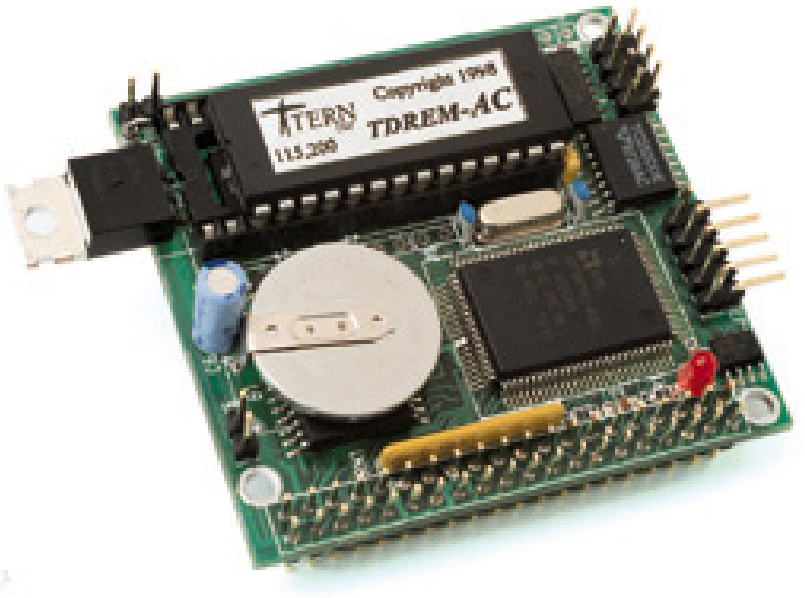
A-Core™

C/C++ Programmable, 16-bit Microprocessor Module
Based on the Am188ES



Technical Manual

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

COPYRIGHT

A-Core, A-Engine, and ACTF are trademarks of TERN, Inc.
Am188ES and Am186ES are trademarks of Advanced Micro Devices, Inc.
ParadigmC/C++ is a trademark of Paradigm Systems.
Microsoft, Windows95\98\2000\NT\ME\XP are trademarks of Microsoft Corporation.
IBM is a trademark of International Business Machines Corporation.

Version 3.0

May 19, 2010

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

© 1997-2010 TERN INC.
1950 5th Street, Davis, CA 95616, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: sales@tern.com ***http://www.tern.com***

Important Notice

TERN is developing complex, high technology integration systems. These systems are integrated with software and hardware that are not 100% defect free. ***TERN products are not designed, intended, authorized, or warranted to be suitable for use in life-support applications, devices, or systems, or in other critical applications. TERN*** and the Buyer agree that ***TERN*** will not be liable for incidental or consequential damages arising from the use of ***TERN*** products. It is the Buyer's responsibility to protect life and property against incidental failure.
TERN reserves the right to make changes and improvements to its products without providing notice.
Temperature readings for controllers are based on the results of limited sample tests; they are provided for design reference use only.

Chapter 1: Introduction

1.1 Functional Description

The ***A-Core*** (AC) is a low-cost, high performance, C/C++ programmable, 16-bit microprocessor core module. It is designed for embedded applications that require compactness, low power consumption, and high reliability. The ***A-Core*** is a minimized version of the ***A-Engine*** with compatible software drivers and C/C++ Evaluation or Development Kits, with reduced mechanical dimension, cost, and power consumption.

The ***A-Core*** can be integrated into an OEM product as a processor core component. It also can be used to build a smart sensor, or as a node in a distributed microprocessor system.

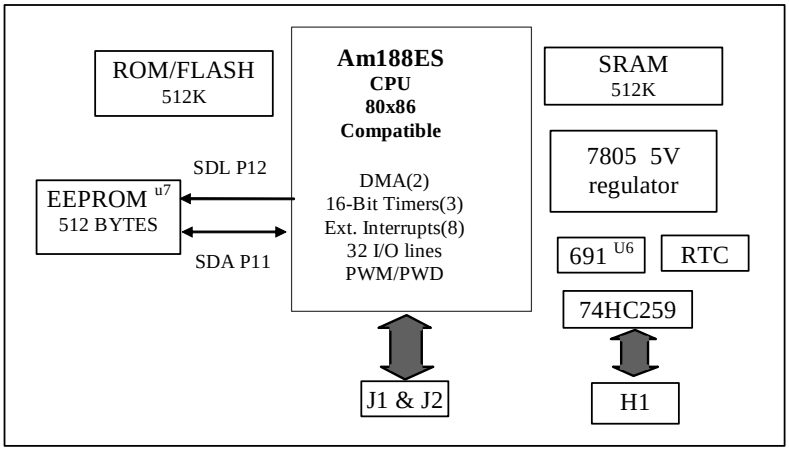


Figure 1.1 Functional block diagram of the A-Core

Measuring 2.2 by 2.3 by 0.3 inches, the ***A-Core*** offers a complete C/C++ programmable computer system with a 16-bit high performance CPU (40 MHz, 188) and operates at 40 MHz system clock with zero-wait-state. Optional features include up to 512K EPROM/Flash and up to 512K battery-backed SRAM. A 512-byte serial EEPROM is included on-board. An optional real-time clock provides information on the year, month, date, hour, minute, and second and an interrupt signal. The ***A-Core*** is available with on-board 5-volt power regulator and RS232 drivers, or the optional ***VE232*** interface board can be used to provide regulated 5V power and RS232 drivers for the AC.

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

There are three 16-bit programmable timers/counters and a watchdog timer. Two timers can be used to count or time external events, at a rate of up to 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.

There are 32 user-programmable I/O pins on the 188 CPU, and eight external interrupt inputs. A supervisor chip (MAX691) with power failure detection, a watchdog timer, an LED, and expansion ports are on-board.

1.2 Features

Standard Features

- Dimensions: 2.2 x 2.3 x 0.3 inches
- Easy to program in Paradigm C/C++
- Power consumption: 160 mA at 5V for 40 MHz
- Power saving mode: 30 mA at 5V for 40 MHz
- Power input: +5V regulated DC with VE232, or
  +9V to +12 V unregulated DC with on-board regulator
- Temperature: -40°C to +85°C
- 16-bit CPU (188 CPU), Intel 80x86 compatible
- High performance, zero-wait-state operation at 40 MHz
- 128KB SRAM, and up to 512KB Flash/ROM
- 2 high-speed PWM outputs and Pulse Width Demodulation
- 32 I/O lines from 188 CPU, 512-byte serial EEPROM
- 8 external interrupt inputs, 3 16-bit timer/counters
- 2 serial ports support 8-bit or 9-bit asynchronous communication
- Supervisor chip (691) for power failure, reset and watchdog

Optional Features

- 512KB SRAM
- Power regulator and RS232 drivers
- Real-time clock RTC72423, lithium coin battery
- VE232 add-on board for regulated 5V power & RS232 drivers

1.3 Physical Description

The physical layout of the A-Core is shown in Figure 1.2.

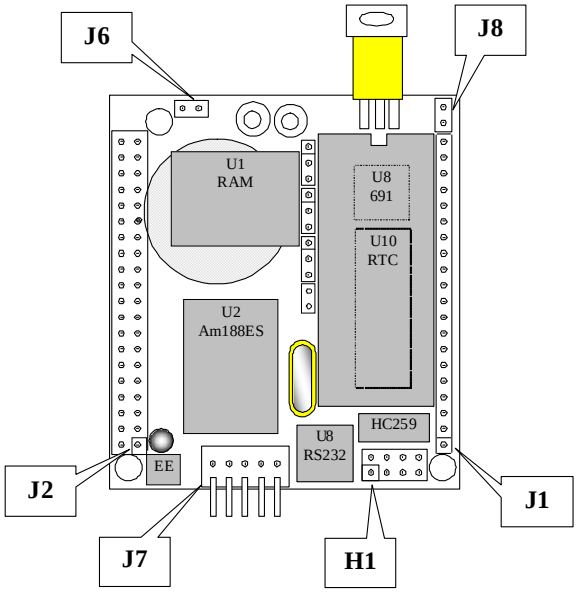
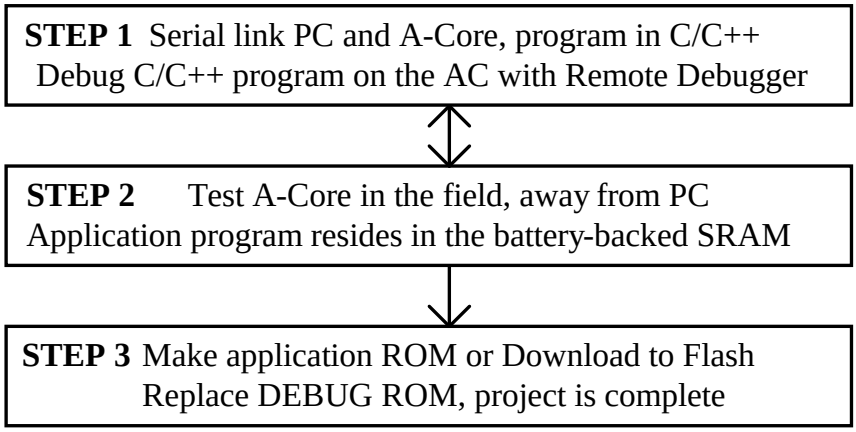


Figure 1.2 Physical layout of the A-Core

1.4 A-Core Programming Overview

Development of application software for the A-Core consists of three easy steps, as shown in the block diagram below.



The A-Core can be programmed from your PC via serial link with an RS232 interface. C/C++ programs can be remotely debugged over the serial link at a rate of 115,000 baud. The C/C++ Evaluation Kit (EV-P) or Development Kit (DV-P) from TERN provides a Paradigm C/C++ compiler, linker, editor, remote debugger, I/O driver libraries, sample programs, and schematics. These kits also include a DEBUG ROM (*AC_0_115)* to communicate with the remote debugger, a debug serial cable to the connect the controller to the PC, and a 9-volt wall transformer.

After remote debugging an application, test run the A-Core in the field, away from the PC, by changing a single jumper, with the application program residing in the battery-backed SRAM. When the field test is complete, application ROMs can be produced to replace the DEBUG ROM. The .HEX or .BIN file can be easily generated with the Paradigm C/C++ software (DV-P kit is required to generate production ROMs). TERN also offers on-board ACTF Flash programming via serial link.

The three steps in the development of a C/C++ application program are explained in detail below.

STEP 1: Debugging

- Write your C/C++ application program. ParadigmC/C++ provides a text editor.
- Connect your controller to your PC via the debug serial cable.
- Use Paradigm C/C++ to compile, link, download, and remote debug your C/C++ application program.

STEP 2: Standalone Field Test.

- Set the jumper on J6 pins 1-2 on the A-Core (Figure 1.3).

- At power-on or reset, if J6 pin 1 (A19) is low, the CPU will run the code that resides in the battery-backed SRAM.
- If a jumper is on J6 pins 1-2 at power-on or reset, the A-Core will operate in STEP 2. If the jumper is off J6 pins 1-2 at power-on or reset, the A-Core will operate in STEP 1. The status of J6 pin 1 (signal A19) of the 188 CPU is only checked at power-on or reset.

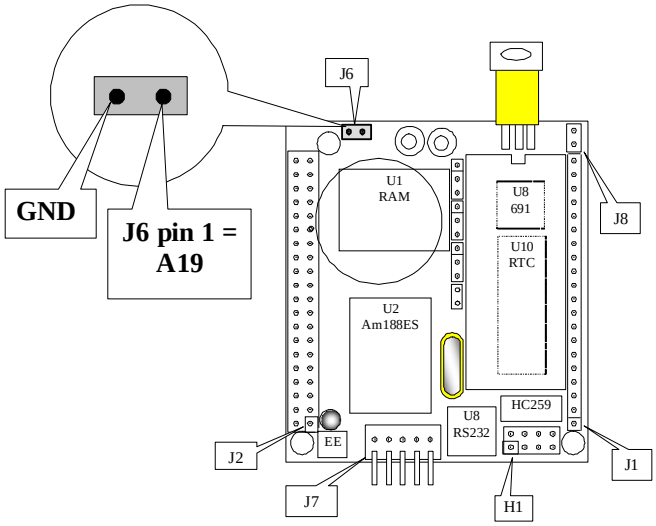


Figure 1.3 Location of Step 2 jumper on the A-Core (J6)

STEP 3: Generate the application .BIN or .HEX file, make production ROMs or download your program to FLASH via ACTF.

- If the Step 2 standalone field test is satisdactory, use ParadigmC/C++ to generate an application ROM to replace the DEBUG ROM (*AC_0_115*). Refer to the EV-P/DV-P manual for ROM generation instructions.

The DV-P Kit is required to complete Step 3.

1.5 VE232

The VE232 is an interface board for the A-Core that provides regulated +5V DC power and RS232 drivers. It converts TTL signals to and from RS232 signals. The VE232 is **NOT** needed if the on-board 5V regulator and RS-232 drivers are installed.

The VE232, shown in Figure 1.4, measures 2.3 x 1.57 inches. A wall transformer (9V, 300 mA) with a center negative DC plug (Ø=2.0 mm) should be used to power the A-Core via the VE232. The VE232 connects to A-Core via H1 (2x10 header). SER0 (J2) and SER1 (J3) on the VE232 are 2x5-pin headers for serial ports SER0 and SER1. SER0 is the default programming port.

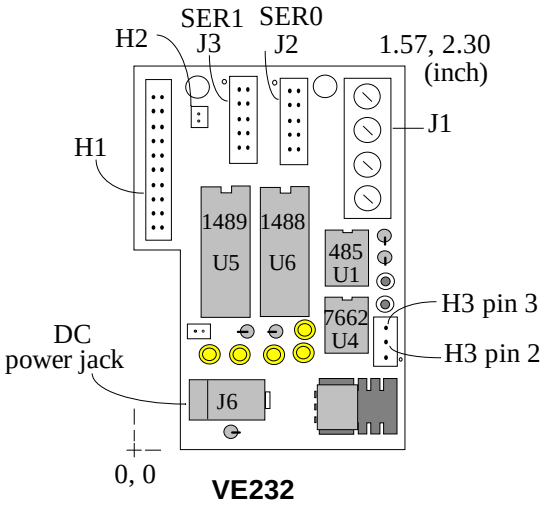


Figure 1.4 The VE232, an interface card for the A-Core

For further information on the VE232, please refer to Appendix B and to the VE232 schematic at the end of this manual.

1.6 Minimum Requirements for A-Core System Development

1.6.1 Minimum Hardware Requirements

- PC or PC-compatible computer with serial COMx port that supports 115,200 baud
- A-Core controller with DEBUG ROM *AC_0_115*
- VE232 interface board **OR** on-board 5V regulator and RS232/RS485 drivers
- Debug serial cable (RS232; DB9 connector for PC COM port and IDC 2x5 connector for controller)
- center negative wall transformer (+9V 500 mA)

1.6.2 Minimum Software Requirements

- TERN EV-P/DV-P Kit CD-ROM
- PC software environment: Windows 95/98/2000/NT/ME/XP

The C/C++ Evaluation Kit (EV-P) and C/C++ Development Kit (DV-P) are available from TERN. The EV-P kit is a limited-functionality version of the DV-P Kit. With the EV-P Kit, the user can program and debug the A-Core in STEP 1 and STEP 2, but cannot run STEP 3. To generate an application ROM/Flash file, make production version ROMs, and complete the project, the Development Kit (DV-P) is needed.

Chapter 2: Installation

2.1 Software Installation

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

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

2.2 Hardware Installation

Overview

- Install VE232 (if applicable):
  H1 connector of VE232 installs on J2 of the A-Core
- Connect debug serial cable:
  For debugging (Step One), place IDE connector on SER0 with red edge of cable at pin 1
- Connect wall transformer:
  Connect 9V wall transformer to power and plug into power jack

Hardware installation for the A-Core consists primarily of connecting the microcontroller to your PC. Either the VE232 must be used to supply regulated power and RS232 drivers, or the on-board regulator and RS-232 drivers are required.

2.2.1 Connecting the VE232 to the A-Core

Install the VE232 interface with the H1 (10x2) socket connector on the upper half of the J2 (dual row header) of the A-Core. Figure 2.1 and Figure 2.2 show the VE232 and the A-Core before and after installation.

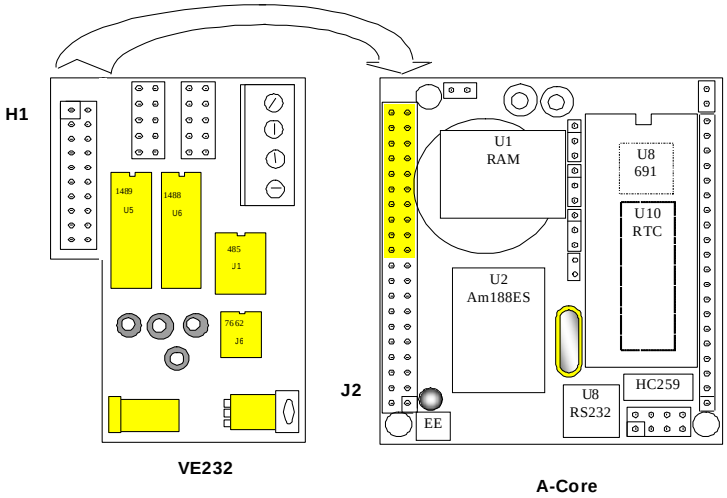
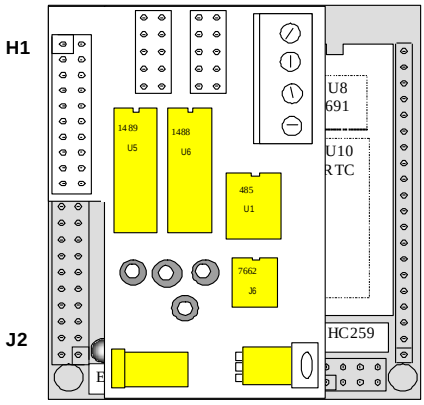


Figure 2.1 Before installing the VE232 on the A-Core



A-Core with VE232

Figure 2.2 After installing the VE232 on the A-Core

2.2.2 Connecting the A-Core to the PC

The following diagram (Figure 2.3) illustrates the connection between the A-Core, VE232 and the PC. The A-Core is linked to the PC via a serial cable (PC-V25).

The *AC_0_115* DEBUG ROM communicates through SER0 by default. Install the 5x2 IDC connector to the V232 on the SER0 header. ***IMPORTANT:*** *Note that the* ***red*** *side of the cable must point to pin 1 of the VE232 J2 header.* The DB9 connector should be connected to the PC's COM Ports (COM1 or COM2).

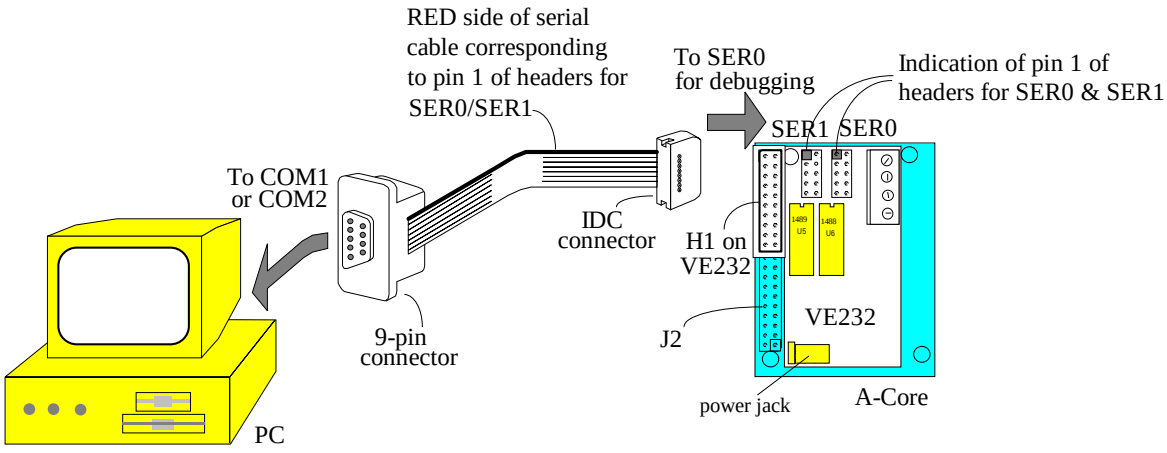


Figure 2.3 Connecting the A-Core and VE232 to the PC

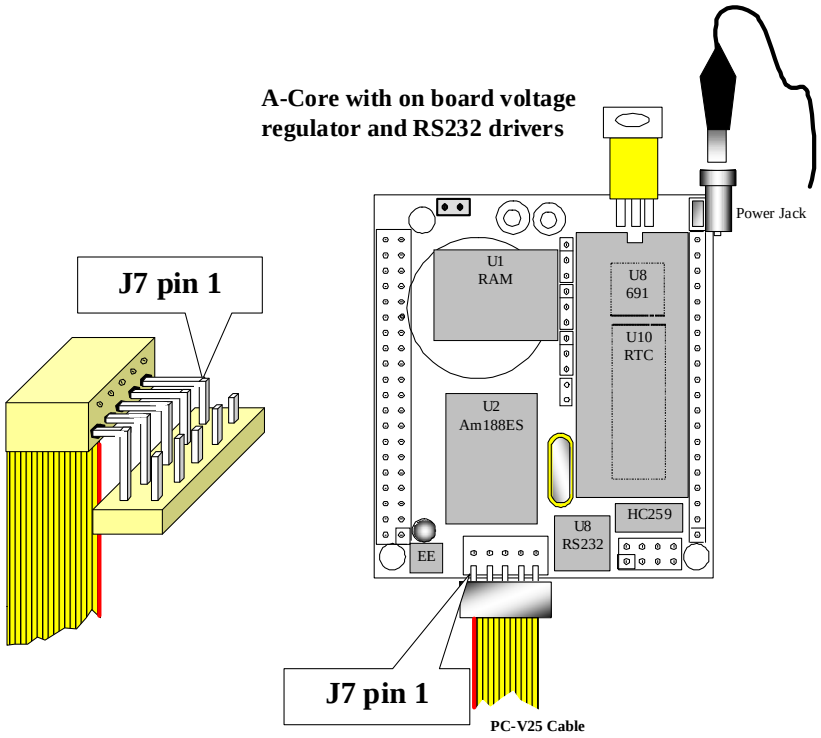


Figure 2.4 Connections for the A-Core with on-board regulator

2.2.3 Powering-on the A-Core

Connect a wall transformer +9V DC output to the VE232 DC power jack. If on-board regulator is installed, connect the transformer output to the DC power jack at J8 on the A-Core (Figure 2.5). The power supply from the wall transformer is connected to a power jack adapter.

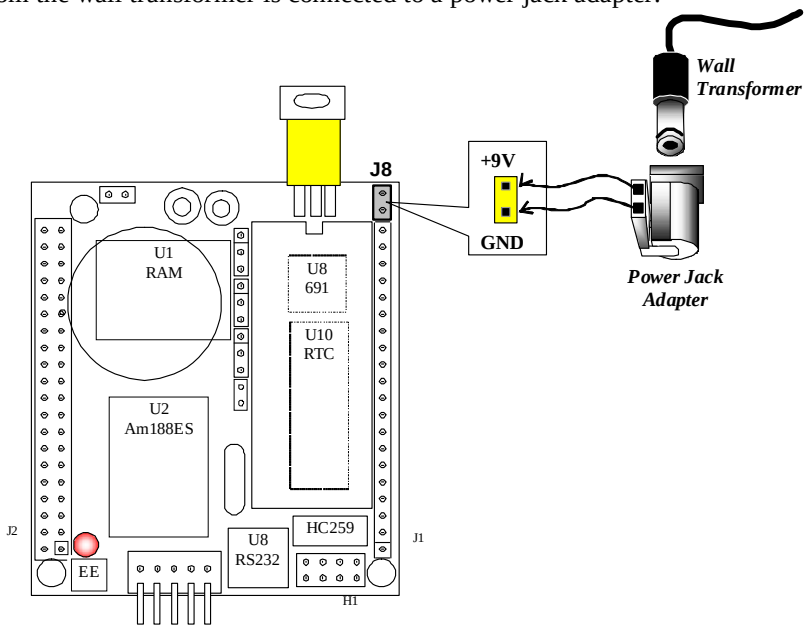


Figure 2.5 Power connection for A-Core with on-board power regulator

At power-on, the on-board LED should blink twice and remain on, as shown below.

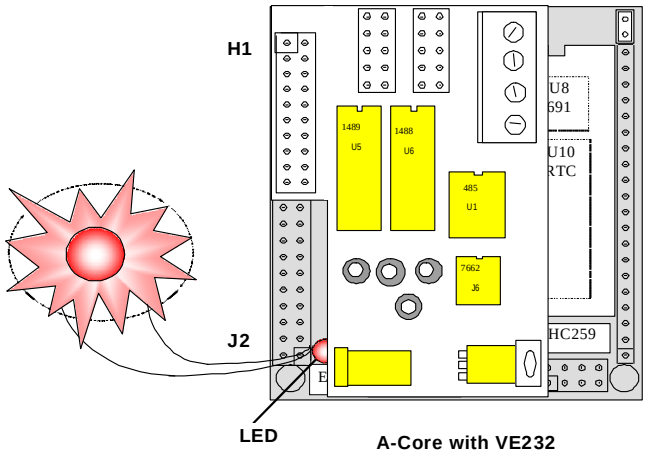


Figure 2.6 LED on the A-Core with VE232

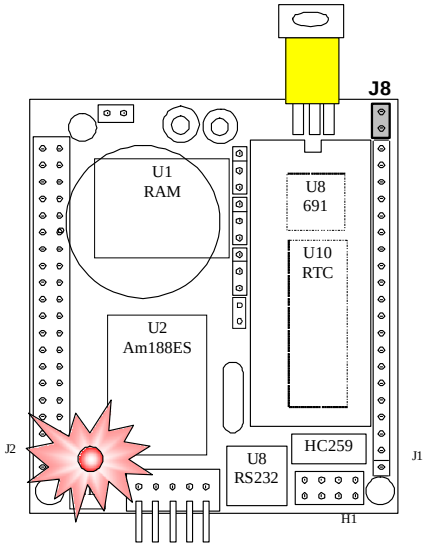


Figure 2.7 LED on the A-Core with on board voltage regulator and RS232 drivers

Chapter 3: Hardware

3.1 188 CPU - Introduction

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

3.2 188 CPU – Features

3.2.1 Clock

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

The system CLKOUTA signal is routed to J2 pin 37, default 40 MHz. The CLKOUTB signal is not connected in the A-Core.

The CLKOUTA remains active during reset and bus hold conditions. The A-Core 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=J2 pin 37.

3.2.2 External Interrupts

There are eight external interrupts: INT0-INT6 and NMI. All eight interrupts are pulled down with a 10K resistor.

INT0, J2 pin 18
INT1, J2 pin 19
INT2, J2 pin 20
INT3, J2 pin 21
INT4, J2 pin 23
INT5=P12=DRQ0, J2 pin 6, used by A-Core as output for LED/EE/HWD
INT6=P13=DRQ1, J2 pin 7
NMI, J2 pin 35

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

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

3.2.3 Asynchronous Serial Ports

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

- Full-duplex operation
- 7-bit, 8-bit and 9-bit data transfers
- Odd, even and no parity

- One stop bit
- Error detection
- Hardware flow control
- DMA transfers to and from serial ports
- Transmit and receive interrupts for each port
- Multidrop 9-bit protocol support
- Maximum baud rate of 1/16 of the CPU clock
- Independent baud rate generators

The software drivers for each serial port implement a ring-buffered DMA receiving and ring-buffered interrupt transmitting arrangement. See the samples files: **`s1_echo.c`** and **`s0_echo.c`**.

3.2.4 Timer Control Unit

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

Timer0 and Timer1 are connected to four external pins:

Timer0 output = P10 = J2 pin 9
Timer0 input = P11 = J2 pin 8
Timer1 output = P1 = J2 pin 10
Timer1 input = P0 = J2 pin 11

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

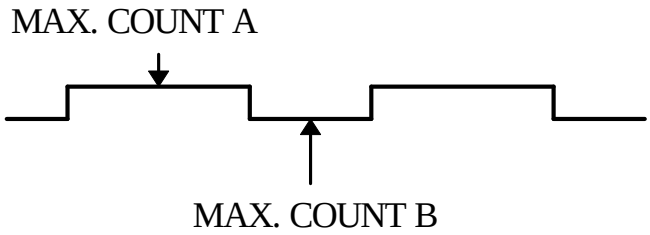
Timer2 is not connected to any external pin. It can be used as an internal timer for real-time coding or time-delay applications. It can also prescale Timer0 and Timer1 or be used as a DMA request source.

The maximum rate at which each timer can operate is 10 MHz, since each timer is serviced on every fourth clock cycle. Timer output takes up to six clock cycles to respond to clock or gate events. See sample programs ***timer0.c*** and ***ae_cnt0.c*** in **`\samples\ae`**.

3.2.5 PWM outputs and PWD

The Timer0 and Timer1 outputs can also be used to generate non-repetitive or variable-duty-cycle waveforms. The timer output takes up to 6 clock cycles to respond to the clock input. Thus the minimum timer output cycle is 25 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 a secondary maximum count register for variable duty cycle output. Using both the primary and secondary maximum count registers lets the timer alternate between two maximum values.



Pulse Width Demodulation can be used to measure the input signal's high and low phases on the INT2=J2 pin 20.

3.2.6 Power-save Mode

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

3.3 188 CPU PIO lines

The 188 CPU has 32 pins available as user-programmable I/O lines. Each of these pins can be used as a user-programmable input or output signal, if the normal shared function is not needed. A PIO line can be configured to operate as an input or output with or without a weak pull-up or pull-down, or as an open-drain output. A pin's behavior, either pull-up or pull-down, is pre-determined and shown below.

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

PIO	***Function***	***Power-On/Reset status***	***A-Core Pin No.***	***A-Core Initial***
P0	Timer1 in	Input with pull-up	J2 pin 11	
P1	Timer1 out	Input with pull-down	J2 pin 10	CLK_1
P2	/PCS6/A2	Input with pull-up	J2 pin 17	/PCS6
P3	/PCS5/A1	Input with pull-up	J2 pin 16	/PCS5
P4	DT/R	Normal	J2 pin 33	Input with pull-up
P5	/DEN/DS	Normal	J2 pin 30	Input with pull-up
P6	SRDY	Normal	J2 pin 38	Input with pull-down
P7	A17	Normal	N/A	A17
P8	A18	Normal	N/A	A18
P9	A19	Normal	J6 pin 1	Input with pull-up
P10	Timer0 out	Input with pull-down	J2 pin 9	Input with pull-down
P11	Timer0 in	Input with pull-up	J2 pin 8	Input with pull-up
P12	DRQ0/INT5	Input with pull-up	J2 pin 6	Output for LED/EE/HWD
P13	DRQ1/INT6	Input with pull-up	J2 pin 7	Input with pull-up
P14	/MCS0	Input with pull-up	J2 pin 29	Input with pull-up
P15	/MCS1	Input with pull-up	J2 pin 24	Input with pull-up
P16	/PCS0	Input with pull-up	J2 pin 14	/PCS0
P17	/PCS1	Input with pull-up	J2 pin 15	PPI, 82C55 select
P18	CTS1/PCS2	Input with pull-up	J2 pin 22	Input with pull-up
P19	RTS1/PCS3	Input with pull-up	J2 pin 31	Input with pull-up
P20	RTS0	Input with pull-up	J2 pin 27	Input with pull-up
P21	CTS0	Input with pull-up	J2 pin 36	Input with pull-up
P22	TxD0	Input with pull-up	J2 pin 34	TxD0
P23	RxD0	Input with pull-up	J2 pin 32	RxD0
P24	/MCS2	Input with pull-up	J2 pin 13	Input with pull-up
P25	/MCS3	Input with pull-up	J2 pin 12	Input with pull-up
P26	UZI	Input with pull-up	J2 pin 4	
P27	TxD1	Input with pull-up	J2 pin 28	TxD1
P28	RxD1	Input with pull-up	J2 pin 26	RxD1
P29	/CLKDIV2	Input with pull-up	J2 pin 5	Input with pullup*
P30	INT4	Input with pull-up	J2 pin 23	Input with pull-up

P31	INT2	Input with pull-up	J2 pin 20	Input with pullup

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

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

Three external interrupt lines are not shared with PIO pins:

INT0 = J2 pin 18
INT1 = J2 pin 19
INT3 = J2 pin 21

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

A-Core initialization on PIO pins in **ae_init()** is listed below:

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

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

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

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

Example:

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

void ***pio_wr***(char bit, char dat);
pio_wr(12,1); set P12 pin high, if P12 is in output mode
pio_wr(12,0); set P12 pin low, if P12 is in output mode

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

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

Signal	**Pin**	**Function**
P2	/PCS6	U4 RTC72423 chip select at base I/O address 0x0600
P9	A19	Step Two jumper
P11	Timer0 input	U7 24C04 EE data input The EE data output can be tri-state, while disabled

Signal	Pin	Function
P12	DRQ0/INT5	Output for LED or U7 serial EE clock or Hit watchdog
P16	/PCS0	U11 74HC259 chip select at base I/O address 0x0000
P22	TxD0	Default SER0 debug
P23	RxD0	Default SER0 debug

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

3.4 I/O Mapped Devices

3.4.1 I/O Space

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

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

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

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

I/O space	Select	Usage	Location
0x0000-0x00ff	/PCS0	74HC259	J2 pin 14=P16
0x0100-0x01ff	/PCS1	USER	J2 pin 15=P17
0x0200-0x02ff	/PCS2	USER	J2 pin 22=CTS1
0x0300-0x03ff	/PCS3	USER	J2 pin 31=RTS1
0x0400-0x04ff	/PCS4	Reserved	
0x0500-0x05ff	/PCS5	USER	J2 pin 16=P3
0x0600-0x06ff	/PCS6	RTC 72423	J2 pin 17=P2

To illustrate how to interface the A-Core with external I/O boards, a simple decoding circuit for interfacing to a 82C55 parallel I/O chip is shown below in Figure 3.1.

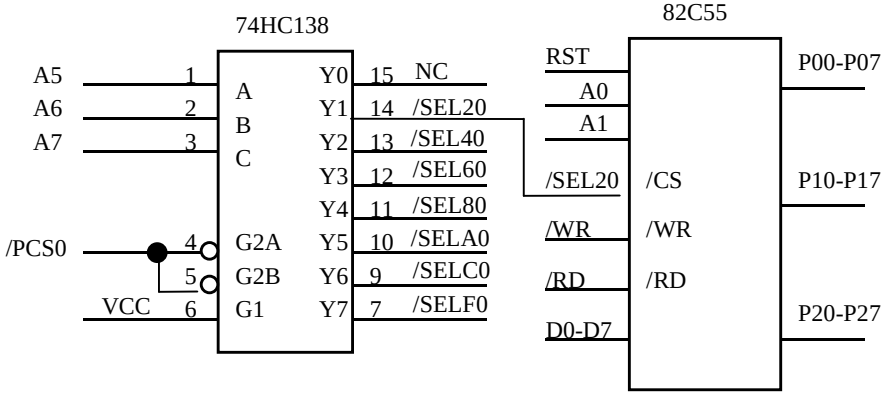


Figure 3.1 Interface the A-Core to external I/O devices

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 0x00 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.

3.4.2 74HC259

The 74HC259 8-bit decoder latch provides eight additional output lines for the A-Core. The 74HC259 is mapped in the I/O address space 0x0000. You may access this device by using the following code.

```
outportb(0x0000 + i, val); // i = output pin, val = 0/1 to set or reset latch.
```

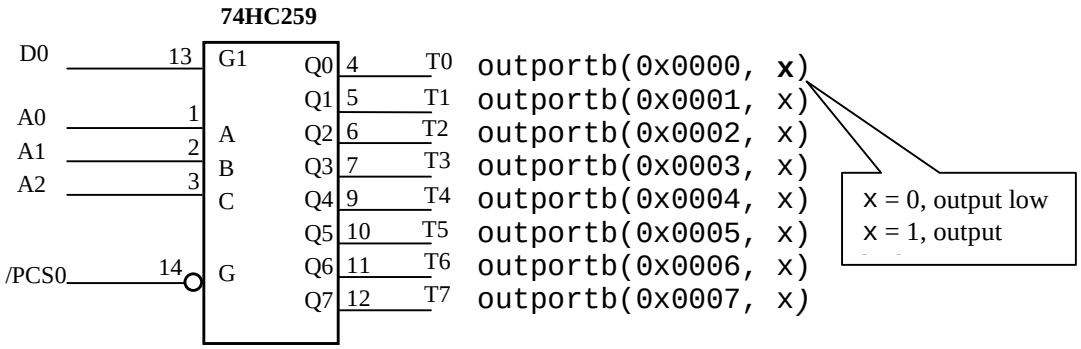


Figure 3.2 74HC259 diagram with corresponding outport addresses

3.4.3 Real-time Clock RTC72423

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

3.5 Other Devices

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

3.5.1 On-board Supervisor with Watchdog Timer

The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the A-Core 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 A-Core (see Figure 3.3). The watchdog timer provides a means of verifying proper software execution. In the user's application program, calls to the function **hitwd()** (a routine that toggles the P12=HWD pin of the MAX691) should be arranged such that the HWD pin is accessed at least once every 1.6 seconds. If the J9 jumper is on and the HWD pin is not accessed within this time-out period, the watchdog timer pulls the WDO pin low, which asserts /RESET. This automatic assertion of /RESET may recover the application program if something is wrong.

After the A-Core is reset, WDO remains low until a transition occurs at the WDI pin of the MAX691. When controllers are shipped from the factory the J9 jumper is off, which disables the watchdog timer.

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

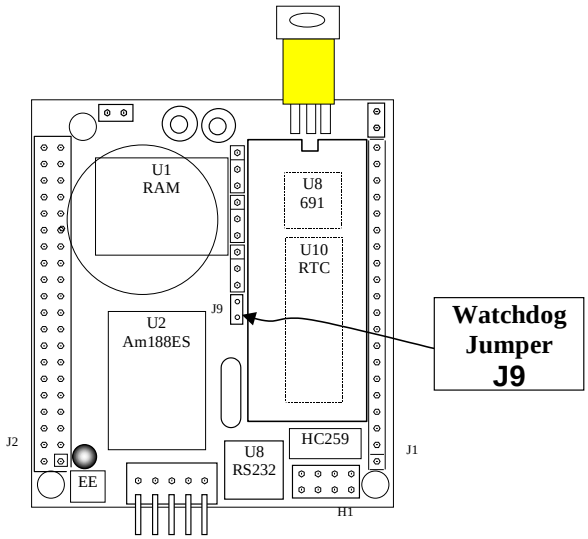


Figure 3.3 Location of watchdog timer enable jumper

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 RTC72423 are backed up. In normal use, the lithium battery should last about 3-5 years without external power being supplied. When the external power is on, the battery-switch-over circuit will select the VCC to connect to the VRAM.

3.5.2 EEPROM

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

3.6 Headers and Connectors

3.6.1 Expansion Headers

The A-Core has one 20x2, one 20x1, and one 4x2 pin header for expansion. Most signals are directly routed to the 188 CPU processor. These signals are 5V only, and any out-of-range voltages will most likely damage the board.

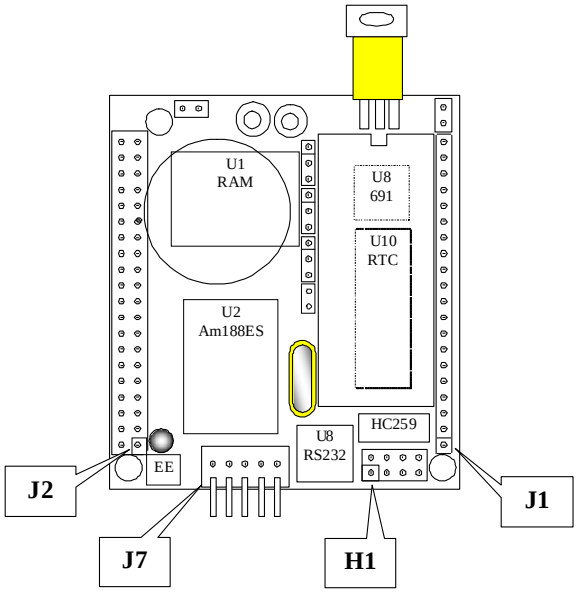


Figure 3.4 Pin 1 locations for J1, J2, J7 and H1

J1 Signal	
VCC	1
D7	2
D6	3
D5	4
D4	5
D3	6
D2	7
D1	8
D0	9
/WR	10
/RD	11
A0	12
A1	13
A2	14
A3	15
A4	16
A5	17
A6	18
A7	19

J2 Signal			
GND	40	39	VCC
P6	38	37	CLK
/CTS0	36	35	NMI
TxD0	34	33	P4
RxD0	32	31	/RTS1
P5	30	29	P14
TxD1	28	27	/RTS0
RxD1	26	25	GND
P15	24	23	INT4
/CTS1	22	21	INT3
INT2	20	19	INT1
INT0	18	17	P2
P3	16	15	P17
P16	14	13	P24
P25	12	11	P0
P1	10	9	P10
P11	8	7	P13
P12	6	5	P29
P26	4	3	/RST = RES

Table 3.3 Signals for J1 (20x1) and J2 (20x2) expansion ports

Signal definitions for J1:

VCC	+5V power supply
GND	Ground
D0-D7	188 CPU 8-bit external data lines
A0-A7	188 CPU address lines
/WR	188 CPU pin 5
/RD	188 CPU pin 6

Signal definitions for J2:

VCC	+5V power supply, < 200 mA
GND	ground
P*xx*	188 CPU PIO pins
TxD0	188 CPU pin 2, transmit data of serial channel 0
RxD0	188 CPU pin 1, receive data of serial channel 0
TxD1	188 CPU pin 98, transmit data of serial channel 1
RxD1	188 CPU pin 99, receive data of serial channel 1
/CTS0	188 CPU pin 100, Clear-to-Send signal for SER0
/CTS1	188 CPU pin 63, Clear-to-Send signal for SER1
/RTS0	188 CPU pin 3, Request-to-Send signal for SER0
/RTS1	188 CPU pin 62, Request-to-Send signal for SER1
INT0-4	Interrupt inputs

H1 Signal			
T5	1	2	T4
T7	3	4	T6
T3	5	6	T2
T1	7	8	T0

Figure 3.5 Signals for H1 (4x2)

Signal definitions for H1:

T0 – T7	TTL output for HC259

J7 Signal			
	1	2	
/TXD0	3	4	/TXD1
/RXD0	5	6	/RXD1
	7	8	
GND	9	10	GND

Figure 3.6 Signals for J7 (5x2)

3.6.2 Jumpers

The following is a list of jumpers and connectors on the A-Core.

Name	**Size**	**Function**	**Possible Configuration**

Name	Size	Function	Possible Configuration
J1	20x1	Main expansion port, A0-A7, D0-D7, /WR, /RD	
J2	20x2	Main expansion port	
J3	3x1	SRAM selection	pin 2-3 SRAM 256K-512K pin 1-2 SRAM 32K-128K
J4	3x1	ROM size selection	pin 1-2: ROM size 32K-128K pin 2-3: ROM size 256K-512K
J5	3x1	EPROM/FLASH selection	pin 1-2: EPROM size 512K pin 2-3 FLASH size 128K/512K, or EPROM 32K-256K
J6	2x1	A19=P9	Step Two Jumper with *AC_0_115* DEBUG ROM
J7	5x2	SER0/SER1 connector	
J9	2x1	Watchdog timer	Enabled if Jumper is on Disabled if jumper is off
H1	4x2	74HC259 TTL output port	

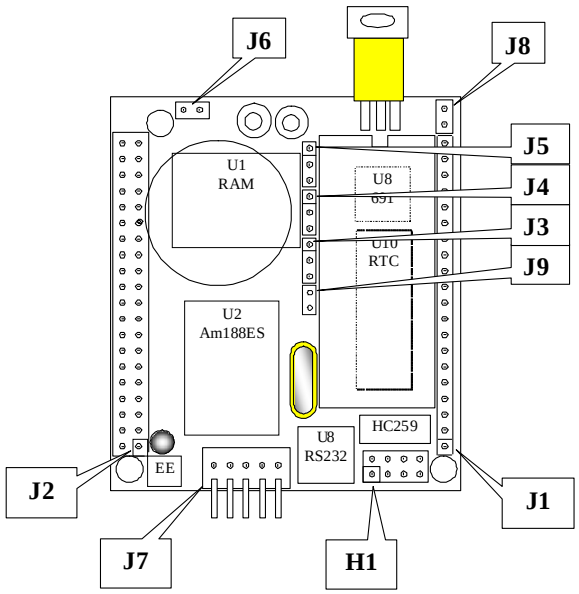


Figure 3.7 Locations of jumpers and connectors on the A-Core

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, system crashes, PC hang-ups, and other problems may result.

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, the PC may hang-up. In extreme cases, a power reset may be required to restart the PC.

As a reference, be aware that the TERN development 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 an application program from TERN libraries as well as simple memory accesses to either code or global and stack data. However, explicit access any address in I/O or memory space is allowed, and will most likely be necessary 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 for every 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** when 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. Most hardware options added to TERN controllers are mapped into I/O space, since memory space is valuable and is reserved for uses related to the code and data. Using I/O mappings, the address is output over the address bus, and the returned 16 or 8-bit value is the return value.

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

4.1 AE.LIB

AE.LIB is a C library for basic A-Core operations. It includes the following modules: AE.OBJ, SER0.OBJ, SER1.OBJ, SCC.OBJ, and AEEE.OBJ. Link AE.LIB in all applications and include the corresponding header files. The following is a list of the header files:

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

Since the A-Core is a scaled version of the A-Engine, not all function prototypes apply. For example, “AE.H” includes routines for ADCs and DACs which are not available on the A-Core.

4.2 Functions in AE.OBJ

4.2.1 A-Core Initialization

ae_init

This function should be called at the beginning of every program running on A-Core core controllers. It provides default initialization and configuration of the various I/O pins, interrupt vectors, sets up

expanded DOS I/O, and provides other processor-specific updates needed at the beginning of every program.

There are certain default pin modes and interrupt settings which may need to be changed based upon the application. With that in mind, the basic effects of **ae_init** are described below. For details regarding register use, refer to the AMD Am188ES Microcontroller User's manual.

- Initialize the upper chip select to support the default ROM. The CPU registers are configured such that:
  - Address space for the ROM is from 0x80000-0xfffff (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 increased performance is required (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.
  - 3 wait state operation. Reducing this value can improve performance.
  - Disables PSRAM, and disables need for external ready.

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

- Initialize MMCS and MPCS so that **MCS0** and **PCS0-PCS6** (except for PCS4) are configured so:
  - **MCS0** is mapped also to a 256K window at 0x80000. If used with MemCard, this chip select line is used for the I/O window.
  - Sets up **PCS5-6** lines as chip-select lines, with three wait state operation.

```
outport(0xffa8, 0xa0bf); // s8, 3 wait states
outport(0xffa6, 0x81ff); // CS0MSKH
```

- Initialize PACS so that **PCS0-PCS3** are configured so that:
  - Sets up **PCS0-3** lines as chip-select lines, with fifteen wait state operation.
  - The chip select lines starts at I/O address 0x0000, with each successive chip select line addressed 0x100 higher in I/O space.

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

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

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

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

void io_wait
Arguments: char wait
Return value: none.

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

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

4.2.2 External Interrupt Initialization

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

TERN provides functions to enable/disable all of the 8 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, write the **EOI** register word with 0x8000.

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

void int*x*_init
Arguments: unsigned char i, void interrupt far(* int*x*_isr) ())
Return value: none

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

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

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

```
void int0_init( unsigned char i, void interrupt far(* int0_isr)() );
void int1_init( unsigned char i, void interrupt far(* int1_isr)() );
void int2_init( unsigned char i, void interrupt far(* int2_isr)() );
```

```
void int3_init( unsigned char i, void interrupt far(* int3_isr)() );
void int4_init( unsigned char i, void interrupt far(* int4_isr)() );
void int5_init( unsigned char i, void interrupt far(* int5_isr)() );
void int6_init( unsigned char i, void interrupt far(* int6_isr)() );
void int7_init( unsigned char i, void interrupt far(* int7_isr)() );
void int8_init( unsigned char i, void interrupt far(* int8_isr)() );
void int9_init( unsigned char i, void interrupt far(* int9_isr)() );
void nmi_init(void interrupt far (* nmi_isr)());
```

4.2.3 I/O Initialization

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

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

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

Please see the sample program **ae_pio.c** in **`tern\186\samples\ae`**. These functions are also 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 from the PIO pins occurs when modifying the PIO registers directly with an **outport** instruction Performance in this case will be around 1-2 us to toggle any pin.

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

void pio_init
Arguments: char bit, char mode
Return value: none

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

mode refers to one of four modes of operation.

- 0 = Normal operation
- 1 = Input with pullup/down
- 2 = Output
- 3 = Input without pullup/down

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 A-Core 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 in power save mode, that means the timers will operate at a reduced speed as well.

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

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

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

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

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

The specific implementations of the timers are described in detail in chapter 8 of the AMD AM188ES User's Manual.

void t0_init
void t1_init
Arguments: int tm, int ta, int tb, void interrupt far(*t_isr)()
Return values: none

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

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

void t2_init
Arguments: int tm, int ta, void interrupt far(*t_isr)()
Return values: none.

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

4.2.5 Other library functions

On-board supervisor MAX691 or LTC691

The watchdog timer offered by the MAX691 or LTC691 offers an excellent way to monitor improper program execution. If the watchdog timer (**J9**) is connected, the function **hitwd()** must be called every 1.6 seconds of program execution. If this is not executed because of a run-time error, such as an infinite loop or stalled interrupt service routine, a hardware reset will occur.

void hitwd
Arguments: none
Return value: none

Resets the supervisor timer for another 1.6 seconds.

void led
Arguments: int ledd
Return value: none

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

Real-Time Clock

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

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

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

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

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

Void rtc_init
Arguments: char* t
Return value: none

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

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

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

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

Delay

In many applications it becomes useful to pause before executing any further code. There are functions provided to make this process easy. For applications that require precision timing, 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 an application needs to re-start the board for any reason. Depending on the current hardware configuration, this might either start executing code from the DEBUG ROM or from some other address.

4.3 Functions in SER0.OBJ/SER1.OBJ

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

The internal asynchronous serial ports are functionally identical. SER0 is used by the DEBUG ROM provided as part of the TERN EV-P/DV-P software kits for communication with the PC. As a result, debugging code directly written for serial port 0 is not allowed.

Two asynchronous serial ports are integrated in the Am188ES CPU: SER0 and SER1. Both ports have baud rates based on the 40 MHz clock, and can operate at a maximum of 1/16 of that clock rate.

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

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

The following table shows the function arguments that express each baud rate, to be used in TERN functions. These are based on a 40 MHz system clock; a 20 MHz system clock would have the baud rates halved.

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

Table 4.1 Baud rate values

After initialization by calling **`s1_init()`**, SER1 is configured as a full-duplex serial port and is ready to transmit/receive serial data at one of the specified 15 baud rates.

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

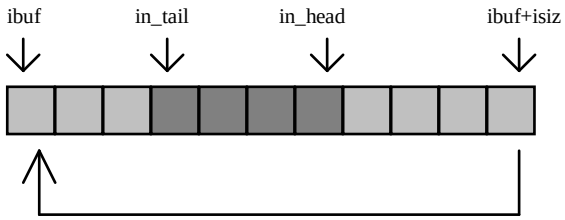


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), and baud rate (**baud**) are specified by the user with **`s1_init()`** with a default mode of 8-bit, 1 stop bit, no parity. After **`s1_init()`**set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port 0/1 Control Register (SP0CT/SP1CT) if necessary, as described in chapter 10 of the Am188ES manual for asynchronous serial ports.

Due to the nature of high-speed baud rates and possible effects from the external environment, serial input data will automatically fill in the buffer circularly without stopping, regardless of overwrite. If the user does not take out the data from the ring buffer with **`getser1()`** before the ring buffer is full, new data will overwrite the old data without warning or control. Thus it is important to provide a sufficiently large buffer if large amounts of data are transferred. For example, if 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. For example, designate a higher baud rate for transmitting data out and a slower baud rate for receiving data. This provides CPU time to do other things, without overrunning the input buffer. Use **`serhit1()`** to check the status of the input buffer and return the offset of the in_head pointer from the in_tail pointer. A return value of 0 indicates no data is available in the buffer.

Use **`getser1()`** to get the serial input data byte by byte using FIFO from the buffer. The in_tail pointer will automatically increment after every **`getser1()`** call. It is not necessary to suspend external devices from sending in serial data with /RTS. Only a hardware reset or **`s1_close()`** can stop this receiving operation.

For transmission, use **`putser1()`** to send out a byte, or use **`putsers1()`** to transmit a character string. Data can be put into the transmit ring buffer, **`s1_out_buf`**, at any time using this method. The transmit ring buffer address (**obuf**) and buffer length (**osiz)** are also specified at the time of initialization. The transmit interrupt service will check the availability of data in the transmit buffer. If there is no more data (the head and tail pointers are equal), it will disable the transmit interrupt. Otherwise, it will continue to take out the data from the out buffer, and transmit. After calling **`putser1()`** and transmit functions, other tasks can be completed with no additional software overhead on the transmitting operation. It will automatically send out all the data specified. After all data has been sent, it will clear the busy flag and be ready for the next transmission.

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

Software Interface

Before using the serial ports, they must be initialized.

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

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

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

s*n*_init
Arguments: unsigned char b, unsigned char* ibuf, int isiz, unsigned char* obuf, int osiz, COM* c
Return value: none

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

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

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

putser*n*
Arguments: unsigned char outch, COM *c

Return value: int return_value

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

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

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

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

serhit*n*
Arguments: COM *c
Return value: int value

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

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

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

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

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

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

Be careful when 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. When 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 check and set the value of these I/O pins appropriate for any form of flow control desired. For details, please refer to the Am188ES 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 serial communications, there are a few functions that can be used to reset default system resources.

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

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

clean_ser*n*
Arguments: COM *c
Return value: none
This flushes the input buffer by resetting the tail and header buffer pointers.

The asynchronous serial I/O ports available on the Am188ES Processor have many other features that might be useful for application use. Please read Chapter 10 of the User's manual for a detailed discussion of other features available.

4.4 Functions in AEEE.OBJ

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

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

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

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

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

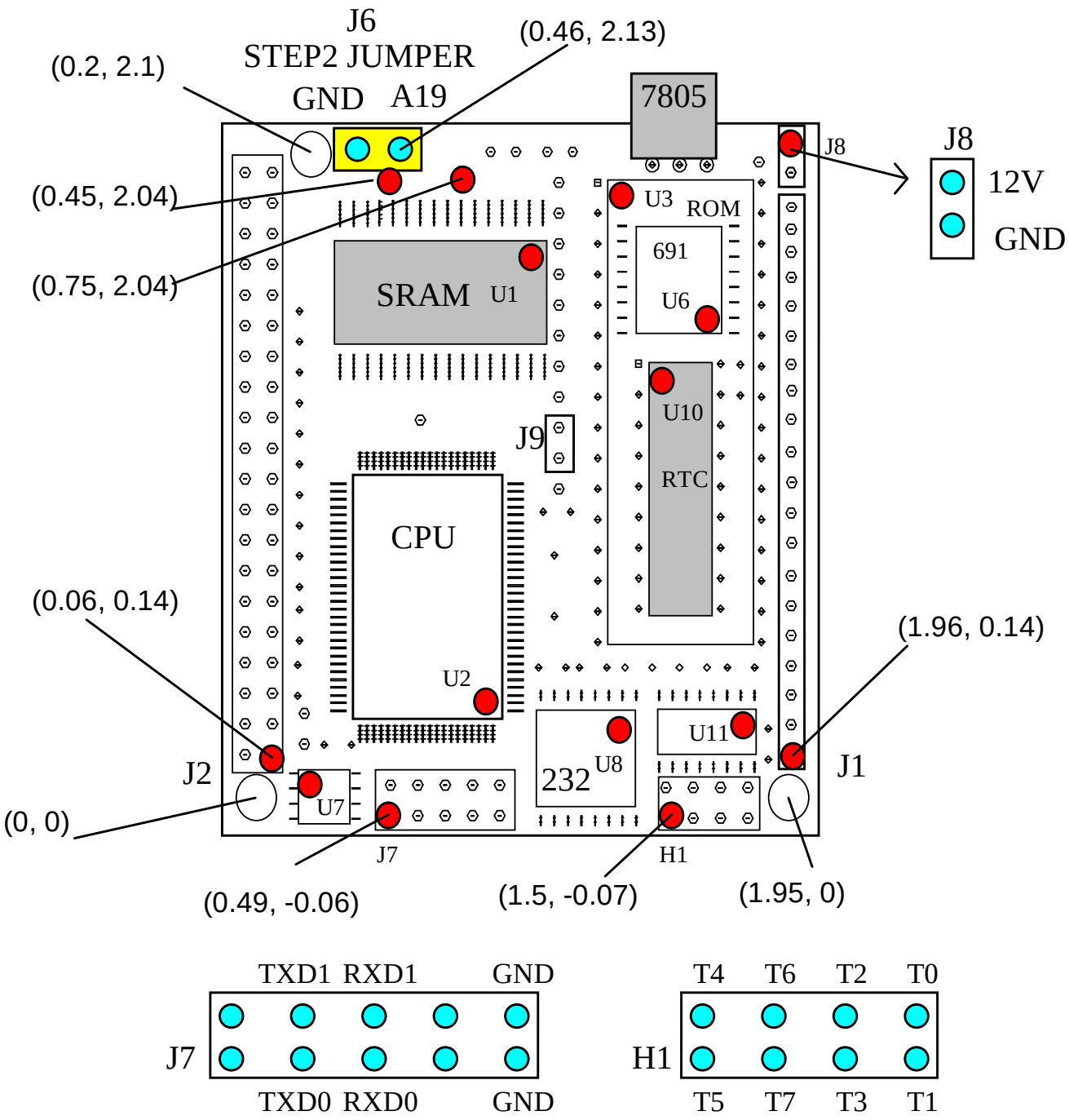
This function is used to write the passed in **dat** to the specified **addr**. The return value is 0 in success.

ee_rd
Arguments: int addr
Return value: int data

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

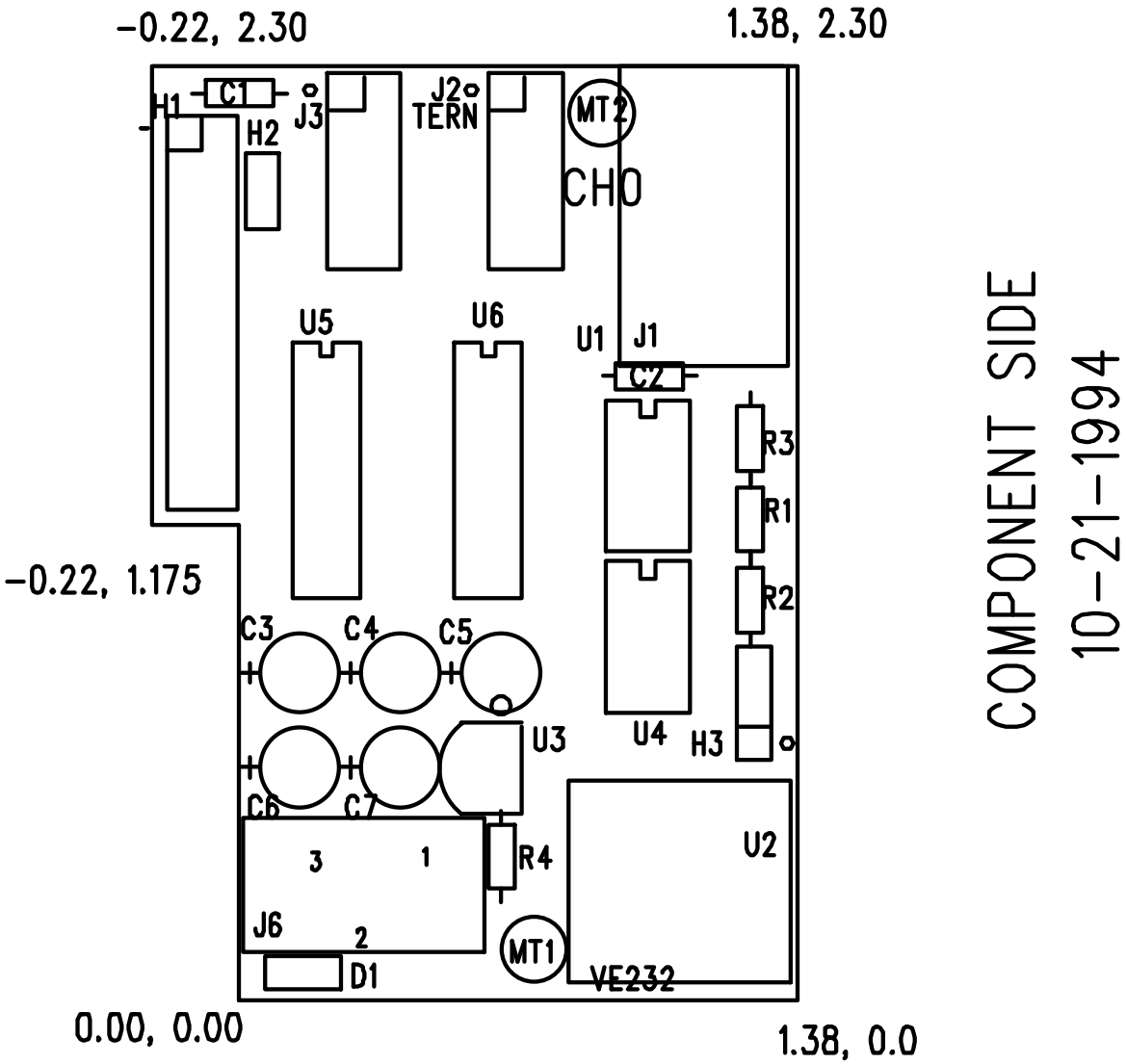
Appendix A: A-Core Layout

All dimensions are in inches.



Appendix B: VE232 Pin Layout

All dimensions are in inches.



Appendix C: RTC72421

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: Serial EEPROM Map

Part of the on-board serial EEPROM locations are used by system software. Application programs must not use these locations.

0x00	Node Address, for networking
0x01	Board Type
0x02	
0x03	
0x04	SER0_receive, used by ser0.c
0x05	SER0_transmit, used by ser0.c
0x06	SER1_receive, used by ser1.c
0x07	SER1_transmit, used by ser1.c
0x10	CS high byte, used by ACTR™
0x11	CS low byte, used by ACTR™
0x12	IP high byte, used by ACTR™
0x13	IP low byte, used by ACTR™
0x14 – 0x1f	Reserved for TERN use
0x20 – 0x1ff	For customer use

Appendix E: Software Glossary

The following is a glossary of library functions for the A-Core.

---

void ae_init(void) ae.h

Initializes the AM188ES 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);
clkb_en(0);
enable();
```

Reference: led.c

---

void ae_reset(void) ae.h

Resets AM188 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 the LED.

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

Reference: led.c

---

void delay0(unsigned int t) ae.h

Software loop delay: *While (t--);*

```
Var:  m – Delay in number of iterations
```

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
void clkb_en(int i)

Enables output clock 'a' and 'b' respectively.

```
Var: i – turns clock on/off.
```

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

Set up 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 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 interrupt far t2_isr(void); ae.h
void interrupt far t1_isr(void);
void interrupt far t0_isr(void);

Timer 0, 1, 2 interrupt service routine.

```
Var: None.
*note:  End of interrupt must be issued in ISR to reset interrupt.
See AMD CPU Manual page 7-27 for EOI register.
```

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 interrupt far nmi_isr(void); ae.h
void interrupt far int0_isr(void);
void interrupt far int1_isr(void);
void interrupt far int2_isr(void);
void interrupt far int3_isr(void);
void interrupt far int4_isr(void);
void interrupt far int5_isr(void);
void interrupt far int6_isr(void);

Interrupt service routine for int0 – int6 and NMI.

```
Var: None.

*note:  End of interrupt must be issued in ISR to reset interrupt.
See AMD CPU Manual page 7-27 for EOI register.
```

Reference: intx.c

void s0_init(unsigned char b, unsigned char* ibuf, int isiz, ser0.h
unsigned char* obuf, int osiz, COM *c) (void);
void s1_init(unsigned char b, unsigned char* ibuf, int isiz, ser1.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
12	115200	57600
13	23400	115200
14	460800	23400
15	921600	460800

Reference: s0_echo.c, s1_echo.c, s1_0.c

---

int putser0(unsigned char ch, COM *c); ser0.h
int putser1(unsigned char ch, COM *c); ser1.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, s1_echo.c, s1_0.c

---

int putsers0(unsigned char *str, COM *c); ser0.h
int putsers1(unsigned char *str, COM *c); ser1.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

Reference: ser1_sin.c

int serhit0(COM *c); ser0.h
int serhit1(COM *c); ser1.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, s1_echo.c, s1_0.c

---

unsigned char getser0(COM *c); ser0.h
unsigned char getser1(COM *c); ser1.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_echo.c, s1_0.c

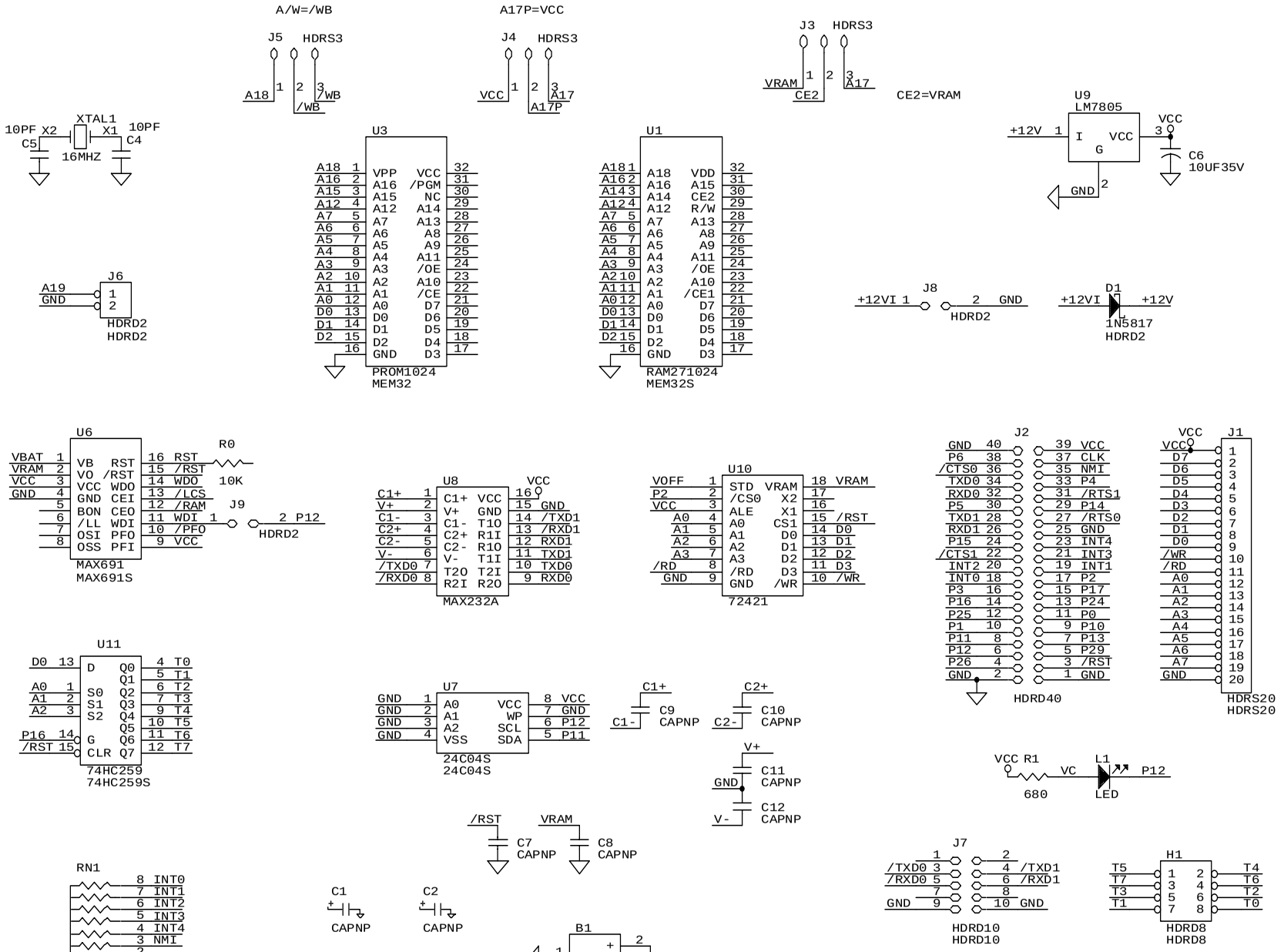
---

int getsers0(COM *c, int len, unsigned char *str); ser0.h
int getsers1(COM *c, int len, unsigned char *str); ser1.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: ser1.h, ser0.h for source code.



STE		
Title	A-CORE-88	
Size B	Document Number AC-MAN.SCH	REV
Date: January 17, 2000	Sheet 1 of 1	