SensorCore™

48 24-bit ADCs, 100M BaseT Ethernet, RS232/485, 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

Version 2.0

October 21, 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

The **SensorCore(*SC*)™** is designed as a low-cost, low-power data logger for the most demanding analog data-acquisition applications. Featuring up to 48 channels of **24-bit ADC**, 2 RS232/RS485 ports, CompactFlash interface, and a high performance 10/100M BaseT Ethernet, the SensorCore out-performs desktop-based acquisition solutions for a fraction of the price.

Measuring only 2"x4.5", the ***SC***'s unique profile allows it to be installed into difficult-to-access physical locations, such as pipes. Even with this limited real estate, the ***SC*** is a full-featured, stand-alone industrial embedded controller.

The ***SC*** is based on a high-performance C/C++ programmable x186ER CPU. It integrates 3 timer/counters, 2 async serial ports, external interrupts, PIOs, and a real-time clock(RTC72423). 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 Power-off feature allowing low voltage(5.1V) battery operation. Optional 2 channels 12-bit DAC analog outputs can be installed.

The ***SC*** has two RS232 serial ports as default, and one can be configured as RS485 operation. The ***SC*** 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 SensorCore allows it to be configured as a HTTP web-server, FTP server/client, etc.

The ***SC*** features three high-speed LTC2448 delta-sigma ADCs, interfaced through a high speed sync serial port.

Each LTC2448 chip offers 8 ch. differential or 16 ch. single-ended input channels. Variable speed/resolution settings can be configured. A peak single-channel output rate of 8 KHz can be achieved. At a sample rate of 1.76KHz, readings are accurate to 18+ bits in experimental conditions.

The LTC2448 works well with 500 ohm and lower impedance sensors, such as 350 ohm Strain gauges, current shunts, RTDs, resistive sensors (500ohm an lower), and 4-20mA current loop sensors with 100 ohm sense resistors.

The ***SC*** will also work directly with thermocouples. We can install a 2.5V precision reference with temperature sensor(LT1019) to minimize input current and providing local temperature measurement for thermocouple applications. There are 8+ million counts of resolution in the input span. If desired, you could put a divider at the input to increase the input range.

The ***SC*** also features an integrated CompactFlash interface. Data can be written into CF at a sufficiently high enough data rate that the ***SC*** 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.

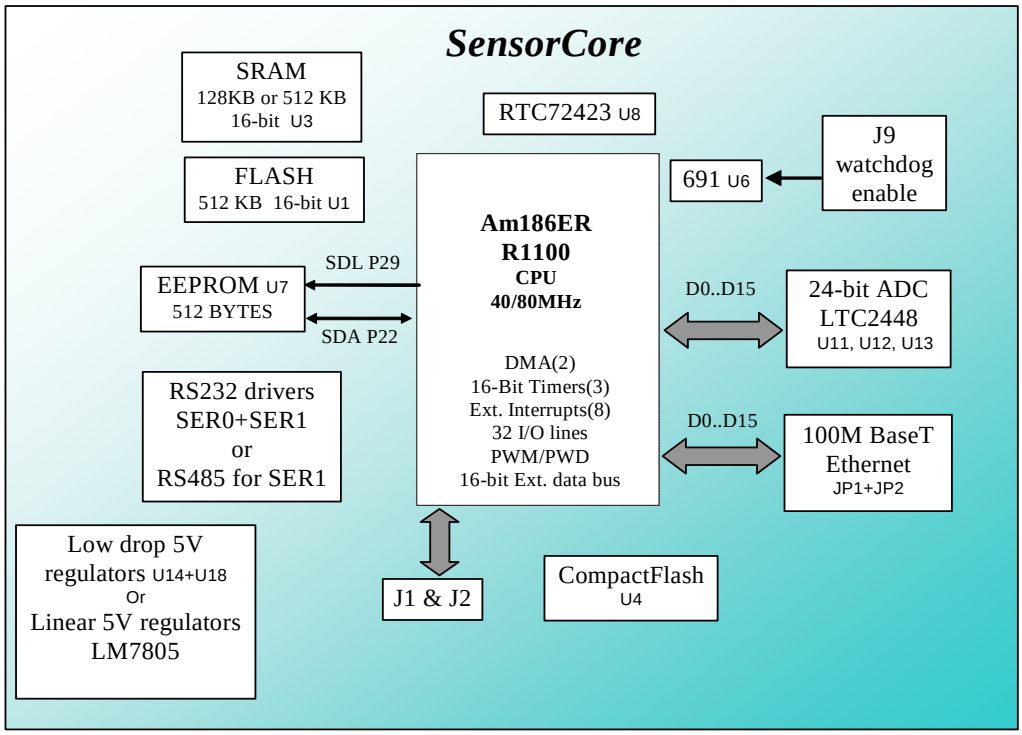


Figure 1.1 Functional block diagram of the SensorCore

The ***SC*** 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. One serial port from the Am186ER/R1100 support high-speed, reliable serial communication at a rate of up to 115,200 baud. A SCC2691 provides the second serial port. Two serial ports support 8-bit and 9-bit communication.

There are three 16-bit programmable timers/counters and a watchdog timer. Two timers can be used to count or time external events, at a rate of up to 10 MHz, 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:
* 2.0 x 4.5”, 160 mA, 9-24V DC power
* Complete C/C++ programmable environment
* 48 channels of 24-bit ADC input, 0-2.5V
* 2 channels 12-bit DAC (0-4.095V), 8 TTL I/Os
* CompactFlash with FAT file system support
* 40/80 MHz 186 CPU with 256 KW Flash, 256 KW SRAM
* 2 RS-232 serial ports; one can be RS232/485
* 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.

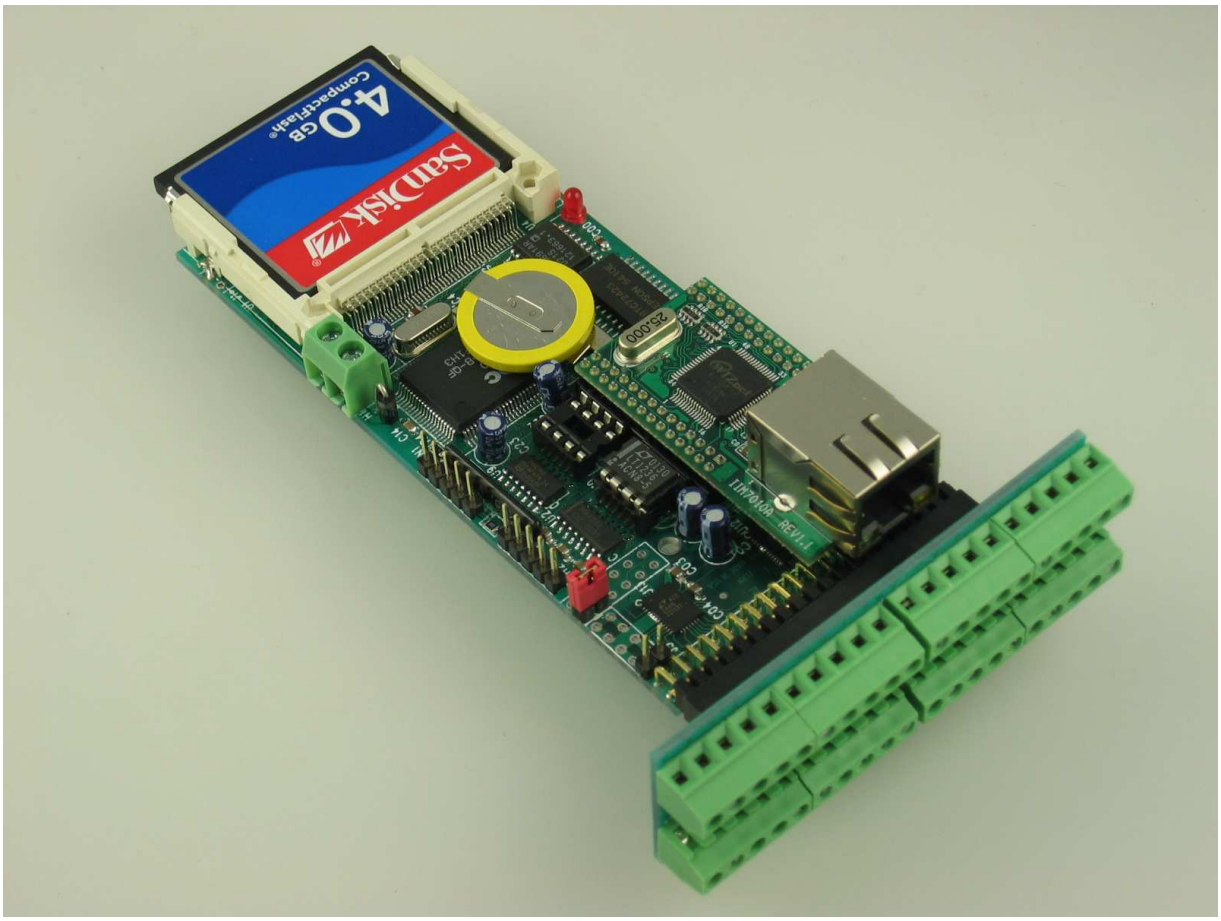


Figure 1.2 Physical layout of the SensorCore

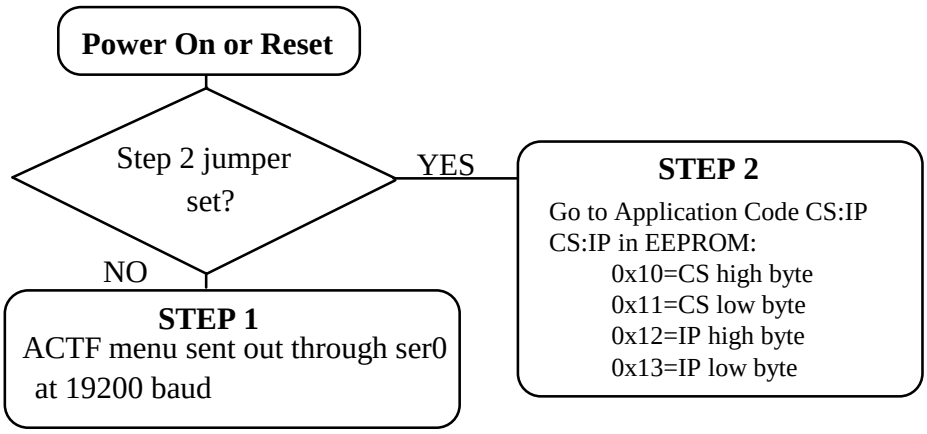


Figure 1.3 Flow chart for ACTF operation

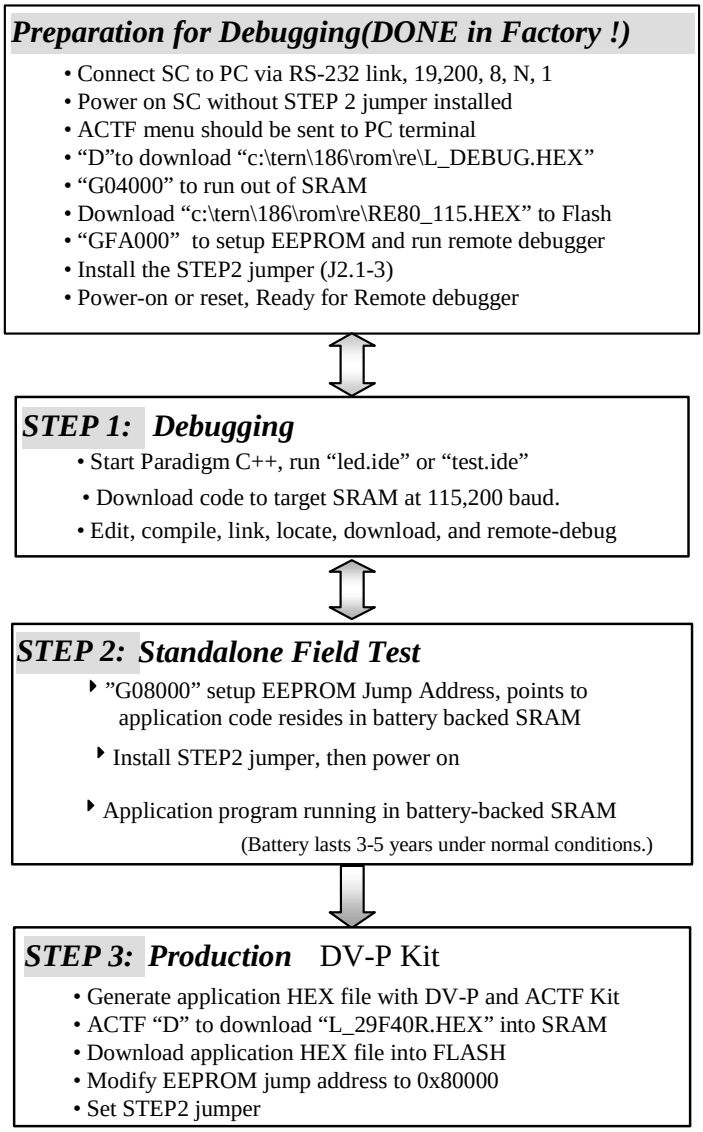
The "ACTF boot loader" resides in the top protected sector of the 512KB on-board Flash chip (29F400).

By default, in the factory, before shipping, the DEBUG kernel (RE80_115.hex) is pre-loaded in the Flash starting at 0xFA000, and the RED STEP2 jumper is installed on J2 pin 38-40, 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 port0 at 19200 baud for a 80MHz SC.
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 "RE80_115.hex" residing at "0xFA000" of the 512KB on-board flash chip.

1.3 SensorCore Programming Overview

Steps for product development:



There is no ROM socket on the SC. 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 (J2 pins 1-3) must be installed for every production-version board.

Step 1 settings

In order to talk to SC with Paradigm C++, the SC must meet these requirements:

1) c:\tern\186\rom\re\RE80_115.HEX or RE84_115.HEX must be pre-loaded into Flash starting address 0xfa000.

2) The SRAM installed must be large enough to hold your program.

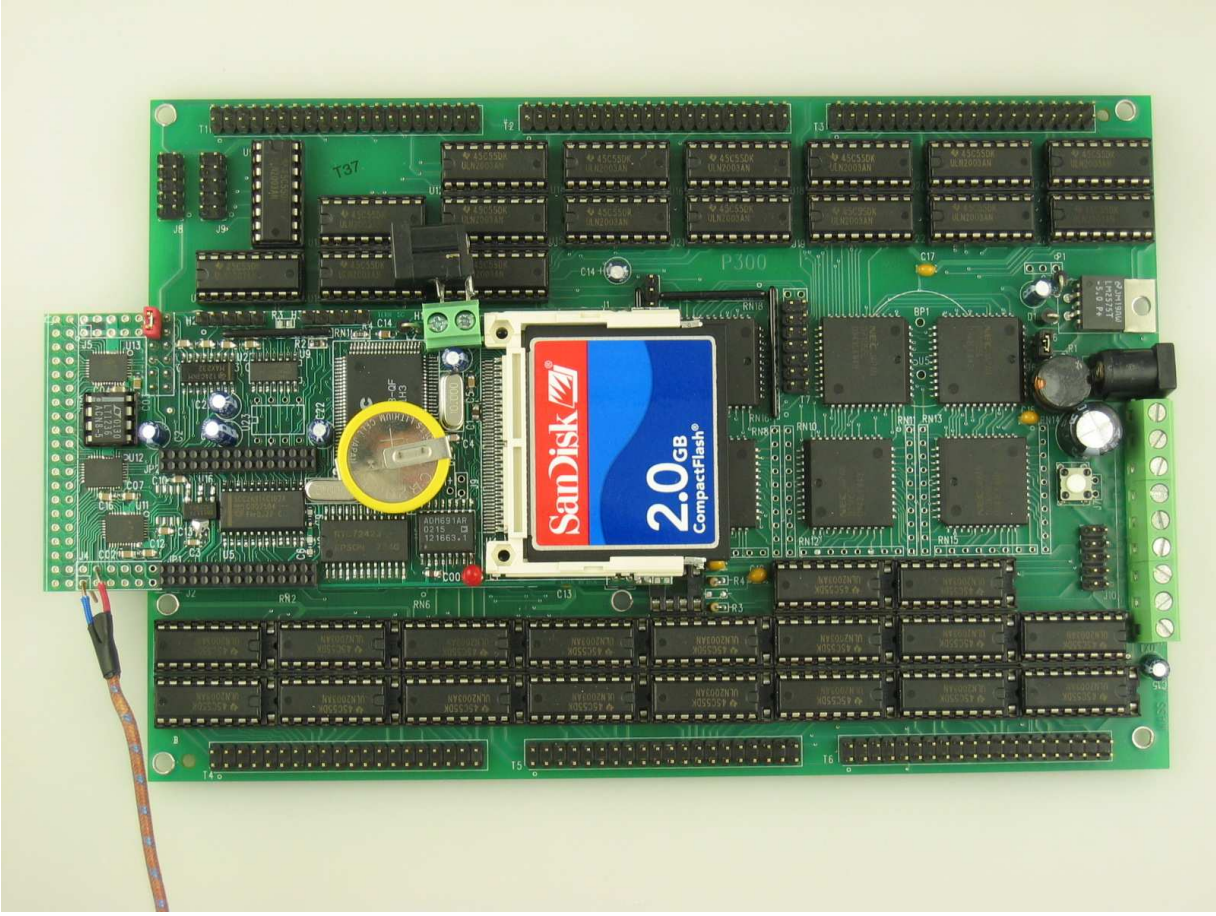
For a 128K SRAM, the physical address is 0x00000-0x01ffff
For a 512K SRAM, the physical address is 0x00000-0x07ffff

3) The on-board EEPROM must have a Jump Address for the RE80_115.HEX with starting address of 0xfa000.

4) The STEP2 jumper must be installed on J2 pins 1-3.

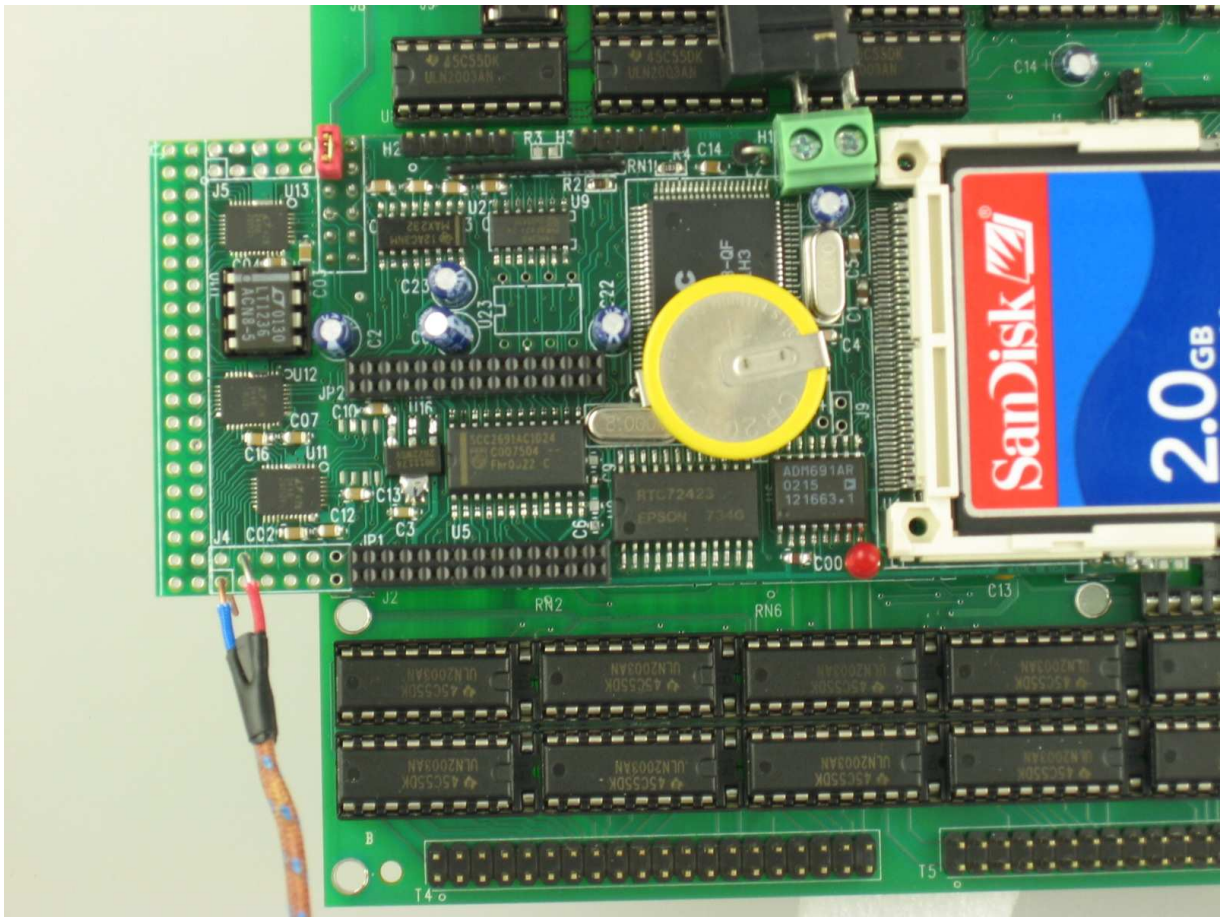
For further information on programming the SensorCore, refer to the manual on the TERN CD under: tern_docs\manuals\software_kit.pdf.

The ***SC*** works with some of TERN expansion boards including the P50, P100, P300.



The LTC2448 works well with 500 ohm and lower impedance sensors, such as 350 ohm Strain gauges, current shunts, RTDs, resistive sensors (500ohm an lower), and 4-20mA current loop sensors with 100 ohm sense resistors.

SC can work directly with thermocouples:



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 with red edge of cable on side closest to J3 (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

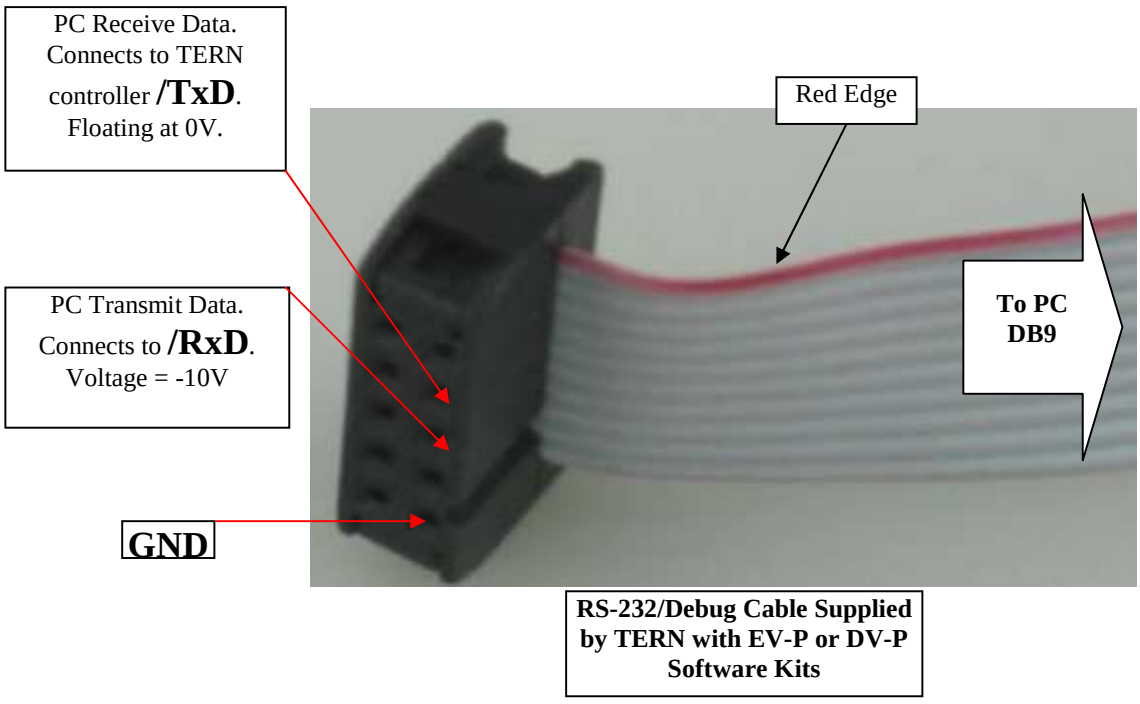
Hardware installation consists primarily of connecting the microcontroller to your PC.

2.2.1 Connecting the SC to the PC

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

The SC communicates through SER0 by default. Install the 5x2 IDC connector on the SER0 H2 pin header. The DB9 connector should be connected to one of your PC's COM Ports (COM1 or COM2).

The following is a picture of the debug cable and its relevant pins. Only /TxD, /RxD, and GND are needed.



RS-232/Debug Cable Supplied by TERN with EV-P or DV-P Software Kits

2.2.2 Powering-on the SC

By factory default setting:
1) The RED STEP2 Jumper is installed. (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. The screw terminal at the corner of the board is positive 12V input and the other terminal is GND (see figure for details). A power jack adapter (seen below) 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 SC. Note that 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.

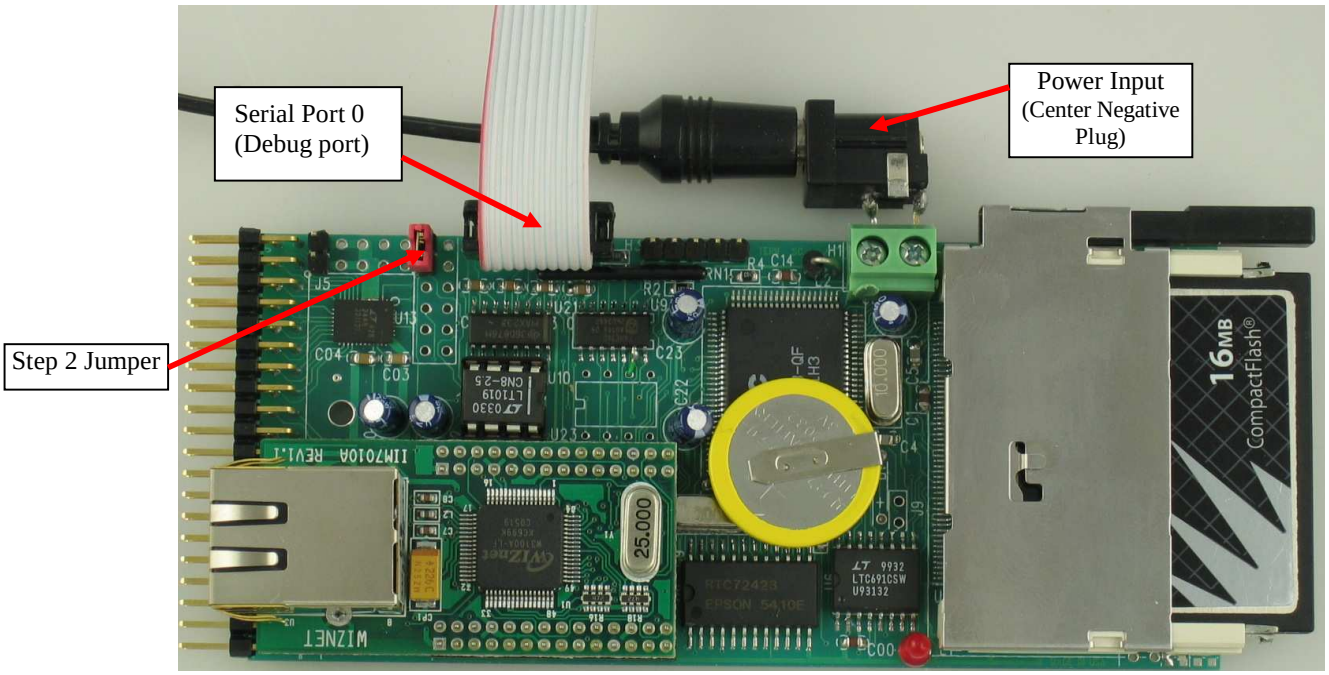


Figure 2.1 Locations of STEP2 Jumper, LED, Power input and DEBUG port

Chapter 3: Hardware

Am186ER AND RDC R1100

The SensorCore is compatible with two different CPUs. Both offer and support the same on-board peripherals as well as the on the CPU itself, aside from a few differences. The Am186ER, from AMD, uses times-four crystal frequency, while the R1100, from RDC, uses times-eight. The SensorCore uses a 10MHz system clock, giving the Am186ER a CPU clock of 40MHz and the R1100 a CPU clock of 80MHz. Both CPUs operate at +3.3V, with lines +5V tolerant. The RDC 1100 supports the same 80C188 microprocessor instruction set as the Am186ER, yet uses an internal RISC core architecture.

Am186ER – Introduction

The Am186ER is based on the industry-standard x86 architecture. The Am186ER controllers are higher-performance, more integrated versions of the 80C188 microprocessors. In addition, the Am186ER has new peripherals. The on-chip system interface logic can minimize total system cost. The Am186ER has one asynchronous serial port, one synchronous serial port, 32 PIOs, a watchdog timer, additional interrupt pins, DMA to and from serial ports, a 16-bit reset configuration register, and enhanced chip-select functionality.

In addition, the Am186ER has 32KB of internal volatile RAM. This provides the user with access to high speed zero wait-state memory. In some instances, users can operate the SensorCore without external SRAM, relying only on the Am186ER's internal RAM.

RDC R1100 – Introduction

The RDC 1100 is based on RISC internal architecture, yet still supports the same 80C188 microprocessor instruction set. It provides faster operation than the Am186ER, allowing it to operate at up to 80MHZ, based a 10MHz system clock and times-eight crystal operation. The RDC R1100 does not offer internal RAM like the Am186ER, so external SRAM is mandatory if using the RDC R1100.

Am186ER – Features

Clock

Due to its integrated clock generation circuitry, the Am186ER microcontroller allows the use of a times-four crystal frequency. The design achieves 40 MHz CPU operation, while using a 10 MHz crystal.

The R1100 offers times-eight crystal frequency, achieving 80MHz operation based on a 10MHz crystal.

The system CLKOUTA signal is routed to J1 pin 4, default 40 MHz. The CLKOUTB signal is not connected in the SensorCore.

CLKOUTA remains active during reset and bus hold conditions. The SensorCore 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.

External Interrupts and Schmitt Trigger Input Buffer

There are six external interrupts: INT0-INT4 and NMI.

/INT0 is used by SCC2691 UART.

/INT1, J2 pin 8, free for user application
INT2, is used by U12 ADC as Busy signal.
/INT3, free for user application.
/INT4, JP1 pin 2, used by Ethernet module.

NMI, U9.10

Four external interrupt inputs, /INT0-1, and /INT3-4 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.

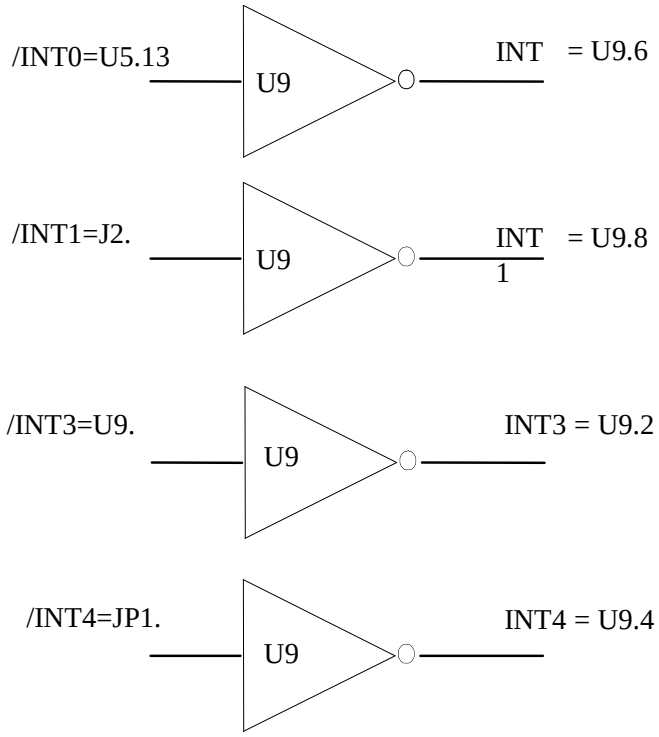


Figure 3.1 External interrupt inputs

Remember that /INT0 is used by the external UART. /INT0 should not be used by application if SER1 is being used.

The SensorCore uses vector interrupt functions to respond to external interrupts. Refer to the Am186ER User's manual for information about interrupt vectors.

Asynchronous Serial Port

The Am186ER and R1100 CPU has one asynchronous serial channel. It supports the following:

- Full-duplex operation
- 7-bit, and 8-bit data transfers
- Odd, even, and no parity

- One or two stop bits
- Error detection
- Hardware flow control
- DMA transfers to and from serial port (Am186ER ONLY)
- Transmit and receive interrupts
- Maximum baud rate of 1/16 of the CPU clock speed
- Independent baud rate generators

The software drivers for the asynch. serial port implement a ring-buffered DMA receiving and ring-buffered interrupt transmitting arrangement. See the sample file *s0_echo.c*

An external SCC2691 UART is located in position U5. For more information about the external UART SCC2691, please refer to the section in this manual on the SCC2691.

Note that while the Am186ER supports DMA transfers to and from its asynchronous serial port, the R1100 does not. Despite this difference, the TERN software drivers for the asynchronous serial port support both CPUs.

Timer Control Unit

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

Timer0 and Timer1 are connected to four external pins:

Timer0 output	= P10	= U15.4
Timer0 input	= P11	= J2 pin 7
Timer1 output	= P1	= J2 pin 12
Timer1 input	= P0	= J2 pin 5

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

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

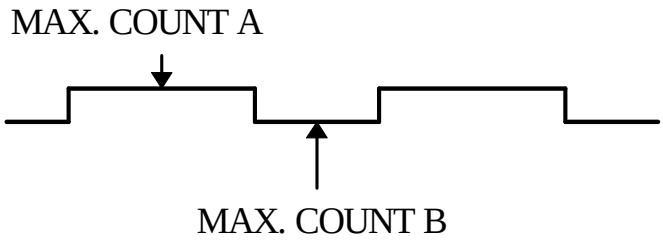
Timer 0 output, P10, is used as the chip select for the DAC7612. Timer 0 should therefore not be used by application unless the DAC7612 is not used.

The maximum rate at which each timer can operate is 10 MHz for the Am186ER and 20MHz for the R1100, since each timer is serviced once every fourth CPU clock cycle. Timer inputs take up to six clock cycles to respond to clock or gate events. See the sample programs ***timer0.c*** and ***ae_cnt0.c*** in the **`\samples\ae`** directory.

PWM outputs

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).

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.



Power-save Mode

The SensorCore is an ideal core module for low power consumption applications. The power-save mode of the Am186ER 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.

The RTC72423 on the SensorCore has a VOFF signal routed to J6 pin 2. VOFF is controlled by the battery-backed RTC72423. The VOFF signal can be programmed by software to be in tri-state or to be active low. The RTC72423 can be programmed in interrupt mode to drive the VOFF pin at 1 second, 1 minute, or 1 hour intervals. The user can use the VOFF line to control an external switching power supply that turns the power supply on/off.

Am186ER PIO lines

The Am186ER 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 Pin No.***	***SensorCore Initial after ae_init(); function call***
P0	Timer1 in	Input with pull-up	J2 pin 5	Input with pull-up
P1	Timer1 out	Input with pull-down	J2 pin 12	Input with pull-up
P2	/PCS6/A2	Input with pull-up	J2 pin 10	/PCS6
P3	/PCS5/A1	Input with pull-up	J2 pin 9	/PCS5
P4	DT/R	Normal	J2 pin 3	Input with pull-up: Step 2
P5	/DEN/DS	Normal	U13.2 ADC	Input with pull-up
P6	SRDY	Normal	J2 pin 6	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	U15.4 DAC	Input with pull-down
P11	Timer0 in	Input with pull-up	J2 pin 7	Input with pull-up

PIO	***Function***	***Power-On/Reset status***	***SensorCore Pin No.***	***SensorCore Initial after ae_init(); function call***
P12	DRQ0	Input with pull-up	J1 pin 26	Output
P13	DRQ1	Input with pull-up	U11.2 ADC	Input with pull-up
P14	/MCS0	Input with pull-up	JP1.5 Ethernet	Input with pull-up
P15	/MCS1	Input with pull-up	J2 pin 4	Input with pull-up
P16	/PCS0	Input with pull-up	J1 pin 19	/PCS0
P17	/PCS1	Input with pull-up	N/A	Input with pull-up
P18	/PCS2	Input with pull-up	U8.2 RTC	Input with pull-up
P19	/PCS3	Input with pull-up	U5.14 UART	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	U7.6 EEPROM	Output
P23	SDEN1	Input with pull-down	U11.36 ADC	Input with pull-down
P24	/MCS2	Input with pull-up	U12.36 ADC	Input with pull-up
P25	/MCS3	Input with pull-up	U13.36 ADC	Input with pull-up
P26	UZI	Input with pull-up	U15.3 DAC	Input with pull-up*
P27	TxD	Input with pull-up	U21.10	TxD0
P28	RxD	Input with pull-up	U21.9	RxD0
P29	S6/CLKSEL1	Input with pull-up	EE,LED,WDI	Output
P30	INT4	Input with pull-up	/INT4 = JP1.2	Input with pull-up
P31	INT2	Input with pull-up	U12.2 ADC	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 **re_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 SensorCore 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
P5	U13.2	U13 ADC Busy Line
P7	A17	Upper address line – **Never use by application**
P8	A18	Upper address line – **Never use by application**
P10	U15.4	U15 DAC Chip select line
P13	U11.2	U11 ADC Busy line
P14	JP1.5	/MCS0 - Ethernet
P18	U8.2	RTC Chip select
P19	U5.14	UART Enable
P20**	SCLK	Synchronous Clock for U11, U12, U13, U15
P21**	SDAT	Serial Interface for U11, U12, U13, U15
P22	SDEN0	Interface with EEPROM
P23	SDEN1	U11.36 ADC
P24	/MCS2	U12.36 ADC
P25	/MCS3	U13.36 ADC
P26*	/CLKSEL2	Used at power-up/reset to determine system clock multiplier
P27		TxD0
P28		RxD0
P29*	/CLKSEL1	Reserved for EEPROM, LED, RTC, and Watchdog timer
P30	INT4	Interrupt used Ethernet
P31	INT2	U12.2 ADC Busy line
/INT0	U5.13	UART interrupt

Table 3.2 Important Notes:

* The Am186ER 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.
** The SCLK and SDAT lines are the synchronous serial port on the Am186ER. Several devices on the **SC** use these lines, including 3 LTC2448 (locations U11, U12, U13), and the DAC7612 (locations U15). The user is free to use the SCLK and SDAT lines for their application only if the ADCs and DACs are disabled first. This is needed so as not to have more than one device trying to occupy the SDAT line simultaneously.

Table 3.3 I/O lines used for on-board components

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 Am186ER and 12.5ns for the R1100. Details regarding this can be found in the Software chapter, and in the Am186ER 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 Am186ER 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*
0x0200-0x02ff	/PCS2	U8 pin 2	RTC
0x0300-0x03ff	/PCS3	U5 pin 14	SCC2691
0x0400-0x04ff	/PCS4		Reserved
0x0500-0x05ff	/PCS5	J2 pin 9=P3	USER
0x0600-0x06ff	/PCS6	J2 pin 10 = 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 SensorCore with external I/O boards, a simple decoding circuit for interfacing to an 82C55 parallel I/O chip is shown in Figure 3.2.

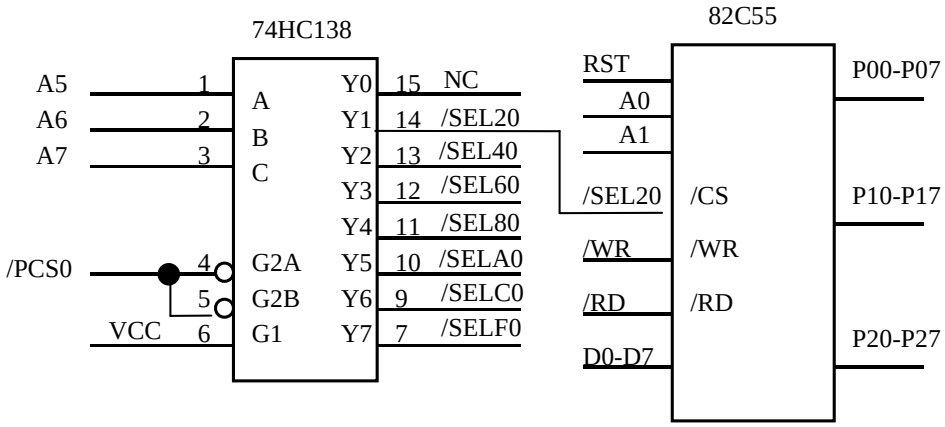


Figure 3.2 Interface the SensorCore to external I/O devices

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.

UART SCC2691

The UART SCC2691 (Signetics, U5) is mapped into the I/O address space at 0x0300. The SCC2691 has a full-duplex asynchronous receiver/transmitter, a quadruple buffered receiver data register, an interrupt control mechanism, programmable data format, selectable baud rate for the receiver and transmitter, a multi-functional and programmable 16-bit counter/timer, an on-chip crystal oscillator, and a multi-purpose input/output including RTS and CTS mechanism.

For more information, refer to Appendix B. The SCC2691 on the SensorCore may be used as a RS485 network 9-bit UART (for the TERN NT-Kit). Its data sheet and sample code are **scc2691.pdf** (in the **\tern_docs\parts** directory) and **\tern\186\samples\sc\ scc_echo.c**, respectively.

EEPROM

A serial EEPROM of 512 bytes (24C04) is installed in U7. The SensorCore uses the P22=SCL (serial clock) and P29=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 SensorCore. 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 SensorCore has several functions: watchdog timer, battery backup, power-on-reset delay, power-supply monitoring, and power-failure warning. These will significantly improve system reliability.

Watchdog Timer

The watchdog timer is activated by setting a jumper on J9 of the 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 P29 = WDI pin of the MAX691) should be arranged such that the WDI pin is accessed at least once every 1.6 seconds. If the J9 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 SensorCore 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 J9 jumper is off, which disables the watchdog timer.

The Am186ER has an internal watchdog timer. This is disabled by default with **ae_init()**.

Watchdog jumper, J9. The J9 header is not populated in this picture.

Figure 3.3 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 RTC72423

If installed, the real-time clock RTC72423 (EPSON, U8) is mapped in the I/O address space 0x0200. It must be backed up with a lithium coin battery. The RTC is accessed via software drivers ***rtc_init()*** or ***rtc_rds()*** (see Appendix C and the Software chapter for details). A sample program is provided **\tern\186\samples\sc\rtc_init.c**. Its data sheet can be found in the \tern_docs\parts directory, filename "rtc7242xam.pdf".

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. An example of a program showing a similar application can be found in ***tern\186\samples\ae\poweroff.c***.

Dual 12-bit DAC (DAC7612U)

The DAC7612 is a dual, 12-bit digital-to-analog converter with guaranteed 12-bit monotonicity performance over the industrial temperature range. It requires a single +5V supply and contains an input shift register, latch, 2.435V reference, a dual DAC, and high speed rail-to-rail amplifiers. For a full-scale step, each output will settle to 1LSB within 7µs.

The DAC7612 uses a four wire serial interface to the CPU. The CPU on the SensorCore uses four lines from the Am186ER to drive the serial interface (Data In, Clock, Chip select and Latch Data) in an 8-lead SOIC package. The SensorCore offers one DAC7612, providing 2 12-bit serial DAC channels. The DAC7612 outputs can support a capacitive load of 500pF.

The DAC is located in the U15 position with analog outputs routed to the J4 pin header, pins 10 & 12. See the schematic at the end of this technical manual.

Refer to data sheet in the **tern_docs\parts** directory of the TERN CD and to sample code in the **tern\186\samples\sc\sc_da.c** directory for additional information.

LTC2448, 24-bit ADC

The ***SC*** features three high-speed LTC2448 delta-sigma ADCs, interfaced through a high speed Sync serial port. Each LTC2448 chip offers 8 ch. differential or 16 ch. single-ended input channels. Variable speed/resolution settings can be configured. A peak single-channel output rate of 8 KHz can be achieved. At a sample rate of 1.76KHz, readings are accurate to 18+ bits in experimental conditions.

The LTC2448 works well with 500 ohm and lower impedance sensors, such as 350 ohm Strain gages, current shunts, RTDs, resistive sensors (500ohm an lower), and 4-20mA current loop sensors with 100 ohm sense resistors. The ***SC*** will also work directly with thermocouples. We can install a 2.5V precision reference with temperature sensor(LT1019) to minimize input current and providing local temperature measurement for thermocouple applications. There are 8+ million counts of resolution in the input span. If desired, you could put a divider at the input to increase the input range.

A sample program **ssi_ad24.c** can be found in the **`c:\tern\186\samples\sc`** directory. See the data sheet, **ltc2448.pdf** in the **tern_docs\parts** directory.

Compact Flash Interface

By utilizing the compact flash interface on the **SC**, 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\sc** directory includes sample code, **sc_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. Refer to **sc.ide** which has the demo built and ready for download.

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 Headers J1 – J5

There are two 20x2 0.1 spacing headers for SensorCore expansion. Most signals are directly routed to the Am186ER processor.

J3 Signal			
E01	1	2	E00
E03	3	4	E02
E05	5	6	E04
E07	7	8	E06
E09	9	10	E08
E11	11	12	E10
E13	13	14	E12
E15	15	16	E14
C01	17	18	C00
C03	19	20	C02
C05	21	22	C04
C07	23	24	C06
C09	25	26	C08
C11	27	28	C10
C13	29	30	C12
C15	31	32	C14
B01	33	34	B00
B03	35	36	B02
B05	37	38	B04
GND	39	40	B07

J1 Signal			
VCC	1	2	GND
	3	4	CLK
	5	6	GND
	7	8	D0
	9	10	D1
	11	12	D2
D15	13	14	D3
/RST	15	16	D4
RST	17	18	D5
P16	19	20	D6
D14	21	22	D7
D13	23	24	GND
	25	26	P12
D12	27	28	A7
/WR	29	30	A6
/RD	31	32	A5
D11	33	34	A4
D10	35	36	A3
D9	37	38	A2
D8	39	40	A1

Table 3.4 Signals for J3 (ADC input) and J1 (Expansion), 20x2 ports

J2 Signal			
GND	1	2	VCC
P4	3	4	P15
P0	5	6	P6
P11	7	8	/INT1
P3	9	10	P2
/INT3	11	12	P1

J5 Signal			
GND	1	2	GND
GND	3	4	GND
GND	5	6	GND
GND	7	8	
REF	9	10	VCC

J4 Signal			
B08	1	2	B06
B10	3	4	B09
B11	5	6	B12
B13	7	8	B14
B15	9	10	VA
GND	11	12	VB

Table 3.5 Signals for J2 (I/O) , J4 (ADC input), and J5 headers

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.

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 RE.LIB

RE.LIB is a C library for basic SC operations. It includes the following modules: AE.OBJ, SER0.OBJ, SER1R.OBJ, and AEEE.OBJ. You need to link to RE.LIB in your applications and include the corresponding header files in your source code. 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, from CPU
SER1R.H	External UART SCC26C92
AEEE.H	on-board EEPROM

Not all functions in the above modules will apply to the SC. For example, “ae.h” was originally created for the A-Engine. Therefore, “ae.h” will include routines for the TLC2543 (for example), not installed on the SC. The user will need to include the header file “sc.h” to provide routines for the SC devices. Although “ae.h” was created for a different controller, it will still be needed for a variety of routines used by the SC, such as timers, interrupts, and others. Refer to the actual header file itself to determine which is needed for a certain application.

4.2 Functions in AE.OBJ

4.2.1 SensorCore Initialization

ae_init

This function should be called at the beginning of every program running on SC 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 Am186ER Microcontroller User's manual.

- Initialize the upper chip select to support the on-board flash. The CPU registers are configured such that:
  - Address space for the Flash 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. Most pins are set up as default input, except for P29 (used for driving the LED), pins for SER0, and others.

```
outport(0xff78,0xc7bc);     // PDIR1, TxD,RxD,PCS0,PCS1,P29&P22 Output
outport(0xff76,0x2040);     // PIOM1
outport(0xff72,0xec7b);     // PDIR0, A18,A17,PCS6,PCS5, P12 Output
outport(0xff70,0x1000);     // PIOM0
```

- Configure the PPI 82C55 to all inputs. You can reset these by writing to the command register.

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

```
outportb(0x0102,0x01);      // I20 high
```

The chip select lines are set to 15 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. Some TERN components, such as the Real-Time-Clock, might fail if the wait state is decreased too dramatically. A function is provided for this purpose.

void io_wait
Arguments: char wait
Return value: none.

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

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

4.2.2 External Interrupt Initialization

There are up to six external interrupt sources on the SC, consisting of five maskable interrupt pins (**INT4-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 9 of the AMD Am186ER Microcontroller User's Manual - or the R1100 user's manual, both available on the CD under the **amd_docs** directory. (**Remember, DMA channels to and from the serial port not available on the R1100.)**

TERN provides functions to enable/disable all of the 5 maskable 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);
```

See Chapter 9 of Am186ER technical manual (tern_docs) for additional details. Sample code is also available in the **tern\186\samples\ae** directory, '**intx.c**'.

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 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 SC. 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, initialize the appropriate 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. (Example, if using the DAC7612, P10 is needed as the chip select, so it will be unavailable for any other purpose while the DAC is being used).

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 14 of the AMD Am186ER User's Manual. Also see Table 3.2 in this manual.

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

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

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, normal operation
- 1, input with pullup/down
- 2, output
- 3, input without pull

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 SC can be used for a variety of applications. All three timers run at ¼ of the processor clock rate, which determines the maximum resolution that can be obtained. Be aware that if you enter power save mode, the timers will operate at a reduced speed as well.

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 10 of the AMD AM186ER User's Manual.

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 for pulse-width modulation with a variable duty cycle. These timers contain two max counters, where the output is high until the counter counts reaches maxcount A before switching and counting 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 10 of the AMD AM186ER 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 if the interrupt bit in the **T0CON/T1CON** is set, with other behavior possible depending on the value specified for the control register. If the interrupt bit is not set, the user can poll the status if the **MC** bit in the timer control registers. Polling the **MC** bit offers a way to monitor timer status without using interrupts.

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, and no I/O pins.

4.2.5 Analog-to-Digital Conversion

Delta-Sigma ADC LTC2448

The 3 LTC2448 ADC units (at locations U11, U12, U13) provide 48 channels of 0-2.5V analog single-ended (24 differential) inputs (or 0-1.25V input with 2.5V reference chip). For details regarding the hardware configuration, see the Hardware chapter.

The following functions will drive the 24-bit ADCs. The order of functions given here should be followed in actual implementation.

void ad24_init(void);

void ad24_setup(unsigned char chip, unsigned int control_byte);

void ad24_ssi_rd(unsigned char* raw);

The control byte, *control_byte,* drives the LTC2448 in 16 channel single-ended mode with value 0xb000.

In code, the control byte is calculated this way:

```
ch_sel=0;        //select channel
control_byte=control_byte+speed[10];    //add speed desired  to 0xb000
control_byte=control_byte+(ch_sel<<8);   //add channel selection w/ 8 bit left shift
```

NOTE: "*ch_sel*" and the desired channel signal do not match up. Instead use this scheme to select the desired signal on the board:

ch_sel	U11	U12	U13
0	B00	C00	E00
1	B02	C02	E02
2	B04	C04	E04
3	B06	C06	E06
4	B08	C08	E08
5	B10	C10	E10
6	B12	C12	E12
7	B14	C14	E14
8	B01	C01	E01
9	B03	C03	E03
10	B05	C05	E05
11	B07	C07	E07
12	B09	C09	E09
13	B11	C11	E11
14	B13	C13	E13
15	B15	C15	E15

The LTC2448 also supports 8 channel differential mode. This can be achieved by changing the control byte passed to the 'ad24_setup' routine to 0xa0000 (speed and channel selection is added on the same way as in single-ended mode). See the LTC2448 data sheet for details on how to define the control byte, '**LTC2448.pdf**' in the **tern_docs\parts** directory.

For a sample file demonstrating the use of the ADC, please see **ssi_ad24.c** in **`tern\186\samples\sc`**.

This sample is also included in the **sc.ide** test project in the **tern\186** directory.

4.2.6 Digital-to-Analog Conversion

Dual DAC7612

The dual DAC7612 uses a serial interface with the CPU for operation. Four control lines are used, /CS = P10, CLK = SCLK(P20), SDI = SDAT(P21), and LD=P26. Each PIO lines must be initialized as output (mode 2) for operation. The user defined function "*sc_da"* is provided to give a one statement interface with the device. The function can be found in the sample file, ***sc_da.c***, in the directory **\tern\186\samples\sc**.

Note: Three 24-bit ADC LTC2448 chips can be installed, but they all must be disabled while using DAC.

void sc_da
Arguments: unsigned int dat
Return value: none

This function drives the DAC at position U15, outputs are VA & VB. The argument *dat* determines which channels are to be written to as well as the value. The values for *dat* are calculated as follows:

dat=0x2000|(0x0fff&dac); for CHA

dat=0x3000|(0x0fff&dac); for CHB

where 0 < dat < 0xfff

See the data sheet. From the root of the installation CD, \tern_docs\parts\dac7612.pdf.

4.2.7 Other library functions

On-board supervisor MAX691 or LTC691

The watchdog timer offered by the MAX691 or LTC691 offers an excellent way to monitor improper program execution. If the watchdog timer (**J9**) 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. See \tern\186\samples\re\re_rtc.c for a sample program. There is a common data structure used to access and use both interfaces.

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

int rtc_rd
Arguments: TIM *r
Return value: int error_code

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

int rtc_rds
Arguments: char* realTime
Return value: int error_code

This function is slightly different from the rtc_rd function. It places the current value of the real time clock into a character string instead of the TIM structure, making it a more convenient function than rtc_rd.

This function places the current value of the real time clock in the char* realTime. The string has a format of "week year10 year1 month10 month1 day10 day1 hour10 hour1 min10 min1 second10 second1". The **rtc_rds** function also places a null terminating character at the end of the time string. It is important to note that you must be sure to make the destination character string long enough to hold the real time clock value plus the null character. A destination character string that is too short will result in the data immediately following the character string in memory to be overwritten, causing unknown results.

For example "3040503142500\0" represents Wednesday May 3, 2004 at 02:25.00 pm. There are only two positions for the year, so the user must decide how to determine the hundreds and thousands digit of the year. Here we just assume "04" correlates to the year 2004.

The length of char * realTime must be at least 14 characters, 13 plus one null terminating character.

This function returns 0 on success and returns 1 in case of error, such as the clock failing to respond.

Void rtc_init
Arguments: char* t
Return value: none

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

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

If, for example, the time to be initialized into the real time clock is Friday June 6, 2003, 10:55:30 am, the byte array would be initialized to: unsigned char t[14] = { 5, 0, 3, 0, 6, 0, 6, 1, 0, 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 ACTF Boot Utility or from some other address.

4.3 Functions in SER0.OBJ

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

The Am186ER only provides one asynchronous serial port. The SC comes standard with the SCC2691, providing one additional asynchronous port. The serial port on the Am186ER will be called SER0, and the UART from the SCC2691 will be referred to as SCC.

This section will discuss functions in **ser0.h** only, as SER0 pertains to the Am186ER.

By default, SER0 is used by the DEBUG kernel (re80_115.hex) for application download/debugging in STEP 1 and STEP 2. **The following examples that will be used, show functions for SER0, but since it is used by the debugger, you cannot directly debug SER0.** This section will describe its operation and software drivers. The following section will discuss SCC, which pertain to the external SCC2691 UART. SCC will be easier to implement in applications, as it can be directly debugged in the Paradigm C/C++ environment.

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 for **SER0 ONLY**. SCC has baud rate based upon different arguments. These are based on a 40 MHz CPU clock (80MHz boards will have all baud rates doubled).

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
16	28,800

Table 4.1 Baud rate values for ser0 only

As of January 25, 2004, the function argument "16" was added for initializing SER0. This new rate provides a baud rate of 28,000 for 40MHz boards, and 57,600 for 80MHz boards.

After initialization by calling **`s0_init()`**, SER0 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, **`ser0_in_buf`** (whose size is specified by the user), will automatically store the receiving serial data stream into the memory by DMA0 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 **`serhit0()`** and take out the data from the buffer with **`getser0()`**, 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.

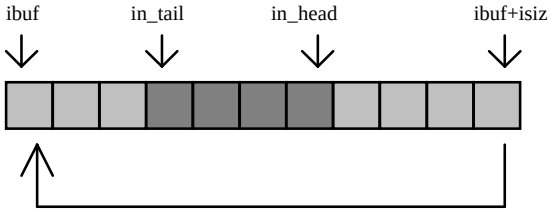


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), and baud rate (**baud**) are specified by the user with **`s0_init()`** with a default mode of 8-bit, 1 stop bit, no parity. After **`s0_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 Control Register (SP0CT) if necessary, as described in chapter 12 of the Am186ER 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 **`getser0()`** 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 **`serhit0()`** 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 **`getser0()`** to get the serial input data byte by byte using FIFO from the buffer. The in_tail pointer will automatically increment after every **`getser0()`** call. It is not necessary to suspend external devices from sending in serial data with /RTS. Only a hardware reset or **`s0_close()`** can stop this receiving operation.

For transmission, you can use **`putser0()`** to send out a byte, or use **`putsers0()`** to transmit a character string. You can put data into the transmit ring buffer, **`s0_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 **`putser0()`** 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.

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.

```
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 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. The following functions are shown as '**putser*n***', where ***n*** is the serial port in use. This section applies only to SER0, thus '**putser0**'.

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, you can re-initialize the serial port with s0_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 port available on the Am186ER processor has many other features that might be useful for your application. If you are interested in having more control, please read Chapter 12 of the manual for a detailed discussion of other features available to you.

4.4 Functions in SCC.OBJ

The functions found in this object file are prototyped in **sc.h** in the **`tern\186\include`** directory.

The SCC is a component that is used to provide a third asynchronous port. It uses an 8 MHz crystal, different from the system clock speed, for driving serial communications. This means the divisors and function arguments for setting up the baud rate for this second port are different than for SER0.

The SCC2691 component has its own 8 MHz crystal providing the clock signal. By default, this is set to 8 MHz to be consistent with earlier TERN controller designs. The highest standard baud rate is 19,200, as shown in the table below. If your application requires a higher standard baud rate (115,200, for example), it is possible to replace this crystal with a custom 3.6864 MHz crystal. A sample file demonstrating how the software would be changed for this application is **scc_echo.c**, found in the **tern\186\samples\sc** directory.

Function Argument	Baud Rate
1	110
2	150
3	300
4	600
5	1200
6	2400

Function Argument	Baud Rate
7	4800
8	9600 (default)
9	19,200
10	31,250
11	62,500
12	125,000
13	250,000

Unlike the other serial ports, DMA transfer is not used to fill the input buffer for SCC. Instead, an interrupt-service-routine is used to place characters into the input buffer. If the processor does not respond to the interrupt—because it is masked, for example—the interrupt service routine might never be able to complete this process. Over time, this means data might be lost in the SCC as bytes overflow.

Special control registers are used to define how the SCC operates. For a detailed description of registers **MR1** and **MR2**, please see '*sc.h*'. In most TERN applications, MR1 is set to ***0x57***, and MR2 is set to ***0x07***. This configures the SCC for no flow control (RTS, CTS not used/checked), no parity, 8-bit, normal operation. Other configurations are also possible, providing self-echo, even-odd parity, up to 2 stop bits, 5 bit operation, as well as automatic hardware flow control.

Initialization occurs in a manner otherwise similar to SER0. A **COM** structure is once again used to hold state information for the serial port. The in-bound and out-bound buffers operate as before, and must be provided upon initialization.

scc_init
Arguments: unsigned char m1, unsigned char m2, unsigned char b, unsigned char* ibuf, int isiz, unsigned char* obuf, int osiz, COM *c
Return value: none

This initializes the SCC2691 serial port to baud rate **b**, as defined in the table above. The values in **m1** and **m2** specify the values to be stored in to **MR1** and **MR2**. As discussed above, these values are normally ***0x57*** and ***0x07***, as shown in TERN sample programs.

ibuf and **isiz** define the input buffer characteristics, and **obuf** and **osiz** define the output buffer.

After initializing the serial port, you must also set up the interrupt service routine. The SCC2691 UART takes up external interrupt **/INT0** on the CPU, and you must set up the appropriate interrupt vector to handle this. An interrupt service routine, **scc_isr()**, has been written to handle the interrupt, and it enables/disables the interrupt as needed to transmit and receive data with the data buffers. So, after initialization, you will need to make a call to do this:

```
int0_init(1, scc_isr);
```

By default, the SCC is disabled for both *transmit* and *receive*. Before using the port, you will need to enable these functionalities.

When using RS232 in full-duplex mode, *transmit* and *receive* functions should both be enabled. Once this is done, you can transmit and receive data as needed. If you do need to do limited flow control, the MPO pin on the J1 header can be used for RTS. For a sample file showing RS232 full duplex communications, please see **ae_scc.c** in the directory **`tern\186\samples\ae`**.

RS485 is slightly more complex to use than RS232. RS485 operation is half-duplex only, which means transmission does not occur concurrently with reception. The RS485 driver will echo back bytes sent to the SCC. As a result, assuming you are using the RS485 driver installed on another TERN peripheral board, you will need to disable *receive* while transmitting. While transmitting, you will also need to place the

RS485 driver in transmission mode as well. This is done by using **scc_rts(1)**. This uses pin MPO (multi-purpose output) found on the J1 header. While you are receiving data, the RS485 driver will need to be placed in receive mode using **scc_rts(0)**. For a sample file showing RS485 communication, please see **ae_rs485.c** in the directory **`tern\186\samples\ae`**.

en485
Arguments: int i
Return value: none

This function sets the pin MPO either high (i = 1) or low (i = 0). The function scc_rts() actually has a similar function, by pulling the same pin high or low, but is intended for use in flow control.

scc_send_e/scc_rec_e
Arguments: none
Return value: none

This function enables transmission or reception on the SCC2691 UART. After initialization, both of these functions are disabled by default. If you are using RS485, only one of these two functions should be enabled at any one time.

scc_send_reset/scc_rec_reset
Arguments: none
Return value: none

This function resets the state of the send and receive function of the SCC2691. One major use of these functions is to disable *transmit* and *receive*. If you are using RS485, you will need to use this feature when transitioning from transmission to reception, or from reception to transmission.

Transmission and reception of data using the SCC is in most ways identical to SER0. The functions used to transmit and receive data are similar. For details regarding these functions, please refer to the previous section.

putser_scc
See: **`putser`*`n`***

putsers_scc
See: **`putsers`*`n`***

getser_scc
See: **`getser`*`n`***

getsers_scc
See: **`getsers`*`n`***

Flow control is also handled in a mostly similar fashion. The CTS pin corresponds to the MPI pin, which is not connected to either one of the headers. The RTS pin corresponds to the MPO pin found on the J1 header.

scc_cts
See: **`s`*`n`*`_cts`**

scc_rts
See: **`s`*`n`*`_rts`**

Other SCC functions are similar to those for SER0.

scc_close
See: **s*n*_close**
serhit_scc
See: **s*n*_hit**
clean_ser_scc
See: **clean_s*n***
Occasionally, it might also be necessary to check the state of the SCC for information regarding errors that might have occurred. By calling **scc_err**, you can check for framing errors, parity errors (if parity is enabled), and overrun errors.

4.5 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 application use.

ee_wr
Arguments: int addr, unsigned char dat
Return value: int status

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

ee_rd
Arguments: int addr
Return value: int data

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

4.6 Other Sample code

The following is a list of other sample code available for the SC. Each will show an example implementation of the specific hardware and are located in the tern\186\samples\sc directory. Most can also be found in the **sc.ide** test project.

tern\186\samples\sc\rtc_init.c // Real Time clock

tern\186\samples\sc\sc_cf.c // file system demo

4.6.1 File system support

TERN libraries support FAT file system for the Compact Flash interface. Refer to Chapter 4 of the FlashCore technical manual (tern_docs\manuals\flashcore.pdf) for a summary of the available routines. The libraries and header files are as follows:

fileio.h

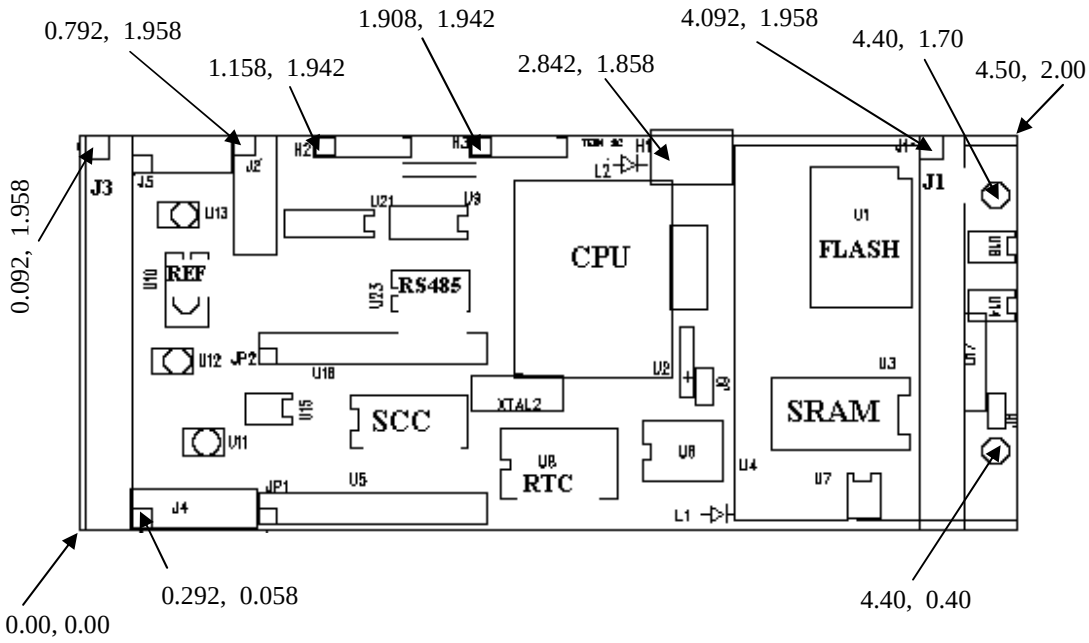
filegio.h

filesy16.lib

mm16.lib

Appendix A: SensorCore(SC) Layout

All dimensions are in inches.



Appendix B: UART SCC2691

1. Pin Description

D0-D7	Data bus, active high, bi-directional, and having 3-State
/CEN	Chip enable, active-low input
/WRN	Write strobe, active-low input
/RDN	Read strobe, active-low input
A0-A2	Address input, active-high address input to select the UART registers
RESET	Reset, active-high input
INTRN	Interrupt request, active-low output
X1/CLK	Crystal 1, crystal or external clock input
X2	Crystal 2, the other side of crystal
RxD	Receive serial data input
TxD	Transmit serial data output
MPO	Multi-purpose output
MPI	Multi-purpose input
Vcc	Power supply, +5 V input
GND	Ground

2. Register Addressing

A2	A1	A0	READ (RDN=0)	WRITE (WRN=0)
0	0	0	MR1,MR2	MR1, MR2
0	0	1	SR	CSR
0	1	0	BRG Test	CR
0	1	1	RHR	THR
1	0	0	1x/16x Test	ACR
1	0	1	ISR	IMR
1	1	0	CTU	CTUR
1	1	1	CTL	CTLR

Note:

ACR = Auxiliary control register
BRG = Baud rate generator
CR = Command register
CSR = Clock select register
CTL = Counter/timer lower
CTLR = Counter/timer lower register
CTU = Counter/timer upper
CTUR = Counter/timer upper register
MR = Mode register
SR = Status register
RHR = Rx holding register
THR = Tx holding register

3. Register Bit Formats

MR1 (Mode Register 1):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RxRTS 0 = no 1 = yes	RxINT 0=RxRDY 1=FFULL	Error 0 = char 1 = block	___Parity Mode___ 00 = with parity 01 = Force parity 10 = No parity 11 = Special mode		Parity Type 0 = Even 1 = Odd In Special mode: 0 = Data 1 = Addr	Bits per Character 00 = 5 01 = 6 10 = 7 11 = 8	

MR2 (Mode Register 2):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Channel Mode	TxRTS	CTS Enable Tx	Stop Bit Length (add 0.5 to cases 0-7 if channel is 5 bits/character)
00 = Normal 01 = Auto echo 10 = Local loop 11 = Remote loop	0 = no 1 = yes	0 = no 1 = yes	0 = 0.563 4 = 0.813 8 = 1.563 C = 1.813 1 = 0.625 5 = 0.875 9 = 1.625 D = 1.875 2 = 0.688 6 = 0.938 A = 1.688 E = 1.938 3 = 0.750 7 = 1.000 B = 1.750 F = 2.000

CSR (Clock Select Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Receiver Clock Select	Transmitter Clock Select
when ACR[7] = 0: 0 = 50 1 = 110 2 = 134.5 3 = 200 4 = 300 5 = 600 6 = 1200 7 = 1050 8 = 2400 9 = 4800 A = 7200 B = 9600 C = 38.4k D = Timer E = MPI-16x F = MPI-1x when ACR[7] = 1: 0 = 75 1 = 110 2 = 134.5 3 = 150 4 = 300 5 = 600 6 = 1200 7 = 2000 8 = 2400 9 = 4800 A = 7200 B = 1800 C = 19.2k D = Timer E = MPI-16x F = MPI-1x	when ACR[7] = 0: 0 = 50 1 = 110 2 = 134.5 3 = 200 4 = 300 5 = 600 6 = 1200 7 = 1050 8 = 2400 9 = 4800 A = 7200 B = 9600 C = 38.4k D = Timer E = MPI-16x F = MPI-1x when ACR[7] = 1: 0 = 75 1 = 110 2 = 134.5 3 = 150 4 = 300 5 = 600 6 = 1200 7 = 2000 8 = 2400 9 = 4800 A = 7200 B = 1800 C = 19.2k D = Timer E = MPI-16x F = MPI-1x

CR (Command Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Miscellaneous Commands		Disable Tx	Enable Tx	Disable Rx	Enable Rx
0 = no command 1 = reset MR pointer 2 = reset receiver 3 = reset transmitter 4 = reset error status 5 = reset break change INT 6 = start break 7 = stop break	8 = start C/T 9 = stop counter A = assert RTSN B = negate RTSN C = reset MPI change INT D = reserved E = reserved F = reserved	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes

SR (Channel Status Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Received Break	Framing Error	Parity Error	Overrun Error	TxEMT	TxRDY	FFULL	RxRDY
0 = no 1 = yes *	0 = no 1 = yes *	0 = no 1 = yes *	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes

Note:
* These status bits are appended to the corresponding data character in the receive FIFO. A read of the status register provides these bits [7:5] from the top of the FIFO together with bits [4:0]. These bits are cleared by a reset error status command. In character mode they are reset when the corresponding data character is read from the FIFO.

ACR (Auxiliary Control Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

BRG Set Select	Counter/Timer Mode and Source	Power-Down Mode	MPO Pin Function Select
0 = Baud rate set 1, see CSR bit format 1 = Baud rate set 2, see CSR bit format	0 = counter, MPI pin 1 = counter, MPI pin divided by 16 2 = counter, TxC-1x clock of the transmitter 3 = counter, crystal or external clock (x1/CLK) 4 = timer, MPI pin 5 = timer, MPI pin divided by 16 6 = timer, crystal or external clock (x1/CLK) 7 = timer, crystal or external clock (x1/CLK) divided by 16	0 = on, power down active 1 = off normal	0 = RTSN 1 = C/TO 2 = TxC (1x) 3 = TxC (16x) 4 = RxC (1x) 5 = RxC (16x) 6 = TxRDY 7 = RxRDY/FFULL

ISR (Interrupt Status Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

MPI Pin Change	MPI Pin Current State	Not Used	Counter Ready	Delta Break	RxRDY/ FFULL	TxEMT	TxRDY
0 = no 1 = yes	0 = low 1 = high		0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes

IMR (Interrupt Mask Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

MPI Change Interrupt	MPI Level Interrupt	Not Used	Counter Ready Interrupt	Delta Break Interrupt	RxRDY/ FFULL Interrupt	TxEMT Interrupt	TxRDY Interrupt
0 = off 1 = 0n	0 = off 1 = 0n		0 = off 1 = 0n	0 = off 1 = 0n	0 = off 1 = 0n	0 = off 1 = 0n	0 = off 1 = 0n

CTUR (Counter/Timer Upper Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
C/T [15]	C/T [14]	C/T [13]	C/T [12]	C/T [11]	C/T [10]	C/T [9]	C/T [8]

CTLR (Counter/Timer Lower Register):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
C/T [7]	C/T [6]	C/T [5]	C/T [4]	C/T [3]	C/T [2]	C/T [1]	C/T[0]

Appendix C: RTC72421 / 72423

Function Table

Address					Data					
A_3	A_2	A_1	A_0	Register	D_3	D_2	D_1	D_0	Count Value	Remarks
0	0	0	0	S_1	s_8	s_4	s_2	s_1	0~9	1-second digit register
0	0	0	1	S_{10}		s_{40}	s_{20}	s_{10}	0~5	10-second digit register
0	0	1	0	MI_1	mi_8	mi_4	mi_2	mi_1	0~9	1-minute digit register
0	0	1	1	MI_{10}		mi_{40}	mi_{20}	mi_{10}	0~5	10-minute digit register
0	1	0	0	H_1	h_8	h_4	h_2	h_1	0~9	1-hour digit register
0	1	0	1	H_{10}		PM/AM	h_{20}	h_{10}	0~2 or 0~1	PM/AM, 10-hour digit register
0	1	1	0	D_1	d_8	d_4	d_2	d_1	0~9	1-day digit register
0	1	1	1	D_{10}			d_{20}	d_{10}	0~3	10-day digit register
1	0	0	0	MO_1	mo_8	mo_4	mo_2	mo_1	0~9	1-month digit register
1	0	0	1	MO_{10}				mo_{10}	0~1	10-month digit register
1	0	1	0	Y_1	y_8	y_4	y_2	y_1	0~9	1-year digit register
1	0	1	1	Y_{10}	y_{80}	y_{40}	y_{20}	y_{10}	0~9	10-year digit register
1	1	0	0	W		w_4	w_2	w_1	0~6	Week register
1	1	0	1	Reg D	30s Adj	IRQ Flag	Busy	Hold		Control register D
1	1	1	0	Reg E	t_1	t_0	INT/ STD	Mask		Control register E
1	1	1	1	Reg F	Test	24/ 12	Stop	Rest		Control register F

Note: 1) INT/STD = Interrupt/Standard, Rest = Reset;

2) Mask AM/PM bit with 10's of hours operations;

3) Busy is read only, IRQ can only be set low ("0");

4)

Data bit	PM/AM	INT/STD	24/12
1	PM	INT	24
0	AM	STD	12

5) Test bit should be "0".

Appendix D: 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 rtc_init(unsigned char * time) ae.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 rtc_rd(TIM *r) ae.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

---

void scc_init(unsigned char m1, unsigned char m2, unsigned char b, unsigned char* ibuf,int isiz, unsigned char* obuf,int osiz, COM *c) sc.h

Serial port 0, 1 initialization.

```
Var:  m1 = SCC691 MR1
      m2 = SCC691 MR2
      b – baud rate. Table below for 8MHz Clock.
      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.
```

m1 bit	Definition
7	(RxRTS) receiver request-to-send control, 0=no, 1=yes
6	(RxINT) receiver interrupt select, 0=RxRDY, 1=FIFO FULL
5	(Error Mode) Error Mode Select, 0 = Char., 1=Block
4-3	(Parity Mode), 00=with, 01=Force, 10=No, 11=Special
2	(Parity Type), 0=Even, 1=Odd
1-0	(# bits) 00=5, 01=6, 10=7, 11=8

m2 bit	Definition
7-6	(Modes) 00=Normal, 01=Echo, 10=Local loop, 11=Remote loop
5	(TxRTS) Transmit RTS control, 0=No, 1= Yes
4	(CTS Enable Tx), 0=No, 1=Yes
3-0	(Stop bit), 0111=1, 1111=2

b	baud (8MHz)
1	110
2	150
3	300
4	600
5	1200
6	2400
7	4800
8	9600
9	19200
10	31250
11	62500
12	125000
13	250000

Reference: scc_echo.c

---

int putser0(unsigned char ch, COM *c); ser0.h
int putser_scc(unsigned char ch, COM *c); sc.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
int putsers_scc(unsigned char ch, COM *c); sc.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
int serhit_scc(COM *c); sc.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
unsigned char getser_scc(COM *c); sc.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

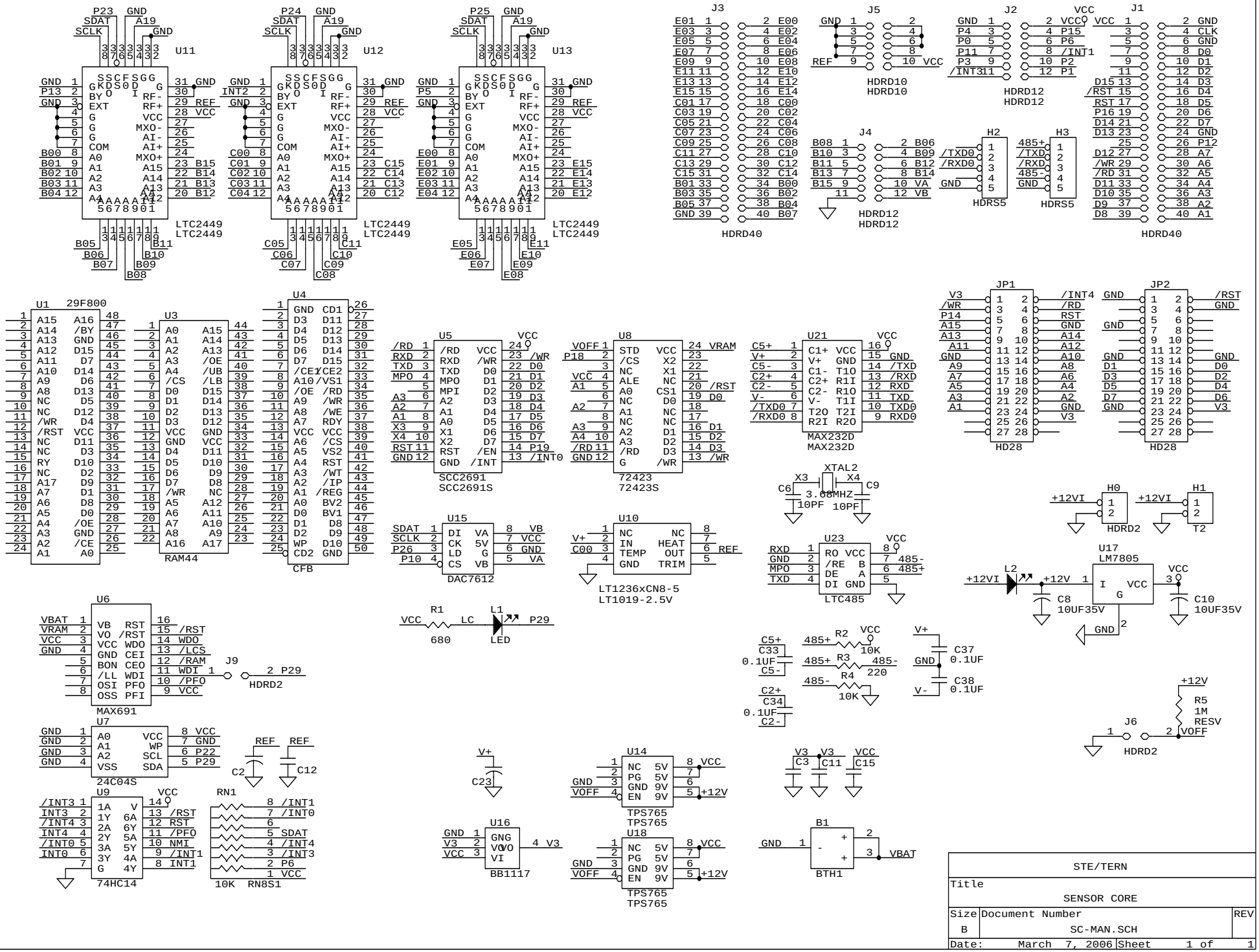
---

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


P23 GND SDAT A19 SCLK GND U11
P24 GND SDAT A19 SCLK GND U12
P25 GND SDAT A19 SCLK GND U13
SSCFSGG GKDS0D G BY O I RF- EXT RF+ G VCC G MXO- G AI- COM AI+ A0 MXO+ A1 A15 A2 A14 A3 A13 A4 A12 AAAAAA11 5678901
LTC2449
LTC2449
J3
HDRD40
J5
HDRD10
HDRD10
J2
HDRD12
HDRD12
J1
HDRD40
J4
HDRD12
HDRD12
H2
H3
HDRS5
HDRS5
U1 29F800
RAM44
U3
CFB
U4
U5
SCC2691
SCC2691S
U8
72423
72423S
U21
MAX232D
MAX232D
JP1
HD28
JP2
HD28
XTAL2
3.68MHZ
10PF
10PF
H0
H1
HDRD2
T2
U17
LM7805
L2
C8 10UF35V
C10 10UF35V
U15
DAC7612
U10
LT1236xCN8-5
LT1019-2.5V
U23
LTC485
U6
MAX691
J9
HDRD2
R1 680
L1 LED
U7
24C04S
C2
C12
U9
74HC14
RN1
10K RN8S1
C33 0.1UF
R2 10K
R3 220
R4 10K
C37 0.1UF
C38 0.1UF
C34 0.1UF
R5 1M RESV
J6
HDRD2
C23
U14
TPS765
TPS765
C3
C11
C15
U16
BB1117
U18
TPS765
TPS765
B1
BTH1
STE/TERN
Title
SENSOR CORE
Size B
Document Number SC-MAN.SCH
REV
Date: March 7, 2006
Sheet 1 of 1