

FlashCore™

Portable mass data storage controller with 16-bit CPU/SRAM/Flash, ADC, I/Os based on a 40MHz 188 CPU, and RS-232 communication



Technical Manual



1950 5th Street, Davis, CA 95616, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: sales@tern.com

<http://www.tern.com>

COPYRIGHT

FlashCore, 586-Engine, A-Engine, i386-Engine, and ACTF are trademarks of TERN, Inc.

Am188ES and Am186ES are trademarks of Advanced Micro Devices, Inc.

Paradigm C/C++ is a trademark of Paradigm Systems.

Borland C/C++ is a trademark of Borland International.

Microsoft, MS-DOS, Windows 95/98/2000/NT/ME/XP are trademarks of Microsoft Corporation.

Version 2.00

October 22, 2010

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of TERN, Inc.



© 1999-2010

1950 5th Street, Davis, CA 95616, USA

Tel: 530-758-0180 Fax: 530-758-0181

Email: sales@tern.com

<http://www.tern.com>

Important Notice

TERN is developing complex, high technology integration systems. These systems are integrated with software and hardware that are not 100% defect free. **TERN products are not designed, intended, authorized, or warranted to be suitable for use in life-support applications, devices, or systems, or in other critical applications.** **TERN** and the Buyer agree that **TERN** will not be liable for incidental or consequential damages arising from the use of **TERN** products. It is the Buyer's responsibility to protect life and property against incidental failure.

TERN reserves the right to make changes and improvements to its products without providing notice.

Temperature readings for controllers are based on the results of limited sample tests; they are provided for design reference use only.

Chapter 1: Introduction

1.1 Functional Description

The *FlashCore*™ provides a simple interface for 50-pin CompactFlash Cards (ranging in size from 8MB - 1 GB Flash cards) which are used to provide non-volatile storage in a wide array of applications, ranging from digital cameras to PDAs. 50-pin CompactFlash cards can also interface with PCs via a standard PCMCIA adapter, making these ideal storage solutions for applications requiring mass data exchange. In addition, the 50-pin CompactFlash cards cost less than 68-pin PCMCIA cards.

The *FlashCore*™ from TERN is available in two different hardware configurations: *FC-0* without CPU and *FC* which includes CPU.

The *FC-0* is designed as a CompactFlash card expansion board for TERN controllers (accessed and controlled via the 20x2 pin J1 header). The *FC-0* provides a 50-pin CF receptacle and a 20x2 pin socket. There is no CPU system on the *FC-0*. It allows you to add mass data storage to TERN controllers such as the *586-Engine*™, *A-Engine*™, and *i386-Engine*™.

The *FC* is a complete standalone C/C++ programmable embedded controller including a 16-bit 188 CPU, 512KB ACTF Flash, 128KB, or 512KB SRAM, 512-byte EEPROM, 2 channel RS-232 driver, 5V regulator, with optional real-time clock, battery, and/or 11 channel 12-bit ADC.

By using the *FlashCore (FC)*, users can easily add widely used CF standard mass data storage cards to their embedded application via RS232, TTL I2C, or parallel interface. TERN supports a complete C/C++ programmable software package (EV-P, or DV-P kit) which includes compiler, remote debugger, samples, and libraries. TERN software supports Linear Block Address mode, 16-bit FAT flash file system, RS-232, TTL I2C or parallel communication. Users can write a file to the CompactFlash card or read a file from the CompactFlash card. Users can also transfer the file to a PC via the PCMCIA port.

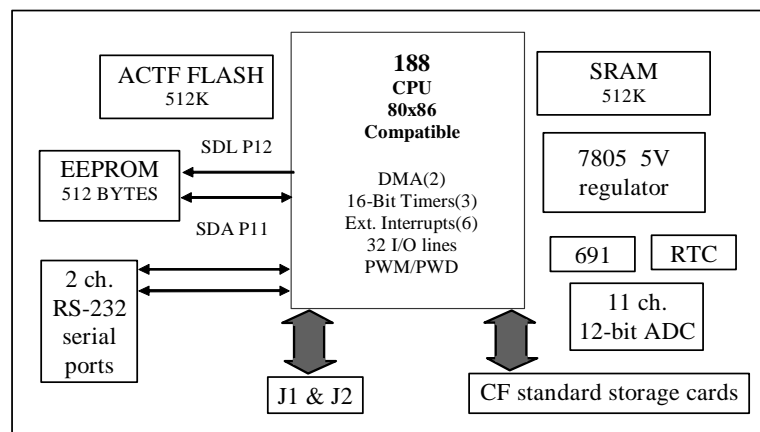


Figure 1.1 Functional block diagram of the FlashCore

Measuring 2.1 by 2.35 inches, the *FlashCore* offers a complete C/C++ programmable computer system with a 16-bit high performance 188 CPU, and operates with a 40 MHz system clock with zero-wait-state. In addition, a 512-byte serial EEPROM is included on-board. Optional features include up to 512K battery-backed SRAM, 11 channel 12-bit ADC, and an optional real-time clock which provides information on the year, month, date, hour, minute, and second. The *FlashCore* also includes an on-board 5-volt power regulator and RS232 drivers.

Two DMA-driven serial ports from the 188 CPU support high-speed, reliable serial communication at a rate of up to 115,200 baud while supporting 8-bit and 9-bit communication.

There are three 16-bit programmable timers/counters and a watchdog timer. Two timers can be used to count or time external events, at a rate of up to 10MHz, or to generate non-repetitive or variable-duty-cycle waveforms as PWM outputs. Pulse Width Demodulation (PWD), a distinctive feature, can be used to measure the width of a signal in both its high and low phases. It can be used in many applications, such as bar-code reading.

There are 32 user-programmable I/O pins on the 188 CPU, and six external interrupt inputs. A supervisor chip with power failure detection, a watchdog timer, an LED, and expansion ports are on-board.

1.2 Features on the FlashCore

Standard Features

- Dimensions: 2.1 x 2.35 inches
- Easy to program in Paradigm C/C++
- Power consumption: 160 mA at 5V
- Power saving mode: 30mA at 5V
- Power input: +9V to +12 V unregulated DC with on-board regulator
- 16-bit CPU, Intel 80x86 compatible
- High performance, zero-wait-state operation
- 512KB ACTF Flash
- 5V regulator and RS-232 drivers
- 2 high-speed PWM outputs and Pulse Width Demodulation
- 32 I/O lines from 188 CPU, 512-byte serial EEPROM
- 6 external interrupt inputs, 3 16-bit timer/counters
- 2 serial ports support 8-bit or 9-bit asynchronous communication
- Supervisor chip (691) for power failure, reset and watchdog
- 50-pin Compact Flash socket for Compact Flash cards of size 8MB-1GB

Optional Features (*surface-mounted components):

- 128KB, or 512KB SRAM*
- Real-time clock RTC72423*, lithium coin battery*
- 11 channel, 10KHz, 12-bit ADC*

1.3 Physical Description

The physical layout of the FlashCore is shown in Figure 1.2.

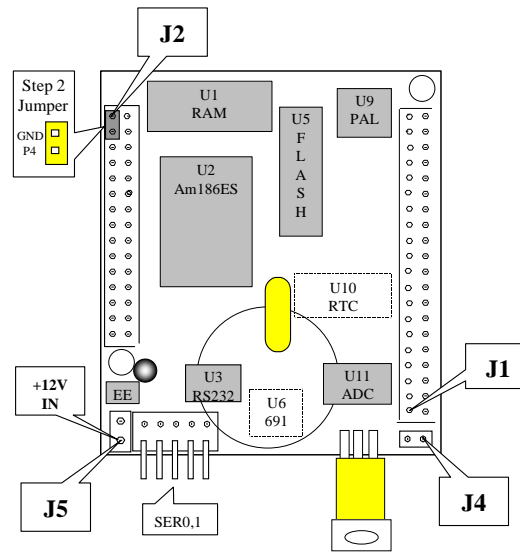
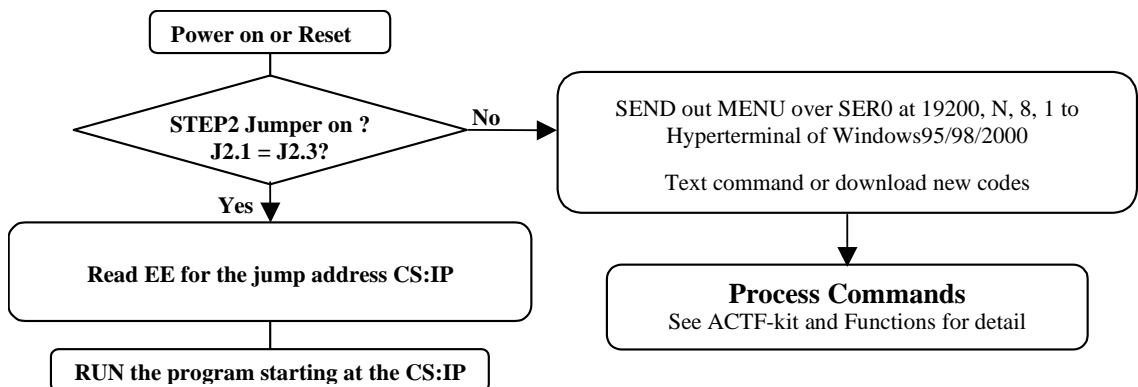


Figure 1.2 Physical layout of the FlashCore

1.4 FlashCore Programming Overview (not FC-0)

At the factory, an ACTF utility is loaded into the upper sector on the on-board flash. This ACTF utility is protected and executes at every power up. Upon power up, the ACTF will perform the process as described by flow chart below. The remainder of this section will be divided into parts: Prepare for Debug Mode (STEP 1), Debug Mode (STEP 1), Standalone Mode (STEP 2), and finally, Production (STEP 3). For your convenience, the preparation for debug mode is done at the factory, meaning you can begin at STEP 1: Debug Mode.



1.4.1 Prepare for Debug Mode (STEP 1):

To run the FC in Step 1, the debug mode, a debug kernel must be loaded into the on-board flash. This is done at the factory for your convenience. This debug kernel must be running to communicate with the Paradigm C/C++ programming environment. It resides in the on-board flash at address 0xE0000. To run the debug kernel and prepare for debug mode do the following:

1. Link the FlashCore to your PC and prepare a hyper terminal session. Configure the terminal to 19,200 Baud, 8 bits, No parity, and 1 stop. Connect to SER0 of the FlashCore.
2. Power on the FlashCore without the STEP 2 installed. The STEP 2 jumper is a red jumper installed on the J2 header pins 1 and 3.
3. At power up, you should see the ACTF Utility menu at your hyper terminal:

```
ACTF/ACTR Copyright(c) 1996 STE CA USA. All rights reserved.
```

```
>C C FUNCTIONS
>D Download an Intel Extend Hex file into SRAM
>G Goto and Run
>H HELP
>M MENU
>U Upload a block of Binary data
```

The “G” command allows you to jump to a location and immediately begin execution. It also sets the start-up jump address. Type “GE0000”, then <enter>. Your FlashCore will jump to that location in the flash and begin to run the debug kernel. The on-board LED will blink twice, then stay on. This indicates the FlashCore is correctly running the debug kernel.

4. Now install the STEP 2 jumper (red jumper installed at J2 pins 1 and 3).
5. Now at start up, the ACTF Utility will check if the STEP 2 jumper is installed. If the STEP 2 jumper is installed, the CPU will fetch the start up jump address (which we set in instruction 4 to point to the debug kernel, 0xE0000) and jump to that address for execution. Your FlashCore is now ready to communicate with the Paradigm C/C++ for Debug Mode. If the STEP 2 jumper is not detected, the ACTF Utility will send out it’s start up menu, and you will be back to instruction 3.

If when you jump to the debug kernel (by typing “GE0000”, then <enter> at the ACTF menu), if you do not see the on-board LED blink twice then stay on, the debug kernel has been erased. It must be loaded again to run STEP 1 and communicate with the Paradigm C/C++ software. See the section 1.5.1 for instructions on how to load the debug kernel.

1.4.2 STEP 1: Debug Mode

After completing the pervious section, your FlashCore is ready to communicate with the Paradigm C/C++ Environment and debug source code. Use samples provided in the c:\tern\186\samples\fc and c:\tern\186\samples\flashcore directories to generate source code. Debug your code as needed. You can then go to STEP 2: Standalone Mode.

1.4.3 STEP 2: Standalone Mode

Now that you have debugged your source code, you are ready to test it in standalone mode. To run standalone, do the following:

1. Remove the STEP 2 jumper. Prepare a hyper terminal session as described by section 1.4.1.

2. At power-on, you will see ACTF menu. (The key is that the STEP 2 jumper is not installed, so the CPU does not fetch the jump address).
3. You now want to jump to your program. In STEP 1 the Paradigm C/C++ environment downloads your program into the SRAM, starting at address 0x08000. We now want to use the same “G” command as before, but jump to your program, not the debug kernel. Type “G08000”, then <enter>. The CPU will then jump to your program in the SRAM for immediate execution. It will also set the start up jump address to 0x08000.
4. Re-install the STEP 2 jumper (J2 pins 1 and 3). Now at every power up, the ACTF utility will see the STEP 2 jumper and fetch the jump address, which now points to your program in the SRAM. Your program will now execute in standalone mode at every power up.
5. When finished with STEP 2: Standalone Mode, you can go back to STEP 1: Debug Mode by repeating instructions 1 & 2. Then use “GE0000”, then <enter> to jump back to the debug kernel. The FlashCore is now ready to communicate with the Paradigm C/C++ Environment.
6. This cycle between STEP 1 and STEP 2 can be done until your program is complete.

1.4.4 STEP 3: Production

This step only applies to those users who have purchased the full Development version of the Paradigm C/C++ Environment.

1. When you have finished development of your program, you are ready to use your source code to generate an Intel Extend HEX file, which can then be burned into the on-board flash for a production version of the FlashCore.
2. Inside Paradigm C/C++, change the config node of your target from “186.cfg” to “actf186.cfg”. This is done by right-mouse clicking on the config node and selecting “Edit Node Attributes”.
3. Open “actf186.cfg” for editing. Follow the instructions at the top of the config file. Save and close.
4. Right-mouse click on the “axe” node of your target and select “Target Expert”. Within the Target Expert window, change **PDREMOTE/ROM** to **No Target/ROM**.
5. Now right-mouse click on the “axe” node and select “Build Node”. You have now generated an Intel Extend Hex file. The name of the file will have the same name as your target, in the same working directory, but with the “.HEX” file extension.

For example, if the name of my target is “My_Program.axe”, then I will have created “My_Program.hex” in the same directory.
6. Go to Section 1.5.2 of this manual and follow the instruction for burning HEX files into the on-board flash.

1.5 Burning HEX Files into the On-board Flash

This section will cover two processes:

- (1) Burning the debug kernel into the flash to prepare for debug mode.

ERASING FLASH EEPROM AM29F040 SECTOR 6

AM29040 EE only takes INTEL EXTEND HEX file starts at 0x80000 !

Ready to receive Intel Extend HEX file at 19200 baud

7. At the terminal menu, select Transfers, then Send Text File. Go to c:\tern\186\rom and select "af_0_115.hex". This is the debug kernel. It will download into the flash starting at address 0xE0000.
8. When it finishes, the ACTF utility will reset and you will see the ACTF menu. Type "GE0000", to jump to and execute the debug kernel. The start up jump address will also be set to 0xE0000.
9. Install the STEP 2 jumper. At power up, your FlashCore will execute the debug kernel and be ready to communicate with Paradigm C/C++ for STEP 1: Debug Mode.

1.5.2 *Burning your application HEX file into the flash*

1. Follow instructions 1-6 of the above section, section 1.5.1. This will prepare the Flash for a HEX file.
2. At the Hyper terminal menu, select Transfers, then Send Text File. Go to the working directory of your project in Paradigm C/C++. Select your Intel Extend HEX file generated by the steps given in the last part of Section 1.4
3. When it finishes downloading, the ACTF utility will reset. Your application will have downloaded into the flash starting at address 0x80000 (not to be confused with 0x08000, the starting address of your program in STEP 2: Standalone Mode in the SRAM).
4. Now all that is needed is to set the jump address to 0x80000. Type "G80000", then <enter>. Your application will then execute out of the flash. The start up jump address is now set to 0x80000.
5. Install the STEP 2 jumper.
6. At every power-up, the CPU will jump to 0x8000 for immediate execution of your program. To get back to debug mode go to section 1.5.1.

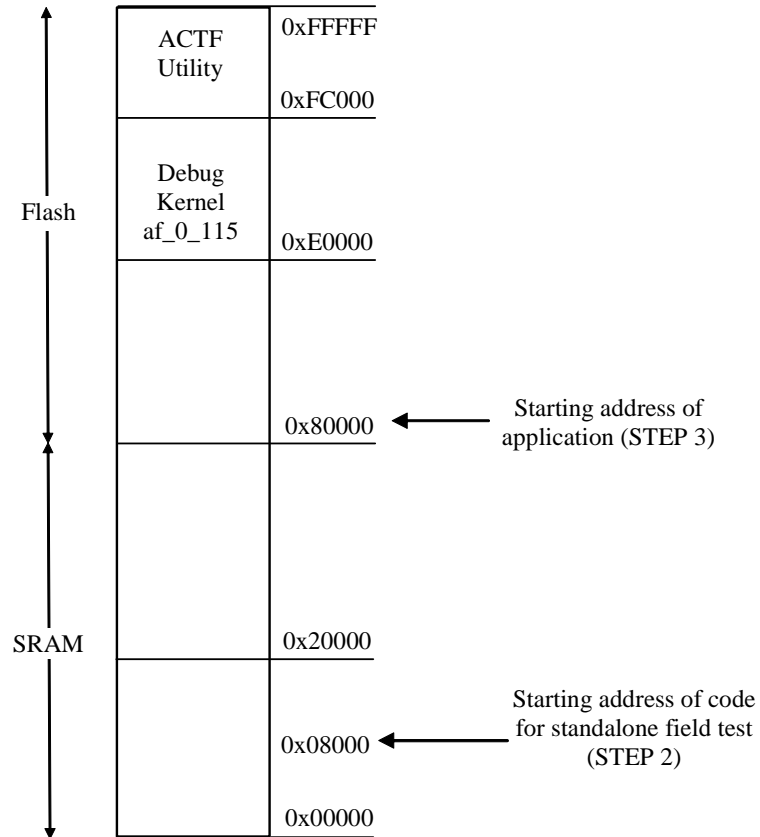
There is no ROM socket on the FC. The User's application program must reside in SRAM for debugging and reside in battery-backed SRAM for the standalone field test.

The on-board Flash 29F040B has 256K words of 16-bits each. It is divided into 8 sectors of 64KB. The top 16KB sector is pre-loaded with ACTF boot strip, and the sector starting 0xE0000 is for loading the remote debug kernel. When application is ready, "lo_ee512.hex" will erase debug kernel, leaving 7 sectors for application use.

The top 16KB ACTF boot strip is protected.

The utility HEX file, "lo_ee512.HEX" will automatically download into SRAM starting at 0x04000 with ACTF-PC HyperTerminal. Use the "D" command to download, and use the "G" command to run.

"lo_ee512.HEX" will erase the bottom seven sectors and load a "AF_0_115.HEX" or "AF_0_384.HEX" into the Flash starting at 0xE0000, and load your application HEX starting at 0x80000. Refer to the ACTF manual for information on how to change the downloading address of your application HEX.



For production, the user must produce a ACTF-downloadable HEX file for the application, based on the DV-P and ACTF Kit. The application HEX file can be loaded into the on-board Flash starting address at 0x80000. To properly generate your application HEX, you must change the config node of your target to “actf186.cfg”, which is found in the /TERN/186/config directory. Then right mouse click on the “.axe” node of your target and select “Target Expert”. This will allow you to change the “TargetConnection” from PDREMOTE/ROM to NoTarget/ROM. Then “Build node” will generate your application “.HEX” file.

The on-board EE must be modified with a “G80000” command while in the ACTF-PC-HyperTerminal Environment.

The “STEP2” jumper (J2 pins 1-3) must be installed for every production-version board.

In order to correctly download a program in STEP1 with Paradigm C/C++, the FC must meet these requirements:

- 1) AF_0_115.HEX must be pre-loaded into Flash starting address 0xE0000(done at factory by default).
- 2) The SRAM installed must be large enough to hold your program.
 - For a 128K SRAM, the physical address is 0x00000-0x01ffff
 - For a 512K SRAM, the physical address is 0x00000-0x07ffff
- 3) The on-board EE must have a correct jump address for the AF_0_115.HEX with starting address of 0xE0000.
- 4) The STEP2 jumper must be installed on J2 pins 1-3.

1.6 Minimum Requirements for FlashCore System Development

1.6.1 Minimum Hardware Requirements

PC or PC-compatible computer with serial COMx port that supports 115,200 baud

FlashCore controller with DEBUG kernel *AF_0_115*

PC-V25 serial cable (RS232; DB9 connector for PC COM port and IDC 2x5 connector for controller)

center negative wall transformer (+9V 500 mA)

1.6.2 Minimum Software Requirements

TERN EV-P/DV-P

PC software environment: Windows95/98/2000/NT/ME/XP

The C/C++ Evaluation Kit (EV-P) and C/C++ Development Kit (DV-P) are available from TERN. The EV-P Kit is a limited-functionality version of the DV-P Kit. With the EV-P Kit, you can program and debug the FlashCore in STEP 1 and STEP 2, but you cannot run STEP 3. In order to generate an application HEX file for downloading to Flash, and complete the project, you will need the Development Kit (DV-P).

Chapter 2: Installation

2.1 Software Installation

Please refer to the Technical manual for the “C/C++ Development Kit and Evaluation Kit for TERN Embedded Microcontrollers” for information on installing software.

The README.TXT file on the TERN EV-P/DV-P CD-ROM contains important information about the installation and evaluation of TERN controllers.

2.2 Hardware Installation

Overview (for FC)

Connect debug cable:

For debugging (Step One), place ICD connector on SER0 (J5) with red edge of cable at pin 1

Connect wall transformer:

Connect 9V wall transformer to power and plug into power jack (installs onto J5 pins 1 and 2)

Overview (for FC-0)

Connect FC-0 to host TERN controller (*A-Engine, i386-Engine, 586-Engine, ...*) via 20x2 J1 pin header. Make sure that J1 pin 1 of FC-0 matches J1 pin 1 of host.

Refer to manual of host controller for appropriate connections for power and RS-232 communication.

2.2.1 Connecting the FlashCore to the PC

FOR FLASHCORE:

The following picture (Figure 2.1) illustrates the connection between the FlashCore and the PC. The FlashCore is linked to the PC via serial cable.

The *AF_0_115.HEX* debug kernel communicates through SER0 by default. Install the 5x2 IDC connector to the SER0 header (J5). **IMPORTANT:** Note that the **red** side of the cable must point to pin 1 of the J5 header. Although pin 1 of J5 is for +12V In, it is still important for the **red** side of the cable to point to pin 1. The 5x2 IDC will just not have the connection at the red side of the connector. The DB9 connector should be connected to one of your PC's COM Ports (COM1 or COM2).

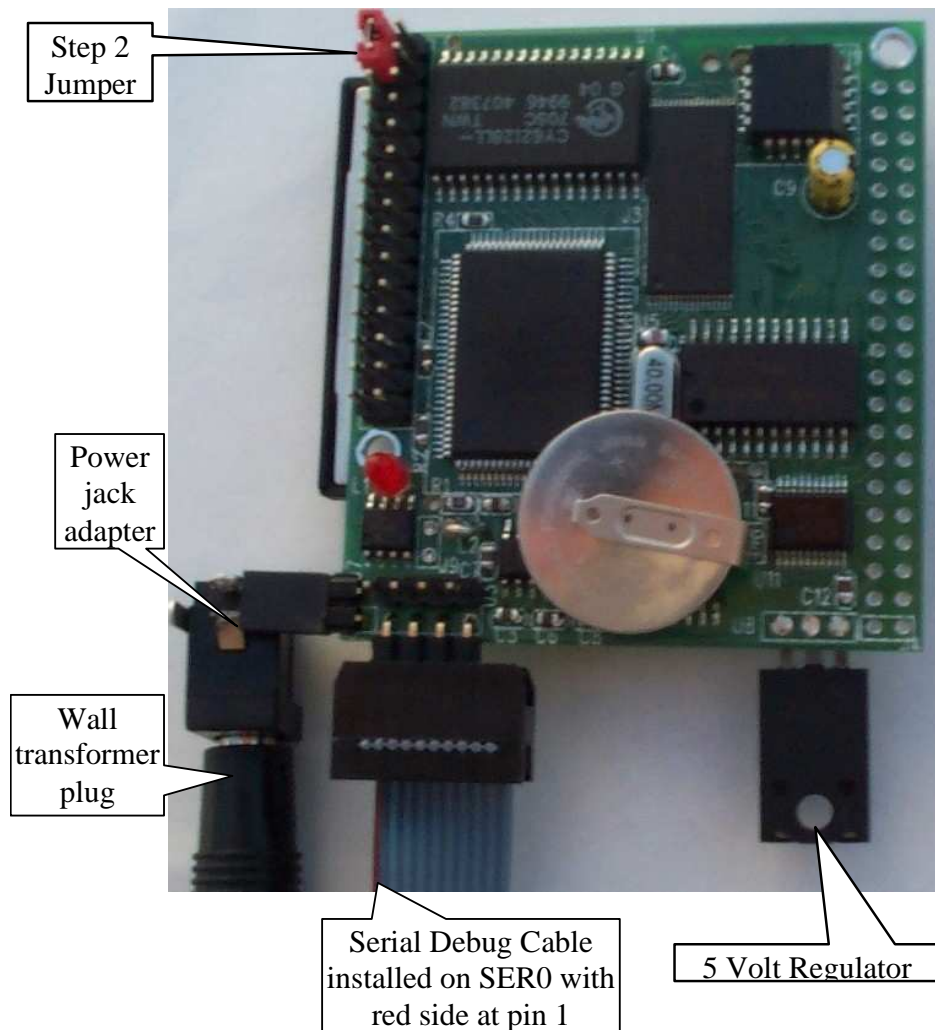


Figure 2.1 Connecting the FlashCore to the PC

FOR FLASHCORE-0:

Figure 2.2 gives an example picture of the FlashCore-0 that has been mounted onto a host controller. Again, this is done via the 20x2 pin header J1. It is important to note the J1 pin 1 of the FC-0 and J1 pin 1 of the host must align.

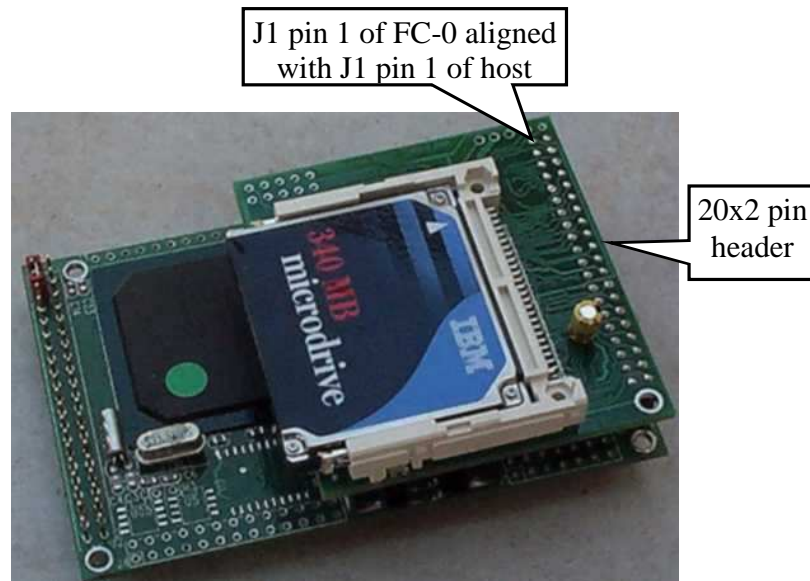


Figure 2.2 Connecting the FlashCore-0 to host controller via 20x2 pin J1

2.2.2 Powering-on the FlashCore

Connect the wall transformer +9V DC output to the FlashCore power jack adapter which then connects to the FlashCore via J5 pin 1(+12V In) and J5 pin 2 (GND). See Figure 2.1.

For the FC-0, the host controller will supply power to the FC-0. Refer the manual of your host controller for correct power connections.

Chapter 3: Hardware

3.1 188 CPU - Introduction

The 188 CPU is based on industry-standard x86 architecture. The 188 CPU controllers are higher-performance, more integrated versions of the 80C188 microprocessors. In addition, the 188 CPU has new peripherals. The on-chip system interface logic can minimize total system cost. The 188 CPU has two asynchronous serial ports, 32 PIOs, a watchdog timer, additional interrupt pins, a pulse width demodulation option, DMA to and from serial ports, a 16-bit reset configuration register, and enhanced chip-select functionality.

3.2 188 CPU – Features

3.2.1 Clock

Due to its integrated clock generation circuitry, the 188 CPU microcontroller allows the use of a times-one crystal frequency. The design achieves 40 MHz CPU operation, while using a 40 MHz crystal.

3.2.2 External Interrupts

There are six external interrupts: INT0-INT4 and INT6. All six interrupts are active high, but since they are not pulled down, it is recommended to add pull down resistors to any external interrupts used so as not to have falsely generated interrupts

- INT0, J2 pin 14
- INT1, J2 pin 11
- INT2, J2 pin 12
- INT3, J2 pin 9
- INT4 = P30, J2 pin 10
- INT6 = P13, J2 pin 21

These external interrupt inputs require a raising edge (LOW-to-HIGH) to generate an interrupt.

The FlashCore uses vector interrupt functions to respond to external interrupts. Refer to the 188 CPU User's manual for information about interrupt vectors.

3.2.3 Asynchronous Serial Ports

The 188 CPU has two asynchronous serial channels: SER0 and SER1. Each asynchronous serial port supports the following:

- Full-duplex operation
- 7-bit, 8-bit and 9-bit data transfers
- Odd, even and no parity
- One stop bit
- Error detection
- Hardware flow control
- DMA transfers to and from serial ports
- Transmit and receive interrupts for each port
- Multidrop 9-bit protocol support
- Maximum baud rate of 1/16 of the CPU clock
- Independent baud rate generators

The software drivers for each serial port implement a ring-buffered DMA receiving and ring-buffered interrupt transmitting arrangement. See the sample files: `s1_echo.c` and `s0_echo.c` in the `tern\186\samples\ae` directory.

3.2.4 Timer Control Unit

The timer/counter unit has three 16-bit programmable timers: Timer0, Timer1, and Timer2.

Timer0 and Timer1 are connected to three external pins:

Timer0 output = P10 = J2 pin 22
 Timer0 input = P11 = NOT ROUTED TO EXTERNAL PIN
 Timer1 output = P1 = J2 pin 25
 Timer1 input = P0 = J2 pin 19

These two timers can be used to count or time external events, or they can generate non-repetitive or variable-duty-cycle waveforms.

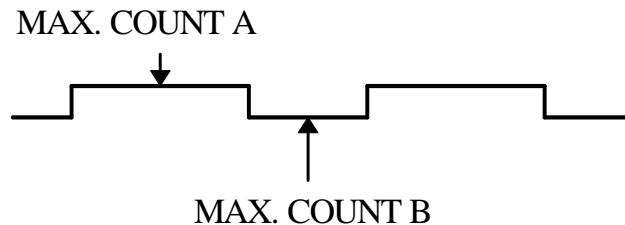
Timer2 is not connected to any external pin. It can be used as an internal timer for real-time coding or time-delay applications. It can also prescale Timer0 and Timer1 or be used as a DMA request source.

The maximum rate at which each timer can operate is 10 MHz, since each timer is serviced on every fourth clock cycle. Timer output takes up to six clock cycles to respond to clock or gate events. See sample programs `timer0.c` and `timer1.c` in `186\samples\ae`.

3.2.5 PWM outputs and PWD

The Timer0 and Timer1 outputs can also be used to generate non-repetitive or variable-duty-cycle waveforms. The timer output takes up to 6 clock cycles to respond to the clock input. Thus the minimum timer output cycle is $25 \text{ ns} \times 6 = 150 \text{ ns}$ (at 40 MHz).

Each timer has a maximum count register that defines the maximum value the timer will reach. Both Timer0 and Timer1 have a secondary maximum count register for variable duty cycle output. Using both the primary and secondary maximum count registers lets the timer alternate between two maximum values.



Pulse Width Demodulation can be used to measure the input signal's high and low phases on the INT2=J2 pin 12.

3.2.6 Power-save Mode

The FlashCore is an ideal core module for low power consumption applications. The power-save mode of the 188 CPU reduces power consumption and heat dissipation, thereby extending battery life in portable systems. In power-save mode, operation of the CPU and internal peripherals continues at a slower clock frequency. When an interrupt occurs, it automatically returns to its normal operating frequency.

3.3 188 CPU PIO lines

The 188 CPU has 32 pins available as user-programmable I/O lines. Each of these pins can be used as a user-programmable input or output signal, if the normal shared function is not needed. A PIO line can be

configured to operate as an input or output with or without a weak pull-up or pull-down, or as an open-drain output. A pins behavior, either pull-up or pull-down, is pre-determined and shown below.

After power-on/reset, PIO pins default to various configurations. The initialization routine provided by TERN libraries reconfigures some of these pins as needed for specific on-board usage as well. These configurations, as well as the processor-internal peripheral usage configurations, are listed below in Table 3.1.

<i>PIO</i>	<i>Function</i>	<i>Power-On/Reset status</i>	<i>FC Pin No.</i>	<i>FlashCore Initial</i>
P0	Timer1 in	Input with pull-up	J2 pin 19	Input with pull-up
P1	Timer1 out	Input with pull-down	J2 pin 25	CLK_1
P2	/PCS6/A2	Input with pull-up	J1 pin 36	RTC select
P3	/PCS5/A1	Input with pull-up	J2 pin 13/J1.38	Used by RTC
P4	DT/R	Normal	J2 pin 3	Input with pull-up Used by Step 2
P5	/DEN/DS	Normal	J2 pin 8	Input with pull-up
P6	SRDY	Normal	J2 pin 6	Input with pull-down
P7	A17	Normal	J3.3	A17
P8	A18	Normal	None	A18
P9	A19	Normal	J2 pin 24	Input with pull-up
P10	Timer0 out	Input with pull-down	J2 pin 22	Input with pull-down
P11	Timer0 in	Input with pull-up	None	Input with pull-up
P12	DRQ0/INT5	Input with pull-up	J9 pin 2	Output for LED/EE/HWD
P13	DRQ1/INT6	Input with pull-up	J2 pin 21	Input with pull-up
P14	/MCS0	Input with pull-up	J2 pin 4	Input with pull-up
P15	/MCS1	Input with pull-up	J2 pin 7	Input with pull-up
P16	/PCS0	Input with pull-up	U9 pin 1	CLK for PAL
P17	/PCS1	Input with pull-up	J2 pin 18	Open for user
P18	CTS1/PCS2	Input with pull-up	J2 pin 15	Input with pull-up
P19	RTS1/PCS3	Input with pull-up	J2 pin 16	Input with pull-up
P20	RTS0	Input with pull-up	J2 pin 23	Input with pull-up
P21	CTS0	Input with pull-up	J2 pin 26	Input with pull-up
P22	TxD0	Input with pull-up	U3 pin 10	TxD0
P23	RxD0	Input with pull-up	U3 pin 9	RxD0
P24	/MCS2	Input with pull-up	J2 pin 17	Input with pull-up
P25	/MCS3	Input with pull-up	J2 pin 20	Input with pull-up
P26	UZI	Input with pull-up	J2 pin 27	Open for user
P27	TxD1	Input with pull-up	U3 pin 11	TxD1
P28	RxD1	Input with pull-up	U3 pin 12	RxD1
P29	/CLKDIV2	Input with pull-up	J2 pin 28	Input with pullup*
P30	INT4	Input with pull-up	J2 pin 10	Input with pull-up
P31	INT2	Input with pull-up	J2 pin 12	Input with pullup

* Note: P26, P29 must NOT be forced low during power-on or reset.

Table 3.1 I/O pin default configuration after power-on or reset

Three external interrupt lines are not shared with PIO pins:

INT0 = J2 pin 14

INT1 = J2 pin 11

INT3 = J2 pin 9

The 32 PIO lines, P0-P31, are configurable via two 16-bit registers, PIOMODE and PIODIRECTION registers. The settings are listed as follows:

<i>MODE</i>	<i>PIOMODE reg.</i>	<i>PIODIRECTION reg.</i>	<i>PIN FUNCTION</i>
0	0	0	Normal operation
1	0	1	INPUT with pull-up/pull-down
2	1	0	OUTPUT
3	1	1	INPUT without pull-up/pull-down

FlashCore initialization on PIO pins in `ae_init()` is listed below:

```
output(0xff78,0xe73c); // PDIR1, TxD0, RxD0, TxD1, RxD1, P16=PCS0, P17=PCS1
output(0xff76,0x0000); // PIOM1
output(0xff72,0xec7b); // PDIR0, P12,A19,A18,A17,P2=PCS6=RTC
output(0xff70,0x1000); // PIOM0, P12=LED
```

The C function in the library `ae_lib` can be used to initialize PIO pins.

```
void pio_init(char bit, char mode);
```

Where bit = 0-31 and mode=0-3, see the table above.

Example:

```
pio_init(12, 2); will set P12 as output
pio_init(1, 0); will set P1 as Timer1 output
```

```
void pio_wr(char bit, char dat);
```

```
pio_wr(12,1); set P12 pin high, if P12 is in output mode
pio_wr(12,0); set P12 pin low, if P12 is in output mode
```

```
unsigned int pio_rd(char port);
```

```
pio_rd (0); return 16-bit status of P0-P15, if corresponding pins is in input mode,
pio_rd (1); return 16-bit status of P16-P31, if corresponding pins is in input mode,
```

Some of the I/O lines are used by the FlashCore system for on-board components (Table 3.2). We suggest that you not use these lines unless you are sure that you are not interfering with the operation of such components (i.e., if the component is not installed).

Signal	Pin	Function
P2	/PCS6	U4 RTC72423 chip select at base I/O address 0x0600
P4	/DT	Step Two jumper
P11	Timer0 input	U7 24C04 EE data input The EE data output can be tri-state, while disabled
P12	DRQ0/INT5	Output for LED or U7 serial EE clock or Hit watchdog
P16	/PCS0	U9 PAL clock at base I/O address 0x0000
P22	TxD0	Default SER0 debug
P23	RxD0	Default SER0 debug

Table 3.2 I/O lines used for on-board components

3.4 I/O Mapped Devices

3.4.1 I/O Space

External I/O devices can use I/O mapping for access. You can access such I/O devices with `inportb(port)` or `outportb(port,dat)`. These functions will transfer one byte or word of data to the specified I/O address. The external I/O space is 64K, ranging from 0x0000 to 0xffff.

The default I/O access time is 15 wait states. You may use the function `void io_wait(char wait)` to define the I/O wait states from 0-15. The system clock is 25 ns, giving a clock speed of 40 MHz. Details regarding this can be found in the Software chapter, and in the 188 CPU User's Manual. Slower components, such as most LCD interfaces, might find the maximum programmable wait state of 15 cycles still insufficient. Due to the high bus speed of the system, some components need to be attached to I/O pins directly.

For details regarding the chip select unit, please see Chapter 5 of the 188 CPU User's Manual.

The table below shows more information about I/O mapping.

I/O space	Select	Usage	Location
0x0000-0x00ff	/PCS0	PAL	U9 pin 1
0x0100-0x01ff	/PCS1	USER	J2 pin 18=P17
0x0200-0x02ff	/PCS2	USER	J2 pin 15=CTS1
0x0300-0x03ff	/PCS3	USER	J2 pin 16=RTS1
0x0400-0x04ff	/PCS4	Reserved	
0x0500-0x05ff	/PCS5	USER	J2 pin 16=P3
0x0600-0x06ff	/PCS6	RTC 72423	U10 pin 2=P2

3.4.2 Real-time Clock RTC72423

If installed, a real-time clock RTC72423 (EPSON, U4) is mapped in the I/O address space 0x0600. It must be backed up with a lithium coin battery. The RTC is accessed via software drivers `rtc_init()` or `rtc_rd()` (see Software chapter for details).

3.5 Other Devices

A number of other devices are also available on the FlashCore. Some of these are optional, and might not be installed on the particular controller you are using. For a discussion regarding the software interface for these components, please see the Software chapter.

3.5.1 On-board Supervisor with Watchdog Timer

The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the FlashCore has several functions: watchdog timer, battery backup, power-on-reset delay, power-supply monitoring, and power-failure warning. These will significantly improve system reliability.

Watchdog Timer

The watchdog timer is activated by setting a jumper on J9 of the FlashCore (see Figure 3.1). The watchdog timer provides a means of verifying proper software execution. In the user's application program, calls to the function `hitwd()` (a routine that toggles the P12=HWD pin of the MAX691) should be arranged such that the HWD pin is accessed at least once every 1.6 seconds. If the J9 jumper is on and the HWD pin is not accessed within this time-out period, the watchdog timer pulls the WDO pin low, which asserts /RESET. This automatic assertion of /RESET may recover the application program if something is wrong. After the FlashCore is reset, WDO remains low until a transition occurs at the WDI pin of the MAX691. When controllers are shipped from the factory the J9 jumper is off, which disables the watchdog timer.

In addition, the 188 CPU has an internal watchdog timer. This is disabled by default with `ae_init()`.

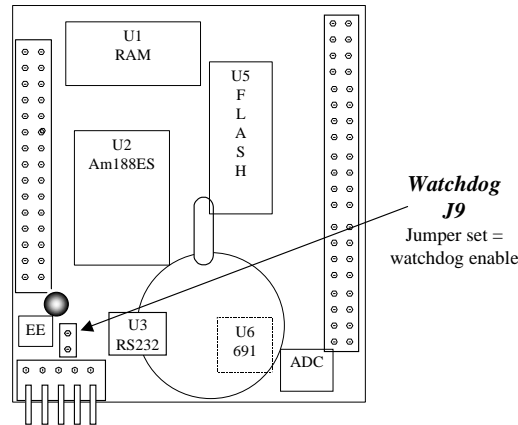


Figure 3.1 Location of watchdog timer enable jumper

Battery Backup Protection

The backup battery protection protects data stored in the SRAM and RTC. The battery-switch-over circuit compares VCC to VBAT (+3 V lithium battery positive pin), and connects whichever is higher to the VRAM (power for SRAM and RTC). Thus, the SRAM and the real-time clock RTC72423 are backed up. In normal use, the lithium battery should last about 3-5 years without external power being supplied. When the external power is on, the battery-switch-over circuit will select the VCC to connect to the VRAM.

3.5.2 12-bit ADC (LTC2543)

The TLC2543 is a 12-bit, switched-capacitor, successive-approximation, 11 channel, serial interface, analog-to-digital converter. Three PAL output lines are used to handle the ADC, with /CS=/AD; CLK=CK; and DIN=DIN.

The ADC digital data output communicates with a host through a serial tri-state output (DOUT). If /AD=/CS is low, the TLC2543 will have output on DOUT. If /AD=/CS is high, the TLC2543 is disabled and DOUT is free. The TLC2543 has an on-chip 14-channel multiplexer that can select any one of 11 inputs or any one of three internal self-test voltages. The sample-and-hold function is automatic. At the end of conversion, the end-of-conversion (EOC) output is not connected, although it goes high to indicate that conversion is complete.

TLC2543 features differential high-impedance inputs that facilitate ratiometric conversion, scaling, and isolation of analog circuitry from logic and supply noise. A switched-capacitor design allows low-error conversion over the full operating temperature range. The analog input signal source impedance should be less than 50 ohms and capable of slewing the analog input voltage into a 60 pf capacitor.

A reference voltage less than VCC (+5V) can be provided for the TLC2543 if additional precision is required. A voltage above 2.5V and less than +5V can be used for this purpose, and can be connected to the REF pin (J1.3)

The CK signal to the ADC is toggled through an output pin from the on-board PAL, and serial access allows a conversion rate of up to approximately 10 KHz.

In order to operate the TLC2543, five I/O lines are used, as listed below:

/CS	Chip select = /AD (U9.13) , high to low transition enables DOUT, DIN and CK. Low to high transition disables DOUT, DIN and CK.
-----	---

DIN	PAL pin 17, serial data input
DOUT	PAL pin 16, serial data output
EOC	Not Connected, End of Conversion, high indicates conversion complete and data is ready
CLK	clock = PAL pin 18
REF+	Upper reference voltage (normally VCC)
REF-	Lower reference voltage (normally GND)
VCC	Power supply, +5 V input
GND	Ground

The analog inputs AD0 and AD1 are available at J4. AD2-AD10 and REF are available at J1. Refer to Figure 3.3 for detailed picture.

3.5.3 EEPROM

A serial EEPROM of 512 bytes (24C04) is installed in U7. The FlashCore uses the P12=SCL (serial clock) and P11=SDA (serial data) to interface with the EEPROM. The EEPROM can be used to store important data, such as a node address, calibration coefficients, and configuration codes. It has typically 1,000,000 erase/write cycles. The data retention is more than 40 years. EEPROM can be read and written by simply calling functions `ee_rd()` and `ee_wr()`.

3.6 Headers and Connectors

3.6.1 Expansion Headers

The FlashCore has one 15x2 and one 20x2 pin header for expansion. Most signals are directly routed to the 188 CPU processor. These signals are 5V only, and any out-of-range voltages will most likely damage the board.

Table 3.3 Signals for J1 (20x2) and J2 (15x2) expansion ports

Signal definitions for J1:

VCC	+5V power supply
GND	Ground
D0-D7	188 CPU 8-bit external data lines
A0-A7	188 CPU address lines
/WR	188 CPU pin 5
/RD	188 CPU pin 6
REF	ADC reference voltage
AD2-10	ADC input pins

Signal definitions for J2:

VCC	+5V power supply, < 200 mA
GND	ground
Pxx	188 CPU PIO pins
/CTS0	188 CPU pin 100, Clear-to-Send signal for SER0
/CTS1	188 CPU pin 63, Clear-to-Send signal for SER1
/RTS0	188 CPU pin 3, Request-to-Send signal for SER0
/RTS1	188 CPU pin 62, Request-to-Send signal for SER1
INT0-4	Interrupt inputs

Signal definitions for J4:

AD0, AD1	Inputs for ADC
----------	----------------

<i>J5 Signal</i>			
+12VI	1	2	GND
/TXD0	3	4	/TXD1
/RXD0	5	6	/RXD1
	7	8	
GND	9	10	GND

Figure 3.2 Signals for J5 (5x2)

3.6.2 Jumpers

The following is a list of jumpers and connectors on the FlashCore.

Name	Size	Function	Possible Configuration
J1	20x2	Main expansion port, A0-A7, D0-D7, /WR, /RD, AD2-AD10	
J2	15x2	Main expansion port	Step 2 Jumper -> J2.1 = J2.3
J4	2x1	AD0 and AD1	
J5	5x2	SER0/SER1 connector, +12V In	Pins 1,2 for +12V In Pins 3,5,9 for SER0 Pins 4,6,10 for SER1
J9	2x1	Watchdog timer	Enabled if Jumper is on Disabled if jumper is off

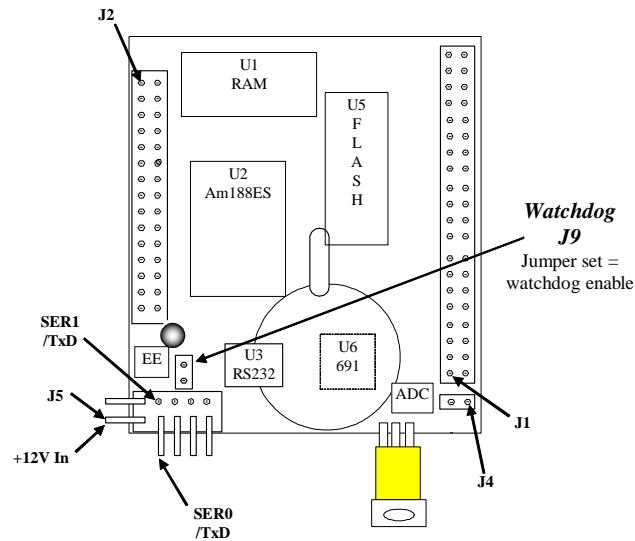


Figure 3.3 Locations of jumpers and connectors on the FlashCore (component view)

3.7 FC-0 Hardware

The FC-0 includes a 50-pin CF receptacle, 20x2 expansion header (J1), and a PAL.

The 50-pin CF receptacle provides a simple interface for 50-pin CompactFlash (CF) Cards. These CF cards, range in size from 8MB-1GB Flash cards.

The 20x2 pin header supplies the an 8-bit address bus and 8-bit data bus for expansion onto other TERN controllers. The pin configuration is as follows:

VCC	1	2	GND
	3	4	
	5	6	
	7	8	D0
	9	10	D1
	11	12	D2
	13	14	D3
/RST	15	16	D4
RST	17	18	D5
P16	19	20	D6
	21	22	D7
	23	24	GND
	25	26	A7
	27	28	A6
/WR	29	30	A5
/RD	31	32	A4
	33	34	A3
	35	36	A2
	37	38	A1
	39	40	A0

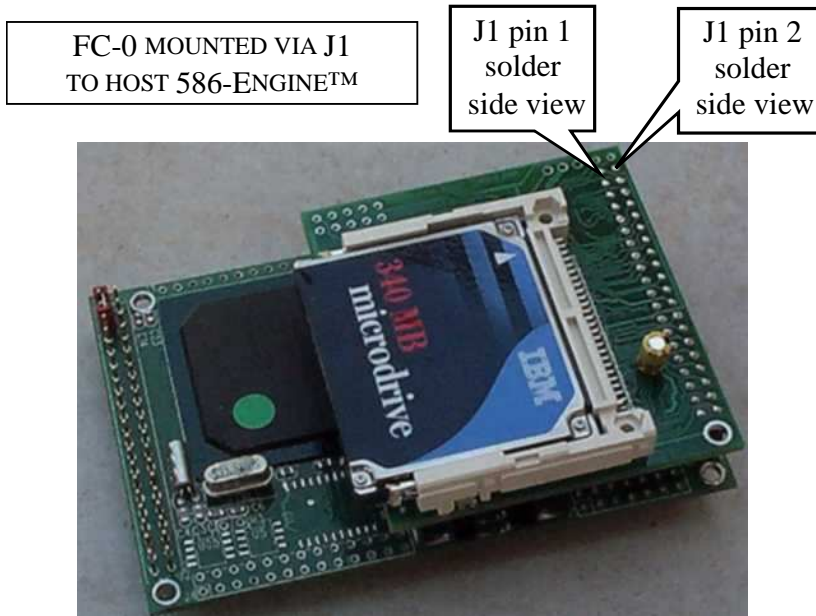


Figure 3.4 Orientation of J1 header from *solder* side view

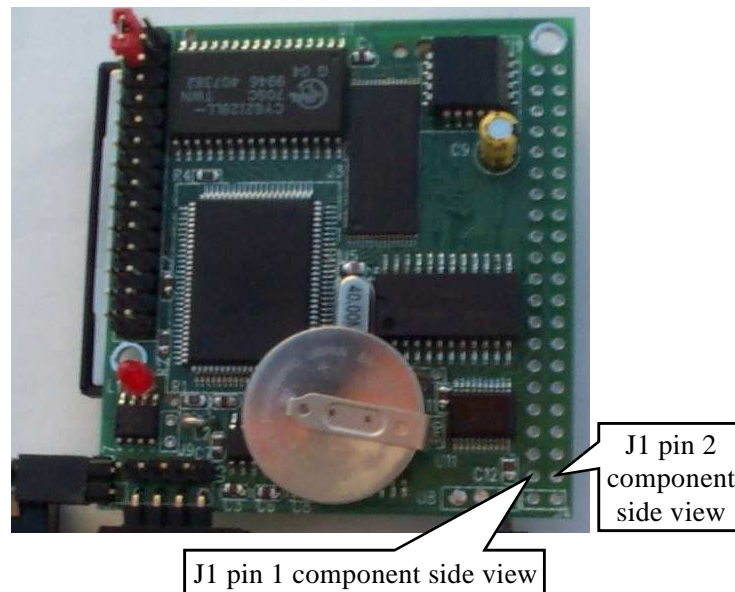


Figure 3.5 Orientation of J1 header from *component* side view

In order to select the FC-0, the following equation must be asserted:

$$/FC \text{ (chip select)} = /P16 * A7 * A6 * A5 * /A4 * /A3 * /A2 * /A1 * /A0.$$

This can be accomplished using any I/O lines available on the driving system, but the address and data bus are still needed. For ease of use, TERN recommends that only TERN engine cards are used to drive the FC-0.

Chapter 4: Software

Please refer to the Technical Manual of the “C/C++ Development Kit for TERN 16-bit Embedded Microcontrollers” for details on debugging and programming tools.

For details regarding software function prototypes and sample files demonstrating their use, please refer to Appendix G, “Software Glossary” of the technical manual for the AE&AEP in \tern_docs\manuals\ from the root directory of your CD.

Guidelines, awareness, and problems in an interrupt driven environment

Although the C/C++ Development Kit provides a simple, low cost solution to application engineers, some guidelines must be followed. If they are not followed, you may experience system crashes, PC hang-ups, and other problems.

The debugging of interrupt handlers with the Remote Debugger can be a challenge. It is possible to debug an interrupt handler, but there is a risk of experiencing problems. Most problems occur in multi-interrupt-driven situations. Because the remote kernel running on the controller is interrupt-driven, it demands interrupt services from the CPU. If an application program enables interrupt and occupies the interrupt controller for longer than the remote debugger can accept, the debugger will time-out. As a result, your PC may hang-up. In extreme cases, a power reset may be required to restart your PC.

For your reference, be aware that our system is remote kernel interrupt-driven for debugging.

The run-time environment on TERN controllers consists of an I/O address space and a memory address space. I/O address space ranges from **0x0000** to **0xffff**, or 64 KB. Memory address space ranges from **0x00000** to **0xfffff** in real-mode, or 1 MB. These are accessed differently, and not all addresses can be translated and handled correctly by hardware. I/O and memory mappings are done in software to define how translations are implemented by the hardware. Implicit accesses to I/O and memory address space occur throughout your program from TERN libraries as well as simple memory accesses to either code or global and stack data. You can, however, explicitly access any address in I/O or memory space, and you will probably need to do so in order to access processor registers and on-board peripheral components (which often reside in I/O space) or non-mapped memory.

This is done with four different sets of similar functions, described below.

poke/pokeb

Arguments: unsigned int segment, unsigned int offset, unsigned int/unsigned char data

Return value: none

These standard C functions are used to place specified data at any memory space location. The **segment** argument is left shifted by four and added to the **offset** argument to indicate the 20-bit address within memory space. **poke** is used for writing 16 bits at a time, and **pokeb** is used for writing 8 bits.

The process of placing data into memory space means that the appropriate address and data are placed on the address and data-bus, and any memory-space mappings in place for this particular range of memory will be used to activate appropriate chip-select lines and the corresponding hardware component responsible for handling this data.

peek/peekb

Arguments: unsigned int segment, unsigned int offset

Return value: unsigned int/unsigned char data

These functions retrieve the data for a specified address in memory space. Once again, the **segment** address is shifted left by four bits and added to the **offset** to find the 20-bit address. This address is then output over the address bus, and the hardware component mapped to that address should return either an 8-bit or 16-bit value over the data bus. If there is no component mapped to that address, this function will return random garbage values every time you try to peek into that address.

outport/outportb

Arguments: unsigned int address, unsigned int/unsigned char data

Return value: none

This function is used to place the **data** into the appropriate **address** in I/O space. It is used most often when working with processor registers that are mapped into I/O space and must be accessed using either one of these functions. This is also the function used in most cases when dealing with user-configured peripheral components.

When dealing with processor registers, be sure to use the correct function. Use **outport** if you are dealing with a 16-bit register.

inport/inportb

Arguments: unsigned int address

Return value: unsigned int/unsigned char data

This function can be used to retrieve data from components in I/O space. You will find that most hardware options added to TERN controllers are mapped into I/O space, since memory space is valuable and is reserved for uses related to the code and data. Using I/O mappings, the address is output over the address bus, and the returned 16 or 8-bit value is the return value.

For further discussion of I/O and memory mappings, please refer to the Hardware chapter of this technical manual.

4.1 AE.LIB

AE.LIB is a C library for basic FlashCore operations. It includes the following modules: AE.OBJ, SER0.OBJ, SER1.OBJ, SCC.OBJ, and AEEE.OBJ. You need to link AE.LIB in your applications and include the corresponding header files. The following is a list of the header files:

Include-file name	Description
AE.H	PPI, timer/counter, ADC, DAC, RTC, Watchdog,
SER0.H	Internal serial port 0
SER1.H	Internal serial port 1
AEEE.H	on-board EEPROM

4.2 Functions in AE.OBJ

4.2.1 FlashCore Initialization

ae_init

This function should be called at the beginning of every program running on FlashCore core controllers. It provides default initialization and configuration of the various I/O pins, interrupt vectors, sets up expanded DOS I/O, and provides other processor-specific updates needed at the beginning of every program.

There are certain default pin modes and interrupt settings you might wish to change. With that in mind, the basic effects of `ae_init` are described below. For details regarding register use, you will want to refer to the AMD Am188ES Microcontroller User's manual.

Initialize the upper chip select to support the default ROM. The CPU registers are configured such that:

Address space for the ROM is from 0x80000-0xffff (to map MemCard I/O window)

512K ROM Block size operation.

Three wait state operation (allowing it to support up to 120 ns ROMs). With 70 ns ROMs, this can actually be set to zero wait state if you require increased performance (at a risk of stability in noisy environments). For details, see the UMCS (Upper Memory Chip Select Register) reference in the processor User's manual.

```
outport(0xffa0, 0x80bf); // UMCS, 512K ROM, 0x80000-0xffff
```

Initialize LCS (*Lower Chip Select*) for use with the SRAM. It is configured so that:

Address space starts 0x00000, with a maximum of 512K RAM.

3 wait state operation. Reducing this value can improve performance.

Disables PSRAM, and disables need for external ready.

```
outport(0xffa2, 0x7fbf); // LMCS, base Mem address 0x0000
```

Initialize MMCS and MPCS so that **MCS0** and **PCS0-PCS6** (except for PCS4) are configured so:

MCS0 is mapped also to a 256K window at 0x80000. If used with MemCard, this chip select line is used for the I/O window.

Sets up **PCS5-6** lines as chip-select lines, with three wait state operation.

```
outport(0xffa8, 0xa0bf); // s8, 3 wait states
```

```
outport(0xffa6, 0x81ff); // CS0MSKH
```

Initialize PACS so that **PCS0-PCS3** are configured so that:

Sets up **PCS0-3** lines as chip-select lines, with fifteen wait state operation.

The chip select lines starts at I/O address 0x0000, with each successive chip select line addressed 0x100 higher in I/O space.

```
outport(0xffa4, 0x007f); // CS0MSKL, 512K, enable CS0 for RAM
```

Configure the two PIO ports for default operation. All pins are set up as default input, except for P12 (used for driving the LED), and peripheral function pins for SER0 and SER1, as well as chip selects for the PPI.

```
outport(0xff78, 0xe73c); // PDIR1, TxD0, RxD0, TxD1, RxD1,
                        // P16=PCS0, P17=PCS1=PPI
outport(0xff76, 0x0000); // PIOM1
outport(0xff72, 0xec7b); // PDIR0, P12,A19,A18,A17,P2=PCS6=RTC
outport(0xff70, 0x1000); // PIOM0, P12=LED
```

Configure the PPI 82C55 to all inputs, except for lines I20-23 which are used as output for the ADC.

You can reset these to inputs if not being used for that function.

```
outportb(0x0103, 0x9a); // all pins are input, I20-23 output
outportb(0x0100, 0);
outportb(0x0101, 0);
outportb(0x0102, 0x01); // I20=ADCS high
```

The chip select lines are by default set to 15 wait state. This makes it possible to interface with many slower external peripheral components. If you require faster I/O access, you can modify this number down as needed. Some TERN components, such as the Real-Time-Clock, might fail if the wait state is decreased too dramatically. A function is provided for this purpose.

void io_wait
Arguments: char wait

Return value: none.

This function sets the current wait state depending on the argument wait.

```
wait=0, wait states = 0, I/O enable for 100 ns
wait=1, wait states = 1, I/O enable for 100+25 ns
wait=2, wait states = 2, I/O enable for 100+50 ns
wait=3, wait states = 3, I/O enable for 100+75 ns
wait=4, wait states = 5, I/O enable for 100+125 ns
wait=5, wait states = 7, I/O enable for 100+175 ns
wait=6, wait states = 9, I/O enable for 100+225 ns
wait=7, wait states = 15, I/O enable for 100+375 ns
```

4.2.2 External Interrupt Initialization

There are up to six external interrupt sources on the FlashCore, consisting of maskable interrupt pins (INT0-INT4, INT6). There are also an additional eight internal interrupt sources not connected to the external pins, consisting of three timers, two DMA channels, both asynchronous serial ports, and the NMI from the watchdog timer. For a detailed discussion involving the ICUs, the user should refer to Chapter 7 of the AMD Am188ES Microcontroller User's Manual.

TERN provides functions to enable/disable all of the 6 external interrupts. The user can call any of the interrupt init functions listed below for this purpose. The first argument indicates whether the particular interrupt should be enabled, and the second is a function pointer to an appropriate interrupt service routine that should be used to handle the interrupt. The TERN libraries will set up the interrupt vectors correctly for the specified external interrupt line.

At the end of interrupt handlers, the appropriate in-service bit for the IR signal currently being handled must be cleared. This can be done using the Nonspecific EOI command. At initialization time, interrupt priority was placed in Fully Nested mode. This means the current highest priority interrupt will be handled first, and a higher priority interrupt will interrupt any current interrupt handlers. So, if the user chooses to clear the in-service bit for the interrupt currently being handled, the interrupt service routine just needs to issue the nonspecific EOI command to clear the current highest priority IR.

To send the nonspecific EOI command, you need to write the EOI register word with 0x8000.

```
outport(0xff22, 0x8000);
```

void intx_init
Arguments: unsigned char i, void interrupt far(* intx_isr) ()

Return value: none

These functions can be used to initialize any one of the external interrupt channels (for pin locations and other physical hardware details, see the Hardware chapter). The first argument *i* indicates whether this particular interrupt should be enabled or disabled. The second argument is a function pointer which will act as the interrupt service routine. The overhead on the interrupt service routine is approximately 20 μ s.

By default, the interrupts are all disabled after initialization. To disable them again, you can repeat the call but pass in 0 as the first argument.

The NMI (Non-Maskable Interrupt) is special in that it can not be masked (disabled). The default ISR will return on interrupt.

```

void int0_init( unsigned char i, void interrupt far(* int0_isr)() );
void int1_init( unsigned char i, void interrupt far(* int1_isr)() );
void int2_init( unsigned char i, void interrupt far(* int2_isr)() );
void int3_init( unsigned char i, void interrupt far(* int3_isr)() );
void int4_init( unsigned char i, void interrupt far(* int4_isr)() );
void int5_init( unsigned char i, void interrupt far(* int5_isr)() );
void int6_init( unsigned char i, void interrupt far(* int6_isr)() );
void int7_init( unsigned char i, void interrupt far(* int7_isr)() );
void int8_init( unsigned char i, void interrupt far(* int8_isr)() );
void int9_init( unsigned char i, void interrupt far(* int9_isr)() );
void nmi_init(void interrupt far (* nmi_isr)());

```

4.2.3 I/O Initialization

There are two ports of 16 I/O pins available on the FlashCore. Hardware details regarding these PIO lines can be found in the Hardware chapter.

There are several functions provided for access to the PIO lines. At the beginning of any application where you choose to use the PIO pins as input/output, you will probably need to initialize these pins in one of the four available modes. Before selecting pins for this purpose, make sure that the peripheral mode operation of the pin is not needed for a different use within the same application.

You should also confirm the PIO usage that is described above within `ae_init()`. During initialization, several lines are reserved for TERN usage and you should understand that these are not available for your application. There are several PIO lines that are used for other on-board purposes. These are all described in some detail in the Hardware chapter of this technical manual. For a detailed discussion toward the I/O ports, please refer to Chapter 11 of the AMD Am188ES User's Manual.

Please see the sample program `ae_pio.c` in `tern\186\samples\ae`. You will also find that these functions are used throughout TERN sample files, as most applications do find it necessary to re-configure the PIO lines.

The function `pio_wr` and `pio_rd` can be quite slow when accessing the PIO pins. Depending on the pin being used, it might require from 5-10 us. The maximum efficiency you can get from the PIO pins occur if you instead modify the PIO registers directly with an `outport` instruction. Performance in this case will be around 1-2 us to toggle any pin.

The data register is 0xff74 for PIO port 0, and 0xff7a for PIO port 1.

void pio_init

Arguments: char bit, char mode

Return value: none

bit refers to any one of the 32 PIO lines, 0-31.

mode refers to one of four modes of operation.

0, High-impedance Input operation

1, Open-drain output operation

2, output

3, peripheral mode

unsigned int pio_rd:

Arguments: char port

Return value: byte indicating PIO status

Each bit of the returned 16-bit value indicates the current I/O value for the PIO pins in the selected port.

void pio_wr:

Arguments: char bit, char dat

Return value: none

Writes the passed in dat value (either 1/0) to the selected PIO.

4.2.4 Timer Units

The three timers present on the FlashCore can be used for a variety of applications. All three timers run at $\frac{1}{4}$ of the processor clock rate, which determines the maximum resolution that can be obtained. Be aware that if you enter power save mode, that means the timers will operate at a reduced speed as well.

These timers are controlled and configured through a mode register which is specified using the software interfaces. The mode register is described in detail in chapter 8 of the AMD AM188ES User's Manual.

Pulse width demodulation is done by setting the PWD bit in the SYSCON register. Before doing this, you will want to specify your interrupt service routines, which are used whenever the incoming digital signal switches from high to low, and low to high.

The timers can be used to time execution of your user defined code by reading the timer values before and after execution of any piece of code. For a sample file demonstrating this application, see the sample file timer.c in the directory tern\186\samples\ae.

Two of the timers, Timer0 and Timer1 can be used to do pulse-width modulation with a variable duty cycle. These timers contain two max counters, where the output is high until the counter counts up to maxcount A before switching and counting up to maxcount B.

It is also possible to use the output of Timer2 to pre-scale one of the other timers, since 16-bit resolution at the maximum clock rate specified gives you only 150 Hz. Only by using Timer2 can you slow this down even further. Sample files demonstrating this are timer02.c and timer12.c in the FlashCore sample file directory.

The specific behavior that you might want to implement is described in detail in chapter 8 of the AMD AM188ES User's Manual.

void t0_init

void t1_init

Arguments: int tm, int ta, int tb, void interrupt far(*t_isr)()

Return values: none

Both of these timers have two maximum counters (MAXCOUNTA/B) available. These can all be specified using ta and tb. The argument tm is the value that you wish placed into the T0CON/T1CON mode registers for configuring the two timers.

The interrupt service routine t_isr specified here is called whenever the full count is reached, with other behavior possible depending on the value specified for the control register.

void t2_init

Arguments: int tm, int ta, void interrupt far(*t_isr)()

Return values: none.

Timer2 behaves like the other timers, except it only has one max counter available.

4.2.5 Other library functions

On-board supervisor MAX691 or LTC691

The watchdog timer offered by the MAX691 or LTC691 offers an excellent way to monitor improper program execution. If the watchdog timer (J9) is connected, the function hitwd() must be called every 1.6

seconds of program execution. If this is not executed because of a run-time error, such as an infinite loop or stalled interrupt service routine, a hardware reset will occur.

void hitwd

Arguments: none

Return value: none

Resets the supervisor timer for another 1.6 seconds.

void led

Arguments: int ledd

Return value: none

Turns the on-board LED on or off according to the value of **ledd**.

Real-Time Clock

The real-time clock can be used to keep track of real time. Backed up by a lithium-coin battery, the real time clock can be accessed and programmed using two interface functions.

There is a common data structure used to access and use both interfaces.

```
typedef struct{
    unsigned char sec1; One second digit.
    unsigned char sec10; Ten second digit.
    unsigned char min1; One minute digit.
    unsigned char min10; Ten minute digit.
    unsigned char hour1; One hour digit.
    unsigned char hour10; Ten hour digit.
    unsigned char day1; One day digit.
    unsigned char day10; Ten day digit.
    unsigned char mon1; One month digit.
    unsigned char mon10; Ten month digit.
    unsigned char year1; One year digit.
    unsigned char year10; Ten year digit.
    unsigned char wk; Day of the week.
} TIM;
```

int rtc_rd

Arguments: TIM *r

Return value: int error_code

This function places the current value of the real time clock within the argument **r** structure. The structure should be allocated by the user. This function returns 0 on success and returns 1 in case of error, such as the clock failing to respond.

Void rtc_init

Arguments: char* t

Return value: none

This function is used to initialize and set a value into the real-time clock. The argument **t** should be a null-terminated byte array that contains the new time value to be used.

The byte array should correspond to { *weekday, year10, year1, month10, month1, day10, day1, hour10, hour1, minute10, minute1, second10, second1, 0* }.

If, for example, the time to be initialized into the real time clock is June 5, 1998, Friday, 13:55:30, the byte array would be initialized to:

```
unsigned char t[14] = { 5, 9, 8, 0, 6, 0, 5, 1, 3, 5, 5, 3, 0 };
```

Delay

In many applications it becomes useful to pause before executing any further code. There are functions provided to make this process easy. For applications that require precision timing, you should use hardware timers provided on-board for this purpose.

void delay0

Arguments: unsigned int t

Return value: none

This function is just a simple software loop. The actual time that it waits depends on processor speed as well as interrupt latency. The code is functionally identical to:

```
while(t) { t--; }
```

Passing in a t value of 600 causes a delay of approximately 1 ms.

void delay_ms

Arguments: unsigned int

Return value: none

This function is similar to delay0, but the passed in argument is in units of milliseconds instead of loop iterations. Again, this function is highly dependent upon the processor speed.

unsigned int crc16

Arguments: unsigned char *wptr, unsigned int count

Return value: unsigned int value

This function returns a simple 16-bit CRC on a byte-array of **count** size pointed to by **wptr**.

void ae_reset

Arguments: none

Return value: none

This function is similar to a hardware reset, and can be used if your program needs to re-start the board for any reason. Depending on the current hardware configuration, this might either start executing code from the DEBUG ROM or from some other address.

4.3 Functions in SER0.OBJ/SER1.OBJ

The functions described in this section are prototyped in the header file ser0.h and ser1.h in the directory `tern\186\include`.

The internal asynchronous serial ports are functionally identical. SER0 is used by the debug kernel provided as part of the TERN EV-P/DV-P software kits for communication with the PC. As a result, you will not be able to debug code directly written for serial port 0.

Two asynchronous serial ports are integrated in the 188 CPU: SER0 and SER1. Both ports have baud rates based on the 40 MHz clock, and can operate at a maximum of 1/16 of that clock rate.

By default, SER0 is used by the DEBUG ROM for application download/debugging in STEP 1 and STEP 2. We will use SER1 as the example in the following discussion; any of the interface functions which are specific to SER1 can be easily changed into function calls for SER0. While selecting a serial port for use, please realize that some pins might be shared with other peripheral functions. This means that in certain limited cases, it might not be possible to use a certain serial port with other on-board controller functions. For details, you should see both chapter 10 of the Am188ES Microprocessor User's Manual and the schematic of the FlashCore provided at the end of this manual.

TERN interface functions make it possible to use one of a number of predetermined baud rates. These baud rates are achieved by specifying a divisor for 1/16 of the processor frequency.

The following table shows the function arguments that express each baud rate, to be used in TERN functions. These are based on a 40 MHz system clock;

Function Argument	Baud Rate
1	110
2	150
3	300
4	600
5	1200
6	2400
7	4800
8	9600
9	19,200 (default)
10	38,400
11	57,600
12	115,200
13	250,000
14	500,000
15	1,250,000

Table 4.1 Baud rate values

After initialization by calling `sl_init()`, SER1 is configured as a full-duplex serial port and is ready to transmit/receive serial data at one of the specified 15 baud rates.

An input buffer, `ser1_in_buf` (whose size is specified by the user), will automatically store the receiving serial data stream into the memory by DMA1 operation. In terms of receiving, there is no software overhead or interrupt latency for user application programs even at the highest baud rate. DMA transfer allows efficient handling of incoming data. The user only has to check the buffer status with `serhit1()` and take out the data from the buffer with `getser1()`, if any. The input buffer is used as a circular ring buffer, as shown in Figure 4.1. However, the transmit operation is interrupt-driven.

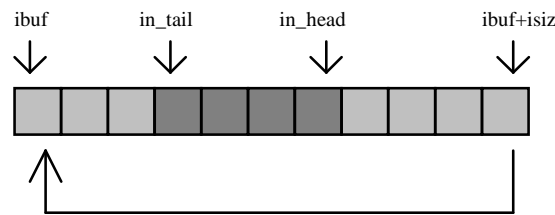


Figure 4.1 Circular ring input buffer

The input buffer (`ibuf`), buffer size (`isiz`), and baud rate (`baud`) are specified by the user with `sl_init()` with a default mode of 8-bit, 1 stop bit, no parity. After `sl_init()` you can set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port O/I Control Register (SPOCT/SPICT) if necessary, as described in chapter 10 of the Am188ES manual for asynchronous serial ports.

Due to the nature of high-speed baud rates and possible effects from the external environment, serial input data will automatically fill in the buffer circularly without stopping, regardless of overwrite. If the user does not take out the data from the ring buffer with `getser1()` before the ring buffer is full, new data will overwrite the old data without warning or control. Thus it is important to provide a sufficiently large buffer if large amounts of data are transferred. For example, if you are receiving data at 9600 baud, a 4 KB buffer will be able to store data for approximately four seconds.

However, it is always important to take out data early from the input buffer, before the ring buffer rolls over. You may designate a higher baud rate for transmitting data out and a slower baud rate for receiving data. This will give you more time to do other things, without overrunning the input buffer. You can use `serhit1()` to check the status of the input buffer and return the offset of the `in_head` pointer from the `in_tail` pointer. A return value of 0 indicates no data is available in the buffer.

You can use `getser1()` to get the serial input data byte by byte using FIFO from the buffer. The `in_tail` pointer will automatically increment after every `getser1()` call. It is not necessary to suspend external devices from sending in serial data with /RTS. Only a hardware reset or `sl_close()` can stop this receiving operation.

For transmission, you can use `putser1()` to send out a byte, or use `putsers1()` to transmit a character string. You can put data into the transmit ring buffer, `sl_out_buf`, at any time using this method. The transmit ring buffer address (`obuf`) and buffer length (`osiz`) are also specified at the time of initialization. The transmit interrupt service will check the availability of data in the transmit buffer. If there is no more data (the head and tail pointers are equal), it will disable the transmit interrupt. Otherwise, it will continue to take out the data from the out buffer, and transmit. After you call `putser1()` and transmit functions, you are free to do other tasks with no additional software overhead on the transmitting operation. It will automatically send out all the data you specify. After all data has been sent, it will clear the busy flag and be ready for the next transmission.

The sample program `ser1_0.c` demonstrates how a protocol translator works. It would receive an input HEX file from SER1 and translate every ':' character to '?' The translated HEX file is then transmitted out of SER0. This sample program can be found in `tern\186\samples\ae`.

Software Interface

Before using the serial ports, they must be initialized.

There is a data structure containing important serial port state information that is passed as argument to the TERN library interface functions. The COM structure should normally be manipulated only by TERN libraries. It is provided to make debugging of the serial communication ports more practical. Since it allows you to monitor the current value of the buffer and associated pointer values, you can watch the transmission process.

The two serial ports have similar software interfaces. Any interface that makes reference to either `s0` or `ser0` can be replaced with `s1` or `ser1`, for example. Each serial port should use its own COM structure, as defined in `ae.h`.

```
typedef struct {
    unsigned char ready;           /* TRUE when ready */
    unsigned char baud;
    unsigned char mode;
    unsigned char iflag;          /* interrupt status */
    unsigned char *in_buf;        /* Input buffer */
    int in_tail;                  /* Input buffer TAIL ptr */
    int in_head;                  /* Input buffer HEAD ptr */
    int in_size;                  /* Input buffer size */
    int in_crcnt;                 /* Input <CR> count */
    unsigned char in_mt;          /* Input buffer FLAG */
    unsigned char in_full;        /* input buffer full */
    unsigned char *out_buf;       /* Output buffer */
    int out_tail;                 /* Output buffer TAIL ptr */
    int out_head;                 /* Output buffer HEAD ptr */
    int out_size;                 /* Output buffer size */
    unsigned char out_full;       /* Output buffer FLAG */
    unsigned char out_mt;         /* Output buffer MT */
    unsigned char tms0;          // transmit macro service operation
    unsigned char rts;
    unsigned char dtr;
    unsigned char en485;
    unsigned char err;
    unsigned char node;
    unsigned char cr; /* scc CR register */
    unsigned char slave;
    unsigned int in_seg;          /* input buffer segment */
    unsigned int in_offs;        /* input buffer offset */
    unsigned int out_seg;        /* output buffer segment */
    unsigned int out_offs;       /* output buffer offset */
    unsigned char byte_delay;    /* V25 macro service byte delay */
} COM;
```

sn_init

Arguments: unsigned char `b`, unsigned char* `ibuf`, int `isiz`, unsigned char* `obuf`, int `osiz`, COM* `c`

Return value: none

This function initializes either `SER0` or `SER1` with the specified parameters. `b` is the baud rate value shown in Table 4.1. Arguments `ibuf` and `isiz` specify the input-data buffer, and `obuf` and `osiz` specify the location and size of the transmit ring buffer.

The serial ports are initialized for 8-bit, 1 stop bit, no parity communication.

There are a couple different functions used for transmission of data. You can actually place data within the output buffer manually, incrementing the head and tail buffer pointers appropriately. If you do not call one of the following functions, however, the driver interrupt for the appropriate serial-port will be disabled, which means that no values will be transmitted. This allows you to control when you wish the transmission of data within the outbound buffer to begin. Once the interrupts are enabled, it is dangerous to manipulate the values of the outbound buffer, as well as the values of the buffer pointer.

putsrn

Arguments: unsigned char `outch`, COM *`c`

Return value: int `return_value`

This function places one byte outch into the transmit buffer for the appropriate serial port. The return value returns one in case of success, and zero in any other case.

putsersn**Arguments:** char* str, COM *c**Return value:** int return_value

This function places a null-terminated character string into the transmit buffer. The return value returns one in case of success, and zero in any other case.

DMA transfer automatically places incoming data into the inbound buffer. serhitn() should be called before trying to retrieve data.

serhitn**Arguments:** COM *c**Return value:** int value

This function returns 1 as **value** if there is anything present in the in-bound buffer for this serial port.

getsern**Arguments:** COM *c**Return value:** unsigned char value

This function returns the current byte from **sn_in_buf**, and increments the **in_tail** pointer. Once again, this function assumes that **serhitn** has been called, and that there is a character present in the buffer.

getsersn**Arguments:** COM c, int len, char* str**Return value:** int value

This function fills the character buffer **str** with at most **len** bytes from the input buffer. It also stops retrieving data from the buffer if a carriage return (ASCII: **0x0d**) is retrieved.

This function makes repeated calls to **getser**, and will block until **len** bytes are retrieved. The return **value** indicates the number of bytes that were placed into the buffer.

Be careful when you are using this function. The returned character string is actually a byte array terminated by a null character. This means that there might actually be multiple null characters in the byte array, and the returned **value** is the only definite indicator of the number of bytes read. Normally, we suggest that the **getsers** and **putsers** functions only be used with ASCII character strings. If you are working with byte arrays, the single-byte versions of these functions are probably more appropriate.

Miscellaneous Serial Communication Functions

One thing to be aware of in both transmission and receiving of data through the serial port is that TERN drivers only use the basic serial-port communication lines for transmitting and receiving data. Hardware flow control in the form of CTS (Clear-To-Send) and RTS (Ready-To-Send) is not implemented. There are, however, functions available that allow you to check and set the value of these I/O pins appropriate for whatever form of flow control you wish to implement. Before using these functions, you should once again be aware that the peripheral pin function you are using might not be selected as needed. For details, please refer to the Am188ES User's Manual.

char sn_cts(void)

Retrieves value of CTS pin.

void *sn_rts*(char b)

Sets the value of RTS to **b**.

Completing Serial Communications

After completing your serial communications, there are a few functions that can be used to reset default system resources.

sn_close

Arguments: COM *c

Return value: none

This closes down the serial port, by shutting down the hardware as well as disabling the interrupt.

clean_sern

Arguments: COM *c

Return value: none

This flushes the input buffer by resetting the tail and header buffer pointers.

The asynchronous serial I/O ports available on the 188 Processor have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 10 of the User's manual for a detailed discussion of other features available to you.

4.4 Functions in AEEE.OBJ

The 512-byte serial EEPROM (24C04) provided on-board provides easy storage of non-volatile program parameters. This is usually an ideal location to store important configuration values that do not need to be changed often. Access to the EEPROM is quite slow, compared to memory access on the rest of the controller.

Part of the EEPROM is reserved for TERN use specifically for this purpose.

Addresses 0x00 to 0x1f on the EEPROM is reserved for system use, including configuration information about the controller itself, jump address for Step 2, and other data that is of a more permanent nature.

The rest of the EEPROM memory space, 0x20 to 0x1ff, is available for your application use.

ee_wr

Arguments: int addr, unsigned char dat

Return value: int status

This function is used to write the passed in **dat** to the specified **addr**. The return value is 0 in success.

ee_rd

Arguments: int addr

Return value: int data

This function returns one byte of data from the specified address.

4.5 FILESYS.LIB

FILESYS.LIB is C library that includes fileio.obj and filegeo.obj that supports data transfers to and from Compact Flash cards installed on the FlashCore or the FC-0.

4.5.1 File System Initialization

```
int fs_initPCFlash(void);
```

This function should be called before any other disk operations. It should also be called if a new card is installed.

This function will return 0 if a card with FAT filesystem is located and initialized. Any other returns indicate the card was 'busy' (not found), or if disk geometry is not correct. If 0 is not return, check <filegeo.h> for error-code description.

4.5.2 File System Access and Modification

A fs_descrip structure is used as a file handle to an open file. The structure might change over time, and you should be careful in accessing any fields directly. This structure is used in many of the function calls that define file management on the FlashCore or FC-0.

A fs_descrip variable might be created in two ways:

- 1) created on the heap via a call to fs_fopen(); this must later be freed by a corresponding call to fs_fclose(), even if an error occurred at some point with the file.
- 2) a call to fs_findfirst(), passing in a fs_descrip variable you've created (in any way). The file isn't actually open, and you are responsible for freeing the variable.

The structure is defined below:

```

*****/
struct fs_descrip {
    unsigned int ff_dirpos; // The number of the directory entry for this file.
    unsigned int ff_start, // The starting cluster.
                ff_current; // The cluster currently being written to.
    char ff_attrib; // Attribute byte, see FA_xxx above.
    char ff_mode; // Either fREAD or fWRITE or fCLOSED.
    unsigned int ff_ctime, ff_cdate; // File created time and date.
    unsigned int ff_mtime, ff_mdate; // File modified time and date.
    unsigned int ff_adata; // File accessed date, no time stored.
    unsigned long ff_fsize, // File size in bytes.
                ff_position; // The 'read' pointer.
    int ff_status; // For passing error information.
}

```

```

char ff_name[FNLEN+1]; // File name, with \0.
unsigned char *ff_buf; // Cluster buffer, sectors must be read and written
// from disk in entirety, so this area buffers them. Created on the heap!
};
*****/

```

struct fs_descrip *fs_fopen(const char *filename, int flags)

Opens and prepares a file for operation. The arguments are as follows:

The flag should be one of the following values:

O_RDONLY : (open file for read only, fails if file doesn't exist),

O_WRONLY : (open/creates a file, fails if file exists),

O_APPEND : (open a file and prepares to append, positioning file pointer at the end of the program)

where the flags are defined as:

```

#define O_RDONLY      0x1
#define O_WRONLY      0x2
#define O_APPEND      0x4

```

The function returns a 'struct fs_descrip' handle to the open file, or NULL if it fails. It is important to note that if a file is successfully opened, it should always be closed using fs_fclose() to free any memory used for the file handle. The function call fs_fclose() will also finalize any modifications to the file.

int fs_findfirst(char *pathname, struct fs_descrip *descrip)

Finds the *first* file entry (including directories and 'labels' corresponding to the argument 'pathname'. The handle for the file is returned in the 'descrip' argument (you must allocate memory for it before making the call). This file is not* actually opened (you don't need to fs_fclose() it later, either).

This function returns one of the following:

fOK: a file was found.

fend: The end of the directory specified in 'pathname' was reached, but no file found.

error code : Check this file and <filegeo.h> for error-code descriptions.

For version 1.0, pathname MUST be " *.*". Any other pathname will generate an error. In later versions, other pathnames/wildcards may be supported. So, a call to fs_findfirst(" *.*",) returns the first file entry in the root directory.

int fs_findnext(struct fs_descrip *fs_descrip);

Given a file descriptor, find the next entry in the file's directory. The details of the file are written into the same argument file descriptor. As before, the file is **not** actually opened.

Return values:

fOK: a file was found.

fEND: The end of the directory specified in 'pathname' was reached, but no file found.

error code : Check this file and <filegeo.h> for error-code descriptions.

Use this function, in combination with fs_findfirst, to iterate through the entries in a directory.

unsigned char fs_fgetc(struct fs_descrip *fs_descrip)

Get a single byte from the opened file pointed to by the file descriptor.

RETURNS:

Normally, next byte of data.

'\0' : Default return value if a read from file is not possible. In this case, check fs_descrip->ff_status to determine the cause. Might be fEOF (end of file), FILLEGAL (illegal operation), or other error code. (fOK indicates the read value was '\0').

unsigned char fs_fputc(const unsigned char s, struct fs_descrip *fs_descrip)

Writes a single byte to the opened file pointed to by the file descriptor.

RETURNS:

Normally, the character that was just written to the file.

'\0' : Default return value if a read from file is not possible.

In this case, check fs_descrip->ff_status to determine the cause.

Might be fEOF (end of file), FILLEGAL (illegal operation),

or other error code (full disk).

char fs_fgets(char *s, int n, struct fs_descrip *fs_descrip);

Gets a string of characters from the opened file 'fs_descrip', of up to n characters. Returns when a newline '\n', or end of file is reached.

RETURNS:

The contents of `fs_descrip->ff_status` (fOK if all is correct).

char fs_fprintf(struct fs_descrip *fs_descrip, const char *format, ...)

Similar to 'printf', writing a formatted string to the opened file pointed by `fs_descrip`.

RETURNS:

The number of characters successfully output.

This function automatically adds carriage return '\r' before newline '\n', as in standard DOS practice.

char fs_fclose(struct fs_descrip *fs_descrip)

Closes a file previously opened with `fs_fopen`, saving any lingering changes, updating directory entries, and freeing memory associated with `fs_descrip` (be sure to only `fs_fclose` file handles created with `fs_fopen`, and not something like `fs_findnext`).

RETURNS:

Returns error code associated with file; the contents of `fs_descrip` are no longer valid after this call, do not check its `ff_status` field.

void fs_StampTimeHMSMDY(struct fs_descrip *fs_descrip, char TDtype, unsigned int hour, unsigned int min, unsigned int sec, unsigned int month, unsigned int day, unsigned int year);

Changes the time stamp for either file 'access', 'modification', or 'creation' for a file pointed to by `fs_descrip` in the directory entry. Since not all systems have RTC, the user is expected to use this function if they wish to use file timestamps. `fs_fopen`, `fs_fclose`, etc... will not. DOS usually stores timestamps in 'packed' storage format (documentation available online).

Appendix A: Kpad – FlashCore Interface

A.1 Signal Definitions

J2 Header: FlashCore

GND	1	2	VCC
P4	3	4	P14
	5	6	P6
P15	7	8	P5
INT3	9	10	P30
INT1	11	12	INT2
P3	13	14	INT0
/CTS1	15	16	/RTS1
P24	17	18	P17
P0	19	20	P25
P13	21	22	P10
/RTS0	23	24	A19
P1	25	26	/CTS0
P26	27	28	P29
GND	29	30	

Signal Definitions for FlashCore
J2 header.

Pins Highlighted Red need to be cut when using a 10x2 connector.

Pins Highlighted in Green indicate the pins to install the 10x2 connector onto.

Also, an external wire must be soldered to bring VCC from J2.2 to J2.30

<i>FlashCore</i>	<i>Function</i>
<i>Pin Name</i>	
VCC	Kpad supply voltage, +5V
GND	Ground
P26, P29, P1, P21(/CTS0), P20(/RTS0), P9(A19), P13, P10	Keypad Scan: Inputs to FlashCore
P0, P25, P24, P17	Outputs from Am188ES. Drive D7-D4 on LCD controller.
P18 (/CTS1)	Drives E (enable) line to LCD
P19 (/RTS1)	Drives RS line to LCD. Shared with I22 for Keypad scan
P3	Output. Keypad scan.
/RD	Input to PAL

<i>H5 Header: Kpad</i>			
I07	1	2	I06
VCC	3	4	GND
I05	5	6	I04
I03	7	8	I02
I01	9	10	I00
I27	11	12	I26
I25	13	14	I24
I23	15	16	I22
I21	17	18	
	19	20	

<i>Kpad Pin Name</i>	<i>Function</i>
VCC	Kpad supply voltage +5V
GND	Ground
I07 – I00	Keypad scan. Tied to Pull-up resistors
I27 – I24	Data lines of LCD controller D7 – D4
I23	LCD controller Enable
I22	LCD controller, mode select Low: Command High: Data
	Keypad Scan
I21	Keypad Scan

Kpad pin layout and description for IO version

A.2 Connections, Pin Mapping

When driving the Kpad with the FlashCore as described by this appendix, one modification must be made to the FlashCore. If your Kpad and FlashCore were ordered together, the necessary modification was already been made before shipment. This applies only if you have ordered the Kpad-FC after your FlashCore. In order to drive the Kpad, it requires a +5V power supply which can be taken directly from the J2 header on the FlashCore. An external wire must be soldered to the FlashCore which ties J2.2 = J2.30. This will bring VCC to J2 pin 30 and then be routed to the Kpad. J2 pin 30 by hardware definition is an open pin, so there will be no compatibility problems. The following table shows the exact connections between the FlashCore and the Kpad.

Pin Mapping: FlashCore ⇔ Kpad

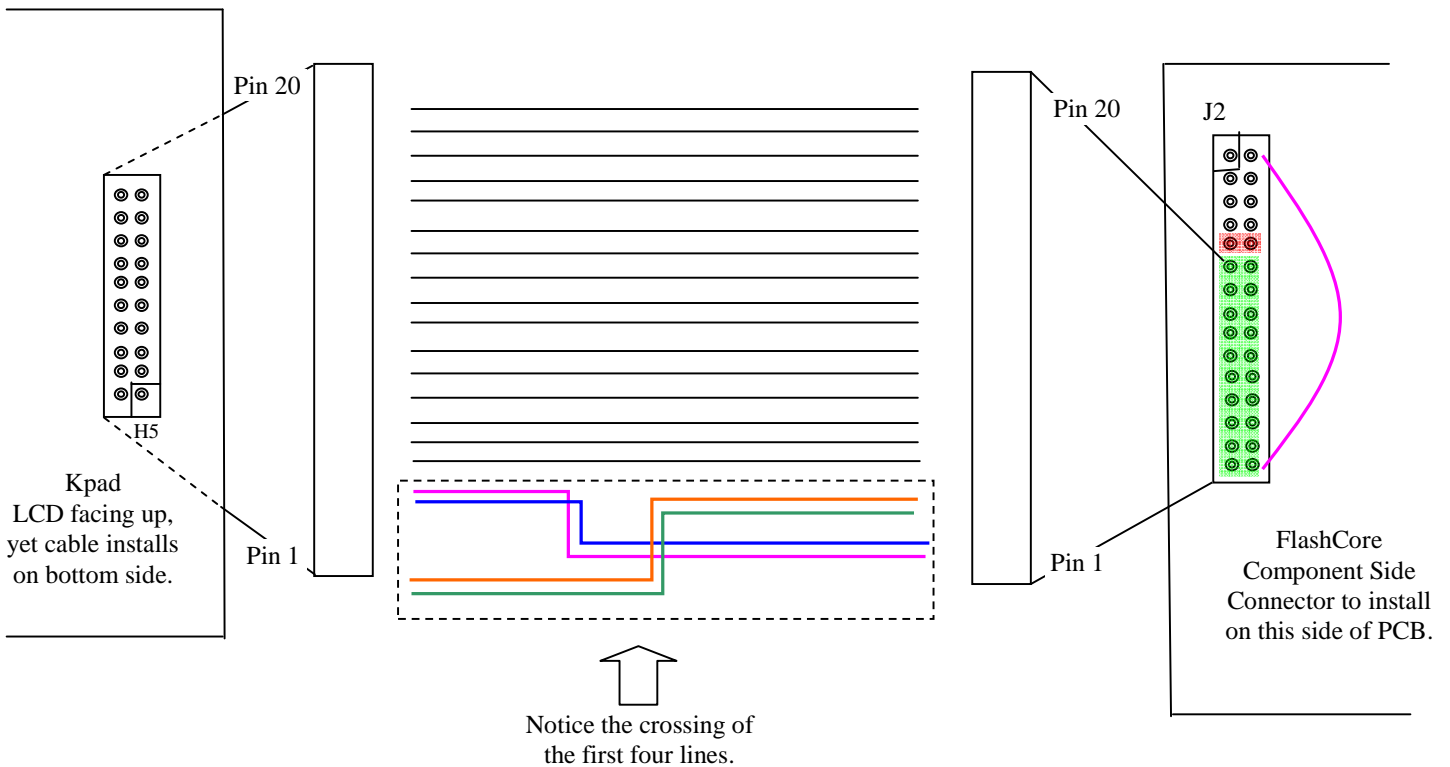
Kpad Signal Name	H5 Pin #	J2 Pin #	FlashCore Signal Name
I07	1	27	P26
I06	2	28	P29
VCC	3	30	VCC
GND	4	29	GND
I05	5	25	P1
I04	6	26	P21
I03	7	23	P20
I02	8	24	P9
I01	9	21	P13
I00	10	22	P10
I27 = D7	11	19	P0
I26 = D6	12	20	P25
I25 = D5	13	17	P24
I24 = D4	14	18	P17
I23 = E	15	15	P18
I22 = RS	16	16	P19
I21 = Row 1	17	13	P3
No Connection	18	14	No Connection

A.3 Flat Cable Specifications

This section will define how a flat cable should be prepared to interface the Kpad based on the sample code for the FlashCore, `tern\186\samples\kpad_fc.c`. This interface is based upon the pin mapping given in the previous section. The following diagram is a simply an aide to help visualize how the cable will connect the Kpad and the FlashCore. It is therefore important to remember the above table takes priority in terms of the final connections.

Header H5 – Kpad
 The H5 pin header will be installed on the underside of the Kpad, opposite of the LCD.
 Thus the flat cable will install on the under side as well.

Because of the Compact Flash Interface on the FC, the pin header on J2 will always be type ‘T’, or component side. This only allows one orientation to install a flat cable.

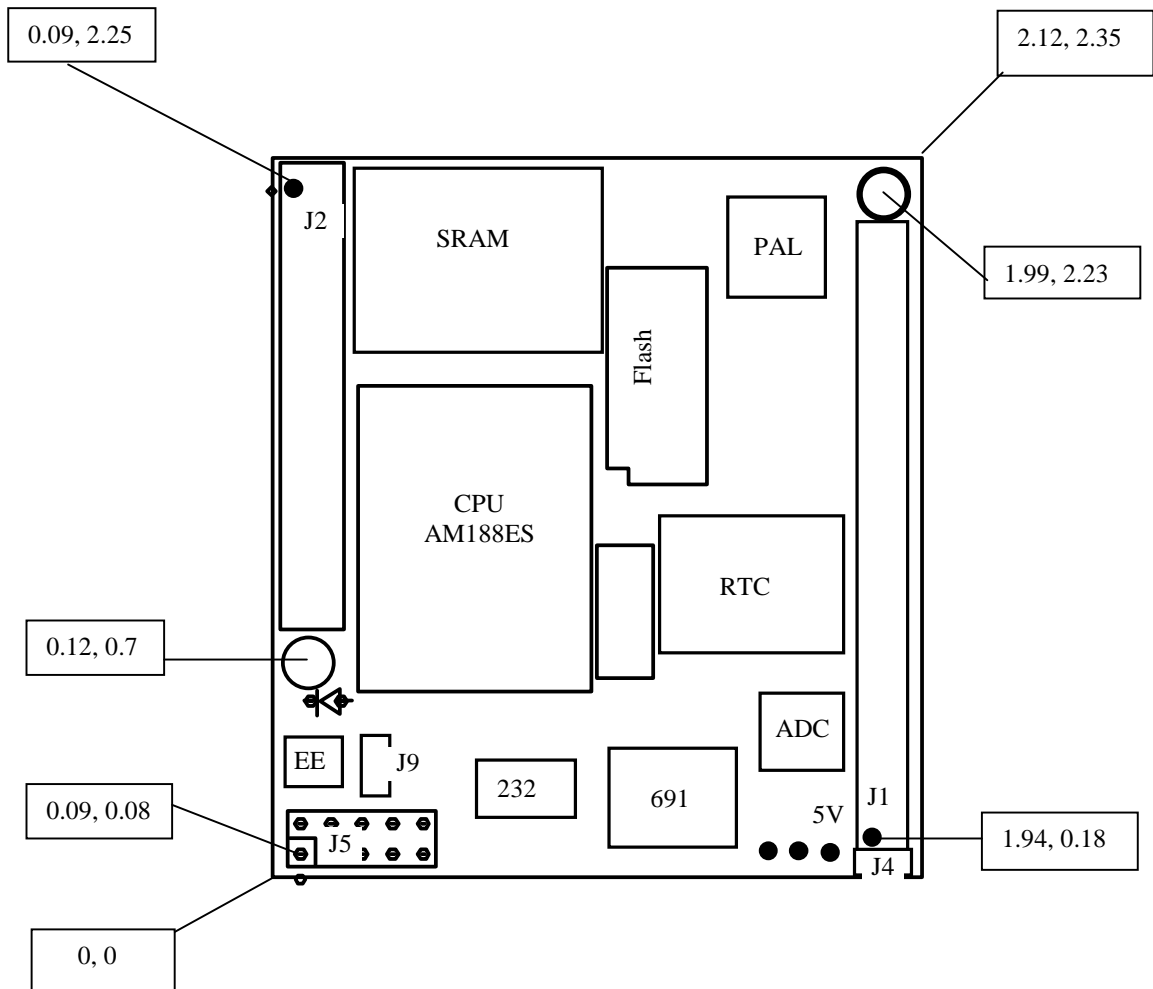


Instructions for flat cable assembly:

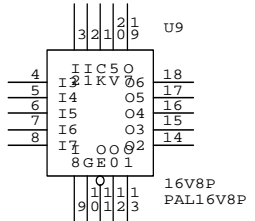
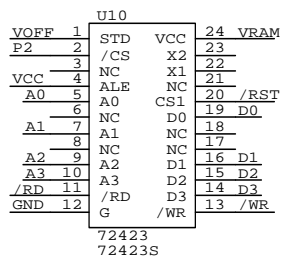
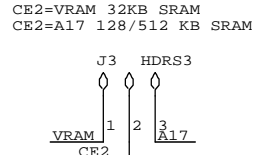
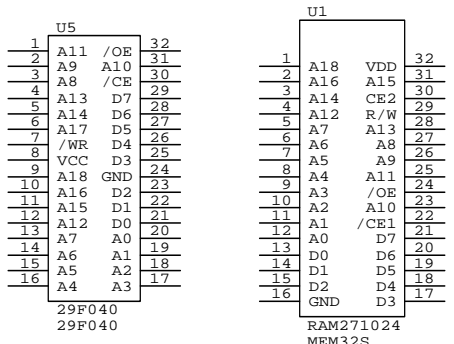
- (1) Use a 20 wire flat cable
- (2) Use two 10x2 connectors.
- (3) Peel back the first and second pair of wires as shown above.
- (4) Cross first four wires as shown above and secure into one connector. This will mount onto the Kpad.
- (5) Secure FlashCore side connector with no modification to the wires.
- (6) Cut pins J2.9 and J2.10 (highlighted in red above) to allow for install of 10x2 connector onto FlashCore’s J2 header
- (7) Install FlashCore side connector flush to bottom of J2 header (highlighted in green above)
- (8) Solder wire from J2.2 to J2.30 on FlashCore (shown in purple above)

IMPORTANT: Refer to tables in this appendix to confirm the correct pin-to-pin configuration.

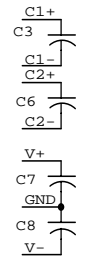
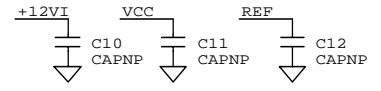
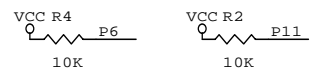
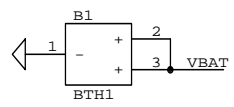
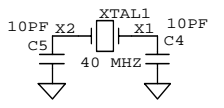
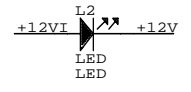
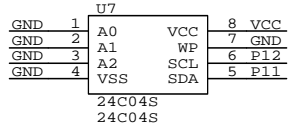
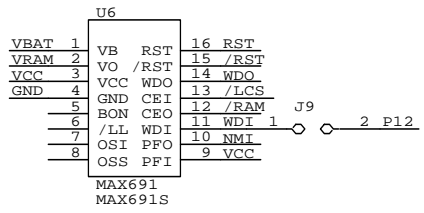
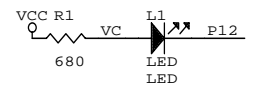
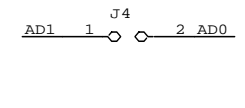
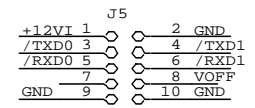
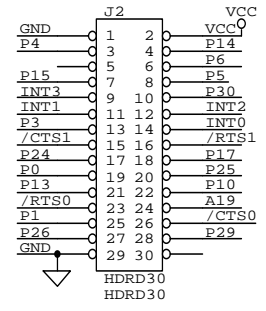
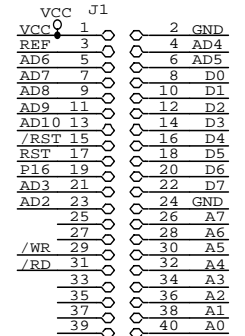
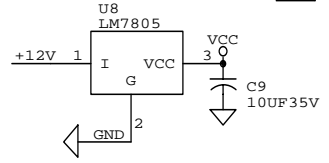
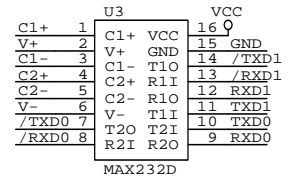
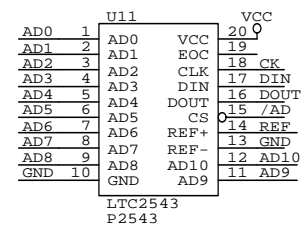
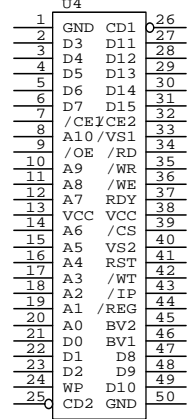
Appendix B: FlashCore Dimensional Layout



All units are in inches



CompactFlash 50 pin CF+ RECEPTACLE



STE		
Title		
FlashCore for CompactFlash Cards		
Size	Document Number	REV
B	F-Core.SCH	
Date:	June 11, 2001	Sheet 1 of 1