586-Engine-P™

C/C++ Programmable, 133 MHz 32-bit Controller
with Floating Point Unit, CompactFlash Interface
and 16-bit ADC / DACs

Technical Manual

TERN INC.
1950 5th Street, Davis, CA 95616, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: sales@tern.com *http://www.tern.com*

Version 2.0

May 19, 2010

© 1993-2010 TERN INC.
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 586-Engine-P™ (***5P***) is a complete C/C++ programmable standalone controller based on a 32-bit 133MHz AMD Elan SC520. The 5P improves upon the standard 586-Engine core by integrating additional peripheral components. Most significantly, the 5P has added voltage regulator and RS232 line drivers, making it a true stand-alone product. The 5P also adds high-speed 16-bit analog I/Os, making it appropriate for a whole new generation of high-performance data acquisition (DAQ) and precision control applications.

INPUTS/OUTPUTS
The SC520 supports 32 programmable multifunctional I/O lines (PIO) that can be used as general discrete I/O. Two industrial-standard 16550-compatible UARTs (RS232) support baud rates up to 1.152 M baud. One synchronous serial interface (SSI) supports full-duplex, high speed bi-directional communication.

A unique 16-bit parallel ADC (AD7655, 0-5V) supports ultra high-speed (1 MHz conversion rate) analog signal acquisition. The AD7655 contains two low noise, high bandwidth track-and-hold amplifiers that allow *simultaneous* sampling on two channels. Each track-and hold amplifier has a multiplexer in front to provide a total of 4 channels analog inputs. The parallel ADC achieves very high throughput by requiring only two CPU I/O operations (one start, one read) to complete a 16-bit ADC reading. With a precision external 2.5V reference, the ADC accepts 0-5V analog inputs at 16-bit resolution of 0-65535 counts.

An octal rail-to-rail digital to analog converter (TLC2600) can be installed to provide eight channels of analog voltage (0-5V) outputs. At power on, all analog outputs are zero with the on-board reset. The DAC chip is accessed through a 3-wire SPI-compatible serial interface, which is connected to the 5Ps high-speed synchronous serial port (clockable up to 50 MHz). Eight built-in analog output buffers can drive rail-to-rail analog voltages up to 15 mA.

OTHER FEATURES
The 5P boots from on-board 256K 16-bit ACTF Flash, and supports up to 256K 16-bit battery-backed SRAM. The 5P supports low cost, removable, up to 2 GB mass storage CompactFlash cards with onboard CompactFlash interface.

The SC520 integrates an Am586 CPU and a high performance ANSI/IEEE 754 compliant hardware floating-point unit (FPU). The FPU provides arithmetic instructions to handle numeric data and transcendental functions for sine, tangent, logarithms, etc, useful for intensive computational applications. It is estimated to be 10-50 times faster than an 8/16-bit controller without a FPU.

Up to 15 external interrupts are supported. There are a total of seven timers including one programmable interval timer (PIT) that provides three 16-bit PIT timers and three 16-bit GP timers, plus a software timer. These timers can support timing or counting external events. The software timer provides a very efficient hardware time base with microsecond resolution. A real-time clock (RTC) provides time-of-day, 100-year calendar and 114 bytes of battery backed RAM.

Signal lines on headers are 3.3V output and 5V input tolerant. Absolutely no voltage greater than 5V should be applied to any pins. With the 388 pin BGA package for the SC520, repair support is not available. The 5P can be powered by a single unregulated DC power from 8V to 30V range with the on-board high-efficiency 5V switching regulator (LM2575). The 5P can also be powered by a regulated 5V without using on-board 5V regulator.

The ***5P*** works with TERN expansion boards including the P100, P300, LittleDrive, and MotionC

Special Note: The core of the Am520 CPU operates at +2.5V and the I/O operation at +3.3V. Also, the input for the I/O is +5V compatable. Stresses above these can cause permanent damage to the SC520 CPU. Operation above these values is not recommended, and can effect device reliability.

1.2 Features

- Dimensions: 3.6 x 2.6 x 0.3 inches
- 133MHz, 32-bit CPU (ElanSC520, AMD), Intel 80x86 compatible
- Easy to program in C/C++
- Power consumption: 110 mA at 24V
- Power input: + 8V to +30V unregulated DC with switching regulator
- 256KW SRAM, 256KW, 114 byte internal CMOS RAM
- 4 channel 16-bit, 1MHz conversion rate, ADC, AD7655, 0-5V input
- 8 channel 16-bit DAC, TLC2600, 0-5V
- High performance floating point coprocessor
- Up to 2GB Compact Flash memory expansion
- 2 serial ports
- 15 external interrupts with programmable priority
- 32 multifunctional I/O lines from ElanSC520, 1 SSI, 7 16-bit timers
- Supervisor (691) for power failure, reset and watchdog
- Lithium coin battery

1.3 Physical Description

The physical layout of the 586-Engine-P is shown in Figure 1.1.

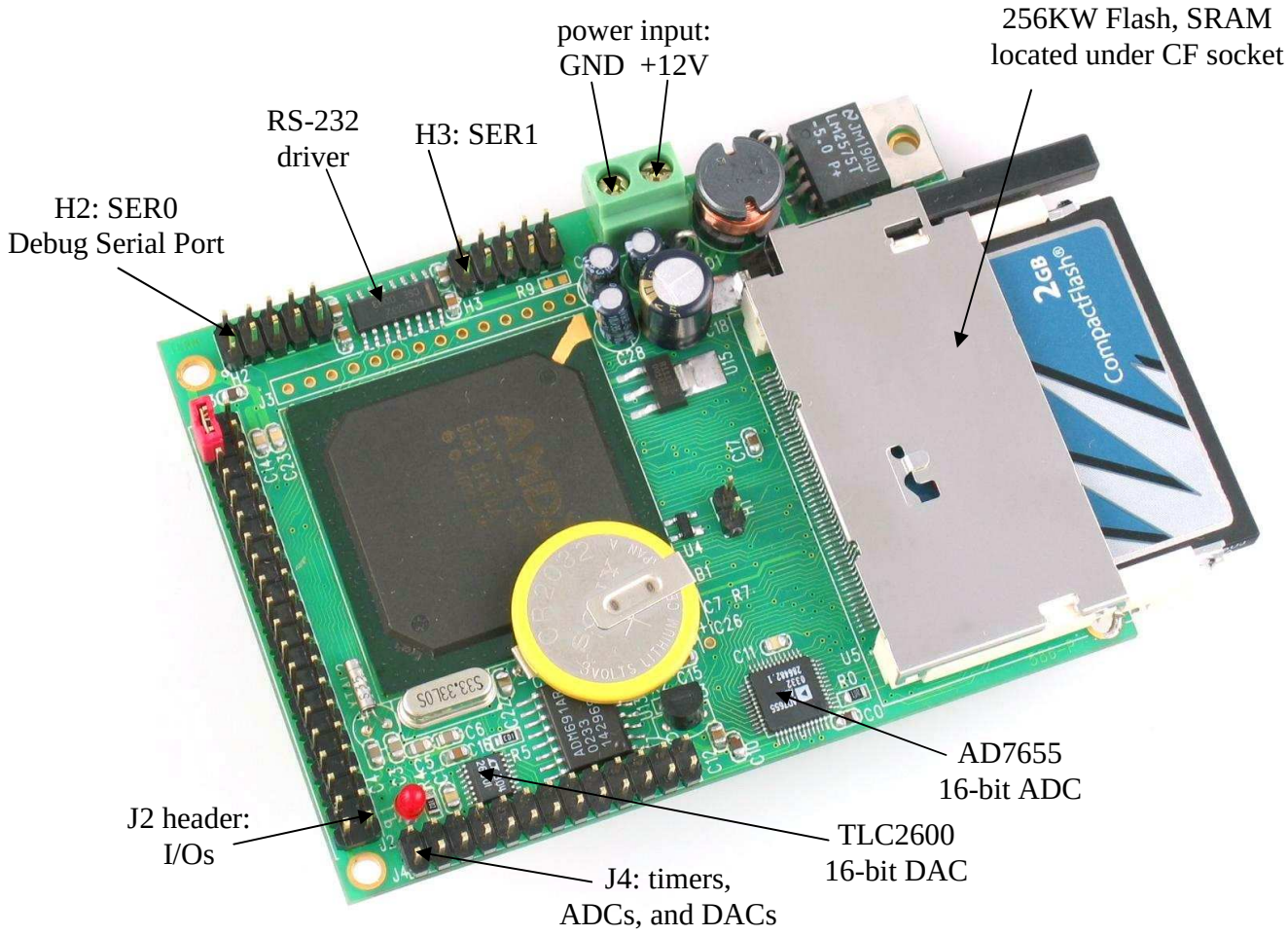


Figure 1.1 Physical layout of the 586-Engine

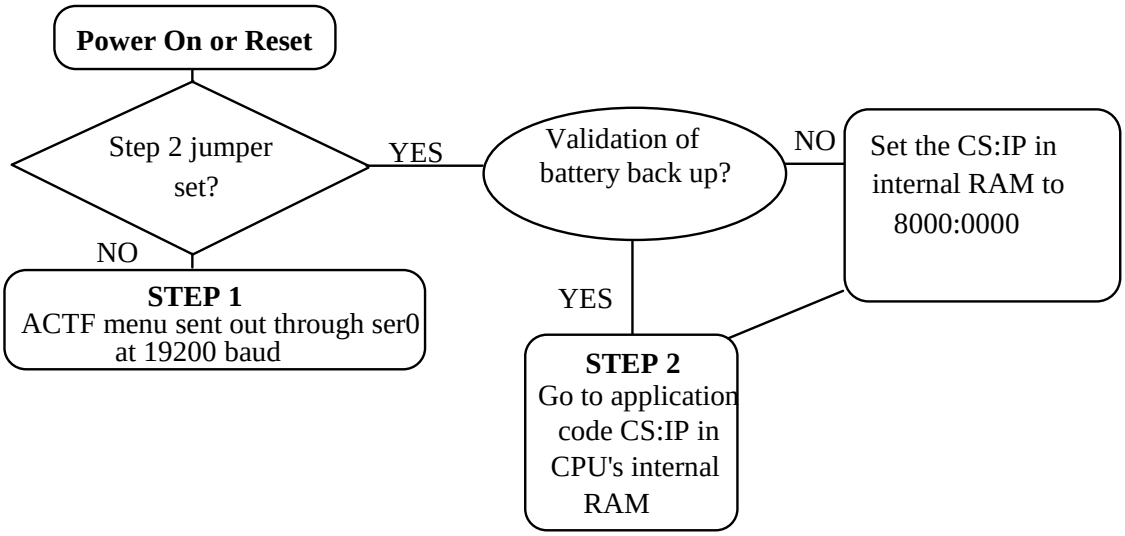


Figure 1.2 Flow chart for ACTF operation

The "ACTF boot loader" resides in the 256KW on-board Flash chip (29F400). At power-on or RESET, the "ACTF" will check the STEP 2 jumper. If STEP 2 jumper is not installed, the ACTF menu will be sent out from serial port0 at 19200 baud. If STEP 2 jumper is installed, the***5P*** will check for a valid battery back-up. If present the ***5P*** will go to the jump address stored in the CPU's 114 bytes of general purpose RAM. Without a valid battery back up, the ***5P*** will write the address 0x80000 to the inernal RAM, and then go to that address.

There is no ROM socket on the ***5P***. The user's application program must reside in SRAM for debugging in STEP1, reside in battery-backed SRAM for the standalone field test in STEP2, and finally be programmed into Flash for a complete product. For production, the user must produce an ACTF-downloadable HEX file for the application, based on the DV-P. The "STEP2" jumper (J2 pins 38-40) must be installed for every production-version board.

Step 1 settings

In order to correctly download a program in STEP1 with Paradigm C++ Debugger, the ***5P*** must meet these requirements:

1) 5860_115.HEX must be pre-loaded into Flash starting address 0x80000.

2) The CPU's 114 bytes of RAM must have the correct jump address pointing at 5860_115.HEX, which is the address 0x80000.

4) The STEP2 jumper must be installed on J2 pins 38-40.

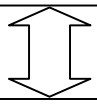
For further information on programming the ***586-Engine-P***, refer to the Software chapter.

1.4 586-Engine-P Programming Overview

Steps for ***5P***-based product development:

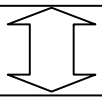
Preparation for debugging:

- Connect 5P to PC via serial link at 19,200, N, 8, 1 with Hyper Terminal
- Power-on without STEP 2 Jumper installed (J2.38 = J2.40)
- ACTF Menu will be sent from 5P to Hyper Terminal
- Type 'D', then <enter> to download. Send \tern\586\rom\l_29f400.hex
- Type 'G04000', then <enter> to execute HEX file just sent; will prepare flash
- Send \tern\586\rom\5860_115.hex
- Type 'G80000', then <enter> to execute debug kernel
- Set STEP 2 Jumper (J2.38 = J2.40)
- Power-on/reset, ready to connect to Paradigm C/C++ for debugging



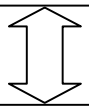
STEP 1: Debugging

- Launch Paradigm C/C++. Open \tern\586\samples\5p\5p_cf.ide.
- Run sample code
- Create application using sample code.
- Use Paradigm C/C++ to edit, compile, link, locate, and remote debug



STEP 2: Standalone field test

- Downloaded code from STEP 1 is located by default at 0x0800:0x0000 in the battery-backed SRAM
- Set Jump Address to point to downloaded code. Remove STEP 2 Jumper and cycle power. Setup Hyper Terminal and see ACTF menu sent from 5P.
- Type 'G08000', then <enter> to jump to and execute application in SRAM
- Set STEP 2 jumper. 5P will execute application at 0x08000 at power-up.
- Test application. Return to STEP 1 as needed



STEP 3: Production (DV-P Kit required)

- Use Paradigm C/C++ to generate ACTF downloadable application HEX file.
- In Paradigm C/C++, open **Target Expert** for application. Set "Target Connection" to "No Target/ROM". Build target. Paradigm C/C++ will generate HEX file.
- Remove STEP 2 Jumper and see ACTF menu at hyper terminal. Download "l_29f400.hex". Same process as above. Type 'G04000" to prepare flash
- Send you application HEX file. Type 'G80000' to execute code and set jump address to point to you application in flash.
- Set STEP2 Jumper

1.5 Minimum Requirements for 586-Engine-P System Development

Minimum Hardware Requirements

- PC or PC-compatible computer with serial COMx port that supports 115,200 baud
- 586-Engine-P controller
- Debug Serial Cable (RS-232; DB9 connector for PC COM port and IDE 2x5 connector for controller)
- Center negative wall transformer (+9V, 500 mA)

Minimum Software Requirements

- TERN EV-P installation CD-ROM and a PC running: Windows 95/98/2000/NT/ME/XP

With the EV-P, you can program and debug the 586-Engine-P in Step One and Step Two, but you cannot run Step Three. In order to generate an application Flash file and complete a project, you will need the Development Kit (DV-P Kit).

Chapter 2: Installation

2.1 Software Installation

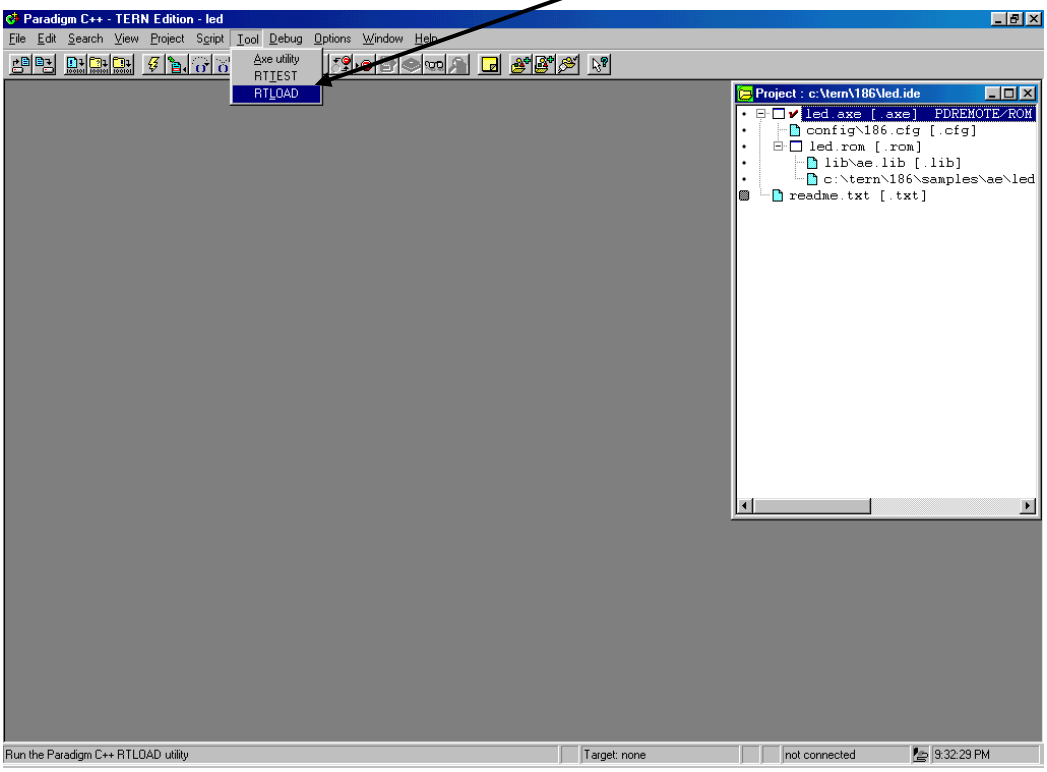
Refer to the Technical manual “EV-P&DV-P Kit” on TERN CD under tern_docs\manual.

By manufacture default upon shipment, the 586-Engine-P will be ready to communicate with Paradigm C++ for debugging, with STEP2 jumper installed, CMOS RAM setup for 0x80000, and 5860_115 debug kernel residing in Flash starting 0x80000. Power on, the on-board LED should blink twice indicating running debug kernel. You DO NOT have to download debug kernel into flash again. You can SKIP the operation discussed in 2.2 below.

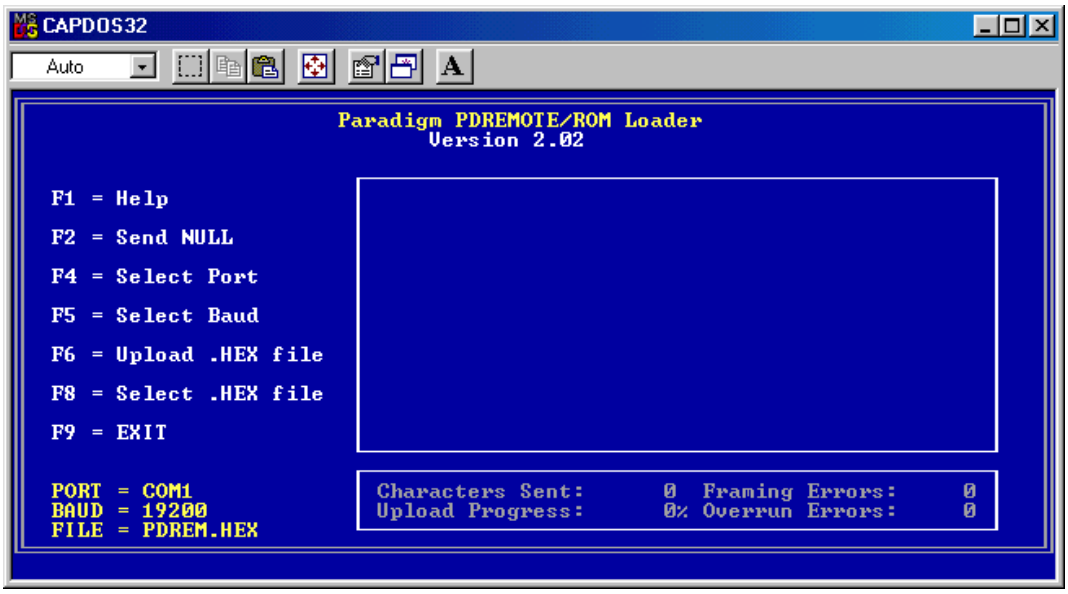
2.2 Prepare 586-Engine-P for Paradigm C++ TERN Edition

This section uses RTLOAD to communicate with the ACTF utility while chapter refers to a hyper terminal. Either works fine.

1) Start Paradigm C++. Select from Top menu: Tool, RTLOAD,

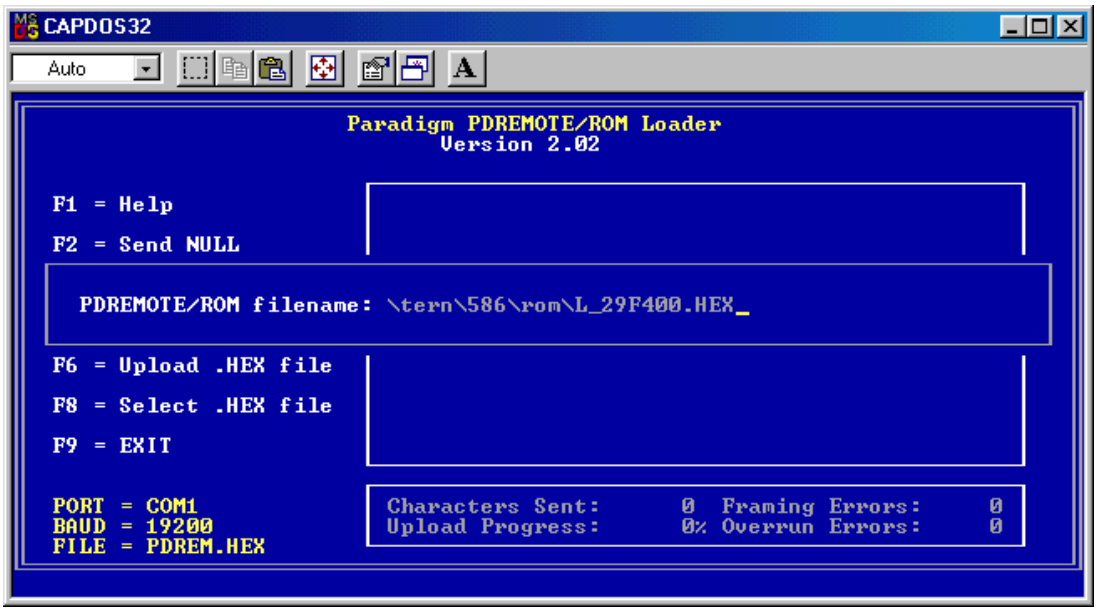


A HEX file Loader window will be shown.

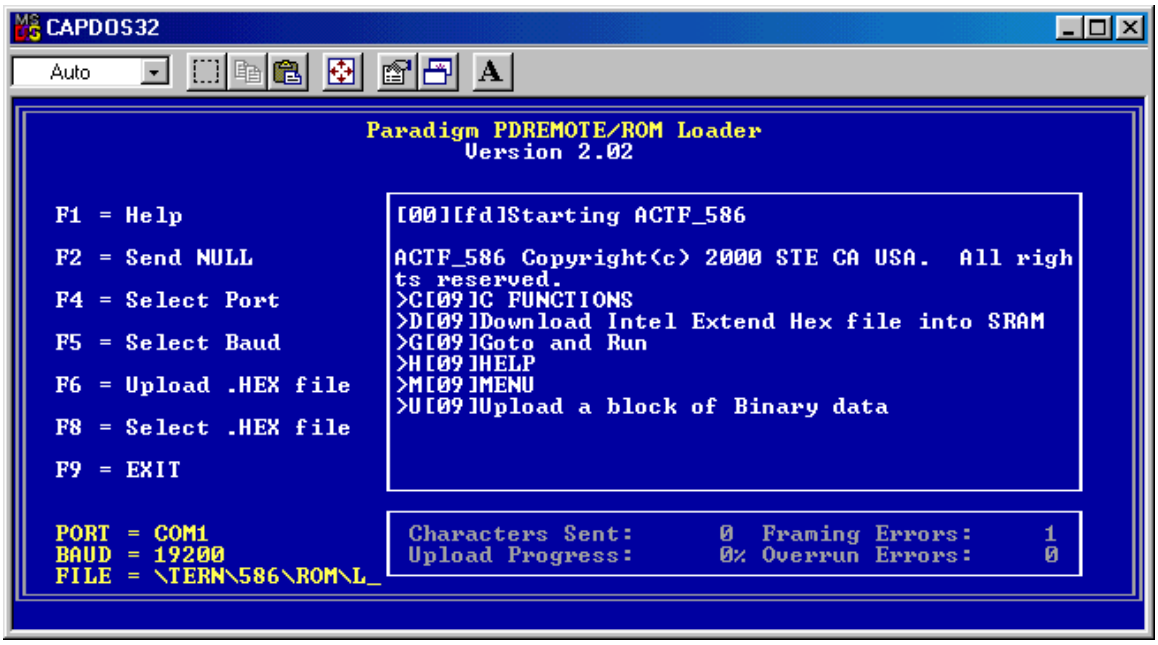


2) F5=Select Baud, to setup 19200.

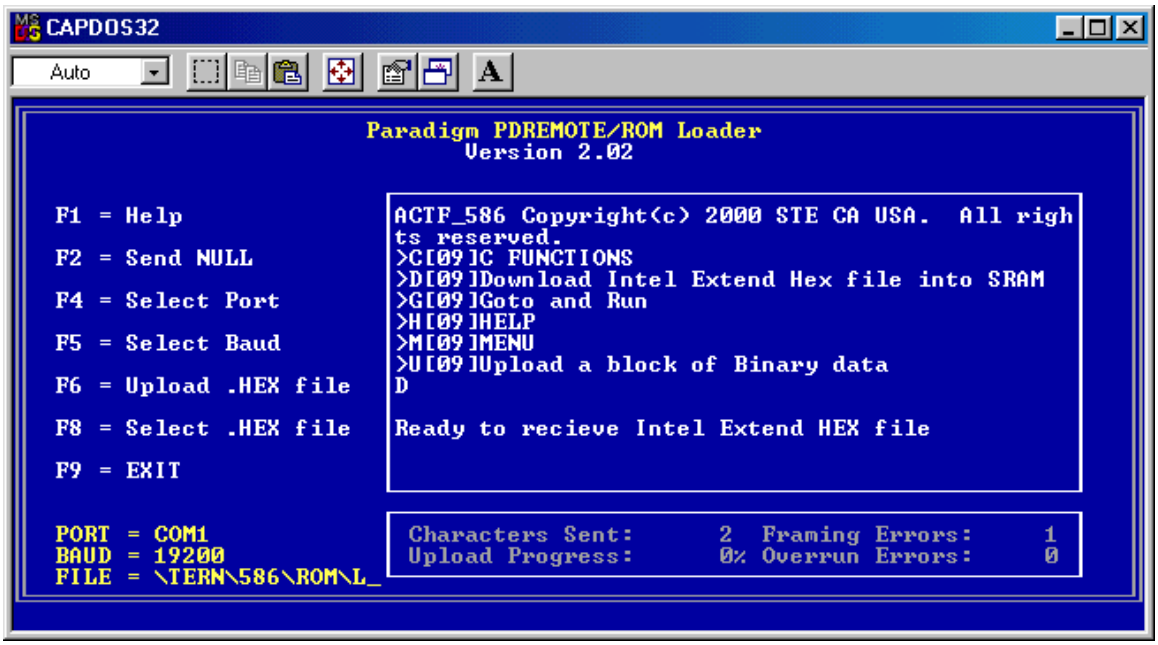
3) F8=Select .HEX file from c:\tern\586\rom\L_29F400.HEX



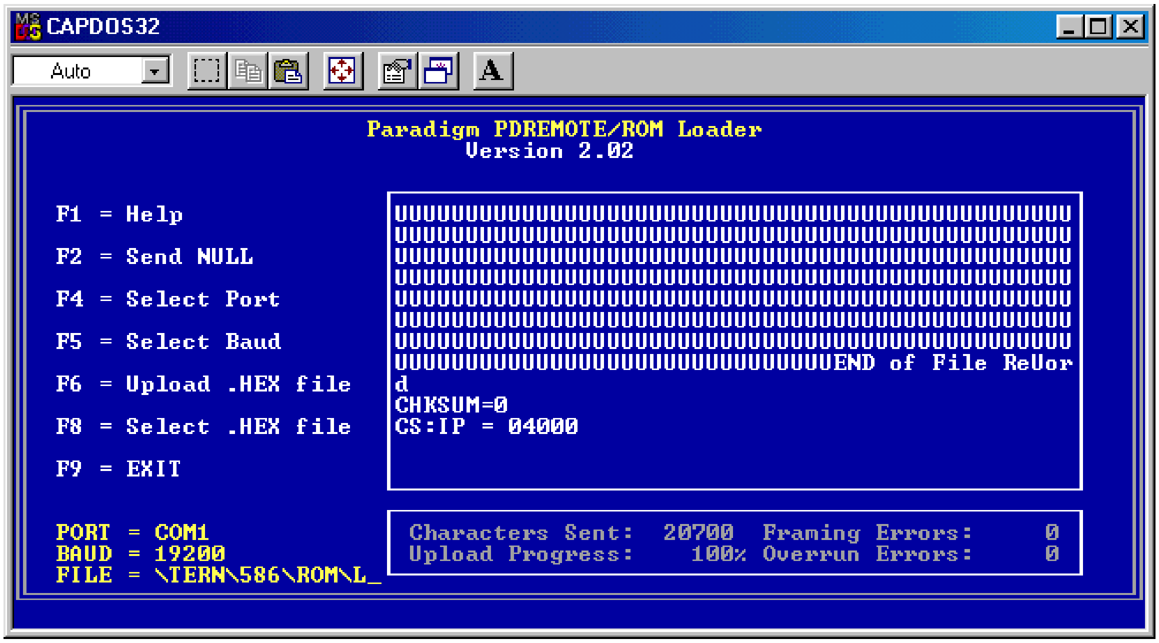
3) Power on the 586-Engine-P with the STEP2 jumper off, the ACTF menu will show up.



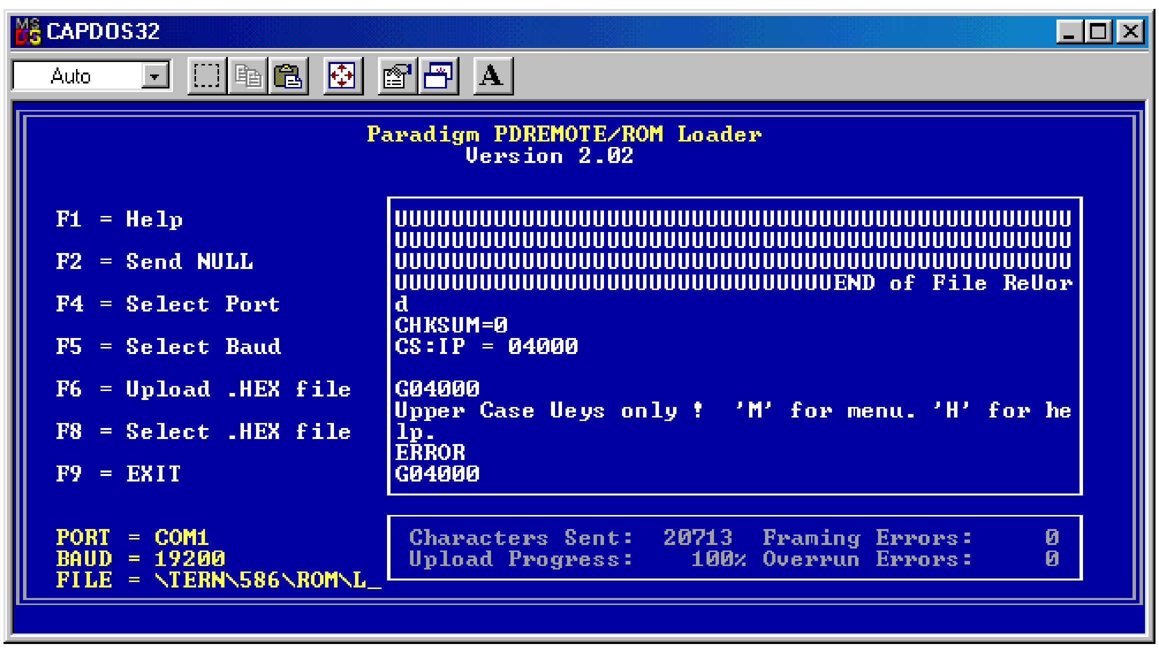
4) Caps Lock on your PC keyboard, Type “D” command, then enter.



5) F6 = Upload .HEX file, from PC to 586 SRAM.

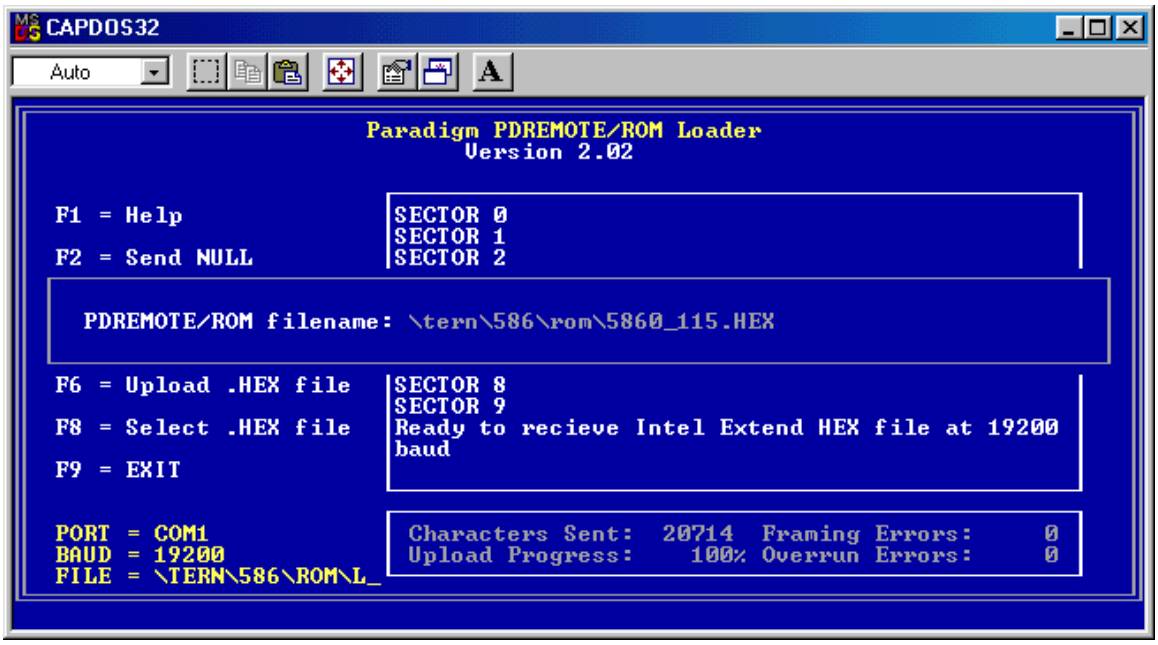


6) Type “G04000” to run the “L_29F400” in SRAM. The first time you type “G04000” you will get an error. Doing the “G04000” again, you will see as below

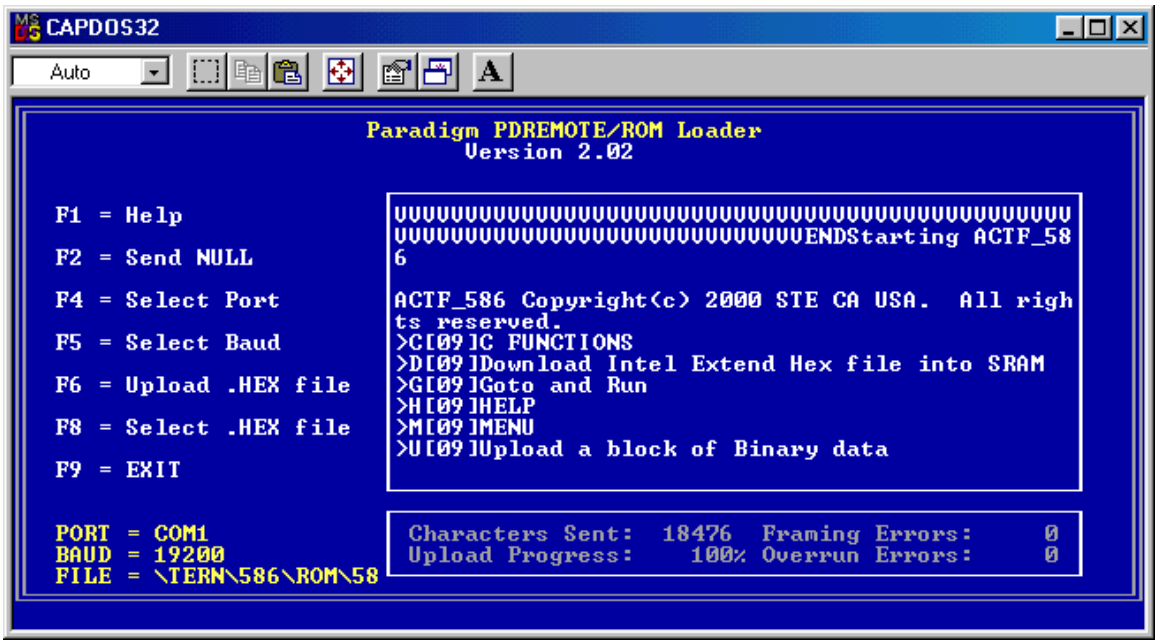


The 29F400 Flash sector 0x80000 to 0xFBFFF will be erased. It will then be ready to program Flash with new DEBUG kernel file (c:\tern\586\rom\5860_115.hex), or user application .HEX.

7) F8= Select .HEX file, from c:\tern\586\rom\5860_115.hex.



8) F6=Upload .HEX file to program the 5860_115 debug kernel into Flash starting address 0x80000.



After programming the Flash, 586-Engine-P will automatically reset.

“G80000” to setup the CMOS RAM Jump Address, and start the DEBUG kernel. The on-board LED should blink twice and then stay on; indicating 586-Engine-P is ready for remote debugging.

Power off the controller. Install the STEP2 Jumper, then power on, the LED blink twice.

Use F9 = Exit.

The 586-Engine-P is ready for using Paradigm C++ TERN Edition to download, debug, and run.

There are several sample projects in the c:\tern\586 directory (default working directory):

tern\586\led.ide: The most basic sample project, just flashes LED

tern\586\test.ide: Gives samples for serial ports, interrupts, timers. etc

tern\586\samples\5p\586p.ide: Shows how to interface CompactFlash

tern\586\samples\5p\5p_cf.ide: Examples for hardware specific to 5p: ADC, DAC for example.

To open projects, go to "File" and open the sample project file, then build and download.

There are many sample programs under c:\tern\586\samples.

After you debug your application code, you can setup the 586-Engine-P to run in Standalone Mode.

Standalone Mode(STEP2):

By default, the Paradigm C++ TERN Edition will download your application code starting at 0x08000 in the battery backed SRAM.

Power off 586-Engine-P. Remove STEP2 jumper. On PC side, click TOOL, RTLOAD.

Power on 586-Engine again without STEP2 jumper, then the ACTF menu should show up.

At the ACTF menu prompt, type "G08000" to setup the Jump Address and run your application.

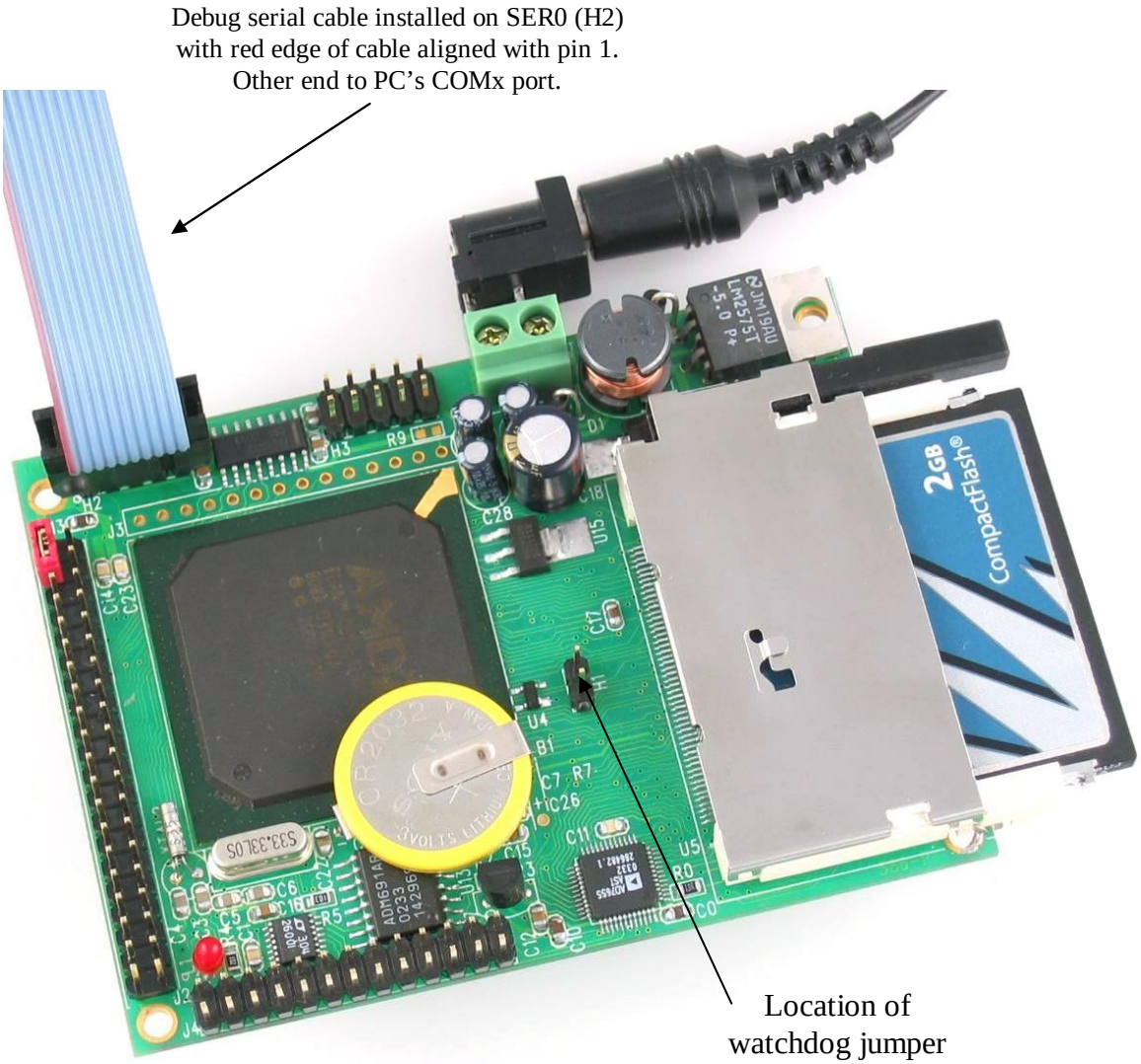
Power off, install the STEP2 Jumper. Then at power on, controller will jump to 0x08000 in SRAM and run your application.

2.3 Hardware Installation

Overview

- Connect debug serial cable:
  For debugging (STEP 1), place IDE connector on SER0 (H2) with red edge of cable at pin 1
- Connect wall transformer:
  Connect 9V wall transformer to power and plug into power jack adapter which installs in 2-pin screw terminal

See diagram below for installation example.


Debug serial cable installed on SER0 (H2)
with red edge of cable aligned with pin 1.
Other end to PC's COMx port.
Location of
watchdog jumper
2GB
CompactFlash®

Chapter 3: Hardware

3.1 *SC520 - Introduction*

The 586-Engine-P is based on AMD Elan SC520 Microcontroller. It includes an industry-standard Am5x86 CPU with floating point unit (FPU). It provides a General-Purpose (GP) bus with programmable timing for 8 and 16-bit devices. A ROM/Flash controller supports on-board high performance code execution. An enhanced programmable interrupt controller (PIC) prioritizes 22 interrupt levels with up to 15 external sources. Two asynchronous UARTs can operate up to 1.15 M bit/s. A **S**ynchronous **S**erial **I**nterface (SSI) offers full-duplex or half-duplex operation to support on-board ADC/DACs and user expansion. A real time clock, a software timer, 3 GP timers, and 3 programmable interval timers are all included. Thirty-two programmable I/O pins are on-board.

Please refer to the SC520 User's Manual, SC520 Data Sheet, and SC520 Register Set Manual included on TERN's CD: **amd_docs\sc520**.

3.2 *SC520 – Features*

3.2.1 Clock

One 32.768 KHz and one 33.333 MHz crystal are installed to provide all the clocks required for CPU, Real time clock, UART, timers, and clock output.

The CLKTEST (CLKT) signal is routed to J1 pin 4. Software can select to output one of 6 internal clocks, including 32.768K, 1.8443 MHz, 18.432 MHz, 1.1882 MHz, 1.47456 MHz, and 36.864 MHz.

On-board ADC (U12) can use 1.8432 MHz as ADC clock.

3.2.2 Programmable interrupt controller and external Interrupts

The Programmable Interrupt Controller (PIC) prioritizes 22 interrupt levels (P1-P22) with up to 15 external sources (GPIRQ0-10 and INTA-D).

A programmable router must be programmed to map internal or external interrupt sources to the master or two of slave interrupt controllers to provide different priorities, from P1 to P22.

All 15 external interrupt requests are programmed as edge sensitive, after 586_init();

An example map for P1 to P22 is listed below and demonstrated in the sample program **586_intx.c** and is included in the pre-built sample project **\tern\586\test.ide**.

```
//    There are 22 interrupt priority levels plus NMI
//    There are 15 external interrupt requests (GPIRQ0-10, /INTA-D)
//    Example internal interrupt map by TERN:
//    P1=Master PIC IR0, interrupt vector=0x40, PIT timer0
//    P2=Master PIC IR1, interrupt vector=0x41, GPIRQ0=PIO23=J2.33
//    P3=slave1 PIC IR0, interrupt vector=0x48, RTC
//    P4=slave1 PIC IR1, interrupt vector=0x49, GPIRQ1=PIO22=J2.23
//    P5=slave1 PIC IR2, interrupt vector=0x4a, GPIRQ2=PIO21=J2.21
//    P6=slave1 PIC IR3, interrupt vector=0x4b, GPIRQ3=PIO20=J2.19
//    P7=slave1 PIC IR4, interrupt vector=0x4c, GPIRQ4=PIO19=J2.20
//    P8=slave1 PIC IR5, interrupt vector=0x4d, FPU
//    P9=slave1 PIC IR6, interrupt vector=0x4e, /INTD=SCC=J3.14
//    P10=slave1 PIC IR7, interrupt vector=0x4f, GP timer1/INTC=J3.13
```

```
//    P11=Master PIC IR3, interrupt vector=0x43, SER2/0
//    P12=Master PIC IR4, interrupt vector=0x44, SER1
//    P13=Slave2 PIC IR0, interrupt vector=0x50, GP timer0
//    P14=Slave2 PIC IR1, interrupt vector=0x51, GPIRQ5=PIO18=J2.17
//    P15=Slave2 PIC IR2, interrupt vector=0x52, GPIRQ6=PIO17=J2.18
//    P16=Slave2 PIC IR3, interrupt vector=0x53, GPIRQ7=PIO16=J2.15
//    P17=Slave2 PIC IR4, interrupt vector=0x54, PIT timer1
//    P18=Slave2 PIC IR5, interrupt vector=0x55, GPIRQ8=PIO15=J2.16
//    P19=Slave2 PIC IR6, interrupt vector=0x56, GPIRQ9=PIO14=J2.6
//    P20=Slave2 PIC IR7, interrupt vector=0x57, GPIRQ10=PIO13=J2.8
//    P21=Master PIC IR6, interrupt vector=0x46, PIT Timer2/INTB=J3.12
//    P22=Master PIC IR7, interrupt vector=0x47, GP timer2/INTA=J3.11
```

See the sample program in c:\tern\586\samples\5e\586_intx.c for more details.

The 586-Engine-P uses vector interrupt functions to respond to external interrupts. Refer to the SC520 User's manual for more information about interrupt vectors.

3.2.3 Asynchronous Serial Ports

The SC520 has two 16450/16550-compatible asynchronous serial channels: SER0/2 and SER1. Both asynchronous serial ports support the following:

- Full-duplex operation,
- 5-, 6-, 7-, and 8-bit data transfers
- Odd, even, and no parity
- 1, 1.5, or 2 stop bits
- Error detection
- Hardware flow control
- Transmit and receive interrupts for each port
- Maximum baud rate, up to 1.152 MHz

The software drivers for each serial port implement a ring-buffered interrupt transmitting and receiving arrangement. See the samples files ***s1_echo.c*** and ***s0_echo.c;*** **s1_echo.c** is included in the pre-built sample project **\tern\586\test.ide**.

3.2.4 GP Timers

Three 16-bit General-Purpose Timers are on-board. Two external inputs, TIN0=J4.4 and TIN1=J4.6, can be used for the GP Timer0 and Timer1 to capture and count external pulses up to 33.333/4 MHz.

Timer 0 and Timer1 can output pulses on TOUT0=J4.3 and TOUT1=J4.5.

GP Timers support interrupt on terminal count, continuous mode, and square wave generation.

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.

See the sample programs ***timer02.c*** and ***tmr_out.c*** in the **`tern\586\samples\5e`** directory. GP timer code is intergrated into several of the samples in the sample project **\tern\586\test.ide.**

3.2.5 PIT Timers

Three 16-bit Programmable Interval Timers (PIT) are on-board. Each PIT channel has one interrupt output. Only PIT2 has an external output pin and can provide square wave output. All PITs supports interrupt on terminal count, hardware-retriggerable one-shot and timer functions. See samples at c:\tern\586\samples\5e directory for 586_pit0.c and pit2_out.c.

3.2.6 Software timers

The “software timer” is actually a hardware timer, which is intended to provide a millisecond timebase with microsecond resolution. Ideal applications for this function include providing a system wide timebase, or a precise measurement of the time between events. The software timer has a 16-bit microsecond counter that increments with a period of one millisecond. This yields a maximum duration of 65.5 seconds. A microsecond latch register that provides the number of microseconds since the last time that the millisecond register was read.

The software timer provides a very efficient hardware timerbase for use by software. It is designed to replace the traditional method of system timebase generation.

Traditional system timebase generation is accomplished by programming a timer to generate a periodic interrupt. The interrupt service routine then increments a counter. This value is often kept in a global variable, which can then be accessed by other code that needs to track time. The problem with this traditional timebase is caused by interrupt latency and possible missed interrupt.

The software timer included can be used to resolve these problems.

See more details on AMD SC520 Users’ Manual, chapter 18.

3.2.7 SSI

A synchronous serial interface (SSI) provides full-duplex and half-duplex, bi-directional communication at a software programmable SSI clock speed, from 64K Hz to 8MHz.

586-Engine-P uses the SSI to interface to the TLC2600 DAC. The user can use the SSI to interface many types of external serial peripheral devices.

See the sample c:\tern\586\samples\5e\ssio.c.

3.2.8 RTC

A battery backed up real-time clock (RTC) is included. The RTC consists of time-of-day clock with alarm and a 100-year calendar. It has also a programmable periodic interrupt and 114 bytes of static user RAM.When the 5P is powered off, the RTC (and 114 bytes of static RAM) is backed-up by the 3V battery installed on the 586-Engine-P.

See samples 586_rtc.c and rtc_pint.c for more details.

3.2.9 Watchdog timer

The Watchdog timer included in SC520 is not disabled. The 586-Engine-P uses a 691 supervisor chip to monitor the 5V power and provides an external watchdog. User can activate the 691 watchdog with a jumper setting on H1. The watchdog timer provided a means to monitor proper software execution. If software becomes stuck in an infinite loop, for example, the watchdog timer can allow the 5P to recover to proper execution. If the watchdog timer is enabled (via setting jumper at H1), it must be reset via software every 1.6 seconds, or sooner. If the watchdog timer is not reset after 1.6 seconds, the 691 supervisor will assert /RST, and reset the 5P. To reset the watchdog timer, use the routine, **hitwd()**. With the watchdog jumper enabled, this routine should be arragned such that it is called every 1.6 seconds are sooner.

3.2.10 PCI, DMA, SDRAM, Write/Read buffer

No SDRAM support on the 586-Engine-P. No support on PCI, DMA, and Write/Read buffer.

3.2.11 SC520 PIOs

The SC520 supports 32 user-programmable I/O lines (PIO). Each of these pins can be used as a user-programmable input or output signal, if the interface function is not needed.

The 586-Engine-P PIO pins are 3.3V output and all inputs are 5V tolerant. Absolutely no voltage greater than 5V should be applied to any pins. Over 5V voltage input can cause permanent damage.

After power-on/reset, PIO pins default to various configurations. The initialization routine, sc_init();, provided by TERN libraries reconfigures some of these pins as needed as:
P27=/GPCS0=J2.37 for 16-bit I/O operation of on-board ADC/DAC
P31=J2.38 as input for STEP2 jumper reading
P0 as output for on-board LED control
P1=/GPBHE=J1.11 as /BHE for 16-bit data bus high byte enable signal

Other 28 PIO pins on the J2 header are free to use. PIO 2-26 and PIO 28,29,30 A PIO line can be configured to operate as an output or an input with a weak internal pull-up or pull-down resistor. A PIO pin's behavior, either pull-up or pull-down, is pre-determined and shown in the table below.
These configurations, as well as the processor-internal peripheral usage configurations, are listed below in Table 3.1.

PIO	***Function***	***Power-On/Reset status***	***586-Engine-P Pin No.***	***586-Engine-P Initial***
P0	GPALE	Input with pull-up	LED L1 pin 2	Output for LED control
P1	/GPBHE	Input with pull-up	J1 pin 11	High byte enable /BHE
P2	GPRDY	Input with pull-up	J2 pin 4	Input with pull-up
P3	GPAEN	Input with pull-up	J2 pin 1	Input with pull-up
P4	GPTC	Input with pull-up	J2 pin 3	Input with pull-up
P5	GPDRQ3	Input with pull-down	J2 pin 5	Input with pull-down
P6	GPDRQ2	Input with pull-down	J2 pin 7	Input with pull-down
P7	GPDRQ1	Input with pull-down	J2 pin 10	Input with pull-down
P8	GPDRQ0	Input with pull-down	J2 pin 9	Input with pull-down
P9	/GPDACK3	Input with pull-up	J2 pin 12	Input with pull-up
P10	/GPDACK2	Input with pull-up	J2 pin 11	Input with pull-up
P11	/GPDACK1	Input with pull-up	J2 pin 14	Input with pull-up
P12	/GPDACK0	Input with pull-up	J2 pin 13	Input with pull-up
P13	GPIRQ10	Input with pull-up	J2 pin 8	Input with pull-up
P14	GPIRQ9	Input with pull-up	J2 pin 6	Input with pull-up
P15	GPIRQ8	Input with pull-up	J2 pin 16	Input with pull-up
P16	GPIRQ7	Input with pull-up	J2 pin 15	Input with pull-up
P17	GPIRQ6	Input with pull-up	J2 pin 18	Input with pull-up
P18	GPIRQ5	Input with pull-up	J2 pin 17	Input with pull-up
P19	GPIRQ4	Input with pull-up	J2 pin 20	Input with pull-up
P20	GPIRQ3	Input with pull-up	J2 pin 19	Input with pull-up
P21	GPIRQ2	Input with pull-up	J2 pin 21	Input with pull-up
P22	GPIRQ1	Input with pull-up	J2 pin 23	Input with pull-up
P23	GPIRQ0	Input with pull-up	J2 pin 33	Input with pull-up
P24	/GPBUFOE	Input with pull-up	J2 pin 24	Input with pull-up
P25	/GPIOCS16	Input with pull-up	J2 pin 25	Input with pull-up
P26	/GPMCS16	Input with pull-up	J2 pin 29	Input with pull-up*
P27	/GPCS0	Input with pull-up	J2 pin 37	16-bit I/O operation
P28	/CTS2	Input with pull-up	J2 pin 36	Input with pull-up
P29	/DSR2	Input with pull-up	J2 pin 35	Input with pull-up

PIO	*Function*	*Power-On/Reset status*	*586-Engine-P Pin No.*	*586-Engine-P Initial*
P30	/DCD2	Input with pull-up	J2 pin 30	Input with pull-up
P31	/RIN2	Input with pull-up	J2 pin 38	STEP2 Jumper

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

C function in the library **586.lib** can be used to initialize and to operate PIO pins.

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

Where bit = 0-31 and mode = 0 (for interface function), 1 (for input), or 2 (for output).

Example: ***pio_init***(0, 2); will set P0 as output

pio_init(1, 0); will set P1 as /GPBHE

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

pio_wr(0,1); set P0 pin high and the LED is off, if P0 is in output mode

pio_wr(0,0); set P0 pin low and the LED is on, if P0 is in output mode

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

pio_rd (0); return 16-bit status of P0-P15, if corresponding pin is in input mode,

pio_rd (1); returns 16-bit status of P16-P31, if corresponding pin is in input mode,

Some of the I/O lines are used by the 586-Engine-P 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
P0	Output	LED control
P1	/GPBHE	High byte D15-D8 data enable of the 16-bit data bus
P27	/GPCS0	General purpose chip select for 16-bit I/O operation
P31	Input	STEP2 jumper

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

3.3 *I/O Mapped Devices*

3.3.1 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 GP bus timing is setup in sc_init(); as:

```
pokeb(MMCR,   _GPCSRT_, 0x01); // set the GP CS recovery time
pokeb(MMCR,   _GPCSPW_, 0x1f); // set the GP CS width
pokeb(MMCR,   _GPCSOFF_, 0x01); // set the GP CS offset
pokeb(MMCR,   _GPRDW_, 0x1f); // set the GP RD pulse width
pokeb(MMCR,   _GPRDOFF_, 0x0); // set the GP RD offset
```

```
pokeb(MMCR,    _GPWRW_, 0x1f); // set the GP WR pulse width
pokeb(MMCR,    _GPWROFF_, 0x0); // set the GP WR offset
```

User may modify the GP bus timing after sc_init();. Details regarding this can be found in the SC520 User's Manual and SC520 Register Set Manual. Slower components, such as most LCD interfaces, might find the maximum programmable wait state of 15 cycles still insufficient.

The 5P uses J4.1=/PITG2=/GPCS3 to select on-board peripherals via the U2 decoder. The base address is 0x2000. The table below shows the I/O mapping for each peripheral.

I/O space	Select	Location	Usage
0x2000-0x2004	/AD	U2.15 & U6.32	AD7655 chip select
0x2020	/CV	U6.35	Begin conversion on ADC
0x2040	HIT	H1.1	Reset watchdog timer
0x20E0	/CF	U5.32	CompactFlash chip select

3.3.2 Eight channel 16-bit DAC (TLC2600)

The TLC2600 is an eight channel 16-bit digital-to-analog converter (DAC) in an SO-8 package. It is complete with a rail-to-rail voltage output amplifier capable of driving up to 15mA. It uses a 3-wire SPI compatable serial interface and has an output range of 0-REF volts, making 1 LSB equal to REF/65535 V. The reference voltage input is routed to J4 pin 11 and by default is shorted to VCC via jumper at J4.11=J4.9. The REF voltage must be greater than GND and less than VCC. The DAC outputs are routed to the J4 pin header, pins 12-19.

The DAC is installed on the 586-Engine-P at location U8 and uses P3 as the chip select. The synchronous serial interface is used to send data to the device. Refer to the sample code, \tern\586\samples\5p\5p_da.c for an example on driving the DAC. The sample is also included in the pre-built sample project \tern\586\samples\5p\5p_cf.ide.

Refer to the DAC data sheet for additional specifications; \tern_docs\parts\ltc2600.pdf.

3.3.3 Four channel, 16-bit ADC

The unique 16-bit parallel ADC (AD7655, 0-5V) supports ultra high-speed (1 MHz conversion rate) analog signal acquisition. The AD7655 contains two low noise, high bandwidth track-and-hold amplifiers that allow *simultaneous* sampling on two channels. Each track-and hold amplifier has a multiplexer in front to provide a total of 4 channels analog inputs. The parallel ADC achieves very high throughput by requiring only two CPU I/O operations (one start, one read) to complete a 16-bit ADC reading. With a precision external 2.5V reference, the ADC accepts 0-5V analog inputs at 16-bit resolution of 0-65,535.

See sample program \tern\586\samples\5p\5p_ad.c for details on reading the ADC. The sample program is also included in the pre-built sample project; \tern\586\samples\5p\5p_cf.ide.

Refer to the data sheet for additional specifications; \tern_docs\parts\ad7655.pdf.

3.4 *Power supplies and Supervisor with Watchdog Timer*

Two supervisor chips monitor 5V and 3.3V and provide power failure detection, a watchdog and system reset. The 2.5V power supply is used for the SC520 core and 3.3V supports SC520 I/O operation. Signal lines on headers are 3.3V output, and 5V maximum input. Absolutely no voltage greater than 5V should be applied to any pins. The 388 pin BGA package of SC520 makes repair support not available. All components are soldered on board for highest reliability.
The ***586-Engine-P™*** can be powered with a single regulated 5V with the on-board 3.3V and 2.5V regulators. Limited by the compact dimension of the 586-Engine-P, the 2.5V and 3.3V on-board regulators may cause excessive heat in a closed enclosure. External off-board regulated 5V, 3.3V, and 2.5V power supplies can power the 586-Engine-P, in order to remove heat from the board. The 586-Engine-P also includes an on-board switching regulator to provide 5V from an unregulated 8-30V input.

A 691 (U7) supervisor chip is used to monitor the 5V power and a MIC8114 (U4) is designed to monitor the 3.3V. The supervisor provides a watchdog timer, battery backup, power-on-reset delay, power-supply monitoring, and power-failure warning. These will significantly improve system reliability.

Watchdog Timer

Setting a jumper on H1 activates the 691 watchdog timer. 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 H1 pin1= HIT) should be arranged such that the HIT pin is accessed at least once every 1.6 seconds. If the H1 jumper is installed and the HIT pin is not accessed within this time-out period, the watchdog timer pulls the WDO pin low, and asserts /RST. This automatic assertion of /RST may recover the application program if something is wrong. When controllers are shipped from the factory the H1 jumper is off, which disables the watchdog timer. See diagram on page 2-7 of this manual for location on watchdod enable jumper.

The SC520's internal watchdog timer is disabled by default with **sc_init()**.

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 *Headers and Connectors*

The 586-Engine-P schematic overrdes any discrepancies with the tables below.

3.5.1 Expansion Headers J1 and J2

There are two 20x2 0.1 spacing headers for 586-Engine-P expansion. Most signals are directly routed to the SC520 processor. These signals are 3.3V and 5V tolerant, and any out-of-range voltages will damage the board.

3.5.2 Header J4

The J4 header on the 586-Engine-P provides the user with access to DAC outputs, ADC inputs, GP timer I/O, plus others.

J2 Signal

Name	Pin #	Pin #	Name
GND	40	39	VCC
P31	38	37	P27
P28	36	35	P29
TxD0	34	33	P23
RxD0	32	31	/RTS1
P30	30	29	P26
TxD1	28	27	/RTS0
RxD1	26	25	P25
P24	24	23	P22
/CTS1	22	21	P21
P19	20	19	P20
P17	18	17	P18
P15	16	15	P16
P11	14	13	P12
P9	12	11	P10
P7	10	9	P8
P13	8	7	P6
P14	6	5	P5
P2	4	3	P4
GND	2	1	P3

J1 signal

Name	Pin #	Pin #	Name
VCC	1	2	GND
MPO	3	4	CLKT
RxD	5	6	GND
TxD	7	8	D0
VOFF	9	10	D1
/BHE	11	12	D2
D15	13	14	D3
/RST	15	16	D4
RST	17	18	D5
/ROM1	19	20	D6
D14	21	22	D7
D13	23	24	GND
	25	26	A7
D12	27	28	A6
/IOWR	29	30	A5
/IORD	31	32	A4
D11	33	34	A3
D10	35	36	A2
D9	37	38	A1
D8	39	40	A0

J4 signal

Name	Pin #	Pin #	Name
/PITG2	1	2	/PITO2
TOUT0	3	4	TIN0
TOUT1	5	6	TIN1
SSO	7	8	SSC
VCC	9	10	SSI
REF1	11	12	V1
V2	13	14	V3
V4	15	16	V5
V6	17	18	V7
V8	19	20	GND
AB2	21	22	AB1
AA2	23	24	AA1
REF	25	26	GND

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 the Software Glossary in Appendix F.

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

4.1 586.LIB

586.LIB is a C library for basic 586-Engine-P operations. It includes the following modules: 586.OBJ, SER0.OBJ, SER1.OBJ, and SCC.OBJ. You need to link 586.LIB in your applications and include the corresponding header files. The following is a list of the header files:

Include-file name	Description
586.H	timer/counter, ADC, DAC, RTC, Watchdog
SER0.H	Internal serial port 0/2
SER1.H	Internal serial port 1
SCC.H	External UART SCC2691

The 586.LIB was originally developed for the 586-Engine (the predecessor to the 586-Einge-P). Function prototypes for the 586-Engine's ADC and DAC were incorporated into 586.lib. The 586-Engine-P does not uses these same routines for its analog I/O. You must refer to the sample code in the 586\samples\5p directory for examples on accessing the analog I/O on the 586-Engine-P. In addition, the 586-Engine-P does not support an SCC2961 UART. Thus the functions defined in 'scc.h' are not valid.

4.2 Functions in 586.OBJ

4.2.1 586-Engine-P Initialization

sc_init

This function should be called at the beginning of every program running on 586-Engine-P. It provides default initialization and configuration of the various I/O pins, interrupt vectors, sets up 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 **sc_init** are described below. For details regarding register use, you will want to refer to the AMD SC520 Microcontroller User's manual.

- Initialize the programmable interrupt controller. Setup the master interrupt controller at vector 0x40. The slave1 interrupt vector at 0x48, and slave 2 interrupt vector at 0x50.
- Initialize /ROMCS1 chip select to support the 8-bit I/O starting 0x1000, and the /GPCS0 to support 16-bit I/O starting address at 0x1800.
- Disable SDRAM
- Initialize default GP-bus chip select timing as

```
pokeb(MMCR, _GPCSRT_, 0x01); // set the GP CS recovery time
pokeb(MMCR, _GPCSPW_, 0x1f); // set the GP CS width
pokeb(MMCR, _GPCSOFF_, 0x01); // set the GP CS offset
pokeb(MMCR, _GPRDW_, 0x1f); // set the GP RD pulse width
pokeb(MMCR, _GPRDOFF_, 0x0); // set the GP RD offset
pokeb(MMCR, _GPWRW_, 0x1f); // set the GP WR pulse width
pokeb(MMCR, _GPWROFF_, 0x0); // set the GP WR offset
```

- Initialize and configure PIO ports for default operation. All pins are set up as default input, except for P31, P27, P2, and P0.

The GP chip selects are set to 0x1f wait states, by default. 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.
A CLKT signal is routed to J1 pin 4 for the on-board ADC and external user clock. The CLKT can be selected as

void clkt_sel **Arguments:** unsigned char clk **Return value:** none.
CLKT=J1.4 output 1.8432 MHz as ADC clock for U12 The CLKT pin is programmed as an output (CLKTEST). When programmed as output, CLKT output one of the 6 internal clocks: void clkt_sel(unsigned char clk); where: clk=000, RTC (32.768 KHz) clk=001, UART (1.8443 MHz) clk=010, UART (18.432 MHz) clk=011, PIT (1.1882 MHz) clk=100, PLL1 (1.47456 MHz) clk=101, PLL2 (36.864 MHz) clk=110-111, CLKT=0

```
void clkt_sel(unsigned char clk);
```

4.2.2 External Interrupt Initialization

The programmable interrupt controller consistes of a system of three individual interrupt controllers (Master, Slave1, and Slave2), each has eight interrupt channels. A total of 22 interrupt priority levels are supported. A programmable interrupt router handles routing of various external and internal interrupt sources to the 22 interuupt channels. TERN recommends an interrupt map as follows;

```
There are 22 interrupt priority levels plus NMI
There are 15 external interrupt requests (GPIRQ0-10, /INTA-D). You must
provide a low to high edge to generate an interrupt
```

Example internal interrupt map by TERN:

P1=Master PIC IR0, interrupt vector=0x40, PIT timer0
P2=Master PIC IR1, interrupt vector=0x41, GPIRQ0=PIO23=J2.33
P3=slave1 PIC IR0, interrupt vector=0x48, RTC
P4=slave1 PIC IR1, interrupt vector=0x49, GPIRQ1=PIO22=J2.23
P5=slave1 PIC IR2, interrupt vector=0x4a, GPIRQ2=PIO21=J2.21
P6=slave1 PIC IR3, interrupt vector=0x4b, GPIRQ3=PIO20=J2.19
P7=slave1 PIC IR4, interrupt vector=0x4c, GPIRQ4=PIO19=J2.20
P8=slave1 PIC IR5, interrupt vector=0x4d, FPU
P9=slave1 PIC IR6, interrupt vector=0x4e, /INTD=SCC=J3.14
P10=slave1 PIC IR7, interrupt vector=0x4f, GP timer1/INTC=J3.13
P11=Master PIC IR3, interrupt vector=0x43, SER2/0
P12=Master PIC IR4, interrupt vector=0x44, SER1
P13=Slave2 PIC IR0, interrupt vector=0x50, GP timer0
P14=Slave2 PIC IR1, interrupt vector=0x51, GPIRQ5=PIO18=J2.17
P15=Slave2 PIC IR2, interrupt vector=0x52, GPIRQ6=PIO17=J2.18
P16=Slave2 PIC IR3, interrupt vector=0x53, GPIRQ7=PIO16=J2.15
P17=Slave2 PIC IR4, interrupt vector=0x54, PIT timer1
P18=Slave2 PIC IR5, interrupt vector=0x55, GPIRQ8=PIO15=J2.16
P19=Slave2 PIC IR6, interrupt vector=0x56, GPIRQ9=PIO14=J2.6
P20=Slave2 PIC IR7, interrupt vector=0x57, GPIRQ10=PIO13=J2.8
P21=Master PIC IR6, interrupt vector=0x46, PIT Timer2/INTB=J3.12
P22=Master PIC IR7, interrupt vector=0x47, /INTA=J3.11

A spurious interrupt is defined as a "Not Valid" interrupt.
A Spurious Interrupt on any IR line generates the same vector number
as an IR7 request. The spurious interrupt, however, does not set the
in-service bit for IR7. Therefore, an IR7 isr must check the isr register to determine the interrupt source was a
valid IR7 (the in-service bit is set),
or a spurious interrupt (the in-service bit is cleared)

Functions

```
void    nmi_init(void); // nmi interrupt handler initialization
void    int0_init(char i, void interrupt far (* int0_isr)())
void    int1_init(char i, void interrupt far (* int1_isr)())
void    int2_init(char i, void interrupt far (* int2_isr)())
void    int3_init(char i, void interrupt far (* int3_isr)())
void    int4_init(char i, void interrupt far (* int4_isr)())
void    int5_init(char i, void interrupt far (* int5_isr)())
void    int6_init(char i, void interrupt far (* int6_isr)())
```

```
void    int7_init(char i, void interrupt far (* int7_isr)())
void    int8_init(char i, void interrupt far (* int8_isr)())
void    int9_init(char i, void interrupt far (* int9_isr)())
void    intD_init(char i, void interrupt far (* intD_isr)())
```

For a detailed discussion involving the interrupt, the user should refer to Chapter 15 of the AMD SC520 Microcontroller User's Manual.

TERN provides functions to enable/disable all of the external interrupts. The user can call any of the interrupt init functions listed 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.

void int*x*_init
Arguments: unsigned char i, void interrupt far(* int*x*_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, when executed, is about 20 μs.

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.

I/O Initialization

Two ports of 16 I/O pins each are available on the 586-Engine-P. Hardware details regarding these PIO lines can be found in the Hardware chapter.

Several functions are 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 three 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 **sc_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 on the I/O ports, please refer to Chapter 23 of the AMD SC520 User's Manual.

Please see the sample program **586_pio.c** in **tern\586\samples\5e**. 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 **poke** instruction Performance in this case will be around 1-2 us to toggle any pin. For example: poke(MMCR, _PIOSET15_0_, m);

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 3 modes of operation.

- 0, Interface operation
- 1, input with pullup/down
- 2, output

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.

4.2.3 GPTimer Units

The three GP timers present on the 586-Engine-P can be used for a variety of applications

These timers are controlled and configured through a mode register that is specified using the software interfaces. The mode register is described in detail in chapter 17 of the AMD SC520 User's Manual.

Two of the timers, **Timer0** and **Timer1** has external pulses output and counter inputs.

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. The sample files ***timer02.c*** and ***timer12.c***, located in ***tern\586\samples\5e***, demonstrate this.

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.

4.2.4 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 (**H1**) jumper is set, 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

A real-time clock is included in the SC520, and 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 are two common data structure used to access and use both interfaces.

```
// SC520 RTC data structure
typedef struct{
        unsigned char sec;
        unsigned char alarm_sec;
        unsigned char min;
        unsigned char alarm_min;
        unsigned char hour;
        unsigned char alarm_hour;
        unsigned char day_week;
        unsigned char day_month;
        unsigned char month;
        unsigned char year;
} RTCTIME;
// Real time data structure
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.
int rtc_rds
Arguments: char* realTime
Return value: int error_code
This function places a string of the current value of the real time clock in the char* realTime.
The text string has a format of "year1000 year100 year10 year1 month10 month1 day10 day1 hour10 hour1 min10 min1 second10 second1". For example" 19991220081020" represents year 1999, December 20th, Eight o'clock, 10 minutes, and 20 seconds.
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, RTCTIME *rtcp
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 RTCTIME data structure will be initialized based on the string **t.**
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, (Software Timer of SC520) 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. NOT accurate at all.

void sc_rst
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\586\include`**.

The internal asynchronous serial ports are functionally identical. SER0 is used by the debug kernel, which is loaded into the surface-mount flash and provided as part of the TERN EV/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, but you can run it in STEP2.

Two asynchronous serial ports are integrated in the SC520: SER0 and SER1. Both ports have baud rates based on the system clock, and can operate at a maximum of 1.152 Mbaud.

By default, SER0 is used by the DEBUG ROM for application download/debugging in STEP 1. We will use SER1 as the example in the following discussion; any of the interface functions that are specific to SER1 can be easily changed into function calls for SER0. For details, you should see both chapter 21 of the SC520 Microprocessor User's Manual and the schematic of the 586-Engine-P 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. The following table shows the function arguments that express each baud rate, to be used in TERN functions. These are based on a 33.333 MHz external crystal. Note: Only up to 115,200 BAUD has been tested in house. (10/25/00)

Function Argument	Baud Rate
1	300
2	600
3	2400
4	4800
5	7200
6	9600
7	14,400
8	19,200
9	38,400
10	57,600

Function Argument	Baud Rate
11	115,200
12	114,000
13	192,000
14	288,000
15	576,000
16	1,152,000

Table 4.1 Baud rate values

After initialization by calling **`s1_init()`**, SER1 is configured as a full-duplex interrupt-driven serial port and is ready to transmit/receive serial data at one of the specified 16 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. 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

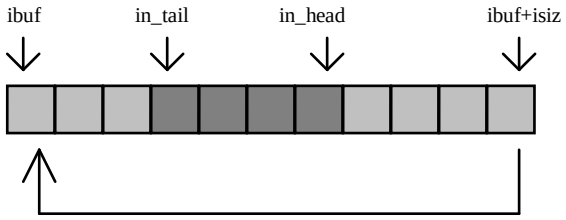


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), and baud rate (**baud**) are specified by the user with **`s1_init()`** with a default mode of 8-bit, 1 stop bit, no parity. After **`s1_init()`** you can set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port 0/1 Control Register if necessary, as described in chapter 21 of the SC520 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 **`s1_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, **`s1_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\586\samples\5e`**.

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 tmso;   // 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_segm;       /* input buffer segment */
  unsigned int in_offs;       /* input buffer offset */
  unsigned int out_segm;       /* output buffer segment */
```

```
  unsigned int out_offs;       /* output buffer offset */
  unsigned char byte_delay;   /* V25 macro service byte delay */
} COM;
```

s*n*_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 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.

putser*n*
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.

putsers*n*
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.

serhit*n*() should be called before trying to retrieve data.

serhit*n*
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.

getser*n*
Arguments: COM *c
Return value: unsigned char value

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

getsers*n*
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 SC520 User's Manual.

char s*n*_cts(void)
Retrieves value of **CTS** pin.

void s*n*_rts(char b)
Sets the value of **RTS** to **b**.

Completing Serial Communications

After completing your serial communications, you can re-initialize the serial port with s1_init(); to reset default system resources.

s*n*_close
Arguments: COM *c
Return value: none

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

The asynchronous serial I/O ports available on the SC520 have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 21 of the manual for a detailed discussion of other features available to you.

4.4 Functions / Routines not in 586.lib

4.4.1 4 channel 16-bit ADC, AD7655

The ADC is accessed using a few different control lines. The control lines are summarized below:

Control Line	Description / Function
/CV	Begin conversion. I/O address 0x2020
/AD	Read conversion. 0x2000 for AB1, AB2 and 0x2004 for AA1 and AA2
P2	MUX select. P2 high selects AA2, AB2. P2 low selects AA1, AB1
A2	A2 = /AD&0x0004.

The following process must be followed to read the ADC:

pio_wr (2 , hi_lo) ;

inport (CV) ;

wait for 2.5 µs

inport (ADA) ;

inport (ADB) ;

Where

hi_lo	ADA	ADB	Read channel
0	A2 = 1		AA1
0		A2 = 0	AB1
1	A2 = 1		AA2
1		A2 = 0	AB2

Refer to the sample program \tern\586\samples\5p\5p_ad.c and data sheet \tern_docs\parts\ad7655.pdf for additional details. Also refer to the pre-built sample project \tern\586\samples\5p\5p_cf.ide.

4.4.2 8 channel 16-bit DAC

To access the DAC use the routine called ‘da_2600’ found in the \tern\586\samples\5p\5p_da.c sample program. You must copy the routine from the source file into you application, as it is not part of a library. The function ‘da_2600’ takes two arguments. The first is the channel to write to, with the valid range 0-7. The second is the output value, ranging from 0 – 65535. See the sample program for additional details.

4.4.3 Compact Flash Interface / File System Access

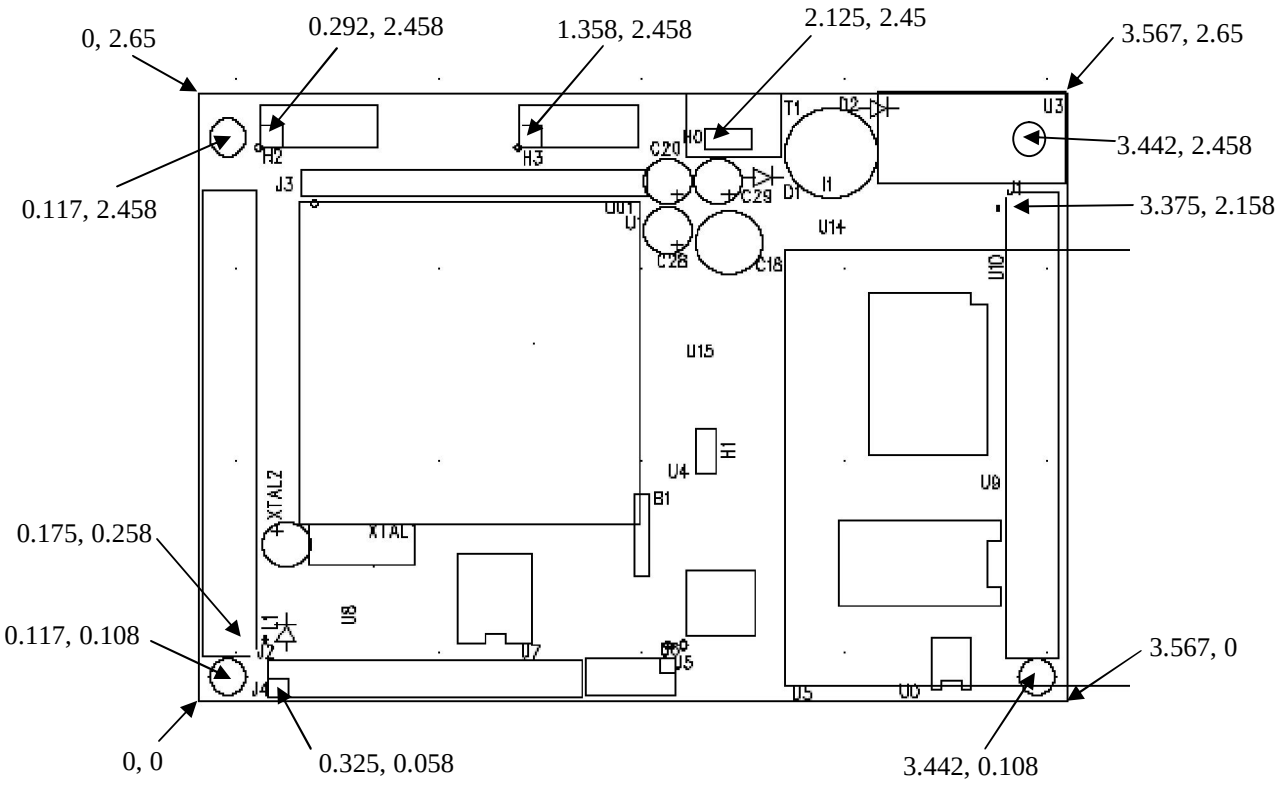
The 586-Engine-P offers a 50-pin compact flash interface. TERN libraries support FAT file systems access to/from the compact flash interface. This allows uses to log data with the 586-Engine-P and then conveniently access the file(s) with a PC.

Refer to \tern\586\include\fileio.h and \tern\586\include\filegeo.h for file system functions and descriptions. Also see \tern\586\samples\5p\586p.ide for a sample program accessing the compact flash interface. For the program in 586p.ide, you will need to install a FAT formatted CF (FAT32 is NOT allowable) card into the 586-Engine-P and link SER1 (H3) with a PC hyper terminal at 19,200, N, 8, 1.

More details about the filesystem library are contained in the file \tern\586\samples\flashcore\readme.txt

Appendix A: 586-Engine-P Layout

The **586-Engine-P** measures 3.567 by 2.65 inches. All dimensions are in inches. For mounting holes, arrows point to the center of the mounting hole. For pin headers, arrows point to pin 1 of the header, and to the center the pin.

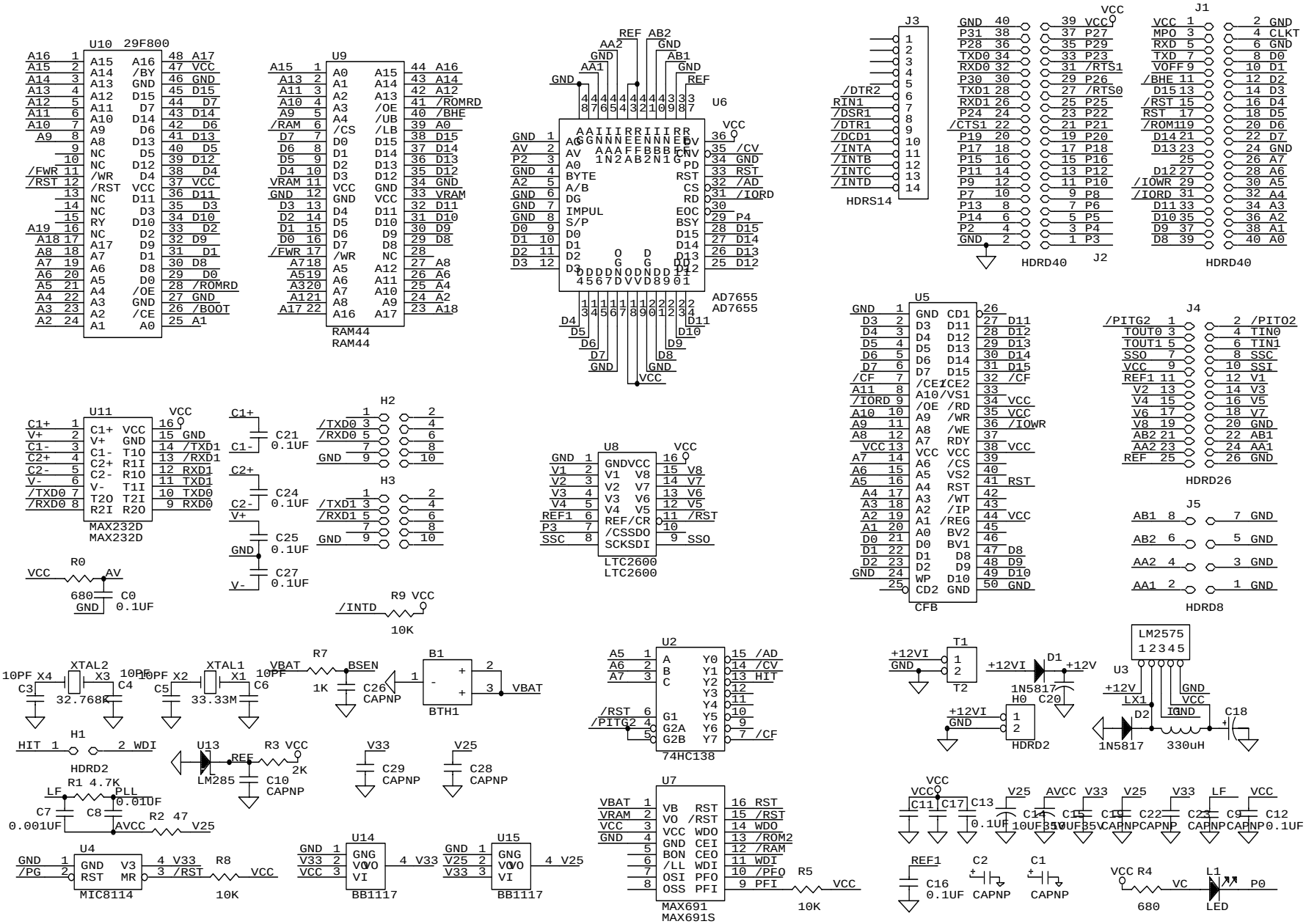


SC520- Internal CMOS RAM usage.

Part of the SC520 internal CMOS RAM locations are used by system software. Application programs must not use these locations.

```
//          With STEP2 Jumper on J2 pin 38-40,
//          586E will run the program starting address at CS:IP
//    There are 114 battery-backed(nonvolatile) CMOS RAM index 0x0E-0x7f
//    Default "Jump Address”=0x08000 for user application code in SRAM
//    Default “Jump Address”=0x80000 for application in Flash.
//
//    CMOS RAM mapping:
//                0x70        CS high= (0x08 for code in SRAM) or
//                                     (0x80 for code in Flash)
//                0x71        CS low=0
//                0x72        IP high=0
//                0x73        IP low=0
//
```

Use HyperTerminal, serial link at 19,200 baud, with no jumper installed.
You may use ACTF "G08000" to set CMOS RAM and run code starting 0x08000 in SRAM.


U10 29F800
U9
RAM44
RAM44
U6
AD7655
AD7655
J3
HDRS14
J1
HDRD40
HDRD40
J2
U5
CFB
J4
HDRD26
J5
HDRD8
U11
MAX232D
MAX232D
H2
H3
U8
LTC2600
LTC2600
R0
680
C0
0.1UF
R9
10K
/INTD
C21
C24
C25
C27
0.1UF
H1
HDRD2
U13
LM285
R3
2K
C10
CAPNP
R1 4.7K
C7
0.001UF
C8
0.01UF
R2 47
U4
MIC8114
10K
U14
BB1117
U15
BB1117
U2
74HC138
U7
MAX691
MAX691S
R5
10K
B1
BTH1
R7
1K
C26
CAPNP
T1
T2
D1
1N5817
U3
LM2575
1N5817
330uH
HDRD2
C18
C11
C17
C13
0.1UF
C14
10UF35V
C15
10UF35V
C22
CAPNP
C23
CAPNP
C9
CAPNP
C12
0.1UF
C16
0.1UF
C2
CAPNP
C1
CAPNP
R4
680
L1
LED
TERN/STE
SC520 586-ENGINE-P
5P-MAN.SCH
March 4, 2004
Sheet 1 of 1