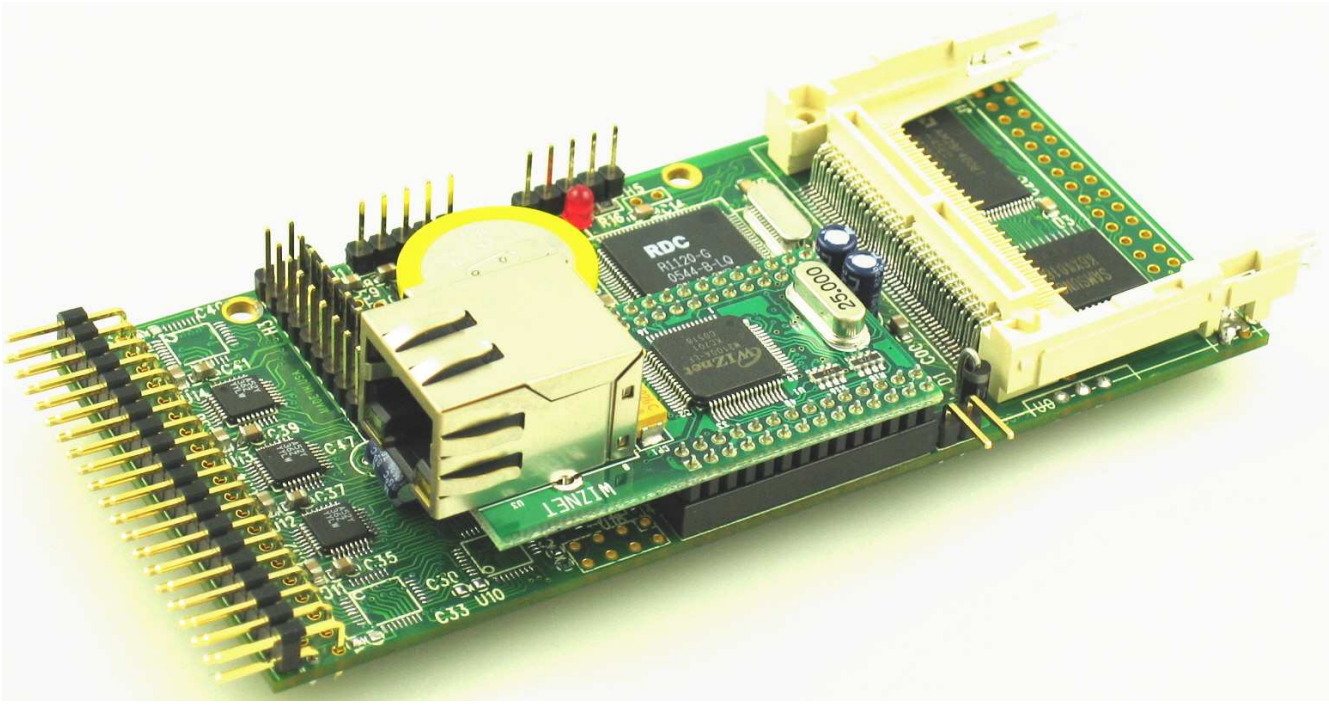
SensorCore-A™

48 12-bit ADCs, 16-bit DAC, 100M BaseT Ethernet, RS232, and CompactFlash



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

COPYRIGHT

SensorCore, E-Engine, A-Engine86, A-Engine, A-Core86, A-Core, i386-Engine, MemCard-A, MotionC, VE232, and ACTF are trademarks of TERN, Inc.
Am188ES and Am186ES are trademarks of Advanced Micro Devices, Inc.
Borland C/C++ is a trademark of Borland International.
Microsoft, MS-DOS, Windows, Windows95, and Windows98 are trademarks of Microsoft Corporation.
IBM is a trademark of International Business Machines Corporation.

Version 2.0

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.

© 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

The **SensorCore-A(*SCA*)™** is designed as a low-cost, low-power data logger for the most demanding analog data-acquisition applications. Featuring up to 48 channels of 12-bit ADC, 2 RS232 ports, CompactFlash interface, and a high performance 10/100M BaseT Ethernet, the ***SCA*** out-performs desktop-based acquisition solutions for a fraction of the price.

The ***SCA***'s unique profile allows it to be installed into difficult-to-access physical locations, such as pipes. Even with this limited real estate, the ***SCA*** is a full-featured, stand-alone industrial embedded controller.

The ***SCA*** is based on a high-performance C/C++ programmable x186ES CPU. It integrates 3 timer/counters, 2 async serial ports, external interrupts, PIOs, and a real-time clock (DS1337). The board is available with up to 512 KB of battery-backed SRAM, 512KB Flash, and 512 bytes EEPROM for non-volatile parameter storage.

The board runs on approximately 150mA at regulated 5V, and can be powered through onboard linear regulator accepting 9-12V DC. An optional low-drop regulator (TPS765) can be installed to provide a power-off feature allowing low voltage(5.1V) battery operation.

The ***SCA*** has two RS232 serial ports as default. The ***SCA*** also features an integrated high-performance 10/100-baseT hardware TCP/IP module, which allows 100KB+ access to TCP/IP networks with minimal CPU load. Sample implementations for the ***SCA*** allows it to be configured as a HTTP web-server, FTP server/client, etc.

The ***SCA*** also features six 300KHz 12-bit (8-channel) ADS7852 ADCs for a total of 48 channels in ADC conversion. Each ADS7852 chip offers 8 channels. A peak single-channel output rate of ~250-300 KHz can be achieved, with approximately at 3-4us read time per channel. Also available is the optional 8 channels 16-bit DAC analog outputs.

The ***SCA*** provides an integrated CompactFlash interface. Data can be written into CF at a sufficiently high enough data rate that the ***SCA*** can record sampling from all 3 chips indefinitely, even at peak sample rates. A 50-pin CompactFlash receptacle can be installed to allow access to mass storage CompactFlash cards (up to 2 GB). Users can easily add mass data storage to their embedded application. C/C++ programmable software package includes FAT16 file system libraries are available, this much room allows more than 1 billion 24-bit samples to be recorded in the field on a single board.

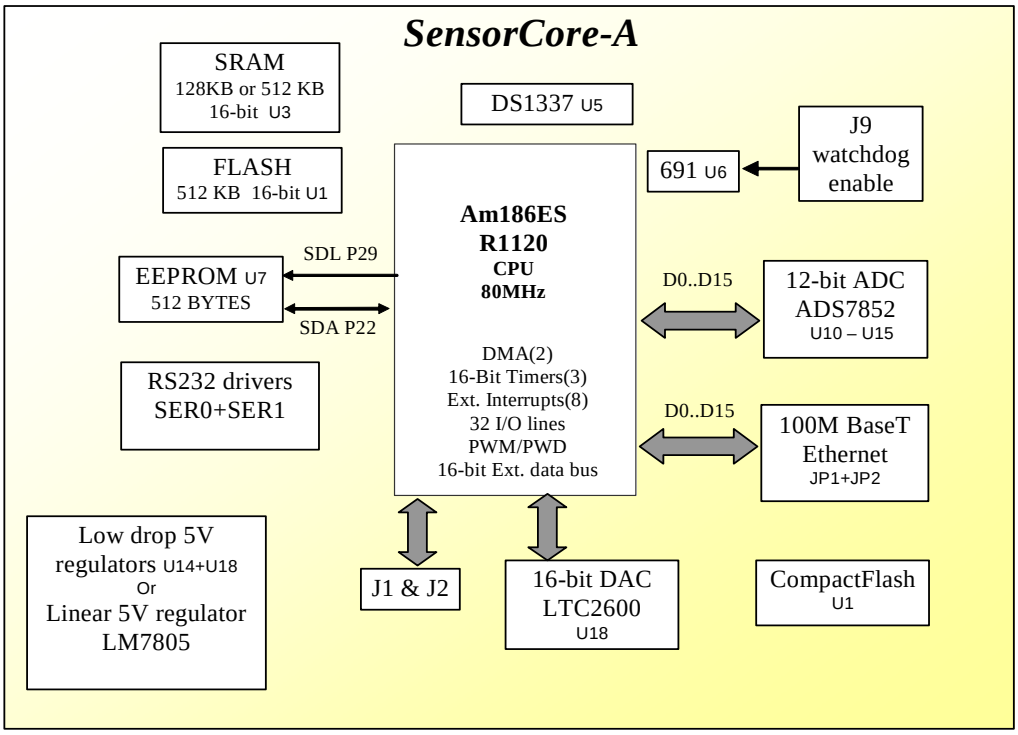


Figure 1.1 Functional block diagram of the SensorCore

The ***SCA*** supports on-board 512 KB 16-bit Flash and up to 512 KB 16-bit battery-backed SRAM. The on-board ACTF Flash has a protected boot loader and can be easily programmed in the field via serial link. Users can download a kernel into the Flash for remote debugging. With the DV-P Kit support, user application codes can be easily field-programmed into and run out of the Flash.

A 512-byte serial EEPROM is included on-board. Two serial ports from the R1120 support high-speed, reliable serial communication at a rate of up to 115,200 baud.

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 20 MHz (for 80 MHz ***SCA***), 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.

The CPU has 32 user-programmable, multifunctional I/Os. Schmitt-trigger inverters are provided for six external interrupt inputs, to increase noise immunity and transform slowly-changing input signals into fast-changing and jitter-free signals. A supervisor chip with power failure detection, a watchdog timer, an LED, and expansion ports are on-board.

Features:
* 150 mA, 9-12V DC power
* Complete C/C++ programmable environment
* 48 channels of 12-bit ADC input, 0-5V
* 8 channels 16-bit DAC (0-5V), 10 TTL I/Os
* CompactFlash with FAT file system support
* 80 MHz 186 CPU with 256 KW Flash, 256 KW SRAM
* 2 RS-232 serial ports; both RS232
* 3 16-bit timer/counters, PWM output, RTC, EE
* Hardware TCP/IP stack for 100M Based-T Ethernet

1.2 Physical Description

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

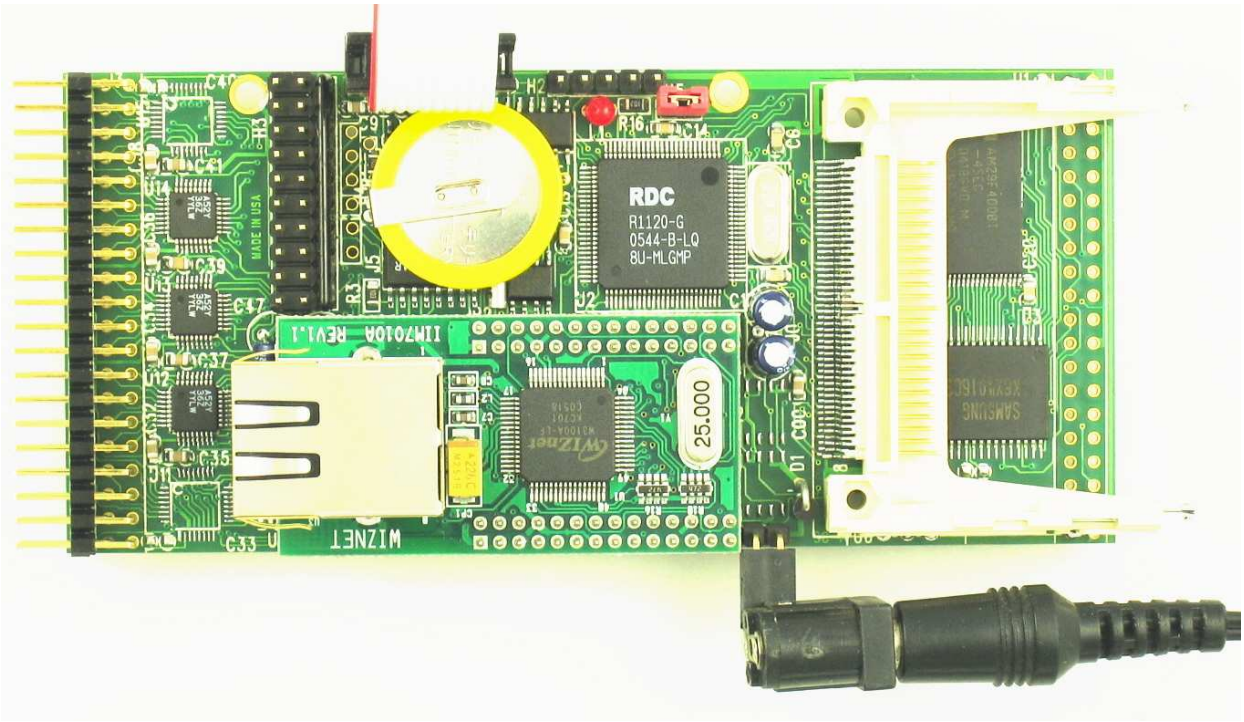


Figure 1.2 Physical layout of the SensorCore

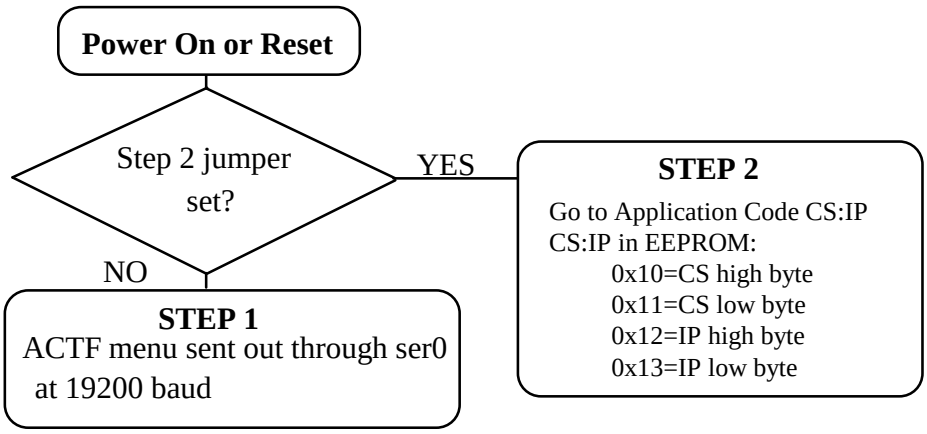


Figure 1.3 Flow chart for ACTF operation

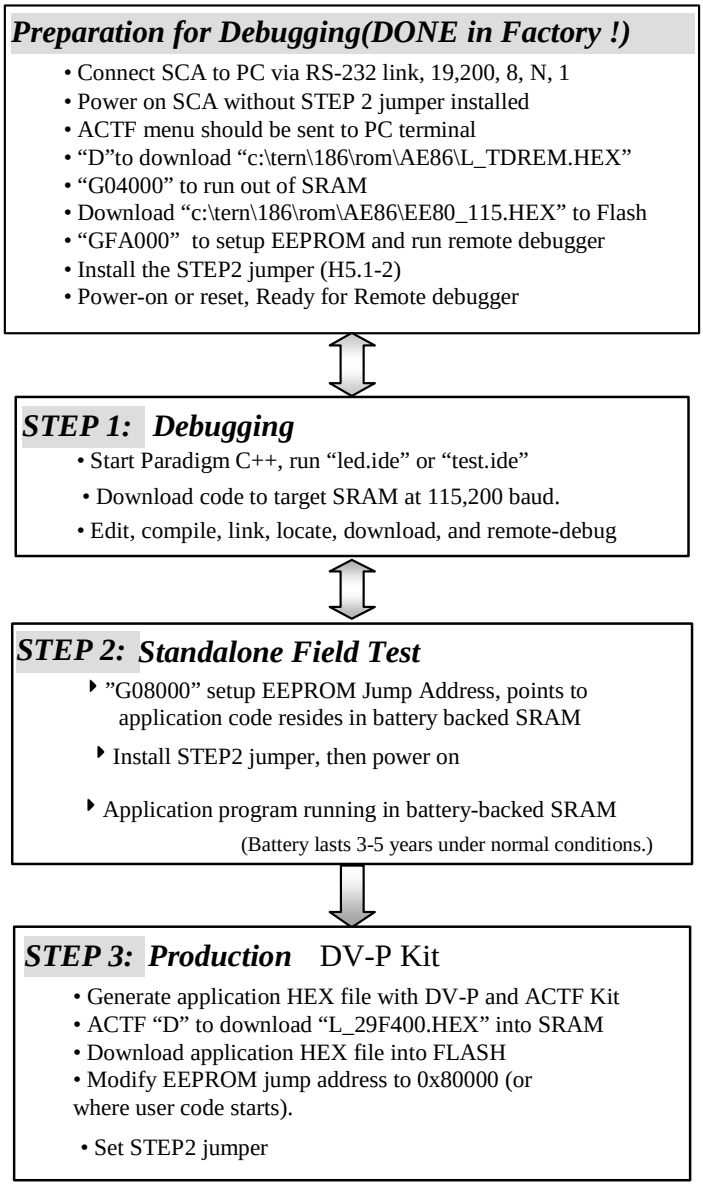
The "ACTF boot loader" resides in the top protected sector of the 512KB on-board Flash chip (29F400).

By default, in the factory, before shipping, the DEBUG kernel (EE80_115.hex) is pre-loaded in the Flash starting at 0xFA000, and the RED STEP2 jumper is installed on H5 pin 1-2, ready for Paradigm C++ debugger. User does not need to download a DEBUG kernel to start with.

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 port 0 at 9600 baud for a 80MHz ***SCA***.
If the STEP 2 jumper is installed, the "jump address" pre-programmed in the on-board serial EEPROM, will be read out and then jump to that address. A DEBUG kernel "EE80_115.hex" residing at "0xFA000" of the 512KB on-board flash chip.

1.3 SensorCore-A Programming Overview

Steps for product development:



There is no ROM socket on the ***SCA***. 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 Kit. The "STEP2" jumper (H5 pins 1-2) must be installed for every production-version board.

Step 1 settings

In order to talk to ***SCA*** with Paradign C++, the ***SCA*** must meet these requirements:

1) c:\tern\186\rom\AE86\EE80_115.HEX must be pre-loaded into Flash starting address 0xfa000.

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 EEPROM must have a Jump Address for the EE80_115.HEX with starting address of 0xfa000.

4) The STEP2 jumper must be installed on H5 pins 1-2.

For further information on programming the ***SCA***, refer to the manual on the TERN CD under: tern_docs\manuals\software_kit.pdf.

Chapter 2: Installation

2.1 Software Installation

Please refer to the “software_kit.pdf” technical manual on the TERN installation CD, under tern_docs\manual\software_kit.pdf, for information on installing software.

2.2 Hardware Installation

Overview

- Connect PC-IDE serial cable:
  For debugging (STEP 1), place IDE connector on SER0 (H1) with red edge of cable on side closest to H3 header(See Fig. 2.1). This DEBUG cable is a 10-pin IDE to DB9 cable, made by TERN.
- Connect wall transformer:
  Connect 9V wall transformer to power and plug into power jack using power jack adapter supplied with EV-P/DV-P Kit

Hardware installation consists primarily of connecting the microcontroller to your PC.

2.2.1 Connecting the SCA to the PC

Figure 2.1 provides the location of the debug serial port and the power jack. The ***SCA*** is linked to the PC via a serial cable (DB9-IDE) which is supplied with the TERN EV-P / DV-P Kits.

The ***SCA*** communicates through SER0 by default. Install the 5x2 IDC connector on the SER0 H1 pin header. The DB9 connector should be connected to one of your PC's COM Ports (COM1 or COM2).

See Appendix B for figures regarding the TERN debug cable.

2.2.2 Powering-on the SCA

By factory default setting:
1) The RED STEP2 Jumper is installed at location H5.1=H5.2. (Default setting in factory)
2) The DEBUG kernel is pre-loaded into the on-board flash starting at address of 0xFA000. (Default setting in factory)
3) The EEPROM is set to jump address of 0xFA000. (Default setting in factory)

Connect +9-12V DC to the DC power terminal (T1 header; see Figure 2.1). A power jack adapter is included with the TERN EV-P/DV-P kit. It can be used to connect the output of the power jack adapter and the ***SCA***. Note: The output of the power jack adapter is center negative.

The on-board LED should blink twice and remain on, indicating the debug kernel is running and ready to communicate with Paradigm C++ TERN Edition for programming and debugging.

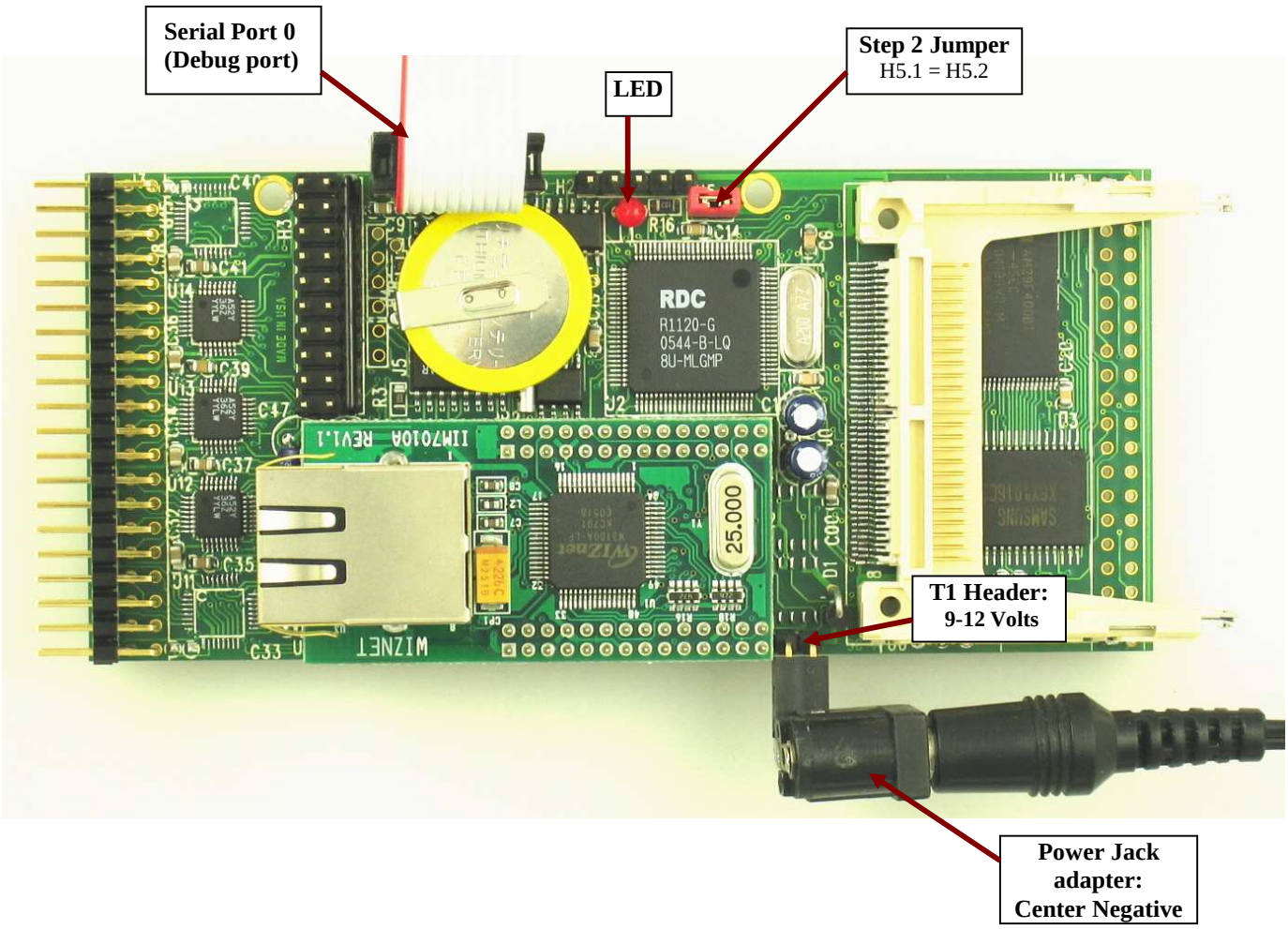


Figure 2.1 Locations of STEP2 Jumper, LED, Power input and DEBUG port

Chapter 3: Hardware

Am186ES/R8820/R1120 - Introduction

The Am186ES is based on industry-standard x86 architecture. The Am186ES controllers uses 16-bit external data bus, are higher-performance, more integrated versions of the 80C188 microprocessors which uses 8-bit external data bus. In addition, the Am186ES has new peripherals. The on-chip system interface logic can minimize total system cost. The Am186ES 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.

R8820 is a drop-in replacement 5V, 40MHz chip for the AM186ES.

R1100 is a 80MHz, 3.3V chip can be installed on the ***SCA***.

By default, the ***SCA*** uses 3.3V 80 MHz R1120 and low power 55-70 ns SRAM with battery backup.
Optional 5V 40 MHz R8820 can be installed.
At 80 MHz, the low power 55 ns SRAM with battery backup works fine but will not be able to support DMA operation.
A fast 10/15/25 ns SRAM (Not low power) can be used to support zero wait state and DMA operation at 80 MHz, but the backup battery will drain in a few days.
There are three pads on the PCB for battery. One pad is ground, and the other two pads allowing a 3V backup lithium battery be installed in two different positions:
1) The battery's positive lead is installed in the pad which is away from the RTC, supporting the RTC only. No battery backup for the SRAM.
2) The battery's positive lead is installed in the pad which is closer to the RTC, supporting both RTC and SRAM.
In the future, when the fast (10 ns) and low standby power SRAM is available, then 80 MHz ***SCA*** can have both RTC and SRAM with battery backup plus the DMA, zero wait state operation.
User can use sample program **c:\tern\186\samples\ee\rdc_id.c** to read the ID register(0xfff4), in order to identify RDC CPU type.

R1100=0xC5D9, R1120=0x85D9, R8820/30=0x04D9(xxD9)

Am186ES – Features

Clock and crystal

Due to its integrated clock generation circuitry, the Am186ES microcontroller allows the use of a times-one crystal frequency. The design achieves 40 MHz CPU operation, while using a 40 MHz crystal.

CLKOUTA remains active during reset and bus hold conditions. The initial function ae_init(); disables CLKOUTA and CLKOUTB with clka_en(0); and clkb_en(0);

You may use clka_en(1); to enable CLKOUTA=CLK=J1 pin 4.

The R8820 uses a 40 MHz crystal.

By default the 3.3V R1120 uses a 20 MHz crystal. The CPU speed is software programmable with the PLL.

At power-on, the on-board ACTF Flash programs the R1120 running at 20 MHz system clock, so a 9600 baud (instead 19,200 baud) is used for ACTF menu.

Two debug kernels are available:

c:\tern\186\rom\ae86\EE40_115.hex, or c:\tern\186\rom\ae86\EE80_115.hex

The EE40_115.hex will run the R1120 at 40 MHz, and the EE80_115.hex will run the R1120 at 80 MHz.

By default, the EE80_115.hex is pre-programmed for the 80 MHz ***SCA***.

User can use software to setup the CPU speed:

```
outport(0xfff8,0x0103);  // PLLCON, 20MHz crystal, 0103=40 MHz, 0107=80MHz
```

External Interrupts and Schmitt Trigger Input Buffer

There are eight external interrupts: INT0-INT6 and NMI.

/INT0, H4 pin 1, free to use.
/INT1, H4 pin 2, free to use.
INT2, H3 pin 6, free to use
/INT3, H4 pin 3, free to use
/INT4, JP1 pin 2, used by 100M BaseT Ethernet
INT5=P12=DRQ0, used by ***SCA*** as output for LED/EEPROM/HWD
INT6=P13=DRQ1, used by ***SCA*** as serial data for RTC/EEPROM
/NMI, H4 pin 4

Some of external interrupt inputs, /INT0, 1, 3, 4 and /NMI, are buffered by Schmitt-trigger inverters (U9, 74HC14), in order to increase noise immunity and transform slowly changing input signals to fast changing and jitter-free signals. As a result of this buffering, these pins are capable of only acting as input.

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

The ***SCA*** uses vector interrupt functions to respond to external interrupts. Refer to the Am186ES User's manual for information about interrupt vectors.

Asynchronous Serial Ports

The Am186ES CPU has two asynchronous serial channels: SER0 and SER1. Both asynchronous serial ports support 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 speed
- 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 samples files ***s1_echo.c*** and ***s0_echo.c***.

Important Note: For 80MHz ***SCA***, DMA functions are not available when by default low power 55 ns SRAM is installed. If install a 25 ns SRAM, 80MHz ***SCA*** can have all DMA functions, but it will drain the backup battery fast. Two battery positive pads allowing the battery be installed:

1) Support both RTC and low power SRAM, or

2) Support only RTC.

Timer Control Unit

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

Timer0 and Timer1 are connected to external pins:

Timer0 output = P10 = H3 pin 14
Timer0 input = P11 = U7 EE pin 5, RTC pin 5
Timer1 output = P1 = U16 pin 7
Timer1 input = P0 = H3 pin 13

Timer0 input P11 is used and shared by on-board EEPROM and RTC, not recommended for other external use.

The timer can be used to count or time external events, or can generate non-repetitive or variable-duty-cycle waveforms.

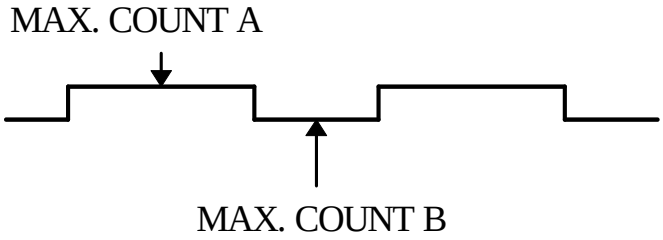
Timer2 is not connected to any external pin. It can be used as an internal timer for real-time coding or time-delay applications. It can also prescale timer 0 and timer 1 or be used as a DMA request source.

The maximum rate at which each timer can operate is 10 MHz (at 40MHz clock) or 20 MHz (at 80 MHz clock), since each timer is serviced once every fourth clock cycle. Timer output takes up to six clock cycles to respond to clock or gate events. See the sample programs ***timer02.c*** and ***ae_cnt1.c*** in the **`tern\186\samples\ae`** directory.

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 ns x 6 = 150 ns (at 40 MHz) and 12.5ns x 6 = 75 ns (at 80 MHz w/ R1120 CPU).

Each timer has a maximum count register that defines the maximum value the timer will reach. Both Timer0 and Timer1 have secondary maximum count registers 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=H3 pin 6.

Power-save Mode

The ***SCA*** can be used for low power consumption applications. The power-save mode of the Am186ES 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.

Am186ES PIO lines

The Am186ES 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 pin's behavior, either pull-up or pull-down, is pre-determined and shown in the table 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.

PIO	***Function***	***Power-On/Reset status***	***SensorCore-A Pin No.***	***SensorCore-A Initial after ae_init(); function call***
P0	Timer1 in	Input with pull-up	H3 pin 13	Input with pull-up
P1	Timer1 out	Input with pull-down	U16 pin 7, ADC's	Input with pull-up
P2	/PCS6/A2	Input with pull-up	H3 pin 7	/PCS6
P3	/PCS5/A1	Input with pull-up	H3 pin 8	/PCS5
P4	DT/R	Normal	H5 pin 1	Input with pull-up: Step 2
P5	/DEN/DS	Normal	H3 pin 3	Input with pull-up
P6	SRDY	Normal	H3 pin 4	Input with external pull-up
P7	A17	Normal	N/A	A17
P8	A18	Normal	N/A	A18
P9	A19	Normal	N/A	Input with pull-up
P10	Timer0 out	Input with pull-down	H3 pin 14	Input with pull-down
P11	Timer0 in	Input with pull-up	EEPROM, RTC	Input with pull-up
P12	DRQ0	Input with pull-up	DAC,LED,RTC,etc.	Output
P13	DRQ1	Input with pull-up	H3 pin 15	Input with pull-up
P14	/MCS0	Input with pull-up	JP1.5 Ethernet	Input with pull-up
P15	/MCS1	Input with pull-up	H3 pin 5	Input with pull-up
P16	/PCS0	Input with pull-up	J1 pin 19	/PCS0
P17	/PCS1	Input with pull-up	U4 pin 4,5	Input with pull-up
P18	/PCS2	Input with pull-up	N/A	Input with pull-up
P19	/PCS3	Input with pull-up	N/A	Input with pull-up
P20	SCLK	Input with pull-up	N/A	Input with pull-up
P21	SDATA	Input with pull-up	N/A	Input with pull-up
P22	SDEN0	Input with pull-down	N/A	Output
P23	SDEN1	Input with pull-down	N/A	Input with pull-down
P24	/MCS2	Input with pull-up	H3 pin 11	Input with pull-up
P25	/MCS3	Input with pull-up	H3 pin 12	Input with pull-up
P26	UZI	Input with pull-up	H3 pin 19	Input with pull-up*
P27	TxD	Input with pull-up	N/A	TxD0
P28	RxD	Input with pull-up	N/A	RxD0
P29	S6/CLKSEL1	Input with pull-up	H3 pin 20, DAC	Output
P30	INT4	Input with pull-up	N/A	Input with pull-up
P31	INT2	Input with pull-up	N/A	Input with pull-up

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

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

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

MODE	PIOMODE reg.	PIODIRECTION reg.	PIN FUNCTION
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

SensorCore initialization on PIO pins in **ae_init()** is listed below:

```
outport(0xff78,0xc7bc);      // PDIR1: TxD, RxD, PCS0, PCS1, P29& P22 Output
outport(0xff76,0x2040);      // PIOM1
outport(0xff72,0xee73);      // PDIR0: A18, A17, PCS6, PCS5, P12 Output
outport(0xff70,0x1040);      // PIOM0
```

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 pin is in input mode,

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

Some of the I/O lines are used by the ***SCA*** system for on-board components. 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**
P1	U10-U15.30	ADC Clock Line
P7	A17	Upper address line – **Never use by application**
P8	A18	Upper address line – **Never use by application**
P11	U5.5,U7.5	RTC & EEPROM data line
P12	J5.2,U7,U5,L1	LED, WDI, EEPROM, RTC
P14	JP1.5	/MCS0 - Ethernet
P17	U4.4,5	U4 HC138
P26*	/CLKSEL2	Used at power-up/reset to determine system clock multiplier
P27		TxD0
P28		RxD0
P29*	/CLKSEL1	Reserved for DAC
P30	INT4	Interrupt used Ethernet

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

Important Notes:

* The Am186ES CPU uses the P26 and the P29 lines to determine the system clock multiplier at power-up or reset. The CPU has internal pull-ups on these lines to select the default multiplier of four-times (AMD) or eight-times (RDC). It is critical that the user allow these lines to remain high during power-up or reset. Failure to do so will result in undesirable operation.

I/O Mapped Devices

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 to 15. The system clock is 100 ns for both CPUs, while the CPU clock is 25ns for the Am186ES and 12.5ns for the R1120. Details regarding this can be found in the Software chapter, and in the Am186ES 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 may need to be attached to I/O pins directly.

For details regarding the chip select unit, please see Chapter 5 of the Am186ES User's Manual.

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

I/O space	Select	Location	Usage
0x0000-0x00ff	/PCS0	J1 pin 19=P16	USER*
0x0100-0x01ff	/PCS1	U4.10-15	ADC's
0x0200-0x02ff	/PCS2	N/A	N/A
0x0300-0x03ff	/PCS3	N/A	N/A
0x0400-0x04ff	/PCS4	N/A	Reserved
0x0500-0x05ff	/PCS5	H3 pin 8=P3	USER
0x0600-0x06ff	/PCS6	H3 pin 7=P2	USER

*PCS0 may be used for other TERN peripheral boards, such as FC-0, P50, P100, MM-A.

To illustrate how to interface the ***SCA*** with external I/O boards, a simple decoding circuit for interfacing to an 82C55 parallel I/O chip is shown in Figure 3.11.

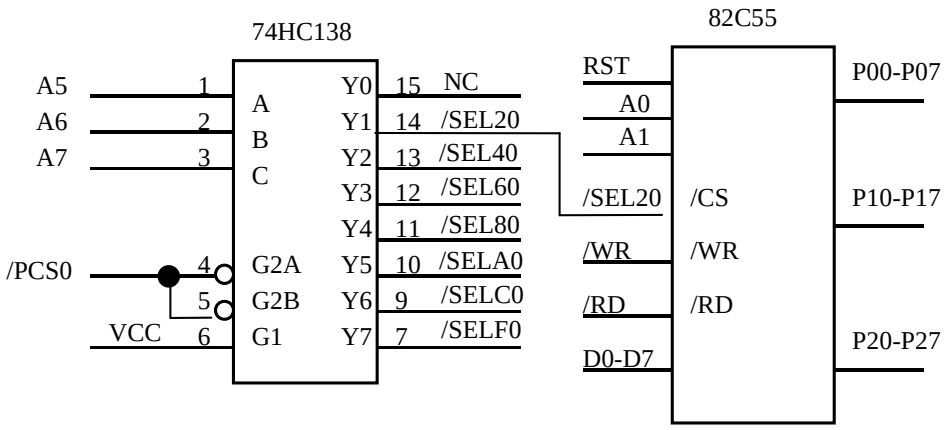


Figure 3.1 Interface the SensorCore to external I/O devices

The function **`ae_init()`** by default initializes the /PCS0 line at base I/O address starting at 0x00. You can read from the 82C55 with ***inportb(0x020)*** or write to the 82C55 with ***outportb(0x020,dat)***. The call to ***inportb(0x020)*** will activate /PCS0, as well as putting the address 0x20 over the address bus. The decoder will select the 82C55 based on address lines A5-7, and the data bus will be used to read the appropriate data from the off-board component.

EEPROM

A serial EEPROM of 512 bytes (24C04) is installed in U7. The ***SCA*** 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 typically has 1,000,000 erase/write cycles. The data retention is more than 40 years. EEPROM can be read and written by simply calling the functions **ee_rd()** and **ee_wr()**.

A range of lower addresses in the EEPROM is reserved for TERN use, 0x00 – 0x1F. The addresses 0x20 to 0x1FF are for user application.

Other Devices

A number of other devices are also available on the ***SCA***. 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.

On-board Supervisor with Watchdog Timer

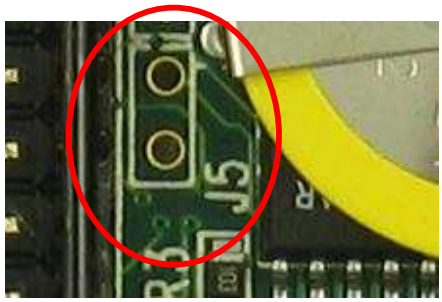
The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the ***SCA*** has several functions: watchdog timer, battery backup, power-on-reset delay, power-supply monitoring, and power-failure warning. These will significantly improve system reliability.

Watchdog Timer

The watchdog timer is activated by setting a jumper on J5 of the SensorCore. 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 = WDI pin of the MAX691) should be arranged such that the WDI pin is accessed at least once every 1.6 seconds. If the J5 jumper is on and the WDI 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 ***SCA*** is reset, the WDO remains low until a transition occurs at the WDI pin of the MAX691. When controllers are shipped from the factory the J5 jumper is off, which disables the watchdog timer.

The Am186ES has an internal watchdog timer. This is disabled by default with **ae_init()**.



Watchdog jumper, J5. The J5 header is not populated in this picture.

Figure 3.2 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 DS1337 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.

Real-time Clock DS1337

The DS1337 serial real-time clock is a low-power clock/calendar with two programmable time-of-day alarms and a programmable square-wave output. Address and data are transferred serially via a 2-wire, bidirectional bus. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The data at the end of the month is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either 24-hour or 12-hour format with AM/PM indicator.

The RTC is accessed via software drivers *rtc1_init()* and *rtc1_rds()*, which have been specifically written for this chip. Refer to sample code in the tern\186\samples\fn directory for fn_rtc.c

It is also possible to configure the real-time clock to raise an output line attached to an external interrupt, at 1/64 second, 1 second, 1 minute, or 1 hour intervals. This can be used in a time-driven application, or the **VOFF** signal can be used to turn on/off the controller using an external switching power supply.

Eight channel 16-bit DAC (LTC2600)

The LTC2600 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 by default shorted to VCC (via PCB trace). The DAC outputs are routed to the J4 pin header, pins 1-8 (found beneath the Ethernet Module).

The DAC is installed on the ***SCA*** at location U18 and uses P29 as the chip select. The synchronous serial interface is used to send data to the device. Refer to the sample code, \tern\186\samples\sca\sca_da16.c for

an example on driving the DAC. The sample is also included in the pre-built sample project \tern\186\samples\sca\sca.ide.

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

AD7852, 300KHz 12-bit ADC

Six AD7852 chips can be installed on the ***SCA,*** giving the option of 48 possible ADC channels! The AD7852 is a 100 ksps, sampling parallel 12-bit A/D converter that draws only 55 mW from a single 5V supply. This device includes 8 channels with sample-and-hold, precision 2.5V internal reference, switched capacitor successive-approximation A/D, and uses an external clock: P1 (timer 1 output).

The input range of the AD7852 is 0-5V. Maximum DC specs include ±2.0 LSB INL and 12-bit no missing codes over temperature. The ADC has a 12-bit data parallel output port that directly interfaces to the full 12-bit data bus D15-D4 for maximum data transfer rate.

The AD7852 requires 16 ADC clocks (or 3.2 µs) conversion time to complete one conversion, based on a 5 MHz ADC clock. The busy signal has an 3.2 µs low period indicating that conversion is in progress. In order to achieve the 300 KHz sample rate, the ***SCA*** must use polling method, not interrupt operation, to acquire data. A sample program **sca_ad.c** can be found in the **`c:\tern\186\samples\sca`** directory.

Figure 3.3 AD7852 (12-bit 8-channel ADC)

Compact Flash Interface

By utilizing the compact flash interface on the ***SCA***, users can easily add widely used 50-pin CF standard mass data storage cards to their embedded application via RS232, TTL I2C, or parallel interface. TERN software supports Linear Block Address mode, 16-bit FAT flash file system, RS-232, TTL I2C or parallel communication. Users can write/read files to/from the CompactFlash card. Users can also transfer files to and from a PC via a Compact Flash card reader. (sandisk.com).

This allows the user to log huge amounts of data from external sources. Files can then be accessed via compact flash reader on a PC.

The **tern\186\samples\sca** directory includes sample build sca_cf (utilizing \tern\186\samples\ee\ee_cf.c), to show reads and writes of raw data by sector. In addition, **tern\186\samples\fn\fs_cmds1.c** is a simple file system demo with serial port based user interface.

100 MHz BaseT Ethernet

A WizNet™ Fast Ethernet Module can be installed to provide 100M Base-T network connectivity. This Ethernet module has a hardware LSI TCP/IP stack. It implements TCP/IP, UDP, ICMP and ARP in hardware, supporting internet protocol DLC and MAC. It has 16KB internal transmit and receiving buffer which is mapped into host processor's direct memory. The host can access the buffer via high speed DMA transfers. The hardware Ethernet module releases internet connectivity and protocol processing from the host processor. It supports 4 independent stack connections simultaneously at a 4Mbps protocol processing speed. An RJ45 8-pin connector is on-board for connecting to 10/100 Base-T Ethernet network. A software library is available for Ethernet connectivity.

Headers and Connectors

Expansion Header J1 / PIO Header H3

There is one 20x2 0.1 spacing headers for SensorCore expansion. Most signals are directly routed to the CPU.

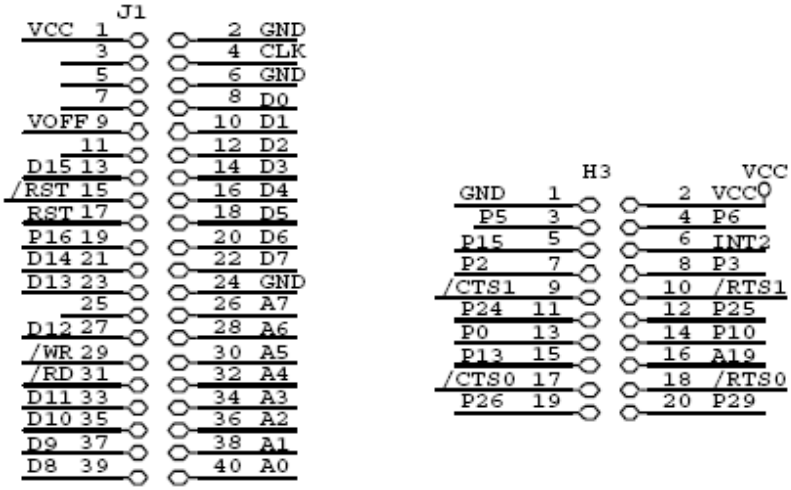


Table 3.3 J1 (Expansion) 20x2 port; H3 (PIO) Header

ADC / DAC Headers

Two headers, 0.1 spacing, are reserved for the 48 possible ADC inputs (**J2** and **J3**). One 4x2 header (**J4**) is reserved for 8 DAC inputs.

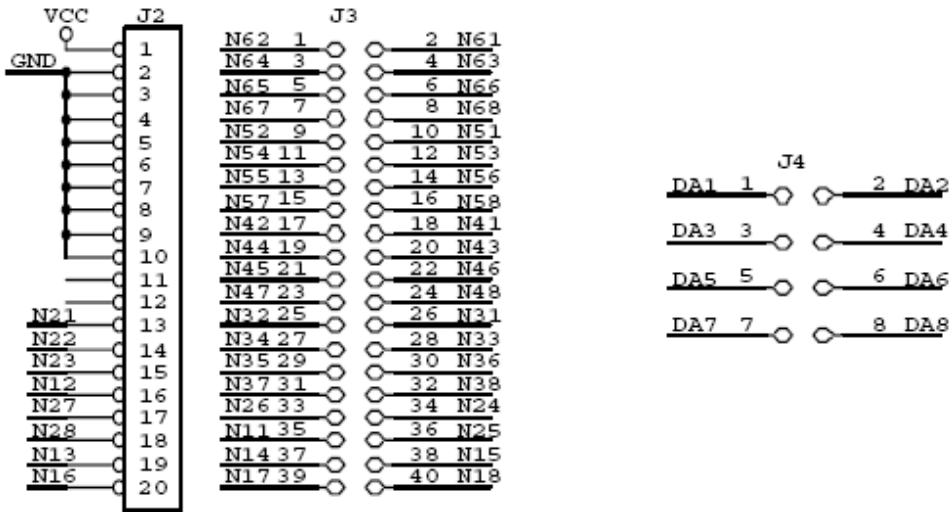


Table 3.4 Signals for J2 & J3 (ADC headers); Signals for J4 (DAC outputs)

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 C.

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 AE.LIB

AE.LIB is a C library for basic ***SCA*** 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
SCC.H	External UART SCC2691
AEEE.H	on-board EEPROM

4.2 Functions in AE.OBJ

4.2.1 SCA Initialization

ae_init

This function should be called at the beginning of every program running on ***SCA*** 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 Am186ES 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 0x80000-0xfffff (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, 0x80000-0xfffff
```

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.
Three 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 states. 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 eight external interrupt sources on the ***SCA***, consisting of seven maskable interrupt pins (**INT6-INT0**) and one non-maskable interrupt (**NMI**). 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 Am186ES Microcontroller User's Manual.

TERN provides functions to enable/disable all of the eight 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 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.

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

Two ports of 16 I/O pins each are available on the ***SCA***. 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 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 Am186ES 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 µs. 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 µs 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 ***SCA*** can be used for a variety of applications. All three timers run at 1/4 of the processor clock rate (10MHz based on 40MHz system clock, or one timer clock per 100ns), 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 Am186ES 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. It is important to note the the interrupt latency generated by the ISRs that handle a signal transition will define the time resolution the user will be able to achieve.

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. The sample files ***timer02.c*** and ***timer12.c***, located in ***tern\186\samples\ae***, demonstrate this.

The specific behavior that you might want to implement is described in detail in chapter 8 of the AMD Am186ES 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 (**J5**) 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

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.

The real time clock only allows storage of two digits of the year code, as reflected below. As a result, application developers should be careful to account for a roll-over in digits in the year 2000. One solution might be to store an offset value in non-volatile storage such as the EEPROM.

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 rtc1_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 rtc1_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 ROM 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 Am186ES 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 One and Step Two. 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 Am186ES Microprocessor User's Manual and the schematic of the ***SCA*** provided on the CD in the **tern_docs\schs** directory.

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 **`s1_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.

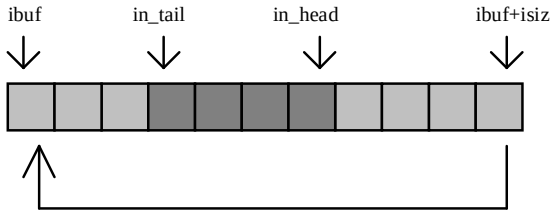


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), and baud rate (**baud**) are specified by the user with **`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 (SP0CT/SP1CT) if necessary, as described in chapter 10 of the Am186ES 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 without overwrite.

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\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 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, and 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.

DMA transfer automatically places incoming data into the inbound buffer. **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 Am186ES 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, there are a few functions that can be used 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.

clean_ser*n*
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 Am186ES 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 AM186ES 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 allows 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 Two, 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

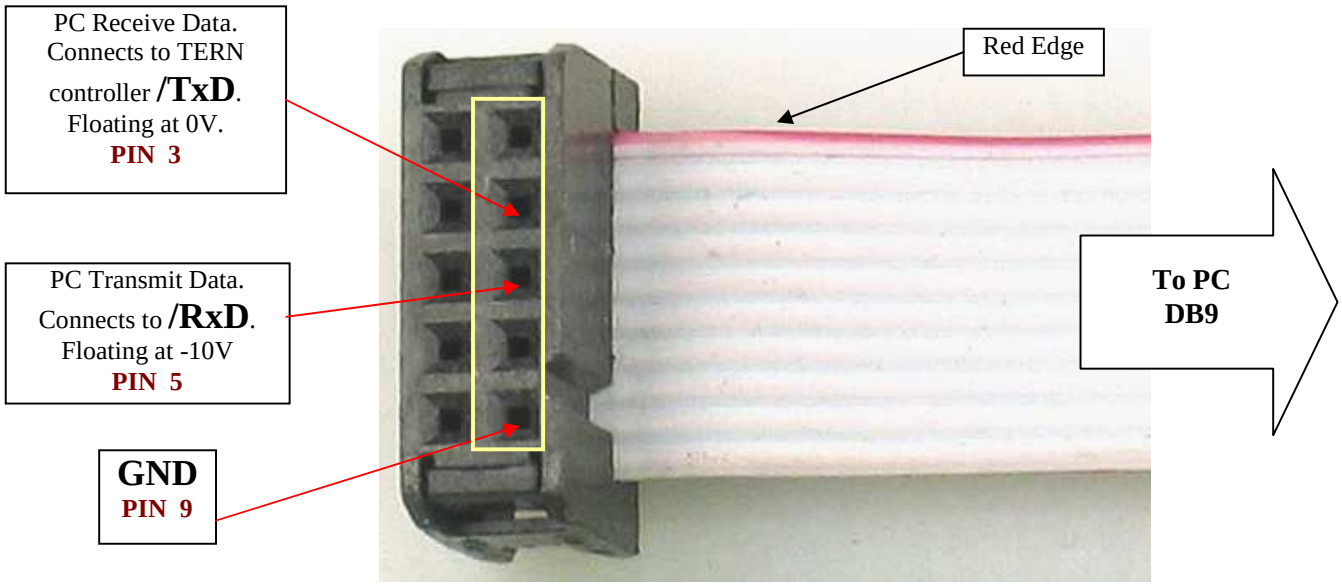
This function returns one byte of data from the specified address.

Appendix A: SensorCore-A Layout

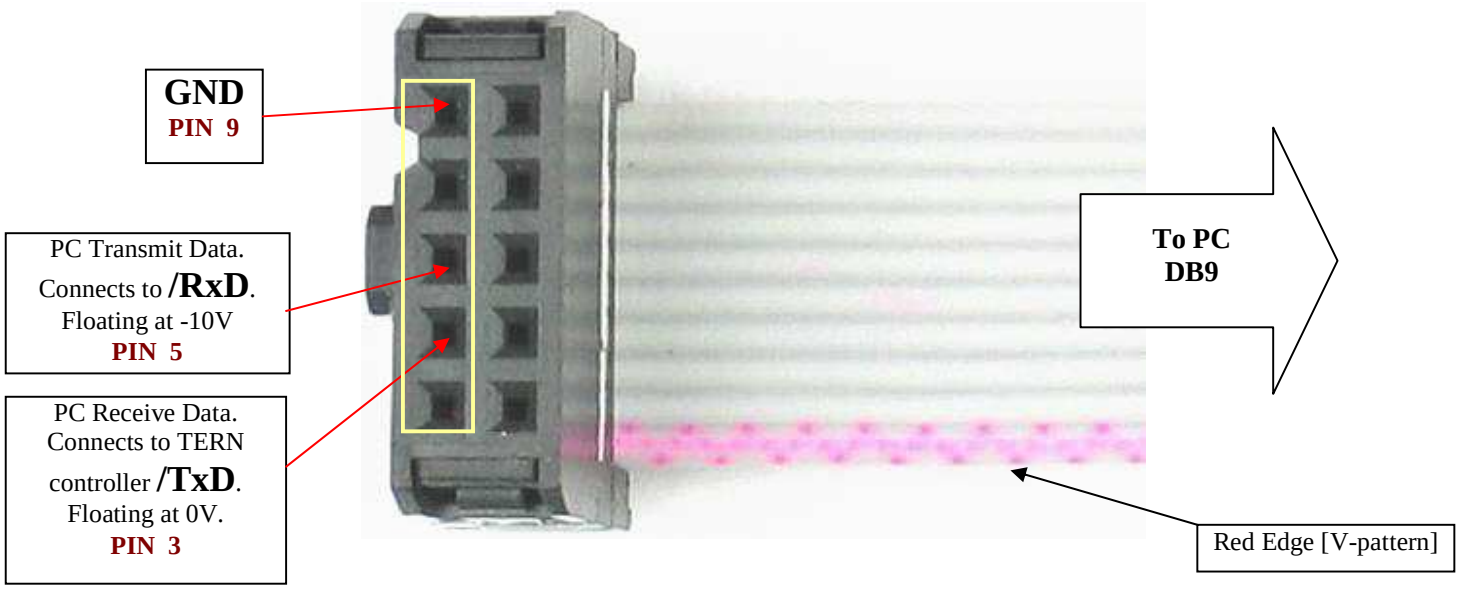
Contact TERN for details.

Appendix B: Tern Serial Port Cables

Note: Almost all Kits (DV-P/EV-P) will contain v2.0 cable, unless there is a size restriction on the controller's serial port header.



Tern RS232 Debug Cable v1.0



Tern RS232 Debug Cable v2.0

Appendix C: Software Glossary

The following is a glossary of library functions for the SensorCore.

---

void ae_init(void) ae.h

Initializes the Am186ES processor. The following is the source code for ***ae_init()***

```
outport(0xffa0,0xc0bf);     // UMCS, 256K ROM, 3 wait states, disable AD15-0
outport(0xffa2,0x7fbc);     // 512K RAM, 0 wait states
outport(0xffa8,0xa0bf); // 256K block, 64K MCS0, PCS I/O
outport(0xffa6,0x81ff);     // MMCS, base 0x80000
outport(0xffa4,0x007f); // PACS, base 0, 15 wait

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

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

Reference: led.c

---

void ae_reset(void) ae.h

Resets Am186ES processor.

---

void delay_ms(int m) ae.h

Approximate microsecond delay. Does not use timer.

```
Var:  m – Delay in approximate ms
```

Reference: led.c

---

void led(int i) ae.h

Toggles P12 used for led.

```
Var:  i - Led on or off
```

Reference: led.c

---

void delay0(unsigned int t) ae.h

Approximate loop delay. Does not use timer.

```
Var:  m – Delay using simple for loop up to t.
```

Reference:

---

void pwr_save_en(int i) ae.h

Enables power save mode which reduces clock speed. Timers and serial ports will be effected. Disabled by external interrupt.

```
Var:  i – 1 enables power save only.  Does not disable.
```

Reference: ae_pwr.c

---

void clka_en(int i) ae.h

Enables signal CLK respectively for external peripheral use.

```
Var: i – 1 enables clock output, 0 disables (saves current when
disabled).
```

Reference:

---

void hitwd(void) ae.h

Hits the watchdog timer using P03. P03 must be connected to WDI of the MAX691 supervisor chip.

Reference: *See Hardware chapter of this manual for more information on the MAX691.*

---

void pio_init(char bit, char mode) ae.h

Initializes a PIO line to the following:
mode=0, Normal operation
mode=1, Input with pullup/down
mode=2, Output
mode=3, input without pull

```
Var:  bit – PIO line 0 - 31
      Mode – above mode select
```

Reference: ae_pio.c

---

void pio_wr(char bit, char dat) ae.h

Writes a bit to a PIO line. PIO line must be in an output mode
mode=0, Normal operation
mode=1, Input with pullup/down
mode=2, Output
mode=3, input without pull

```
Var:  bit – PIO line 0 - 31
      dat – 1/0
```

Reference: ae_pio.c

---

unsigned int pio_rd(char port) ae.h

Reads a 16 bit PIO port.

```
Var:  port – 0: PIO 0 - 15
             1: PIO 16 – 31
```

Reference: ae_pio.c

---

void outport(int portid, int value) dos.h

Writes 16-bit *value* to I/O address *portid*.

```
Var:  portid – I/O address
      value – 16 bit value
```

Reference: ae_ppi.c

---

void outportb(int portid, int value) dos.h

Writes 8-bit *value* to I/O address *portid*.

```
Var:  portid – I/O address
      value – 8 bit value
```

Reference: ae_ppi.c

---

int inport(int portid) dos.h

Reads from an I/O address *portid*. Returns 16-bit value.

```
Var:  portid – I/O address
```

Reference: ae_ppi.c

---

int inportb(int portid) dos.h

Reads from an I/O address *portid*. Returns 8-bit value.

```
Var:  portid – I/O address
```

Reference: ae_ppi.c

---

int ee_wr(int addr, unsigned char dat) aeee.h

Writes to the serial EEPROM.

```
Var:  addr – EEPROM data address
      dat - data
```

Reference: ae_ee.c

---

int ee_rd(int addr) aeee.h

Reads from the serial EEPROM. Returns 8-bit data

```
Var:  addr – EEPROM data address
```

Reference: ae_ee.c

---

void io_wait(char wait) ae.h

Setup I/O wait states for I/O instructions.

```
Var:  wait – wait duration {0…7}
      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
```

Reference:

---

void rtc1_init(unsigned char * time) fn.h

Sets real time clock date, year and time.

```
Var:  time – time and date string
      String sequence is the following:
            time[0] = weekday
            time[1] = year10
            time[2] = year1
            time[3] = mon10
            time[4] = mon1
            time[5] = day10
            time[6] = day1
            time[7] = hour10
            time[8] = hour1
            time[9] = min10
            time[10] = min1
            time[11] = sec10
            time[12] = sec1
unsigned char time[]={2,9,8,0,7,0,1,1,3,1,0,2,0};
/* Tuesday, July 01, 1998, 13:10:20 */
```

Reference: rtc_init.c

---

int rtc1_rd(TIM *r) fn.h

Reads from the real time clock.

```
Var:  *r – Struct type TIM for all of the RTC data
      typedef struct{
            unsigned char sec1, sec10,  min1, min10, hour1, hour10;
            unsigned char day1, day10, mon1, mon10, year1, year10;
            unsigned char wk;
            } TIM;
```

Reference: rtc.c

---

void t2_init(int tm, int ta, void interrupt far(*t2_isr)()); ae.h

void t1_init(int tm, int ta, int tb, void interrupt far(*t1_isr)());
void t0_init(int tm, int ta, int tb, void interrupt far(*t0_isr)());

Timer 0, 1, 2 initialization.

```
Var:  tm – Timer mode. See pg. 8-3 and 8-5 of the AMD CPU Manual
      ta – Count time a (1/4 clock speed).
      tb – Count time b for timer 0 and 1 only (1/4 clock).
           Time a and b establish timer duty cycle (PWM). See
           hardware chapter.
      t#_isr – pointer to timer interrupt routine.
```

Reference: timer.c, timer1.c, timer02.c, timer2.c, timer0.c timer12.c

---

void nmi_init(void interrupt far (* nmi_isr)()); ae.h
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)());

Initialization for interrupts 0 through 6 and NMI (Non-Maskable Interrupt).

```
Var:  i – 1: enable, 0: disable.
      int#_isr – pointer to interrupt service.
```

Reference: intx.c

---

void s0_init(unsigned char b, unsigned char* ibuf, int isiz, ser0.h
unsigned char* obuf, int osiz, COM *c) (void);

Serial port 0, 1 initialization.

```
Var:  b – baud rate. Table below for 40MHz and 20MHz Clocks.
      ibuf – pointer to input buffer array
      isiz – input buffer size
      obuf – pointer to output buffer array
      osiz – ouput buffer size
      c – pointer to serial port structure. See AE.H for COM
      structure.
```

b	baud (40MHz)	baud (20MHz)
1	110	55
2	150	110
3	300	150
4	600	300
5	1200	600
6	2400	1200
7	4800	2400
8	9600	4800
9	19200	9600
10	38400	19200
11	57600	38400

b	baud (40MHz)	baud (20MHz)
12	115200	57600
13	23400	115200
14	460800	23400
15	921600	460800

Reference: s0_echo.c

---

int putser0(unsigned char ch, COM *c); ser0.h

Output 1 character to serial port. Character will be sent to serial output with interrupt isr.

```
Var:  ch – character to output
      c – pointer to serial port structure
```

Reference: s0_echo.c

---

int putsers0(unsigned char *str, COM *c); ser0.h

Output a character string to serial port. Character will be sent to serial output with interrupt isr.

```
Var:  str – pointer to output character string
      c – pointer to serial port structure
```

---

int serhit0(COM *c); ser0.h

Checks input buffer for new input characters. Returns 1 if new character is in input buffer, else 0.

```
Var:  c – pointer to serial port structure
```

Reference: s0_echo.c

---

unsigned char getser0(COM *c); ser0.h

Retrieve 1 character from the input buffer. Assumes that *serhit* routine was evaluated.

```
Var:  c – pointer to serial port structure
```

Reference: s0_echo.c, s1_0.c

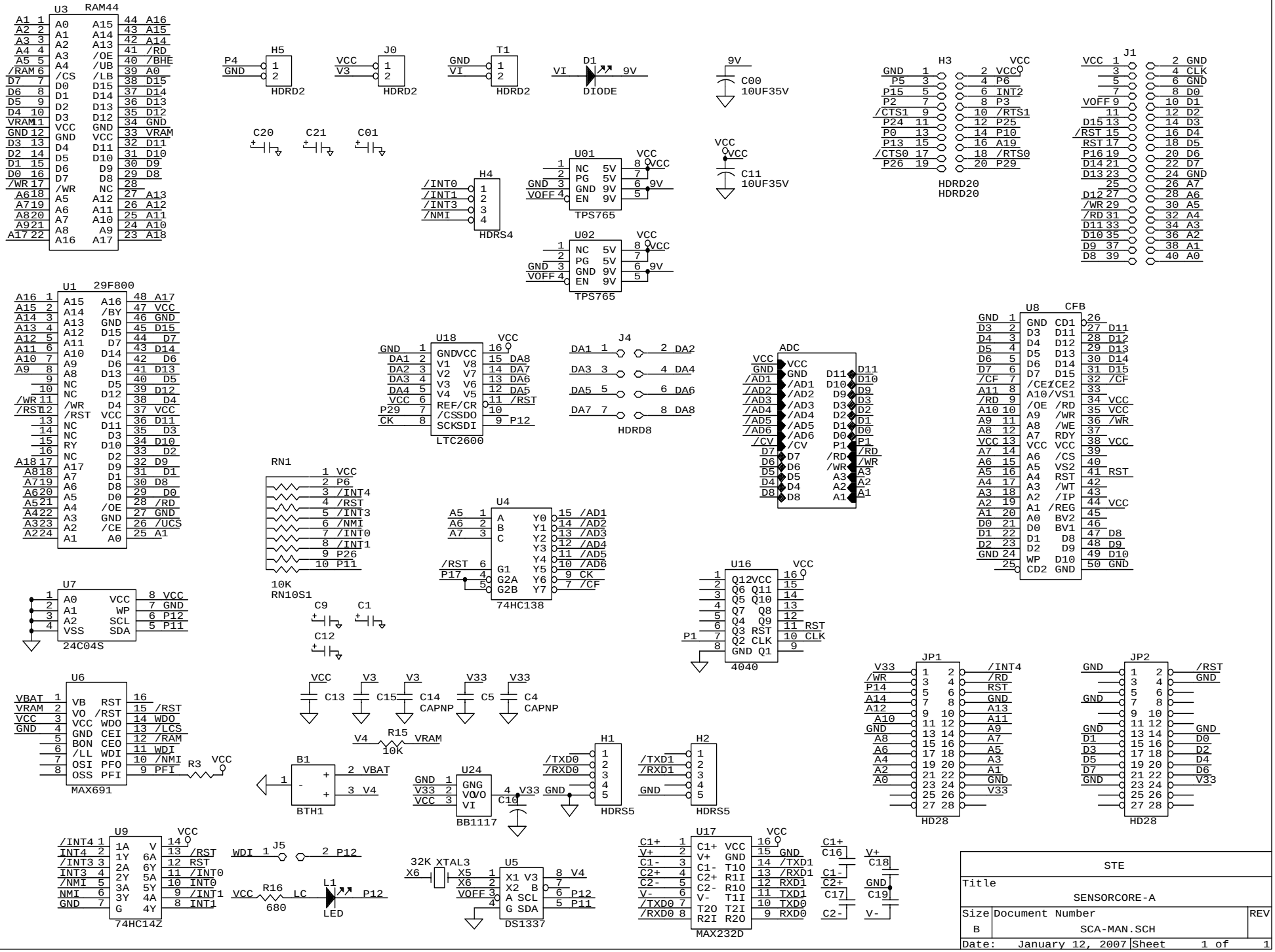
---

int getsers0(COM *c, int len, unsigned char *str); ser0.h

Retrieves a fixed length character string from the input buffer. If the buffer contains less characters than the length requested, *str* will contain only the remaining characters from the buffer. Appends a '\0' character to the end of *str*. Returns the retrieved string length.

```
Var:  c – pointer to serial port structure
      len – desired string length
      str – pointer to output character string
```

Reference: ser0.h for source code.


U3 RAM44
H5 HDRD2
J0 HDRD2
T1 HDRD2
D1 DIODE
9V
C00 10UF35V
H3 HDRD20
J1
C20
C21
C01
H4 HDRS4
U01 TPS765
U02 TPS765
VCC
C11 10UF35V
U1 29F800
U18 LTC2600
J4 HDRD8
ADC
U8 CFB
RN1 10K RN10S1
U4 74HC138
U7 24C04S
C9
C1
C12
U16 4040
JP1 HD28
JP2 HD28
U6 MAX691
C13
C15
C14 CAPNP
C5
C4 CAPNP
R15 10K
B1 BTH1
U24 BB1117
C10
H1 HDRS5
H2 HDRS5
U9 74HC14Z
J5
R16 680
L1 LED
32K XTAL3
U5 DS1337
U17 MAX232D
C16
C18
C17
C19
STE
SENSORCORE-A
SCA-MAN.SCH
Date: January 12, 2007
Sheet 1 of 1