

Flash Memory Map

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 and compilation of Forth source code or C-compiler S records from the PC to the QED Board. The two maps are very similar; the effect of changing from the Standard to the Download map is to "swap" the locations of pages 4, 5 and 6 between the flash and the RAM. The reason for this will become clear after we review the page assignments and the compilation procedure.

The table in Figure 6 summarizes the user-modifiable pages in the flash memory map.

"Standard Map"		**"Download Map"**	
S1 Flash	**S2 RAM**	**S1 Flash**	**S2 RAM**
Pages 4 – 6	**Pages 1 – 3**	**Pages 1 – 3**	**Page 4 – 6**
Page 7	Page 8	Page 7	Page 8
Page 13 (reserved for kernel)	Page 9	Page 13 (reserved for kernel)	Page 9
	Pages 10 – 11		Pages 10 – 11
	Page 15 at 3000 - 7FFF hex		Page 15 at 3000 - 7FFF hex

Figure 6. User-modifiable pages in the flash memory map, assuming the standard 256K flash in S1 and the optional 256K RAM in S2. If a 32K RAM is installed in S2, only page 15 (3000-7FFF) and common RAM are available in S2. If a 128K RAM is installed in S2, pages 1, 2, 3, 15 and common RAM are available in the standard map in the S2 device. Pages that can be swapped are shown in bold. Pages are listed as decimal numbers.

Flash Programming Functions Reserve Half of On-chip RAM

As mentioned above, the functions that program the flash memory use the 68HC11's on-chip RAM starting at hex B200. The remaining on-chip RAM at B000 to B1FF remains available to the user.

Software Development Using Flash Memory

Compilation Procedure for C Programmers

If You Have a New Version of the C Compiler

If your C compiler was shipped after the third quarter of 1998, compiling a program proceeds exactly as with the QED-3 Board as described in "Getting Started with the QED Board Using the Control C Programming Language". Automated commands contained in the download file will determine whether flash is present on the QED-FLASH Board, and will automatically establish the download map, load the code into RAM, transfer the code to flash, and re-establish the standard map. If you are curious, you may want to review the following instructions for users of older compilers to see what is going on "under the hood".

If You Have a Previous Version of the C Compiler

If your C compiler was shipped before the third quarter of 1998, contact Mosaic Industries to obtain update files that automate compilation for the QED-Flash Board. If your compiled code is larger than 32 Kbytes, the compiler update files are required for proper operation.

Alternatively, if your application fits on one page, you can interactively type a few commands from the TERMINAL program as described in the following steps so that even a non-updated compiler will work with the QED-Flash Board. It is assumed that your board has the standard complement of memory during program development, comprising 256K flash in S1, and 128K RAM in S2, and that your application fits in 32 Kbytes so that you can use the "hammer" icon to compile your code.

1. Compile your program in the normal manner using the C compiler. Use the default "hammer" icon which places code only on page 4 and results in the fastest execution time.

2. Before sending the .txt download file to the QED Board, enter the terminal window and interactively type the command
   ```
   DOWNLOAD.MAP
   ```
   to establish the download memory map as shown in the last 2 columns of Figure 6.

3. Send your .txt download file (created during compilation) to the QED Board.

4. After successfully downloading your code, transfer the code and names to flash using the `PAGE.TO.FLASH` routine. All of your code and names are located on page 4 and the debug functions are on page 5; simply type at the terminal
   ```
   4    PAGE.TO.FLASH
   5    PAGE.TO.FLASH
   ```
   which transfers pages 4 and 5 in RAM to the parallel pages 1 and 2 in flash.

   The `PAGE.TO.FLASH` routine will print an error message if you pass it an illegal page, or if the flash cannot be programmed (for example, if the flash is write protected because DIP switch 2 is OFF). The page parameter passed to the routine must be the source page in RAM where the code has been compiled. `PAGE.TO.FLASH` transfers all 32 Kbytes of the source page to the "parallel" page in flash memory. At this point, two copies of your code and

names exist in memory: one copy is in RAM, and the other resides in flash. But only the RAM version can be executed properly, because it resides at the same pages at which it was compiled. The next step will make the flash-resident copy of the code executable.

5. Re-establish the standard memory map by typing at your terminal
   ```
   STANDARD.MAP
   ```
   This remaps pages 4, 5, and 6 so that they are addressed in the flash device, and maps pages 1, 2, and 3 to be addressable in the RAM. At this point, the flash-resident code on page 4 (which is now in flash) can be executed. The RAM in pages 1, 2, and 3 are available; they can be written over or used by the application program.

6. Run the program by 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 QED Board to simplify debugging). You can also execute an autostart command to cause a specified function to be automatically called upon each restart. To place the autostart vector in EEPROM inside the processor, execute the command:
   ```
   CFA.FOR  MAIN   AUTOSTART
   ```
   To place the autostart vector in flash in socket S1, execute the command:
   ```
   CFA.FOR  MAIN   PRIORITY.AUTOSTART
   ```
   For most applications, the `PRIORITY.AUTOSTART` option is preferable because it locates the autostart vector with the code in the flash memory device.

C EXAMPLE SESSION

Here's how easy it is to compile a C program into flash:

First compile a source code file. For example, to run a "Hello World" program, your source code file might be named HIWORLD.C with the following contents:

```
#include <\mosaic\allqed.h>  // this include statement should
// appear at the top of each source // code file.

_Q void HelloWorld( void )
{   printf("\nHello Everyone!\n");
}

void main( void )
{ HelloWorld();
}
```

Use the standard "hammer" icon to compile the file onto a single page (page 4). Those with C compilers shipped in late 1998 can just send the HIWORLD.TXT file to the QED Board using the Terminal program.

We recommend using the QEDTERM.EXE terminal program that is available on the Mosaic Industries web site (www.mosaic-industries.com) or on the Demo and Driver Disk that accompanies each new QED-Flash Board; it runs under Windows 95, 98, and NT. Customers running Windows 3.x can use the Microsoft Windows TERMINAL.EXE program.

For those with older versions of the C compiler, type the following commands from the terminal to the QED Board:

```
DOWNLOAD.MAP
```

<use the terminal's Send Text File feature to send HIWORLD.TXT to the QED Board>

```
4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
STANDARD.MAP
MAIN        // executes the program!
```

You've compiled your program, downloaded it into RAM, transferred it to flash, and executed it. If you want to set up an autostart routine, follow the standard procedure as explained above, or consult the final chapter of the "Getting Started with the QED Board Using the Control C Programming Language" manual.

Compilation Procedure for Forth Programmers

This section lists the steps required to compile and execute a Forth program on a flash-carrying QED-FLASH Board. It is assumed that your board has the standard complement of memory during program development, comprising 256K flash in S1, and 128K RAM in S2.

1. Establish the download memory map as shown in the last 2 columns of Figure 6 by executing:
   ```
   DOWNLOAD.MAP
   ```
   This command can be typed interactively from the terminal program, or can be the first statement in your download file.

2. Establish memory locations for code and names on pages 4, 5, and 6. One valid way to do this is by executing
   ```
   4 USE.PAGE
   ```
   which is equivalent to the following set of commands:
   ```
   HEX
   0000 4 DP X!       \ code starts at address 0 on page 4;
                      \ 20 Kbytes available
   5000 4 NP X!       \ names start at address 5000 on page 4;
                      \ 12 Kbytes available
   8E00 0 VP X!       \ variables start at address 8E00 in
                      \ common RAM
   4A00 0F 7FFF 0F IS.HEAP \ 13.5 Kbyte heap at addresses 4A00-
                           \ 7FFF on page F
   ```
   Typically, the `USE.PAGE` command or other equivalent commands are the first commands in the source code file. To customize the locations of these memory areas, you can include edited versions of these commands in your source code file. Make sure these commands come BEFORE the first `ANEW` statement in your code. For example, to locate up to 32 Kbytes of code on page 4, and up to 32 Kbytes of names on page 5, you can execute:
   ```
   HEX
   4 USE.PAGE         \ set default use.page locations
   0 5 NP X!          \ move names section to page 5,
                      \ leaving all of page 4 for code.
   ```
   or, equivalently,
   ```
   HEX
   0000 4 DP X!       \ code starts at address 0 on page 4;
                      \ 32 Kbytes available
   0000 5 NP X!       \ names start at address 0000 on page 5;
                      \ 32 Kbytes available
   8E00 0 VP X!       \ variables start at address 8E00 in
                      \ common RAM
   ```

```
4A00 0F 7FFF 0F IS.HEAP \ 13.5 Kbyte heap at addresses 4A00-
                        \ 7FFF on page F
```

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

```
ANEW  MY.CODE     \ choose any name you want here
: HELLO.WORLD  ( -- )  CR ." Hi Everyone!" ;
                  \  < all of your source code goes here >
```

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

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

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

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

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

5. Re-establish the standard memory map by executing

```
STANDARD.MAP
```

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

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

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

where <name> is the name of the designated function. To place the autostart vector in flash in socket S1, execute the command:

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

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

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

```
NO.AUTOSTART      \ if you installed autostart, undo it
4 PAGE.TO.RAM     \ move code from page 4 in flash to page 1
                  \ in RAM
5 PAGE.TO.RAM     \ (only needed if page 5 used);
                  \ move page 5 flash to page 2 RAM
DOWNLOAD.MAP      \ get ready for download
<send source code file here>
```

```
4 PAGE.TO.FLASH   \ copy updated page 4 code to flash
5 PAGE.TO.FLASH   \ copy updated page 5 code to flash
STANDARD.MAP      \ code is now executable at pages 4 and 5
                  \ in flash
CFA.FOR <name> PRIORITY.AUTOSTART   \ optional
```

FORTH EXAMPLE SESSION

Here's how easy it is to compile a Forth program into flash:

```
DOWNLOAD.MAP
4 USE.PAGE
ANEW  MY.CODE
: HELLO.WORLD  ( -- )  CR ." Hi Everyone!" ;
4 PAGE.TO.FLASH
STANDARD.MAP
HELLO.WORLD       \ executes the program!
```

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

Software Enhancements

The software enhancements and features found on the QED-Flash Board include:

- Changes in A/D conversion software
- Timing considerations for the software UART
- Accessing the battery-backed real time watch
- Stepper motor control utilities

Each of these is described in detail below. This section also presents an update describing how C programmers can now call kernel functions from within interrupt service routines.

Note that all of these software features are present on the QED-3 Board as well as the QED-Flash Board, so those who are familiar with the QED-3 Board may skip to the next section. The discussion of these software feaures is presented here as a cogent update to the QED Software Manual. All glossary entries for the relevant software routines are correct and updated in the Forth and C Glossaries.

Changes in A/D Conversion

This subsection is targeted to Forth programmers. The A/D drivers on the QED-Flash Board are unchanged from those on the QED-3 Board, and all glossary entries for these routines are correct in the Forth and C Glossaries. If you are using Control C, A/D Conversion is covered in Chapter 8 of “Getting Started with the QED Board Using the Control C Programming Language”.

The Forth stack pictures of the routines that control multiple conversions by the 8-bit and 12-bit A/D converters have been changed. The new stack pictures include an unsigned parameter "u1" that controls the interval between successive samples. The revised routines and their stack pictures are as follows:

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

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

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

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

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

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

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

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

has been given a new meaning that is backwardly compatible with the previous software version while adding support for the conversion of bipolar analog signals. In the prior V2.01 software, the "flag" was true (= -1) for single-ended A/D conversions, and false (=0) for differential conversions. Moreover, all conversions were "unipolar", meaning that the allowed input signal range did not include negative voltages.

In QED kernel version 4.xx, the software supports both unipolar and bipolar conversions, and the "flag" in the stack pictures of the routines above has the following meaning:

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

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

Timing Considerations for the Software UART

The QED Board has two asynchronous serial ports. The primary serial port (serial1) is supported by the 68HC11's on-chip hardware UART, and does not require interrupts to work properly. Serial1 is the default serial port used by the QED-Forth operating system, and its support code has not been changed.

The QED Board's 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 kernel PROM on the QED Board contains a complete set of high level driver routines for the serial2 port, and these functions are summarized in the QED-Forth and Control C glossaries.

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 QED Board 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, it is highly likely that 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 and QED-Forth Glossaries each contain a list of functions that temporarily disable interrupts, and the glossary entries give further information regarding how long interrupts are disabled. In most cases the times are less than 25 µs which does not pose a problem. However, note that the `READ.WATCH` and `SET.WATCH` functions (`ReadWatch()` and `SetWatch()` in C) disable interrupts for about a millisecond (ms), and the functions that write to EEPROM disable interrupts for 20 ms per programmed byte. Be sure to account for these effects when designing your application.

We have built sophisticated instruments based on the QED Board 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.

Accessing the Battery-backed Real Time Watch

QED-Forth

In QED kernel version 4.xx, interrupts are now disabled during the entire operation of the functions `READ.WATCH` and `SET.WATCH`, which access the battery-backed real time clock/calendar. This change makes these two routines fully re-entrant in QED-Forth multitasking applications.

With a 16 MHz crystal on the QED Board, READ.WATCH now disables interrupts for approximately 0.8 milliseconds (ms), and SET.WATCH disables interrupts for approximately 1.0 ms. These times are about double the prior values. All of the input and output parameters of SET.WATCH and READ.WATCH are passed on the Forth data stack. These routines are now fully re-entrant, which means that they can be simultaneously called from multiple tasks in a single QED-Forth application without producing any conflicts.

Control C

The Control C versions of these functions are named ReadWatch() and SetWatch(). In C, these two routines access a fixed pre-allocated structure in the 68HC11's on-chip RAM using structure macros defined in the WATCH.H header file. For example, the following code reads the watch and stores the current minute (after the hour) in the variable named current_minute:

```
static int  current_minute;
ReadWatch();
current_minute = WATCH_MINUTE;
```

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 WATCH_MINUTE 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.

Summary of Multitasking Considerations for Watch Access Routines

In summary, while the QED-Forth versions of READ.WATCH and SET.WATCH are now fully re-entrant in version 4.xx, additional care must still be taken for those who wish to call the watch routines from multiple tasks that are coded in the Control C programming language.

Stepper Motor Control Utilities

Three functions have been added to the kernel to support stepper motor control. These QED-Forth functions are named CREATE.RAMP, SPEED.TO.DUTY, and STEP.MANAGER. The Control C versions of the functions are named CreateRamp(), SpeedToDuty(), and StepManager(). A high level source code file (available in both Forth and C on the Demo and Driver Disk that accompanies each QED-Flash starter system) and an accompanying applications note provide and document a variety of motor control functions to manage ramping, speed and step control. The C version of the high level source code file named STEPPERS.C is also present in the \FABIUS\QEDCODE directory of the C compiler.

Overview of the Stepper Control Code

The stepper code is explained in more detail in the high level source code listing and its application note. These documents are available from Mosaic Industries. A brief overview is presented here.

Conceptually, the stepper motor control code comprises the following elements:

1. Two arrays of structures that are accessed by both the low-level support functions and the high level code. One array holds a status structure for each stepper motor, and the other array defines a speed ramp for each stepper motor.
2. Three support functions in the QED kernel version 4.xx PROM (presented in the Glossary section at the end of this document). The Control C versions of the functions are declared in the header file STEPPER.H in the \FABIUS\INCLUDE\MOSAIC directory of the C compiler.
3. A high level source code file (available in Forth and C) that defines a set of useful motion control functions.

Stepper Motor Data Structures

A 1-dimensional array of status structures is called STATUS.ARRAY in Forth, and status_array in C. For each motor, the array specifies the extended address and bit mask of the port that controls the motor, the state of the motor (disabled, stopped, in a ramp, at a final speed, etc.), a signed 32 bit step counter, a direction flag, the bit masks that specify the step patterns, the default steady speed, jog/start speed, acceleration and deceleration, additional parameters used by the utility routines, and a pointer to the RAMP.ARRAY.

The array that specifies the number of steps to be taken at each speed for each motor is called the RAMP.ARRAY (ramp_array in C). Each row corresponds to one stepper motor, and each entry in a row contains a two element structure that specifies a number of steps and a corresponding speed (represented as a duty cycle as explained below).

Utility Functions

Two of the utility functions are used to write to the RAMP.ARRAY. SPEED.TO.DUTY (named SpeedToDuty() in C) converts a specified speed in steps per second to a "duty cycle" measure used internally by the motor control code. The CREATE.RAMP function (named CreateRamp() in C) initializes the RAMP.ARRAY. It expects as input parameters the starting speed, ending speed, linear acceleration, ticks per second of the 68HC11's time base, a starting ramp address, and the number of

entries in the ramp. It uses a linear ramp algorithm to initialize the appropriate elements in RAMP.ARRAY, and returns the total number of steps in the resulting ramp.

The third utility function is named STEP.MANAGER (StepManager() in C). This function should be called from a periodic interrupt service routine; the default time base is once per millisecond. STEP.MANAGER expects the base address of the STATUS.ARRAY in the Y register. For each enabled motor, it writes the appropriate pattern at the appropriate duty cycle to the motor port to attain the speed specified in the motor's RAMP.ARRAY. This assembly coded routine executes in approximately 120 µs per enabled stepper motor. Thus running four stepper motors at a maximum speed of 1000 full- or half-steps per second requires approximately half of the 68HC11's available time (480 µs interrupt service time every 1000 µs).

High Level Stepper Motor Control Code

The high level control code declares and allocates the STATUS.ARRAY and RAMP.ARRAY, sets up the clock interrupt service routine using output compare 3 (this can be changed by the user), defines routines that initialize the STATUS.ARRAY, and defines a versatile set of functions to facilitate motion control. Sample function names are 1STEP, JOG.STEPS, STEPS.AT.SPEED, CHANGE.SPEED, SOFT.STOP, and ESTOP (the corresponding names in Control C are similar). The well commented high level source code file and the accompanying application note describe these functions in detail.

Glossary Entries for Stepper Motor Utilities

The glossary entries for the three new kernel functions are presented in the Glossary of Additional Functions at the end of this document.

Calling Kernel Functions from within Interrupt Routines in C

The chapter titled "Timekeeping and Interrupts" in the "Getting Started with the QED Board Using the Control C Programming Language" book contains a warning against 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 \FABIUS\QEDCODE directory of the C compiler.

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.

Caution: _forth Functions Can Not Be Nested in C

Many functions at are callable from C are actually of the _forth type. This includes functions that are in the kernel on the QED Board, or are part of software distributions such as the Graphical User Interface Toolkit or the ATA Flash Card Software Package. 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 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.

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 often fails when the device goes into production because the variables and arrays are not properly initialized when power is cycled. Only 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 perform initializations within functions wherever possible. This approach will avoid hard-to-diagnose field failures that result from corrupted data in the 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.

Kernel Changes

Additions to the Kernel to Support Flash Memory and ATA Flash Cards

Six new functions have been added to facilitate access to the flash memory on the QED-Flash Board. The C function names (shown without parentheses) are:

```
DownloadMap     PageToFlash     PageToRam
StandardMap     ToFlash         WhichMap
```

Access to these library routines are present starting with C compilers shipped in the last half of 1998. The corresponding Forth function names are:

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

Four new keywords (function and variable names) have been added to Kernel V4.05 to facilitate access to "ATA flash" PC Cards, also known as PCMCIA flash cards. These credit-card-sized devices provide removable flash memory in sizes from 1 Megabyte to hundreds of Megabytes per card. The QED Memory Interface Board (MIB) interfaces to the QED-Flash Board's address/data header, and provides a read/write PC Card (PCC) socket, plus 128K RAM with optional battery backup, and 256K on-board flash. The keywords present in the kernel to support the Memory Interface Board are:

```
Link_File_IO    pcc_bank    PCC_Offset_To_Xaddr    PCC_Present
```

In Kernel versions 4.05 and later, the names of these functions for C and Forth are identical. The glossary entries for all of these functions are presented in the next section.

A separate software distribution called the ATA Flash Card Software Package available from Mosaic Industries enables C and Forth programs to manage a comprehensive file system using the MIB. The software implements a DOS/Windows compatible FAT file system, plus a set of file I/O operations that allows file creation, reading, writing, copying, renaming, deleting, and more. A powerful file processing feature allows you to automatically process (interpret and execute) specified files when the QED Board starts up. This facilitates automated software upgrades or data exchanges. Call us for details.

Note that Kernel V4.01 contains some Forth keywords that were used to support the now-obsolete "linear flash" style PCMCIA cards. These names are PCC.BANK, PCC.C@, PCC.@, PCC.2@, PCC.CMOVE, PCC.OFFSET>XADDR, and PCC.PRESENT?. Those interested in accessing PCMCIA cards via the Memory Interface Board should contact Mosaic Industries to obtain an upgrade to kernel version 4.05.

New C Macros

Three new macros are now available in the \fabius\include\mosaic\types.h file to simplify the manipulation of xaddresses (32-bit extended addresses) and their constituent 16-bit addresses and pages. The new C macros are::

```
TO_XADDR      XADDR_TO_ADDR       XADDR_TO_PAGE
```

The glossary entries for these macros are presented in the next section.

New C #include Files to Set and Clear the Priority Autostart

Two new include-able files are now available in the \fabius\include\mosaic directory to simplify the setting or clearing of the PRIORITY.AUTOSTART vector. Simply including the file named

SET_AUTO.H

in the application's C source code file causes MAIN to be automatically executed each time the QED Board is powered up or restarted. This is equivalent to typing

CFA.FOR MAIN PRIORITY.AUTOSTART

from the terminal after the *.txt file has been sent to the QED Board, as explained in the documentation (for example, in the Turnkey Application Program chapter of the Getting Started With C manual).

Similarly, including the file named

CLR_AUTO.H

in the application's C source code file erases the priority autostart vector at the top of page 4. As before, the priority autostart may also be erased by typing

NO.AUTOSTART

from the terminal.

```
The names of these files are:
          \fabius\include\mosaic\set_auto.h
          \fabius\include\mosaic\clr_auto.h
```

These files are included in the latest versions of the GUI (Graphical User Interface) Toolkit and the ATA Flash Card Software Package. Contact Mosaic Industries for more information.

Implementation Changes

Changes in Priority Autostart

To the programmer, the functionality of priority autostart is the same as before. It allows a specified function to be automatically executed each time the QED Board powers up or resets. In the QED-FLASH Board, the C functions `PriorityAutostart()` and `NoAutostart()` and the corresponding Forth functions `PRIORITY.AUTOSTART` and `NO.AUTOSTART` are now "flash-smart". These routines check whether the DIP switches 3 and 4 are both asserted (which indicates that flash is in socket S1). If so, they write or erase the autostart pattern at the top of page 4 in flash. Note that an autostart routine should not include a call to `PriorityAutostart()` or `PRIORITY.AUTOSTART`. This would cause the autostart pattern to be written to flash upon each startup or reset, which might eventually exceed the 10,000 write/erase cycle limit of the flash chip.

DAC Access Is Slightly Slower

The previous Forth function (`>DAC`) and the equivalent C function `FastSetDAC()` executed in approximately 40 microseconds, and the full multitasking versions `SetDAC()` and `>DAC` executed in about 125 microseconds. The QED-FLASH Board maps the chip enable of the onboard D/A converter to page 14, which adds about 6 extra microseconds to the execution time of DAC accesses compared to the previous mapping in common memory.

Average Speed of Page Changes in C Is Much Faster

This modification affects only C programs compiled with the multi-page "bricks icon" required for C applications over 32 Kbytes in size. C programs compiled with the standard "hammer" icon reside in a single page, and so never have to use the page change routine. These programs run at maximum speed.

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.

The QED-Flash kernel includes a new version of the C page change routine that takes advantage of this situation, performing a page change only if it is needed. In prior kernels, all function calls compiled using the multi-page "bricks" icon required 43 microseconds per call. In the new kernel, 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 significantly improved.

Bug Fixes

The Forth functions

```
F2*   F2/   FSCALE
```

have been modified to test whether the input is 0.0 before performing the specified operation. This prevents a valid zero from being converted to an invalid floating point number. These routines are not used by C programmers.

Glossary of Additional Functions

Each glossary entry lists the C name with its prototype declaration, and the equivalent Forth name with its stack picture. The entries are alphabetized using the C names.

int **CreateRamp** (int start_speed, int end_speed, int acceleration, int ticks_per_sec, RAMP_ELEMENT* starting_ramp_addr, int speeds_per_ramp) (C)

CREATE.RAMP (start_speed\end_speed\accel\ticks_per_sec\starting_ramp_addr\ speeds_per_ramp -- steps_in_ramp) (Forth)

Writes `speed_per_ramp` +1 entries into the `RAMP.ARRAY` starting at the specified `starting_ramp_addr` to attain the specified starting and ending speeds and acceleration (or deceleration). Returns the number of steps in the created ramp. `start_speed`, `end_speed` and `acceleration` are all interpreted as positive numbers. Speeds are in units of steps per second if the motor is configured for full stepping, or halfsteps per second if the motor is configured for half stepping. The acceleration is in units of (half) steps per second per second. Speeds are clamped to the attainable range (between 0 and `ticks_per_second`), and the acceleration is clamped such that a maximum of 10 seconds is spent at any one transient speed in a ramp. Each ramp entry comprises a `step_limit` which specifies the number of steps to be taken at the speed, and a `duty_cycle` which specifies the speed (see the glossary entry for `SPEED.TO.DUTY`). If the specified `speeds_per_ramp` = 0, this function simply writes a "final" speed by setting the `step_limit` to 0. For non-zero `speeds_per_ramp`, this routine writes the specified number of ramp entries, plus an additional entry at the final speed with the `step_limit` set to 0 which tells the `STEP.MANAGER` that this is the final speed in the ramp. Note that higher level calling routines can write over the final speed, or concatenate two ramps to achieve a speed profile that ramps up to a steady speed for a specified number of steps, and then smoothly ramps down to a stopped state.

void **DownloadMap** (void) (C)

DOWNLOAD.MAP (--) (Forth)

Sets a flag in EEPROM and changes the state of a latch in the onboard PALs to put the download memory map into effect on flash-equipped QED-Flash Boards. After execution of this routine, and upon each subsequent reset or restart, pages 4, 5, and 6 are addressed in the S2 RAM, and pages 1, 2, and 3 are addressed in the S1 flash memory. This allows code (and Forth names) to be compiled into RAM on pages 4, 5 and 6 and then transferred to flash using the `PageToFlash()` (`PAGE.TO.FLASH` in Forth) function. To establish the standard memory map, see the glossary entry for `StandardMap()` (`STANDARD.MAP` in Forth). Note that the standard map is active after a "factory cleanup" operation.

void **Link_File_IO**(void) (C)

Link_File_IO (--) (Forth)

This function (present in kernel versions starting with V4.05) links in the Forth headers of the ATA Flash Card Software Package into the linked list of Forth names. See the MIB/ATA Flash Card Users Manual for information on how to use this function.

void **PageToFlash** (int source_page) (C)

PAGE.TO.FLASH (source_page --) (Forth)

Transfers the 32 Kbyte contents of the specified RAM `source_page` to the parallel page in flash. If the current memory map is the "download map", then valid source pages are 4, 5, or 6: page 4 RAM is transferred to page 1 flash, page 5 RAM is transferred to page 2 flash, and page 6 RAM is transferred to page 3 flash. If the current memory map is the "standard map", then valid source pages are 1, 2, or 3: page 1 RAM is transferred to page 4 flash, page 2 RAM is transferred to page 5 flash, and page 3 RAM is transferred to page 6 flash. An "invalid input parameter" error is issued if an invalid `source_page` is specified. A "can't program flash" error is issued if the flash cannot be programmed;

make sure that the "Write Enable Flash" DIP switch # 2 is ON and that DIP switches 3 and 4 are also ON. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user.

void **PageToRAM** (int source_page) (C)
PAGE.TO.RAM (source_page --) (Forth)

Transfers the 32 Kbyte contents of the specified flash `source_page` to the parallel page in RAM. If the current memory map is the "download map", then valid source pages are 1, 2, or 3: page 1 flash is transferred to page 4 RAM, page 2 flash is transferred to page 5 RAM, and page 3 flash is transferred to page 6 RAM. If the current memory map is the "standard map", then valid source pages are 4, 5, or 6: page 4 flash is transferred to page 1 RAM, page 5 flash is transferred to page 2 RAM, and page 6 flash is transferred to page 3 RAM. An "invalid input parameter" error is issued if an invalid `source_page` is specified.

static int **pcc_bank** (C)
PCC_Bank (-- xaddr) (Forth)

A system variable holding the current 5-bit pattern that determines which of 32 1-Megabyte banks is accessible in the PC Card. The user typically does not need to use this variable, and the high order PCMCIA address bits that it references are not even used to access ATA flash cards (they are only needed for linear flash or RAM cards). The contents of this variable and the corresponding pcc bank bits are zeroed by Init_File_System and Init_File_IO as described in the MIB/ATA Flash Card Users Manual; please consult that document for more information.

Implementation Details: To initialize the PCC bank bits, store the constant 0x01F0 into the pcc_bank variable, and then call PCC_Offset_To_Xaddr specifying an offset parameter of zero.

xaddr **PccOffsetToXaddr** (long offset) (C)
PCC.OFFSET>XADDR (d.offset -- xaddr) (Forth)

Returns the extended address in the QED Board's memory map corresponding to the specified 32-bit offset address from the beginning of a linear PC card, and also sets the pcc_bank bits appropriately to access the correct 1 Mbyte bank in the card. The user typically does not need to call this low-level routine, and the high order PCMCIA address bits that it references are not even used to access ATA flash cards (they are only needed for linear flash or RAM cards). Please consult the MIB/ATA Flash Card Users Manual for more information.

Implementation Details: Using a banked memory scheme, the QED Memory Interface Board (MIB) maps any RAM, battery-backed RAM, or linear flash PC Card up to 32 Megabytes in size at pages hex 80 through BF in the QED Board's address space. To accommodate slow memory timing, each byte in the PC Card is mapped onto 2 bytes of the QED Board's memory map. This function translates the intuitively simple offset to the hardware-dependent xaddress and pcc_bank bits (see pcc_bank).

int **PccPresent** (void) (C)
PCC.PRESENT? (-- flag) (Forth)

Returns a true (-1) flag if a PC Card (also called a PCMCIA Card) is installed in the Memory Interface Board; otherwise returns a false (0) flag.

Implementation details: Reads the card status byte mapped at address 0x0000 on page 0x30. A true flag is returned if the active-low card detection bits 0 and 1 are low, and the

battery voltage detection bits 4 and 5 are high, and the /WAIT signal at bit 7 is inactive high.

int **SpeedToDuty** (int steps_per_second, int ticks_per_second) (C)
SPEED.TO.DUTY (steps_per_second \ ticks_per_second -- duty_cycle) (Forth)

Returns an integer representation of a duty cycle which specifies the step rate of the stepper motor. The first input parameter is the number of steps per second if full stepping, or the number of halfsteps per second if half stepping. The second input parameter is the number of clock ticks per second; the default is 1000 ticks per second. The output parameter can be interpreted as a fraction with the radix point to the left of the most significant bit. A 100% duty cycle is represented by hex 0xFFFF, and this tells the STEP.MANAGER to output a new step pattern on every tick of the interrupt clock (e.g., once per millisecond, corresponding to 1000 (half) steps per second). A duty cycle of hex 0x8000 means a new step pattern is written to the motor port every other clock tick; a duty cycle of hex 0x0100 dictates one step every 256 clock ticks; and a duty cycle of 0000 means corresponds to a stopped state with no step pattern updates.

void **StandardMap** (void) (C)
STANDARD.MAP (--) (Forth)

Sets a flag in EEPROM and changes the state of a latch in the onboard PALs to put the standard memory map into effect on flash-equipped QED-FLASH Boards. After execution of this routine, and upon each subsequent reset or restart, pages 1, 2, and 3 are addressed in the S2 RAM, and pages 4, 5, and 6 are addressed in the S1 flash memory. After code is downloaded to RAM and transferred to flash using the PageToFlash() (PAGE.TO.FLASH in Forth) function, establishing the standard map allows code resident on pages 4, 5 and 6 to be executed. To establish the download memory map, see the glossary entry for DownloadMap() (DOWNLOAD.MAP in Forth). Note that the standard map is active after a "factory cleanup" operation.

void **StepManager** (void) (C)
STEP.MANAGER (--) (Forth)

Expects the base address of the STATUS.ARRAY in the Y register. Based on the information in the STATUS.ARRAY and the RAMP.ARRAY, for each enabled motor STEP.MANAGER writes the appropriate step pattern at the specified duty cycle to the motor port to attain the speed specified in the motor's RAMP.ARRAY. This function is meant to be called from a periodic interrupt service routine typically associated with an output compare (OC) interrupt; the default time base is once per millisecond generated by the OC3 interrupt, with a resulting maximum speed of 1000 full- or half-steps per second. This assembly coded routine executes in approximately 120 µs per enabled stepper motor. Thus running four stepper motors at a maximum speed of 1000 full- or half-steps per second requires approximately half of the 68HC11's available time (480 µs interrupt service time every 1000 µs).

int **ToFlash** (xaddr src_xaddr, xaddr dest_xaddr, uint num) (C)
TO.FLASH (src_xaddr \ dest_xaddr \ num -- success_flag) (Forth)

Transfers num bytes (0 <= num <= 65,535) starting at the specified source extended address, to the specified destination extended address in flash. The source may be anywhere in memory; it may even be in the flash which is being programmed. The destination must be in flash. Returns a flag equal to -1 if the programming was successful, or 0 if the programming failed. Reasons for failure include improper DIP switch settings, or a destination that is not in a programmable page in flash memory. Recall that DIP switches number 2, 3, and 4 must be ON to allow writes to the flash. (If

any locations in the flash are programmed more than 10,000 times, the cell may wear out causing a failure flag to be returned). Assuming that the standard 256 Kbyte flash is present on the board, writable flash pages include pages 4, 5 and 6 for the standard map, and pages 1, 2, and 3 for the download memory map. Page 7 is always in flash and writable; it provides an excellent location for data storage. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user.

xaddr **TO_XADDR**(uint address, int page) (C)
This C macro combines the specified 16-bit address and page into a single 32-bit extended address. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.

int **WhichMap** (void) (C)
WHICH.MAP (-- [0] or [1]) (Forth)
Returns a 0 if the current memory map is the "standard map", and returns a 1 if the current map is the "download map" on flash-carrying boards. If the standard map is active, pages 4, 5, and 6 are addressed in the S1 flash, and pages 1, 2, and 3 are addressed in the S2 RAM. If the download map is active, pages 4, 5, and 6 are addressed in the S2 RAM, and pages 1, 2, and 3 are addressed in the S1 flash memory. This routine allows a user or program to verify which map is currently being used. After a "factory cleanup" operation, the standard map is active.

uint **XADDR_TO_ADDR** (xaddr xaddress) (C)
This C macro converts the specified 32-bit extended address into its constituent 16-bit address. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.

uint **XADDR_TO_PAGE** (xaddr xaddress) (C)
This C macro converts the specified 32-bit extended address into its constituent page. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.

Glossary Addendum: Custom Routines Added to the QED V4.0x Kernel

BUFFER>SPI (xaddr\+n --) (Forth)
This routine is headerless; its xcfa is stored at address 7FF8 on page C.
Writes to the SPI the contents of the buffer specified by xaddr and +n, where xaddr is the starting address, and +n is the number of bytes (0 <= +n <= 32,768). The buffer must not cross a page boundary. This routine does not `GET` or `RELEASE` the `SPI.RESOURCE`, nor does it modify the configuration of the SPI or activate any chip selects. If required, these additional functions must be performed by the calling program. This routine is optimized for speed, and executes at 9 microseconds per byte. To access this routine from C, contact Mosaic Industries. To access this routine from Forth, add the following definition to your Forth source code:

```
HEX  : BUFFER>SPI  7FF8  C X@  EXECUTE ;
```

CALC.CHECKSUM (xaddr\+n -- checksum | n MUST be even) (Forth)
This routine is headerless; its xcfa is stored at address 7FFC on page C.
Calculates a 16-bit checksum for the buffer specified by xaddr and +n, where xaddr is the starting address, and +n is the number of bytes (0 <= +n <= 32,768). The buffer must not cross a page boundary, and n must be an even number of bytes. The checksum is

calculated by initializing a 16-bit accumulator to zero, then adding in turn each 2-byte number in the buffer to the accumulator; the checksum is the final value of the accumulator. Using this routine provides a method of checking whether the contents of an area of memory have changed since a prior checksum was calculated. This routine is optimized for speed, and executes at less than 3 microseconds per byte. To access this routine from C, contact Mosaic Industries. To access this routine from Forth, add the following definition to your Forth source code:

```
HEX  : CALC.CHECKSUM     7FFC  C X@  EXECUTE ;
```

Appendix A: QED-Flash Connector Pinouts

The pinouts of all of the connectors on the QED-Flash Board are presented below**.** To locate the connectors on the board, consult Figure A**.**1 and the white silkscreened labels on the "memory side" of the QED-Flash Board**.** On the "processor side" of the board, the legend indicates the location of pin 1 and the highest numbered pin on each connector**.**

Note that some signals have compound names to suggest multiple functions**.** For example, PPB5/Display.RS on the keypad/display connector is a signal that can be configured as a general purpose digital I/O line named PPB5, or as a signal that controls the reset line of the display.

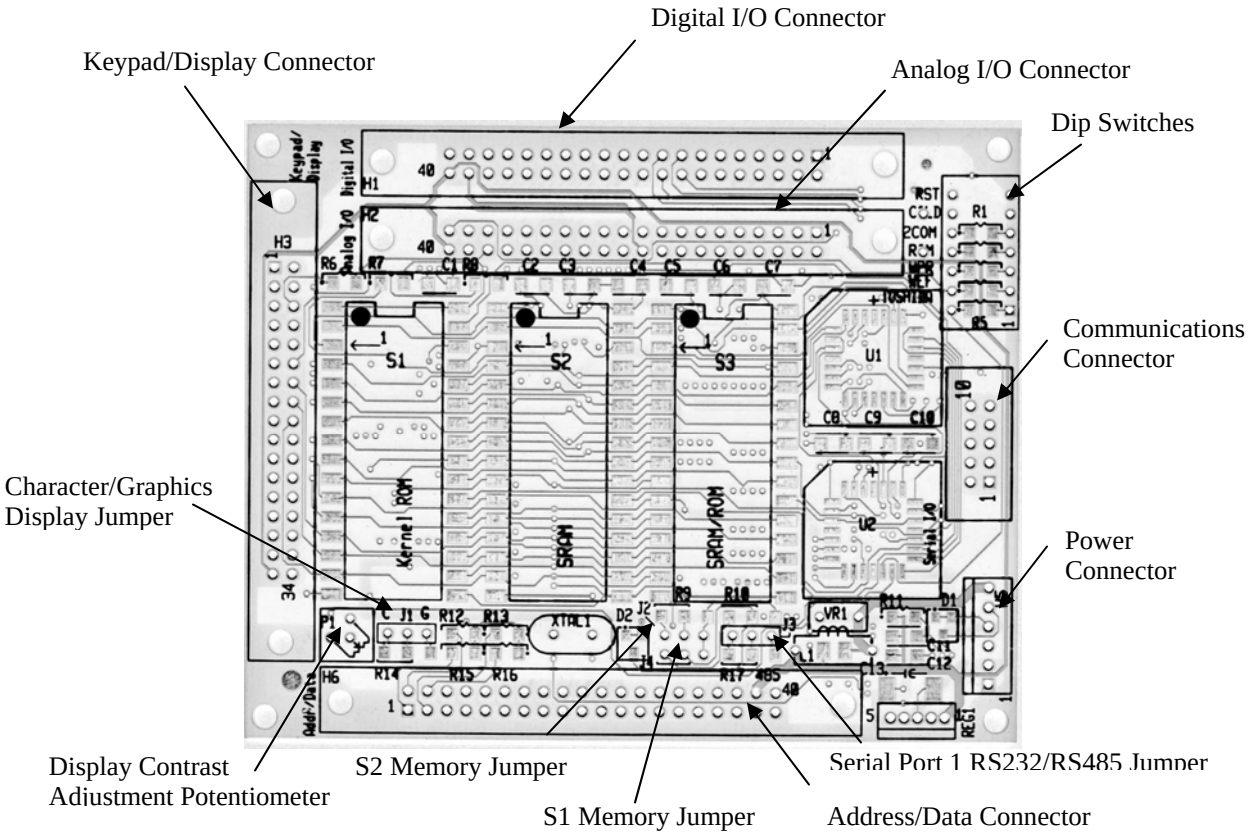


Figure A**.**1 Diagram of the memory side of the QED-Flash Board with each of the key components and connectors labeled**.**

QED Communications Connector

TxD1 -	1	2	- RxD1
DGND -	3	4	- DGND
XCVR- -	5	6	- XCVR+
TxD2 -	7	8	- RxD2
DGND -	9	10	- DGND

QED Power Connector

/Shutdown	- 1
AGND	- 2
+5VAN	- 3
DGND	- 4
+5V	- 5
6-12VDC Raw Vin	- 6

Address/Data Connector

PG7	-	1	2	-	PG6
PG5	-	3	4	-	PG4
PG3	-	5	6	-	PG2
PG1	-	7	8	-	PG0
A15	-	9	10	-	A14
A13	-	11	12	-	A12
A11	-	13	14	-	A10
A9	-	15	16	-	A8
A7	-	17	18	-	A6
A5	-	19	20	-	A4
A3	-	21	22	-	A2
A1	-	23	24	-	A0
D7	-	25	26	-	D6
D5	-	27	28	-	D4
D3	-	29	30	-	D2
D1	-	31	32	-	D0
/RESET	-	33	34	-	R//W
/OE	-	35	36	-	/WE
E	-	37	38	-	V+Raw
DGND	-	39	40	-	+5V

Keypad/Display Connector

PPB4/KPR4	-	1	2	-	GND
PPB3/KPR3	-	3	4	-	+5V
PPC3/KPC3	-	5	6	-	Vcontrast
PPC0/KPC0	-	7	8	-	A1
PPB2/KPR2	-	9	10	-	R//W
PPC1/KPC1	-	11	12	-	Display**.**E
PPC2/KPC2	-	13	14	-	D0
PPB1/KPR1	-	15	16	-	D1
PPB0/KPR0	-	17	18	-	D2
DGND	-	19	20	-	D3
HC0*	-	21	22	-	D4
HC1*	-	23	24	-	D5
HC2*	-	25	26	-	D6
HC3*	-	27	28	-	D7
DGND	-	29	30	-	DGND
SNUB	-	31	32	-	+5VDD
V+RAW	-	33	34	-	VEE

* HC0 is generated by PAL, HC1-HC3 are generated using PB5-PB7 respectively

Digital I/O Connector

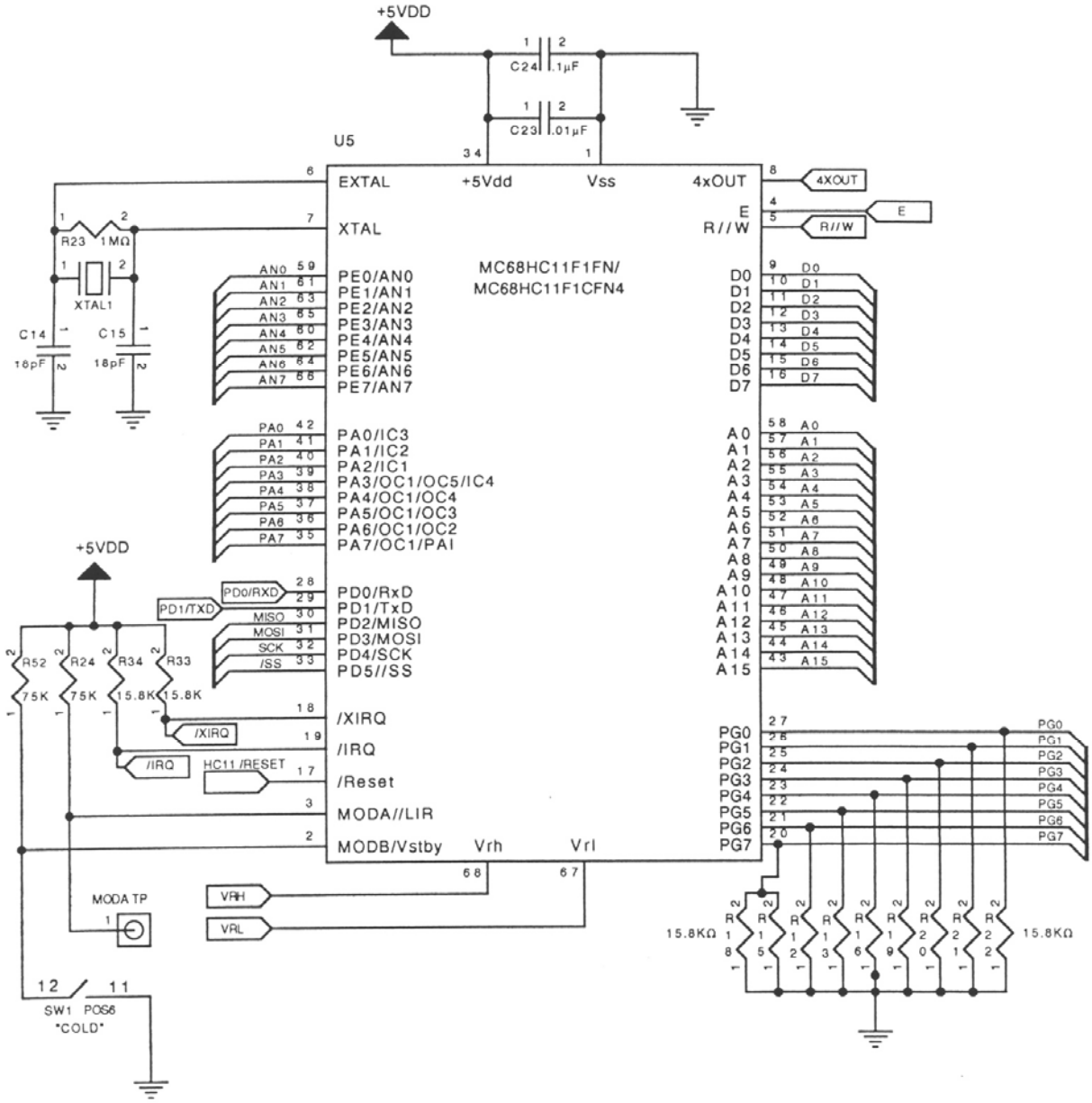
GND	- 1	2	-	+5V
PA7	- 3	4	-	PA6
PA5	- 5	6	-	PA4
PA3	- 7	8	-	PA2
PA1	- 9	10	-	PA0
PD5//SS	- 11	12	-	PD4/SCK
PD3/MOSI	- 13	14	-	PD2/MISO
E	- 15	16	-	R//W
/OE	- 17	18	-	/WE
/XIRQ	- 19	20	-	/IRQ
PFI Input	- 21	22	-	4xOut
/Reset	- 23	24	-	/Shutdown
PPC7	- 25	26	-	PPC6
PPC5	- 27	28	-	PPC4/RS485.XMIT
PPA7	- 29	30	-	PPA6
PPA5	- 31	32	-	PPA4
PPA3	- 33	34	-	PPA2
PPA1	- 35	36	-	PPA0
AGND	- 37	38	-	+5VAN
DGND	- 39	40	-	V+Raw

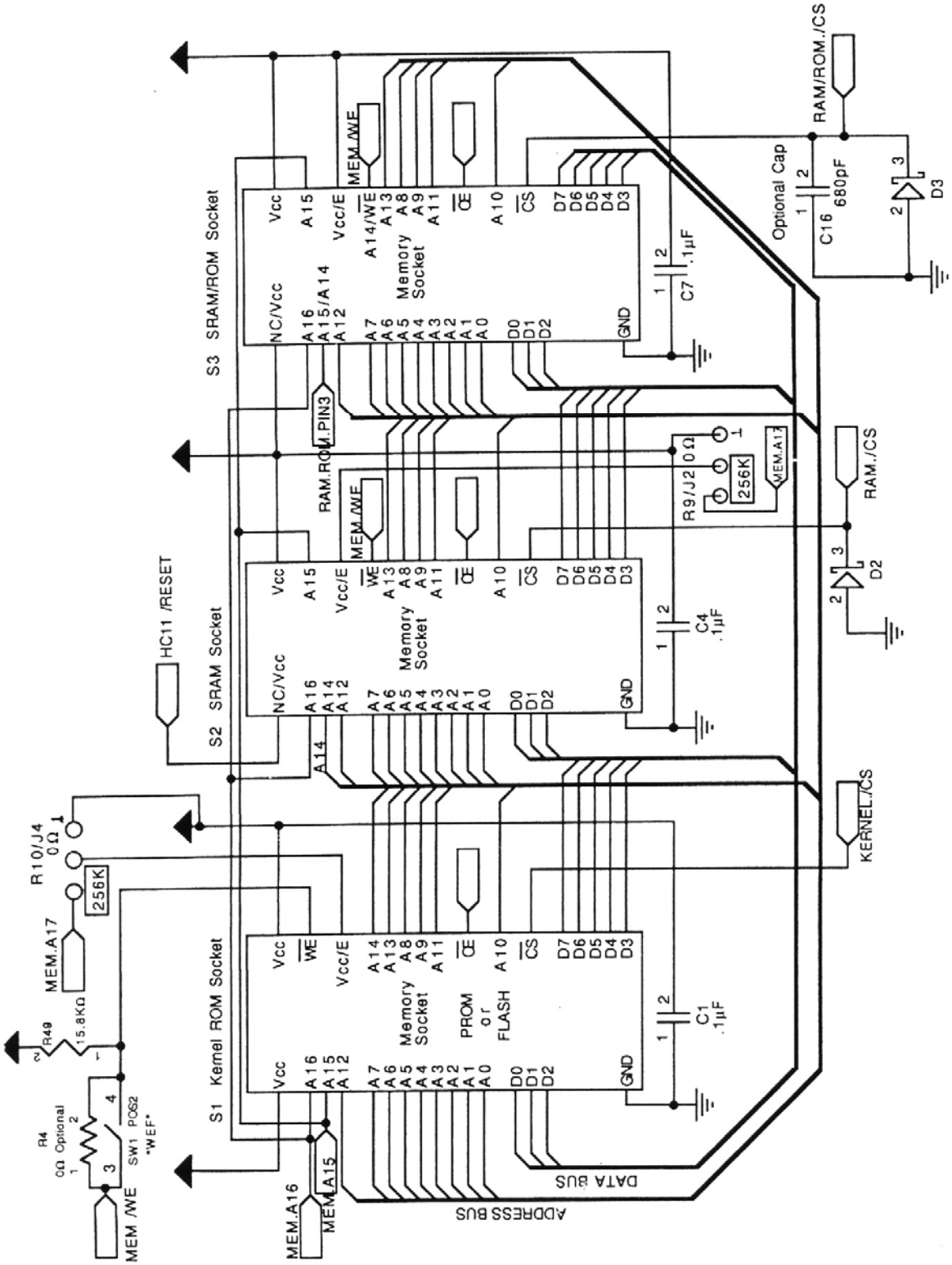
Analog I/O Connector

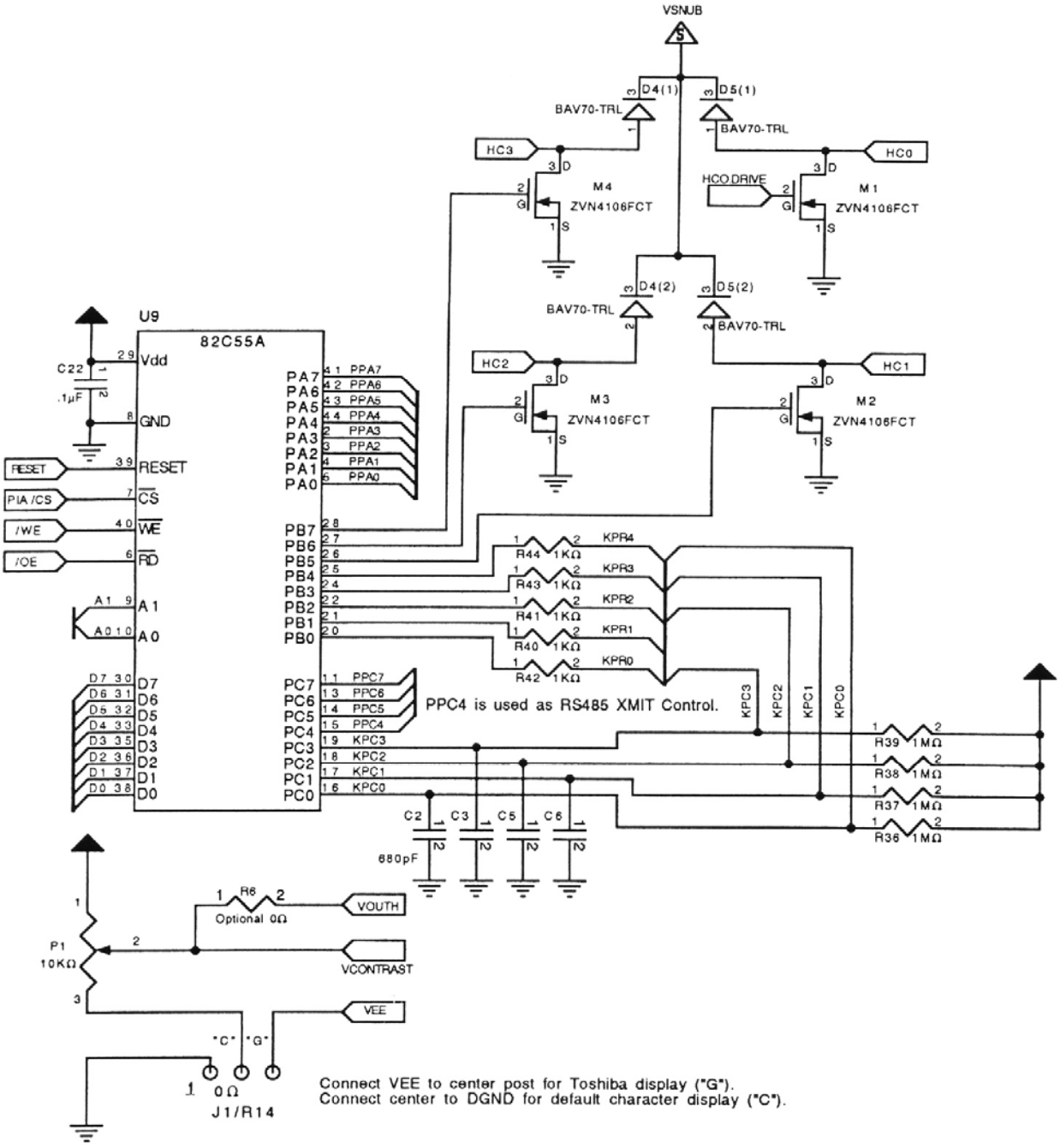
Vrl	- 1	2	-	Vrh
PE7/AN7	- 3	4	-	PE6/AN6
PE5/AN5	- 5	6	-	PE4/AN4
PE3/AN3	- 7	8	-	PE2/AN2
PE1/AN1	- 9	10	-	PE0/AN0
12AN7	- 11	12	-	12AN6
12AN5	- 13	14	-	12AN4
12AN3	- 15	16	-	12AN2
12AN1	- 17	18	-	12AN0
AGND	- 19	20	-	+5VAN
Vin1	- 21	22	-	Vout1
Vin2	- 23	24	-	Vout2
Vin3	- 25	26	-	Vout3
Vin4	- 27	28	-	Vout4
Vin5	- 29	30	-	Vout5
Vin6	- 31	32	-	Vout6
Vin7	- 33	34	-	Vout7
Vin8	- 35	36	-	Vout8
1.5Vref	- 37	38	-	+5V
Analog Bus V-	- 39	40	-	V+Raw

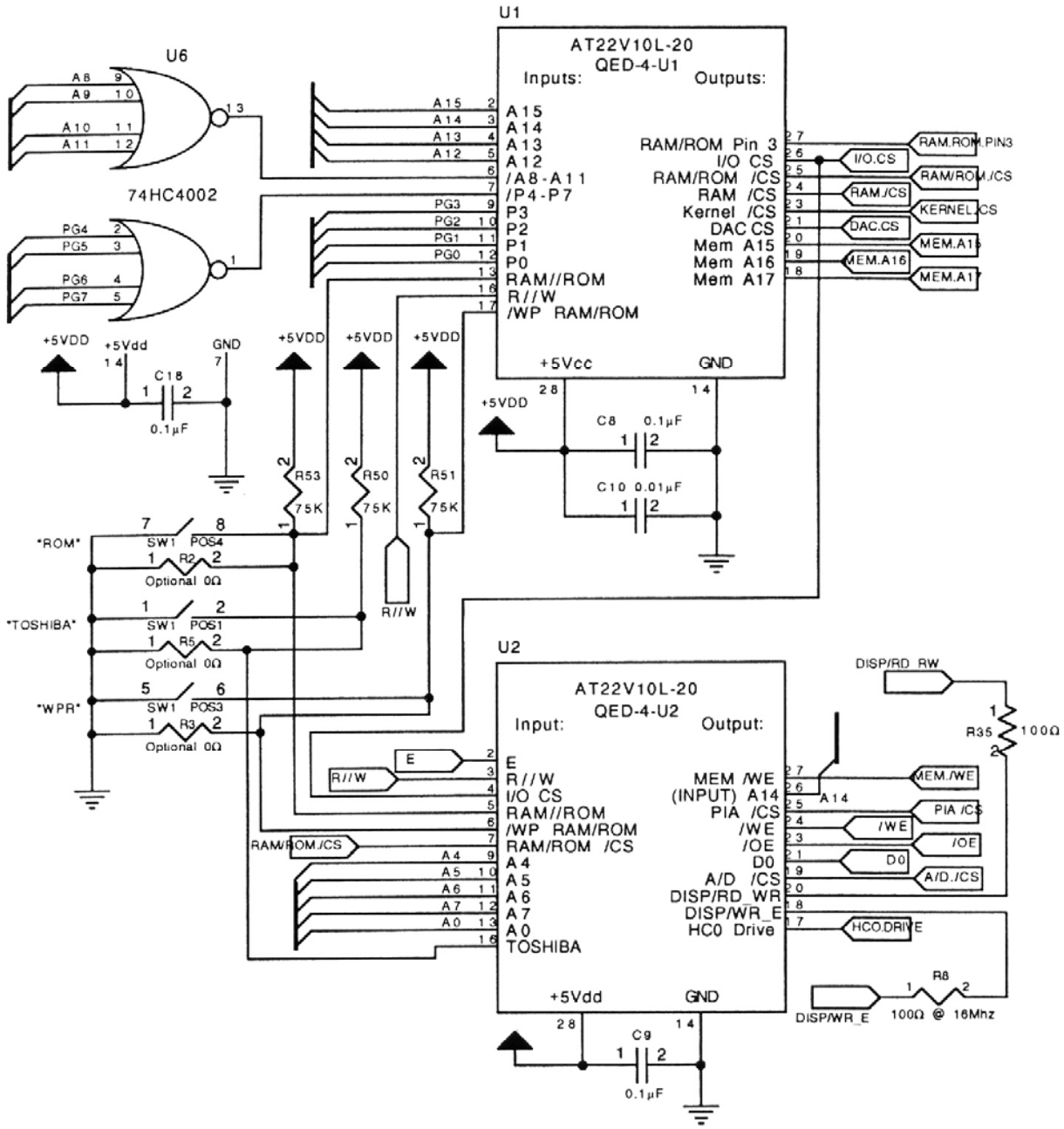
Appendix B: QED-Flash Board Schematics

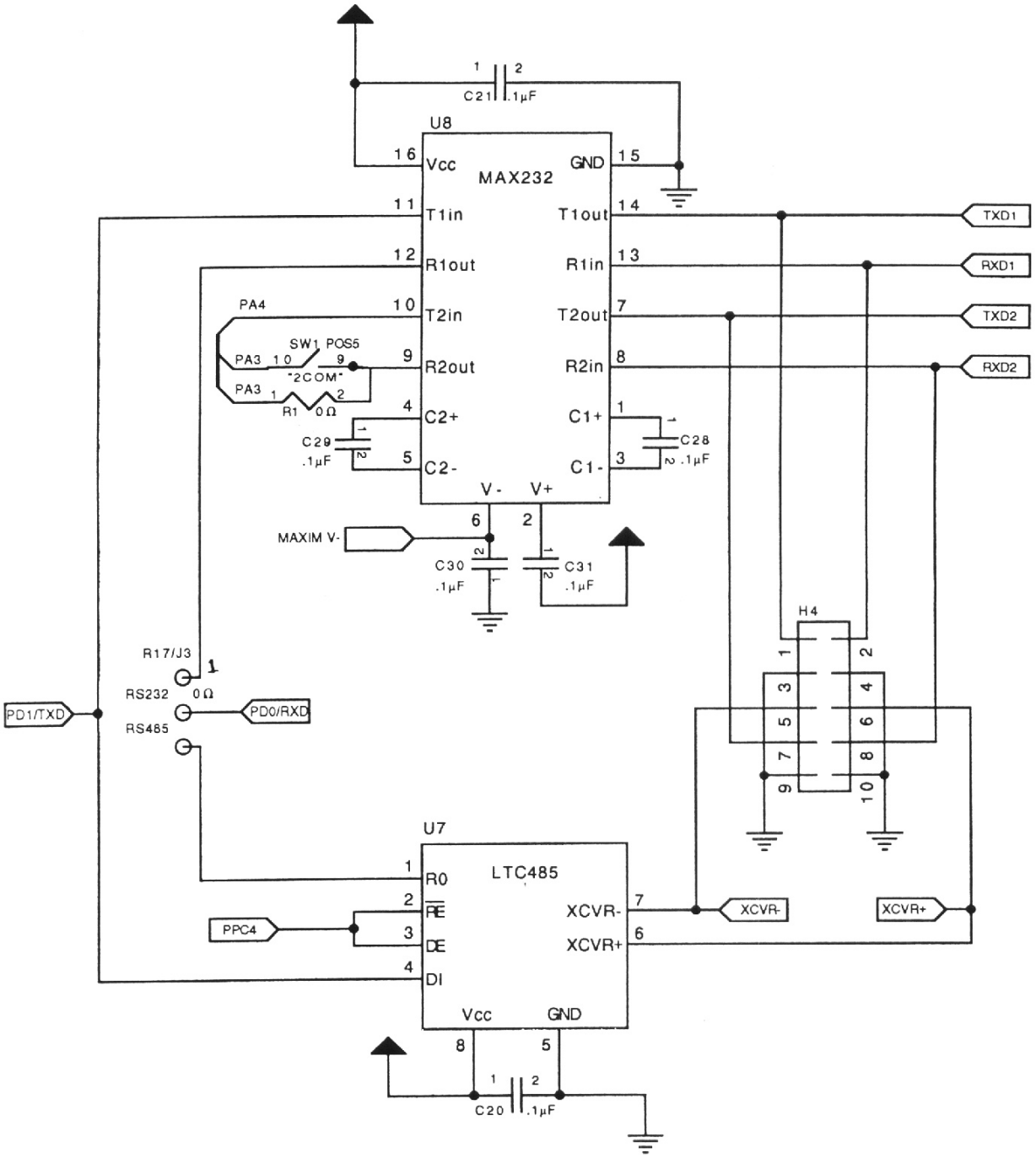
Microprocessor



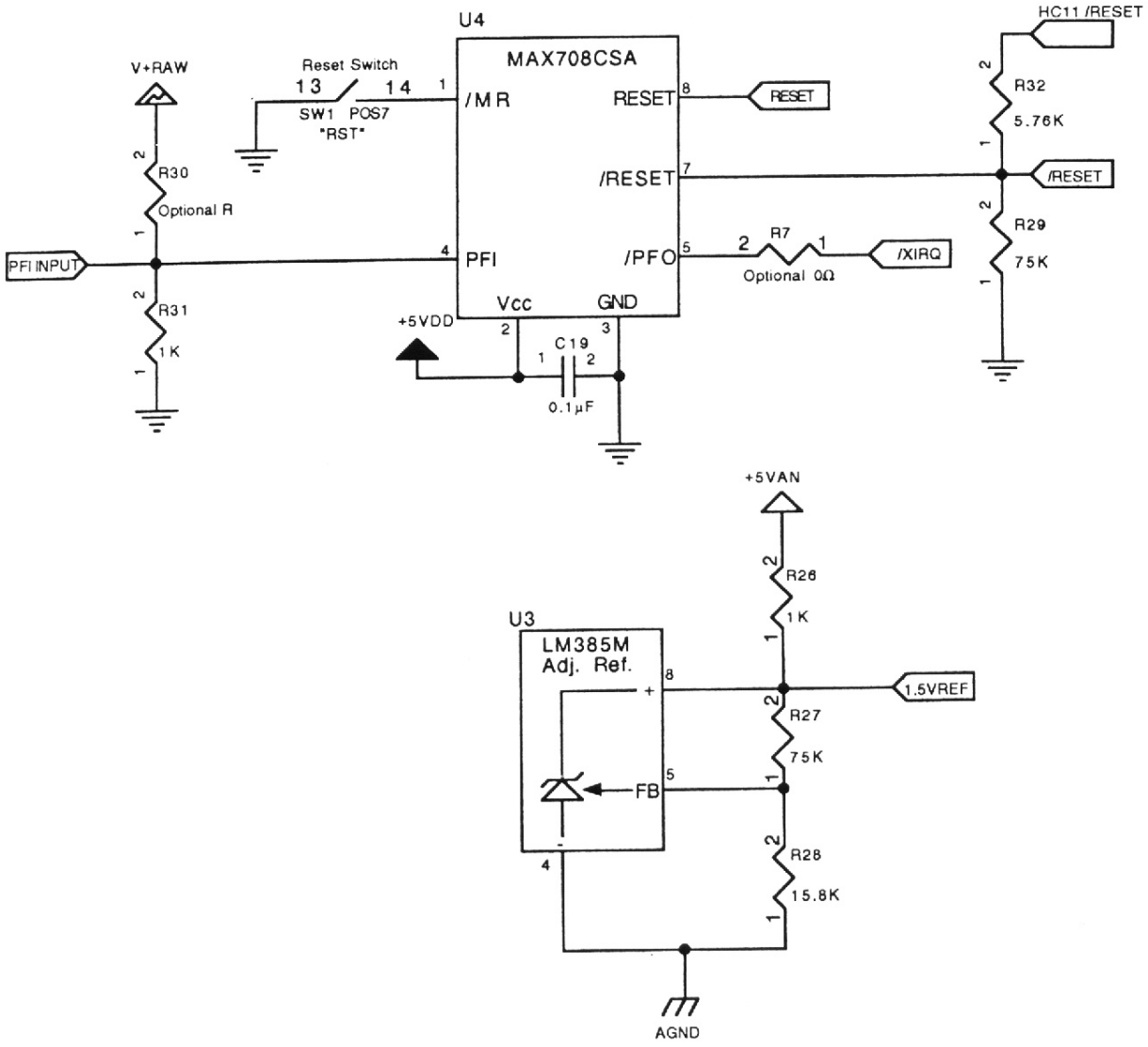

S3 SRAM/ROM Socket
S2 SRAM Socket
S1 Kernel ROM Socket
Memory Socket
PROM or FLASH
RAM/ROM./CS
RAM./CS
KERNEL./CS
Optional Cap
C16 680pF
C7 .1µF
C4 .1µF
C1 .1µF
D3
D2
R9/J2 0Ω
256K
MEM.A17
R10/J4 0Ω
R49 15.8KΩ
R4 0Ω Optional
SW1 POS2 "WEF"
MEM /WE
MEM.A16
MEM.A15
HC11 /RESET
RAM.ROM.PIN3
DATA BUS
ADDRESS BUS


VSNUB
D4(1)
D5(1)
BAV70-TRL
BAV70-TRL
HC3
HC0
M4
ZVN4106FCT
HCO DRIVE
M1
ZVN4106FCT
D4(2)
D5(2)
BAV70-TRL
BAV70-TRL
HC2
HC1
M3
ZVN4106FCT
M2
ZVN4106FCT
U9
82C55A
C22
.1µF
Vdd
GND
RESET
CS
WE
RD
A1
A0
D7
D6
D5
D4
D3
D2
D1
D0
PA7
PA6
PA5
PA4
PA3
PA2
PA1
PA0
PB7
PB6
PB5
PB4
PB3
PB2
PB1
PB0
PC7
PC6
PC5
PC4
PC3
PC2
PC1
PC0
PPA7
PPA6
PPA5
PPA4
PPA3
PPA2
PPA1
PPA0
PPC7
PPC6
PPC5
PPC4
KPC3
KPC2
KPC1
KPC0
RESET
PIA /CS
/WE
/OE
R44 1KΩ
R43 1KΩ
R41 1KΩ
R40 1KΩ
R42 1KΩ
KPR4
KPR3
KPR2
KPR1
KPR0
PPC4 is used as RS485 XMIT Control.
KPC3
KPC2
KPC1
KPC0
R39 1MΩ
R38 1MΩ
R37 1MΩ
R36 1MΩ
C2
C3
C5
C6
680pF
R6
Optional 0Ω
VOUTH
P1
10KΩ
VCONTRAST
VEE
"C"
"G"
0Ω
J1/R14
Connect VEE to center post for Toshiba display ("G").
Connect center to DGND for default character display ("C").

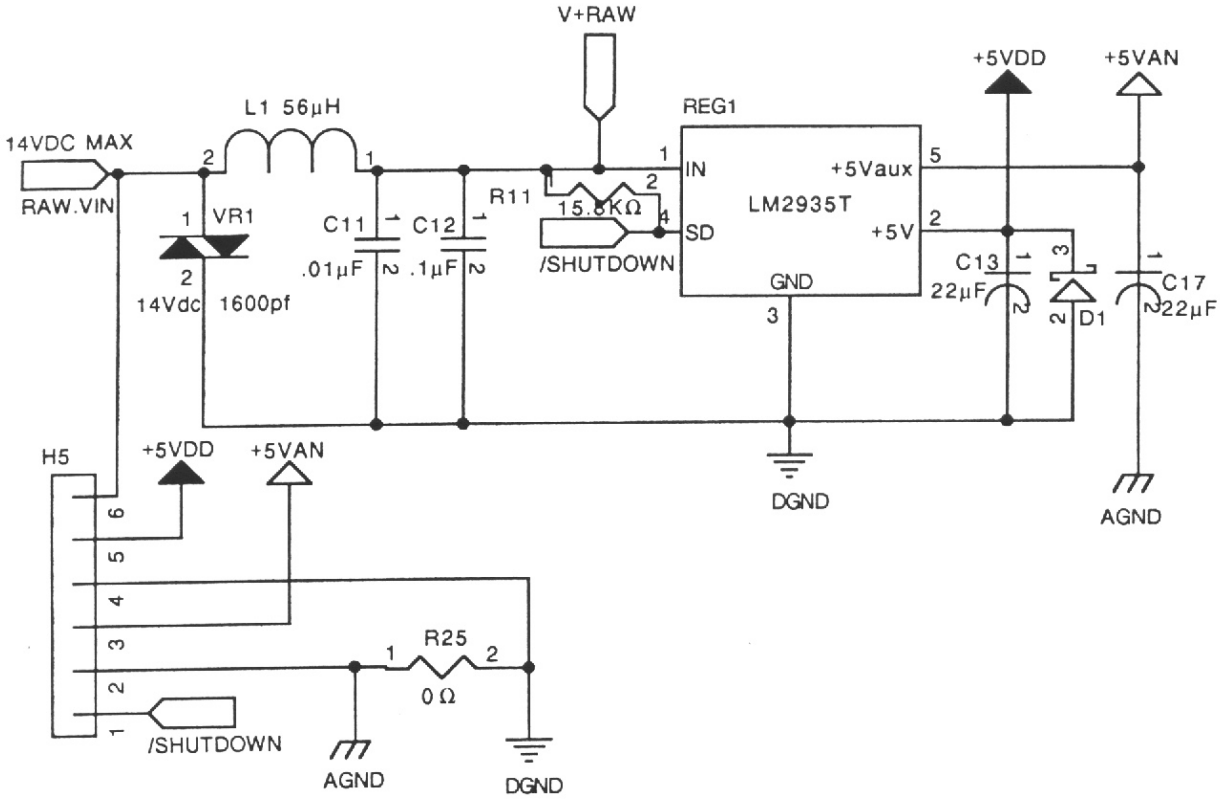

U1
AT22V10L-20
QED-4-U1
Inputs:
Outputs:
U6
74HC4002
A15
A14
A13
A12
/A8-A11
/P4-P7
P3
P2
P1
P0
RAM//ROM
R//W
/WP RAM/ROM
RAM/ROM Pin 3
I/O CS
RAM/ROM /CS
RAM /CS
Kernel /CS
DAC CS
Mem A15
Mem A16
Mem A17
RAM.ROM.PIN3
I/O.CS
RAM/ROM./CS
RAM./CS
KERNEL.CS
DAC.CS
MEM.A15
MEM.A16
MEM.A17
+5VDD
+5Vdd
+5Vcc
GND
C18
0.1µF
C8
C10 0.01µF
R53
R50
R51
75K
"ROM"
"TOSHIBA"
"WPR"
SW1
POS4
POS1
POS3
R2
R5
R3
Optional 0Ω
R//W
U2
AT22V10L-20
QED-4-U2
Input:
Output:
E
R//W
I/O CS
RAM//ROM
/WP RAM/ROM
RAM/ROM /CS
A4
A5
A6
A7
A0
TOSHIBA
MEM /WE
(INPUT) A14
PIA /CS
/WE
/OE
D0
A/D /CS
DISP/RD_WR
DISP/WR_E
HC0 Drive
A14
MEM./WE
PIA./CS
/WE
/OE
D0
A/D./CS
HC0.DRIVE
DISP/RD_RW
R35
100Ω
R8
100Ω @ 16Mhz
C9
0.1µF

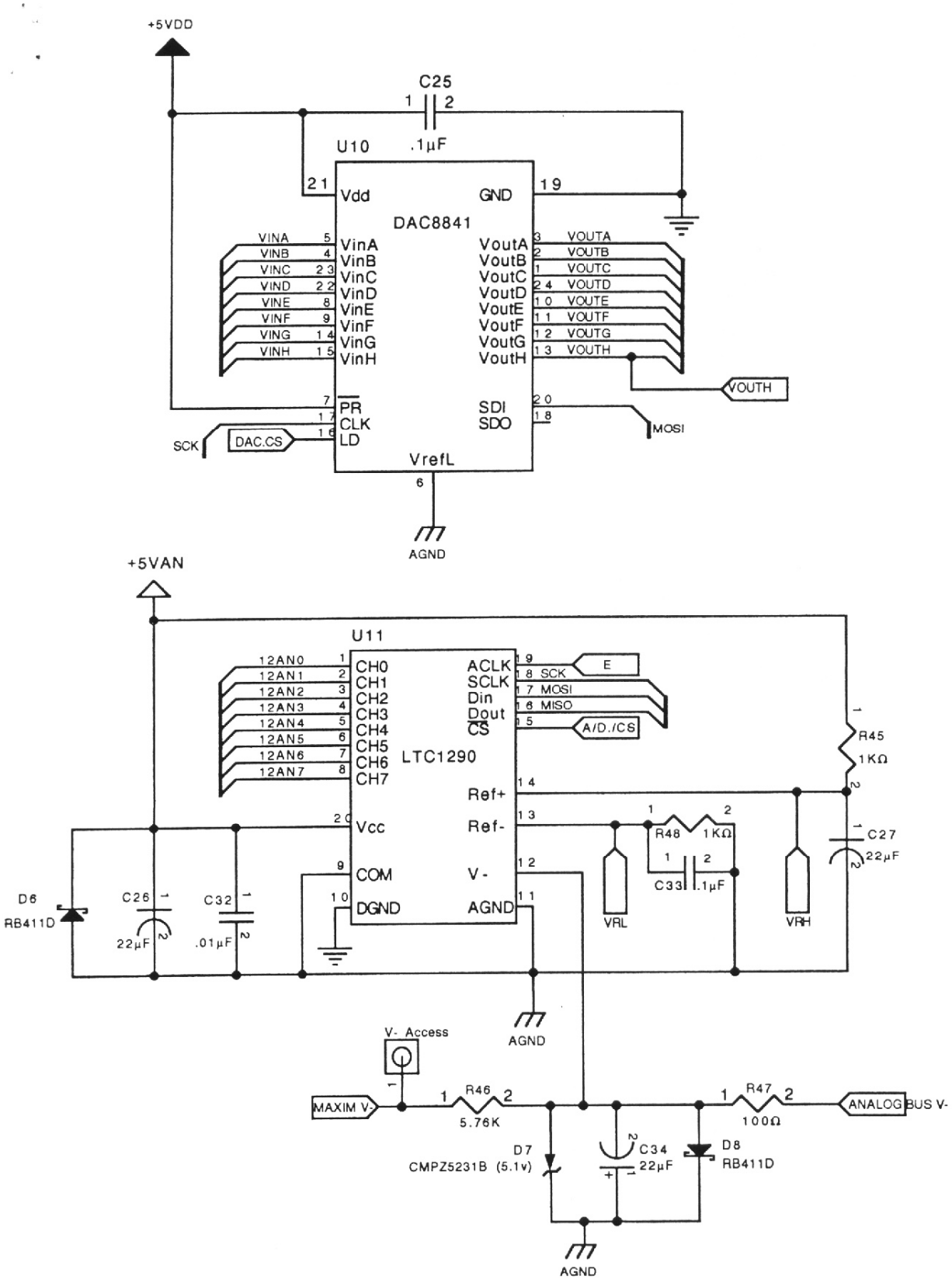

C21 .1µF
U8
MAX232
Vcc
GND
T1in
T1out
R1out
R1in
T2in
T2out
R2out
R2in
C2+
C1+
C2-
C1-
V-
V+
C28 .1µF
C29 .1µF
C30 .1µF
C31 .1µF
MAXIM V-
PA4
SW1 POS5
PA3
"2COM"
R1 0Ω
TXD1
RXD1
TXD2
RXD2
H4
R17/J3
RS232 0Ω
RS485
PD1/TXD
PD0/RXD
U7
LTC485
R0
RE
DE
DI
XCVR-
XCVR+
PPC4
C20 .1µF

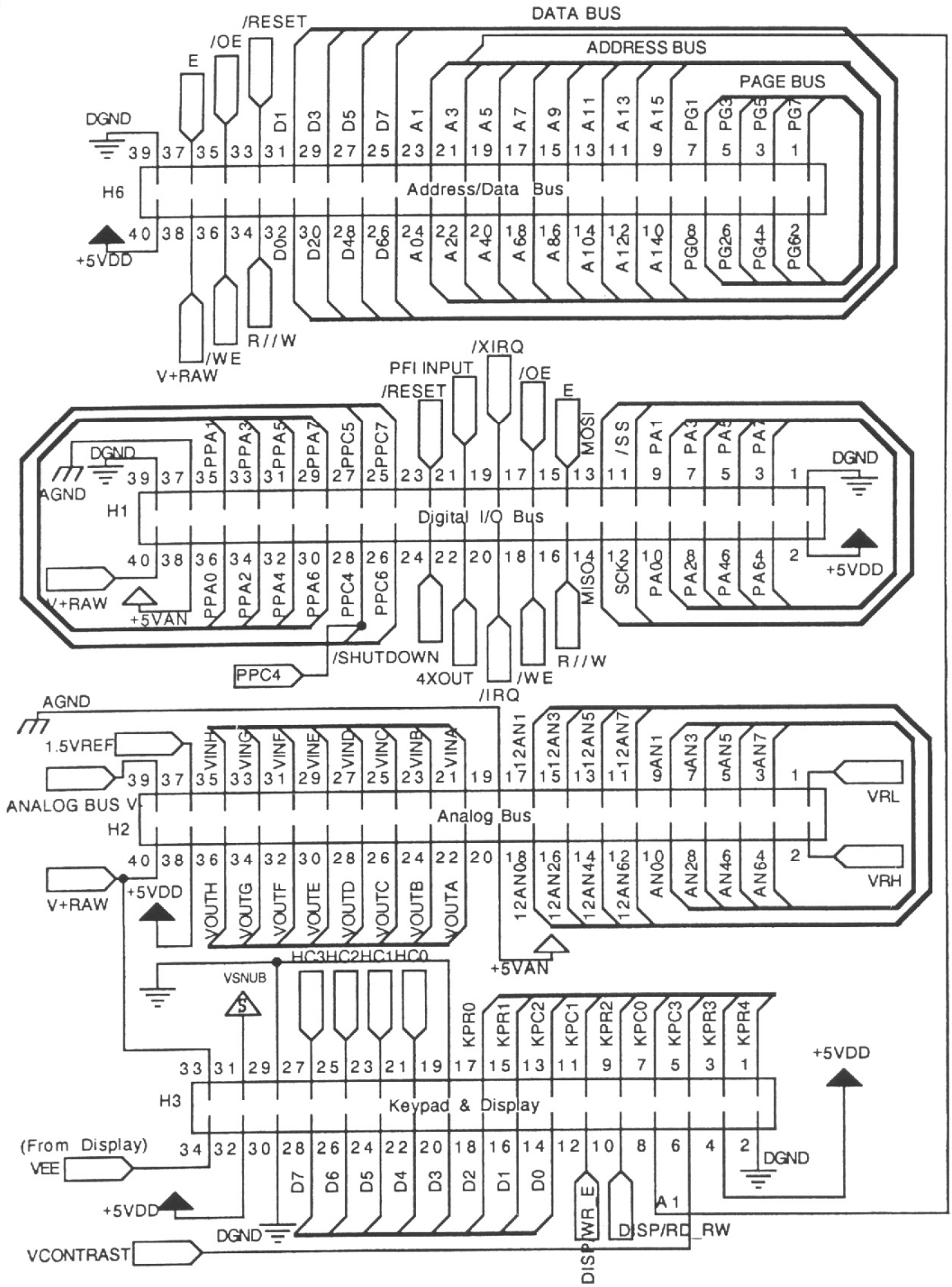
Reset Circuitry



Power Circuitry




+5VDD
C25
.1µF
U10
DAC8841
Vdd
GND
VinA
VinB
VinC
VinD
VinE
VinF
VinG
VinH
VoutA
VoutB
VoutC
VoutD
VoutE
VoutF
VoutG
VoutH
VINA
VINB
VINC
VIND
VINE
VINF
VING
VINH
VOUTA
VOUTB
VOUTC
VOUTD
VOUTE
VOUTF
VOUTG
VOUTH
PR
CLK
LD
SDI
SDO
SCK
DAC.CS
MOSI
VrefL
AGND
+5VAN
U11
LTC1290
CH0
CH1
CH2
CH3
CH4
CH5
CH6
CH7
12AN0
12AN1
12AN2
12AN3
12AN4
12AN5
12AN6
12AN7
ACLK
SCLK
Din
Dout
CS
E
SCK
MOSI
MISO
A/D./CS
Ref+
Ref-
Vcc
COM
V-
DGND
AGND
R45
1KΩ
R48
1KΩ
C27
22µF
C33
.1µF
VRL
VRH
D6
RB411D
C26
22µF
C32
.01µF
V- Access
MAXIM V-
R46
5.76K
R47
100Ω
ANALOG BUS V-
D7
CMPZ5231B (5.1v)
C34
22µF
D8
RB411D
AGND


DATA BUS
ADDRESS BUS
PAGE BUS
/RESET
/OE
E
DGND
H6
Address/Data Bus
+5VDD
R//W
/WE
V+RAW
/XIRQ
PFI INPUT
/RESET
/OE
E
MOSI
/SS
DGND
AGND
H1
Digital I/O Bus
V+RAW
+5VAN
/SHUTDOWN
PPC4
4XOUT
/WE
R//W
/IRQ
AGND
1.5VREF
ANALOG BUS V-
H2
Analog Bus
VRL
VRH
V+RAW
+5VDD
+5VAN
HC3HC2HC1HC0
VSNUB
H3
Keypad & Display
+5VDD
(From Display)
VEE
+5VDD
DGND
VCONTRAST
DISP WR
DISP/RD_RW
A1
DGND